Java Garbage Collection Scheduling in Utility Accrual Scheduling Environments

Shahrooz Feizabadi
Godmar Back
{shahrooz,gback}@cs.vt.edu
Virginia Tech
Blacksburg, VA 24061, U.S.A

ABSTRACT

Convenience, reliability, and effectiveness of automatic memory management have long been established in modern systems and programming languages such as Java. The timeliness requirements of real-time systems, however, impose specific demands on the operational parameters of the garbage collector. The memory requirements of real-time tasks must be accommodated with a predictable impact on the timeline, and under the purview of the scheduler.

Utility Accrual is a method of dynamic overload scheduling that is designed to respond to CPU overload conditions by producing a schedule that heuristically maximize a predefined metric of utility. There also exists in such systems the possibility of memory overload situations in which the cumulative memory demand exceeds the amount of memory available.

This paper presents a utility accrual algorithm for uniprocessor CPU and garbage collection scheduling that addresses memory overload conditions. By tightly linking CPU and memory allocation, the scheduler can now appropriately respond to overload along both dimensions. This scheduler is the first of its kind to enable the use of automatic memory management in a utility accrual system. Experimental results using actual Java application profiles indicate the viability of this model.

1. INTRODUCTION

Some real-time systems must function in environments with a high degree of uncertainty. No significant a priori knowledge of the system's operating conditions can be assumed. Scheduling of real-time tasks must therefore be accordingly adjusted to the unfolding of the environment's dynamic, context-dependent parameters. CPU load is one such parameter which varies as a function of the number and demands of real-time tasks present in the system.

Utility Accrual (UA) scheduling is one method of adapting to CPU overload conditions while maintaining predictable temporal behavior. Such schedulers are designed to produce deterministic real-time guarantees during underload, e.g., meeting all deadlines up to 100% CPU load. As the load increases beyond 100%, however, these dynamic schedulers will select only a time-feasible subset of the contending tasks based on the attributes of each task. We have previously demonstrated [6] the viability of implementing complex UA schedulers under real-time Java.

Absent system-level awareness, memory is often treated as a monolithic resource by the memory manager where allocation requests are indiscriminately serviced. Similar to the CPU, the aggregate demand on system memory can dynamically exceed availability and lead to memory overload conditions. A memory manager, tightly coupled to the scheduler, would be able to make better decisions given its widened view of the system. Conversely, the awareness of the system's memory status likewise enables the scheduler to enact more sophisticated admission policies, consider memory constraints at scheduling points, and appropriately budget for the memory operations time requirements. Given the inherent execution overhead of the garbage collector (GC) and its potentially disruptive activation, it is necessary for real-time schedulers to monitor and manage the GC as accurately as possible.

This paper presents CADUS (Collector Aware Dynamic Utility accrual Scheduler), a scalable scheduling algorithm intended to produce predictable, graceful, performance degradation during CPU and/or memory overload conditions. The paper makes the following contributions:

The CADUS algorithm — During CPU overload the algorithm chooses only the most "desirable" tasks for execution; it adjusts its admission policy during memory overload; and, it calculates and budgets for garbage collection overhead while scheduling the collector appropriately.

The TPUD task ordering — Three-dimensional Potential Utility Density (TPUD) is a performance metric to produce a ranking of tasks based on their CPU time and memory requirements combined with their potential utility to the system as a whole. Furthermore, this metric can be tuned to introduce a bias towards CPU or memory based on the task's cycle demands and allocation patterns.

JTRES 2005 San Diego, CA, USA

The PRD task ordering — The Potential Reclamation Density (PRD) is a metric to evaluate a task's capacity for memory reclamation.

Workload Evaluation — We profiled and traced actual Java programs and evaluated the scheduler's behavior against those.

2. BACKGROUND

2.1 Utility Accrual Scheduling

We assume a dynamic system in which tasks can arrive at any time. Each task is associated with an expected utility, which represents an application-specific quality of service metric. This metric is expressed by a Time/Utility Function (TUF) which maps a task's completion time [10, 9] to a utility gain value. Two example TUFs are shown in Figure 1. The gray blocks in the figure represents the execution time of the task starting at t_s and completing execution at t_c, and can be anywhere between the task arrival time t_a and the task deadline t_d.

Figure 1 (a) illustrates a step-TUF corresponding to a hard deadline where a task yields full utility prior to the deadline and zero afterwards. While deadline-based systems have a wide range of applications, gradually declined TUF such as the one shown in Figure 1 (b) are also used.

Scheduling disciplines such as Earliest Deadline First (EDF) can provide optimal sequencing of tasks up to 100% processor utilization. Beyond 100% load, however, UA schedulers must select a sequence of tasks from a "feasible" subset of all contending tasks $J = \{j_1, \ldots, j_n\}$, based on the expected utility gained from completing these tasks. The objective of the scheduler is to maximize system-wide accrued utility: $U_\sigma = \sum_{i=1}^{|\sigma|} u_i(t_c)$. An optimal schedule, $\sigma_{optimal}$, is a sequence of tasks that yields maximal utility.

Because finding an optimal schedule is $\mathcal{NP}$-hard [5], dynamic, on-line, real-time UA schedulers must adopt heuristic approaches with limited time complexity. CADUS accomplishes this by using a hybrid greedy/combinatorial approach.

2.2 Memory Management and Garbage Collection Model

In previous work [1], we have demonstrated that it is feasible to implement multiple logical heaps within a single Java runtime system. In such a system, each task is provided

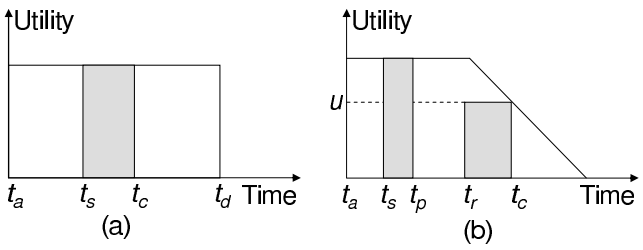

Figure 1: Examples of Time/Utility Functions (TUFs)

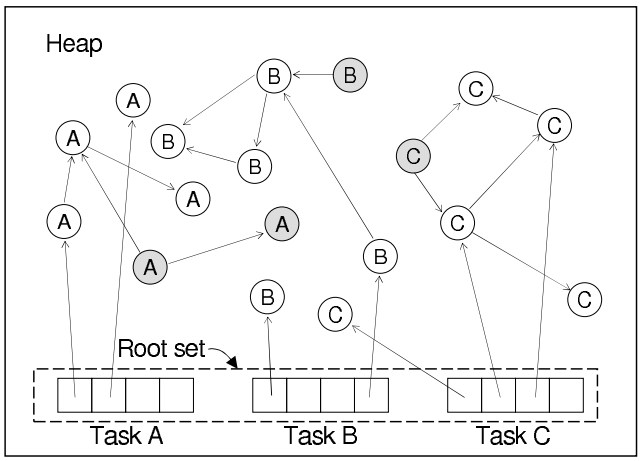

Figure 2: Logical Task Heap Model

with its own logical heap that represents a portion of the physical heap. We have shown that each task's heap can be separately garbage collected if there are no cross-heap references between tasks. We have developed a scheme for managing or avoiding such cross-heap references. After a task instance terminates, its objects must be returned to the physical heap. If there are multiple instances of a periodic task, each instance starts with a blank slate. Our system uses a central heap for permanent, shared objects. In this paper, we assume a system with the above properties. An example of a physical heap with logical heaps for 3 tasks A, B, and C is shown in Figure 2.

Allocation requests are fulfilled from the shared physical heap. A garbage collector is responsible for reclaiming unused memory. In systems without real-time requirements, a collection is triggered when an allocation request by a mutator fails and more memory is needed to satisfy the request. For real-time systems, however, such reactively triggered GC is often unsuitable as the mutator could run out of memory at an unpredictable time and be forced to yield the processor to the collector for an unknown period [11].

Solving this problem is the subject of real-time garbage collection algorithms [3, 7, 2]. These approaches address two questions: how to build an allocator and collector such that the time required by the operations necessary to perform allocation and collection is "bounded by a small constant" [3]. Second, and equally important, is the question of garbage collection scheduling: deciding when to collect and for how long. Known scheduling strategies can be divided in *work-based* and *time-based* approaches. Work-based approaches schedule collection work based on the amount of memory a mutator has allocated, while time-based approaches schedule collection work based on the amount of time the mutator has progressed.

CADUS presents a scheduling strategy for garbage collection in which the expected utility of a task, in relation to its time and memory allocation requirements, determines when the task—and any necessary collection work—should be scheduled. As such, this paper does not present a new real-time garbage collection algorithm, but rather a strategy

of how to integrate the scheduling of garbage collection in the context of a utility accrual framework. We believe that CADUS's scheduling strategy could be used with different collection mechanisms, provided the following requirements on the collector are fulfilled.

We require (1) that the collector is precise, such that a full GC cycle could reclaim all garbage. We require (2) that time bounds are known for the collector's operation, and that these time bounds can be computed from the memory allocation profile of a task as follows

$$GC_{time,i} <= f_m(m_{live,i}) + f_s(m_{alloc,i} - m_{live,i})$$

f_m accounts for the cost involved in root scanning and tracing the reachability graph, as well as possible context switch cost to and from the collector. We assume that the number of roots is small and has no significant impact on total GC time. f_s accounts for the time spent reclaiming objects, which is determined by the amount of garbage $g = m_{alloc,i} - m_{live,i}$; this time is assumed to include the time needed to reinitialize the memory for subsequent use. See Section 4.1 for how we determine realistic f_m and f_s for our experiments.

We require (3) that the collector be preemptible: should a new task arrive while a collection is in progress, the CADUS must have the option of preempting an ongoing collection. It is acceptable that the collector does not return any memory in this case; we assume negligible preemption delay.

We believe that variants of Baker's real-time treadmill collector [3], such as the collector used in the SPIN OS [14], could be used in our model; for a detailed discussion of the implementation trade-offs regarding collector overhead and mutator utilization we refer to [2].

2.3 Allocation Profiles

We obtain the memory allocation profile of a task by tracing the task's memory consumption. The cumulative memory allocation of a task is a monotonically increasing function during the task's life time. Two examples are shown in Figure 3. The JPEG decompression task uses `com.sun.image.codec.jpeg.JPEGImageDecoder.decodeAsBufferedImage` to decode an image, while the Matrix decomposition task constructs matrices of varying but known sizes and performs three decompositions using routines from the JAMA matrix library. In this paper, we assume an input-independent allocation profile, as is the case for the two examples shown. Note that the JPEG-Decoder's profile is independent of the size of the images being decoded, except for a scaling factor. In general, worst-case allocation requirement analysis would be required, analogous to worst-case execution time analysis to determine a task's CPU cost. Static techniques to determine allocation amounts, such as those described by Mann et al [17], may be applicable as well.

3. THE CADUS ALGORITHM

The CADUS algorithm is an on-line, preemptive, dynamic, heuristic, soft real-time, utility accrual scheduling algorithm. It is invoked at the following scheduling points: task arrival, task departure, resource request, and resource release.

During CPU and/or memory overload conditions—CADUS's intended operating environment—the algorithm constructs a schedule with the objective of maximizing utility. Only a subset of tasks can satisfy their timing constraints during overload. This feasible subset of all tasks is chosen based on a metric that reflects their relative worth. One commonly used metric is Potential Utility Density (PUD), defined as the utility yield u_i of a task at completion time, divided by its remaining execution time t_i. This is analogous to the notion of "return on investment": cycle for cycle, a higher PUD task is more beneficial than a lower PUD task.

We extend PUD along a third dimension by defining Three-dimensional Potential Utility Density (TPUD), as the ratio of utility yield over combined remaining execution time t_i and remaining maximum allocation memory requirement m_i, where m_i includes the amount of memory currently kept alive by that task plus all objects it would allocate if let run to completion. TPUD is thus defined as

$$TPUD(u_i, t_i, m_i) = \frac{u_i}{f(t_i, m_i)}$$

where $f(t_i, m_i)$ represents the relative cost contributions of time and memory. Different choices of f allow for different assignments of relative weights to CPU and memory bandwidth. We use a linear function $f(t_i, m_i) = m_i + kt_i$ where k represents the allocation rate in bytes/cycles of a representative task in the system.

The rationale for this choice is that two tasks with equal utility and typical allocation rates should be able to trade time for memory and vice versa and obtain the same TPUD. However, tasks with higher allocation rates should have to give up more memory to achieve the same TPUD for a smaller increase in execution time.

When creating a feasible schedule, CADUS may have to include garbage collections or shed load to obtain sufficient memory to satisfy the memory requirements of the tasks that become part of the schedule. For this purpose, we define PRD, or Potential Reclamation Density, of a preempted task i as follows:

$$PRD(i) = \begin{cases} \frac{m_{alloc,i} - m_{live,i}}{GC_{time,i}} & \text{if task } i \text{ is collected} \\ \frac{m_{alloc,i}}{f_s(m_{alloc,i})} & \text{if task } i \text{ is killed} \end{cases}$$

This function reflects the choices CADUS has to obtain more memory: it can either garbage collect an existing task i with a yield equal to the amount of garbage that task's logical heap contains, or it can terminate the task, yielding all its memory. In the first case, the time required is the time to complete a GC cycle for the chosen task; in the second case, it is the time required to sweep and reinitialize all its memory. Intuitively, the maximum PRD task provides the highest potential rate of memory return for the least amount of work — the biggest memory bang for the cycles spent. The PRD performance metric establishes a mechanism to distinguish tasks based on how quickly, and to what extent, they could replenish memory.

3.1 Algorithm Description

A high-level outline of CADUS is provided in Algorithm 3.1. CADUS maintains two task queues at all times: the ready queue and the ineligible queue. The ineligible queue is composed of tasks that are no longer allowed access to the CPU:

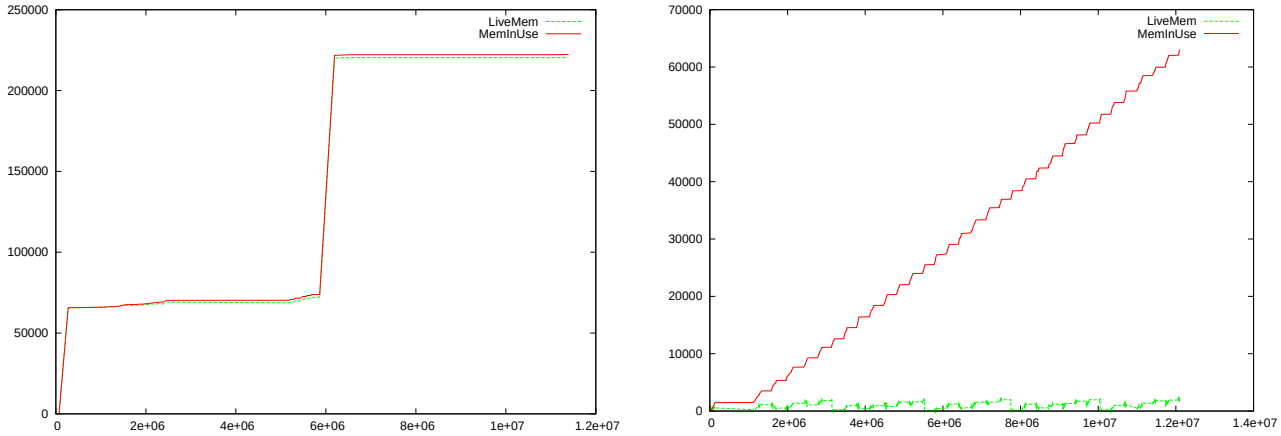

Figure 3: Allocation Profile of a JPEGDecompression Task (left) and Matrix Decomposition (right). The y-axis is in bytes and the x-axis is in cycles.

(1) Tasks that completed execution and voluntarily released the CPU but whose memory has not been collected; (2) Terminated tasks; and (3) Time-infeasible tasks. Time infeasibility is defined as a task's inability to potentially gain positive utility at completion even if granted system resources now. At invocation, CADUS purges from the ready queue (i.e., moves to the ineligible queue) all tasks that have become time-infeasible.

CADUS needs to construct a schedule that is a subset of the ready queue and is feasible with respect to time, resource, and memory constraints. A schedule σ includes a sequence of tasks to be run, interspersed with task termination actions and garbage collections. A key observation is that unlike such scheduling algorithms as DASA, CADUS cannot decide on the subset of the tasks that will be part of the feasible schedule independent of the order in which they are scheduled, because the schedule order determines the available memory and the amount of GC work that must be performed between tasks. For this reason, CADUS attempts to construct feasible schedules by successively considering subsets of tasks with increasing deadline order (or, in the case of non-step TUFs, in optimal completion order) and checking them for feasibility at each step.

For each subset under consideration, CADUS checks for time, resource, and memory feasibility. If the subset is found to be feasible, then the task with the next farther deadline is tentatively included and an attempt is made to make the combined set feasible. If the combined set is feasible and yields a higher utility than the previous tentative set, the task is included into the set. If the combined set is not feasible, tasks are dropped from the set, starting with the lowest TPUD task, until a feasible set has been found. (Once a feasible set has been found, as an optional optimization, we try to flesh out the set by re-including previously skipped tasks if possible.)

For a given subset, feasibility is checked in three steps. First, time feasibility is checked: a schedule is time-feasible by a given deadline if its combined time requirement is less than the time remaining to that deadline, and all tasks would yield a positive utility if granted the CPU in deadline order. We use the time feasibility check to quickly reject schedules that include too many tasks even before resource and memory constraints are taken into account.

Second, resource feasibility is checked. We obtain a tentative schedule by topologically sorting the resource dependency graph, breaking ties by deadline order. If the resource dependency graph contains cycles, a deadlock condition is flagged. If any task in the set under consideration is dependent on a task outside the set (say if the outside task holds a resource on which an in-subset task is blocked), a tentative "kill action" is added to the schedule before the dependent task. The resulting topological sorted schedule is checked for time feasibility. If the schedule is time and resource feasible, we check for its memory feasibility.

A task is considered memory feasible if its remaining allocation requirements can be assured for the remainder of the task's execution time. CADUS will not context switch into a task unless it can provide sufficient memory for it up-front. Here, the objective is to "snowplow" the road ahead of the task so that once activated, the task can be assured of no delays due to a reactively triggered collection. A memory feasible schedule is one composed of memory feasible tasks.

Should the tentative schedule be found memory infeasible, CADUS identifies the slack (if any) in the schedule, and attempts to interpose enough collection work in the schedule to make it memory feasible. The algorithm considers tasks in the tentative schedule in ascending deadline order. The slack between now and the task's start time is calculated such that once activated, the task will complete at its deadline. CADUS knows, from a task's memory allocation profile, the amount of memory required by the task to finish execution. The slack just calculated tells CADUS how much time it has to free the needed memory. CADUS must now determine if this is possible.

A "reclamation plan" is devised to determine the possibility of sufficient GC action in the slack ahead of the task. To construct this plan, we consider a set of *memory donors* for

Algorithm 1: CADUS

Input: J: Unordered set of all jobs (tasks) in the system

Output: σ_{final}: Task sequence (schedule) to be dispatched in order

/*Notation: Sets and subsets are represented as J and S. Ordered sets (sequences and subsequences) are represented as σ's. */

begin

$S_{tentative} \longleftarrow \emptyset$

$\sigma_{PRD} \longleftarrow J$ Sorted in descending PRD order

foreach $j \in J$ **do**

Purge j if time-infeasible: $J \longleftarrow J - \{j\}$

1 /*Topological sort to determine resource dependencies, breaking ties by deadline */

$\sigma_{topsort} \longleftarrow topSort(S_{cand})$

$\sigma_{DL} \longleftarrow J$ Sorted in ascending deadline order

$\sigma_{TPUD} \longleftarrow J$ Sorted in descending TPUD order

/*Consider tasks in deadline order: */

2 **foreach** $j \in \sigma_{DL}$ **do**

$S_{cand} \longleftarrow S_{tentative} \cup \{j\}$

$\sigma_{cand} \longleftarrow$ makeTimeAndMemoryFeasible(S_{cand})

if $\sigma_{cand} \neq \emptyset$ **then**

/*Tentative schedule does not exceed system capacity yet */

$S_{tentative} \longleftarrow S_{cand}$

$\sigma_{tentative} \longleftarrow \sigma_{cand}$

continue;

/*System capacity exceeded, sacrifice lowest-TPUD task and re-evaluate */

$\sigma_{tpud} \longleftarrow S_{cand}$ Sorted in ascending TPUD order

$S_{dropped} \longleftarrow \emptyset$

3 **while** $j \in \sigma_{tpud}$ *and* $\sigma_{cand} = \emptyset$ **do**

$S_{cand} \longleftarrow S_{cand} - \{j\}$

$S_{dropped} \longleftarrow S_{dropped} \cup \{j\}$

4 $\sigma_{cand} \longleftarrow$ makeTimeAndMemoryFeasible(S_{cand})

/*Optional optimization -- see if previously dropped tasks can now be accommodated */

$\sigma_{DTPUD} \longleftarrow S_{dropped}$ Sorted in descending TPUD order

foreach $(j \in \sigma_{DTPUD})$ **do**

$\sigma_{tmp} =$makeTimeAndMemoryFeasible($S_{cand} \cup j$)

if $\sigma_{tmp} \neq \emptyset$ **then**

$S_{cand} \longleftarrow S_{cand} \cup j$

$\sigma_{cand} \longleftarrow \sigma_{tmp}$

/*Accept new schedule only if it yields higher utility: */

if *utility*(S_{cand}) > *utility*($S_{tentative}$) **then**

$S_{tentative} \longleftarrow S_{cand}$

$\sigma_{tentative} \longleftarrow \sigma_{cand}$

$\sigma_{final} \longleftarrow \sigma_{tentative}$

end

Algorithm 2: makeTimeAndMemoryFeasible

Input: S_{cand}: Set of possible candidates for inclusion in final schedule

Output: σ: Ordered sequence of time and memory feasible tasks in S_{cand} interdispersed with reclamation and kill actions; $\emptyset$ otherwise

begin

$\sigma_{feasible} \longleftarrow \emptyset$

$potMemDonors \longleftarrow \emptyset$

if *notTimeFeasible*(S_{cand}) **then**

return $\emptyset$

/*Retrieve topological order of cand set $\oplus$ denotes sequence concatenation: */

$\sigma_{tent} \longleftarrow \oplus\{\forall i : i \in \sigma_{topsort} \wedge i \in S_{cand}\}$

$S_{blockers} \longleftarrow \{j : j$ holds resource(s) needed by $S_{cand}\}$

foreach $j \in S_{blockers}$ **do**

$\sigma_{feasible} \longleftarrow \sigma_{feasible} \oplus \{kill(j)\}$

/*Memory can be reclaimed from 3 possible sources:
1 - All memory held by an ineligible task;
2 - Kill any task, reclaim all its memory, or;
3 - Harvest the garbage generated by any task;
*/

$potMemDonors \longleftarrow \{\ gc(t) : \forall t \in ineligibleQueue\}$

$potMemDonors \longleftarrow \cup \{\ kill(t) : \forall t \in readyQueue\}$

$potMemDonors \longleftarrow \cup \{\ gc(t) : \forall t \in readyQueue\}$

/*Subtract tasks we care about: */

$potMemDonors \longleftarrow potMemDonors - \{kill(t) : \forall t \in S_{cand}\}$

/*Descending PRD order lookup in σ_{PRD}: */

$\sigma_{donors} \longleftarrow orderByPRD(potMemDonors)$

5 **foreach** $j \in \sigma_{tent}$ **do**

$memNeeded \longleftarrow j.deficit$

$memDonors \longleftarrow \emptyset$

6 **while** $j \in \sigma_{donors} \wedge memDonors.memYield <$ $memNeeded$ **do**

7 **if** $memDonors.reclamationTime + j.reclamationTime > slack$ **then**

continue;

else

$memDonors \longleftarrow memDonors \cup \{j\}$

$potMemDonors \longleftarrow$ $potMemDonors - \{j\}$

if $memDonors.memYield < memNeeded$ **then**

/*Cannot make the schedule memory feasible */

return $\emptyset$

$\sigma_{feasible} \longleftarrow \sigma_{feasible} \oplus memDonors \oplus j$

return $\sigma_{feasible}$

end

inclusion in the schedule. Potential memory donors include tasks that have already run to completion, but whose memory has not been reclaimed, and tasks that are currently preempted and that have a positive memory yield if they were garbage collected now. PRD values are calculated for each donor: for preempted tasks, the PRD corresponding to reclaiming the garbage generated by the task is computed as well as the PRD corresponding to the termination of the task to reclaim all memory held by it. For ineligible, but uncollected tasks the PRD is computed based on reclaiming their memory. Tasks are then ordered by descending PRD values.

The reclamation plan is composed of GC work, in the slack period, on behalf of tasks in the PRD sequence to free sufficient memory if possible. In other words, CADUS attempts to fill the slack with just enough GC work (based on PRD order) to meet the requirements of the task ahead. If the choice based on highest PRD would make the following task miss its deadline, it is skipped and the next-lower PRD choice is examined to see if it would yield enough memory while keeping the next task time feasible. If a task is found to be memory feasible at its position in the schedule, it is added to the set of memory donors such that subsequent tasks have the option of reclaiming its memory to satisfy their memory requirements. Algorithm 3.1 outlines the construction of the reclamation plan.

The resulting schedule is time, resource, and memory feasible. Execution continues with the first item in the schedule, which could be starting a new task, starting a collection on behalf of a preempted or terminated task, or killing a task in order to obtain a resource it holds or to reap its memory.

CADUS dispatches tasks in deadline order during underload conditions. As such, it reduces to EDF [15] for CPU utilizations of up to 100%, and therefore provides the same optimality. Furthermore, given its pro-active GC scheme, CADUS can provide the assurance that once a task is activated, it will not block on GC. Accuracy of the task's allocation profile, however, bears directly on the strength of the "GC safety" assurance provided.

3.2 Algorithm Complexity

CADUS has a computational complexity of $O(rn^3)$ where n is the number of tasks, and r is the number of resources in the system. Initially, several sorting operations are performed in sequence, each costing $O(n \log n)$. Subsequently, at the top level, the algorithm iterates over all tasks in deadline order (Alg. 3.1, (step 1) at a cost of $O(n)$. The inner `while` loop (step 2) in which tasks are dropped until a time-and-memory-feasible schedule is found can execute at most n times. In practice, however, it would rarely execute n times: the worst case of n would occur only if all tasks in the candidate set would need to be dropped in order to accommodate the task just encountered.

The `makeTimeAndMemoryFeasible` method (step 3) is thus invoked at most $O(n^2)$ times. Step 4 of `makeTimeAndMemoryFeasible` dominates this method, and involves rn operations. The `for` loop in step 5 executes at most n times if condition 7 is always false, because σ_{tent} and $potMemDonors$ are both bounded by n. This condition checks whether a task that has a nominally higher PRD should be skipped in the reclamation plan because its garbage collection would take too long. We can place an upper bound on the number of checks we perform (not shown in the algorithm); after that bound is reached, we drop the offending high-PRD task from $potMemDonors$. By including this precautionary measure to handle this however unlikely case, the overall worst-case complexity of CADUS is therefore $O(rn^3)$.

On average, we expect a complexity of $O(rn^2)$, assuming that only a limited number of tasks needs to be dropped to find a feasible schedule, which should be the case unless the algorithm is operating under extreme overload. We are currently working on a version of the algorithm in which the new version of σ_{tent} in loop 2 would be computed by merging the previous σ_{tent} with the next task j, which will reduce the complexity further.

4. RESULTS

We built a simulator to evaluate the effectiveness of CADUS both against synthetic as well as real workloads.

4.1 Tracing Real Workloads

We tested CADUS against traces obtained from real workloads. We used Sun's JDK 1.5.02 running on a Linux 3GHz Pentium 4 machine with 1GB of memory. To obtain an estimate of a task's mutator cost, we used the Pentium cycle counter and measured the time spent in the section of interest. Prior to starting the measurements, we ran the task several times to allow the Hotspot JVM to select the methods involved for just-in-time compilation. We sized the JVM's heap such that no garbage collection occurred during the run.

To obtain the memory traces of a task, we wrote a JVMTI agent. This agent interposes on all allocations a task performs, including requests from the VM and requests stemming from native (JNI) code. At each allocation, it performs a complete traversal of all reachable objects and counts and records their size. In addition, we record the size of the objects allocated and create an object size profile of each task.

To obtain realistic assumptions regarding how long a garbage collection would take, we timed the duration of full collections for a workload where we varied the size of the live memory in steps, and triggered full collections at each step (the JVM uses a generational collector which does not traverse the entire live memory during minor collections.) We recorded the output of verbosegc and fit a quadratic equation to this curve with a correlation coefficient $r^2 > 0.999$. We determined the minimum cost for a collection with zero live memory by observing the duration of a minor GC cycle with an empty Eden space. This cost (about $118\mu s$) represents the assumed minimum amount a collection would take if a task holds no live memory.

To obtain realistic assumptions regarding sweep cost, we assumed that during a sweep an object must be unlinked from a linked list and its content be zeroed. The relative overhead of both operations depends on the size of the object. We therefore measured the sweep cost for objects of different sizes. To determine the overall sweep cost of a task, we

Benchmark	Cost in Mcycles	Max Live (KB)	MaxAlloc (KB)	Sweep (Mcycles)	Allocations	Rate bytes/cycle
Grep (small)	8.2	4.73	308.1	1.43	2397	0.0387
Grep (large)	83.7	36.48	7477.9	49.18	20079	0.0914
Matrix (small)	12.1	2.11	61.6	0.30	1798	0.0052
Matrix (medium)	34.4	2.38	128.6	0.63	3625	0.0038
Matrix (large)	73.7	2.71	198.1	0.96	5370	0.0028
JPEG (mini)	11.4	215.28	217.1	1.20	103	0.0195
JPEG (800)	174.5	1945.91	1947.8	8.55	103	0.0114
JPEG (1024)	274.3	3142.91	3144.8	13.64	103	0.0117
JPEG (1280)	464.4	5190.91	5192.8	22.34	103	0.0114

Table 1: Workload Characterization of Different Java Tasks

multiplied the size listed in the object size profile with their frequency and the measured sweep cost for objects of that particular size.

We implemented three benchmarks: a JPEG decoder task, a Grep task that performs a regular expression match over a given string, and a Matrix task that computes the eigenvalue, LU, and QR-decompositions for a number of "magic square" matrices. Table 1 shows the results. Note that the sweep costs are significant, in particular for tasks with high allocation rates. For instance, Grep (large) finishes in 83.7 Mcycles without garbage collection, but it would take 49.18 Mcycles to unlink and zero out the 7477.9 KB of garbage it generates. Current RAM technology provides a peak bandwidth between 2 and 6 GB/s, which means that sweep times will likely remain significant.

4.2 Simulation

We developed a discrete-event based simulation framework that combines process-based and event-based simulation. Events include task arrivals; task themselves are simulated using a multi-threaded, process-based approach that directly reflects a task's execution. A discrete event engine manages the threads implementing the tasks in the system and calls into the scheduler whenever a scheduling event occurs. Schedulers must react to scheduling events; they also must provide a dispatcher function that tells the simulator which task to schedule next. Our simulator also supports modeling resources and keeps track of the resource dependency graph for use by the scheduler.

We implemented CADUS as well as DASA [5], LBESA [16], and GUS [13] in our framework. Figure 4 shows the behavior of CADUS compared to these schedulers in the memory underload case. It can be seen that CADUS handles various CPU loads similar or better to the others; this is not surprising since CADUS explores more of the search space than those algorithms.

The true distinction comes from the fact that these schedulers are not memory aware; therefore, we had to make decisions as to how they would react in the presence of limited memory. We allowed for reactive GC to simulate involuntary GC pauses. In addition, memory-unaware schedulers will always sweep a task's memory immediately after it completes.

We are currently working on analyzing the behavior of CADUS for varying memory loads. For now, we give two scenarios to illustrate the usefulness of collection-aware scheduling. In scenario 1, two tasks arrive at time 0 with a period of 24.5Mc: an instance of JPEG(mini) with a utility 30, and an instance of Matrix(small) with a utility of 20. Including sweep costs, the offered load from this task set is greater than 1 - however, it is still possible to finish both tasks at least in some of the periods by scheduling JPEG(mini) first, then running Matrix(small) and postponing its sweep cost until the deadline. Eventually, the sweeping will have to be done, but being aware of how much memory is available at any given point in time can be used to achieve higher overall utility by postponing sweeping when possible. This behavior is shown in Figure 5. A memory-unaware scheduler such a DASA, will be forced to sweep right after executing JPEG(mini), pushing Matrix(small) past its deadline during every single period - even though at least in some periods it would be possible to just run it in the provided heap of size 400,000bytes.

A second example is shown in Figure 6. In this scenario, a heapsize of 320K is used. A JPEG(mini) task with low utility arrives at time 0, both schedulers decide to schedule this task at that time. At time 8Mc, an instance of Grep(small) arrives with a deadline of 18Mc with a very high utility. CADUS recognizes that in order to fulfill this task's memory requirements, it must kill the JPEG(mini) instance which by that time has already allocated the buffer used to hold the decoded image, hence has almost reached it maximum live size. CADUS kills and sweeps JPEG(mini) at time 8Mc, allowing Grep(small) to run to completion in its deadline. Our memory-unaware version of DASA, on the other hand, also realizes that JPEG(mini) should be preempted in favor of Grep(small) at time 8Mc, but it does not know that there is not enough memory for Grep(small) to finish. Instead, Grep(small) will fill up the heap, and repeatedly trigger a reactive GC. Once these collections have freed up enough memory, Grep(small)'s deadline has expired and the task is purged. To make matters worse, JPEG(mini) is by that point also already past that deadline, yielding an overall utility of 0 for the memory-unaware case.

5. RELATED WORK

Henriksson [8] first studied explicit time-based scheduling of garbage collection. His GC is decoupled from the application and brought under the control of the scheduler. High-

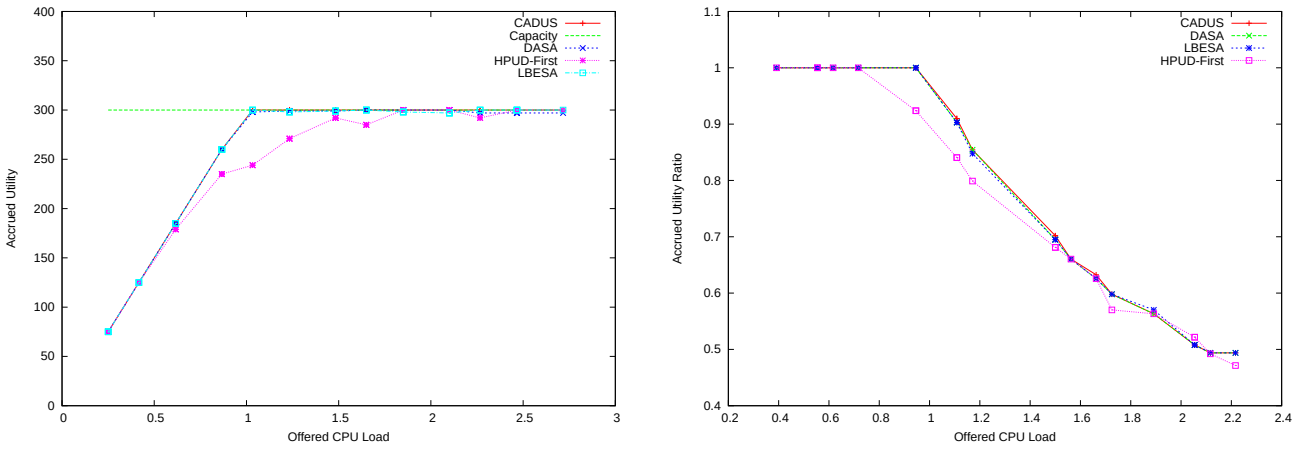

Figure 4: The left figure shows Offered Load vs Accrued Utility with no memory overload – 300 is the highest utility obtainable if the CPU is fully utilized and all started tasks run to completion. The right figures shows the Comparative Accrued Utility Ratio – in underload situations all schedulers accrue a utility equal to the offered load.

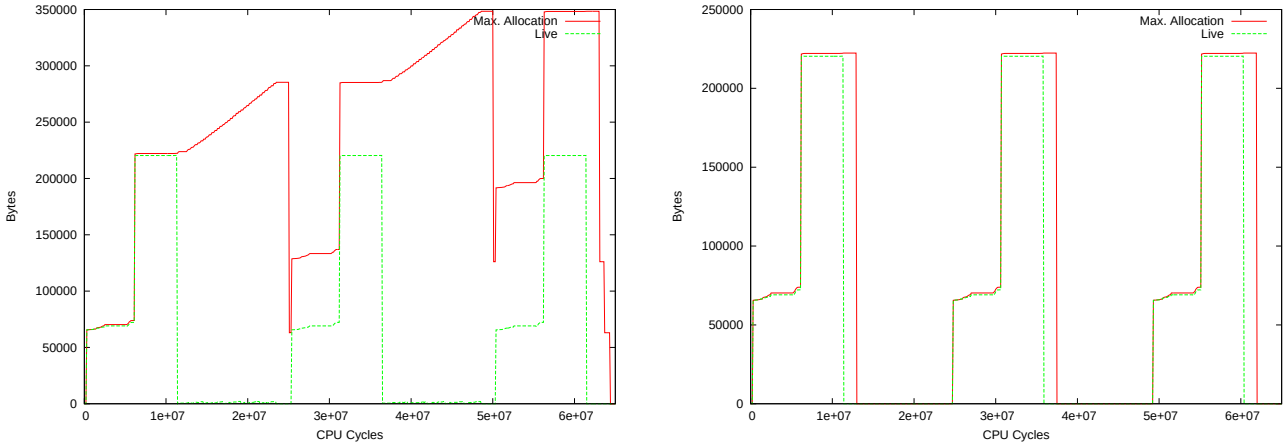

Figure 5: CADUS (shown left) is able to complete 3 instances of JPEG(mini) and two instances of Matrix(small). DASA (shown right), on the other hand sweeps promptly after each task and never runs any Matrix(small) instance because it has become time-infeasible by the time the sweep is completed.

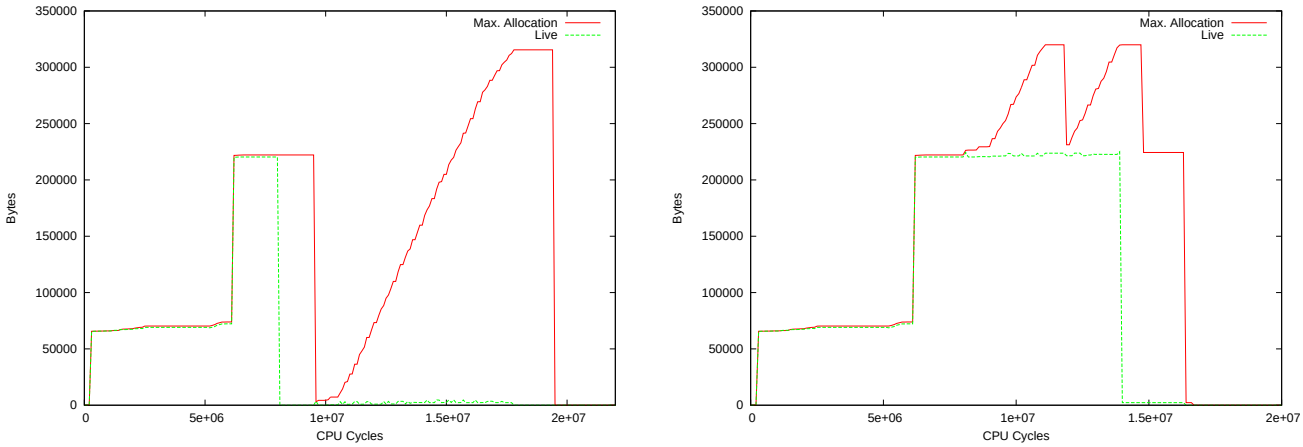

Figure 6: CADUS (shown left) preempts and kills JPEG(mini) at 8Mc and runs the higher utility task Grep(small). DASA (shown right) preempts, but does not kill JPEG(mini) at 8Mc and thus triggers repeated GC that cause it to miss both deadlines, yielding no utility.

priority periodic processes are ensured of memory availability by scheduling enough collection work to immediately follow the process. Low priority processes are relegated to the background and serviced by reactively triggered GC whose execution is interleaved with the mutator(s). As this system enters overload, execution of lower priority tasks is suppressed in favor of providing CPU bandwidth for the high-priority task and its corresponding GC. By comparison, CADUS does not require that GC always follow recurring high priority tasks - the scheduler has flexibility to react to different and dynamic task mixes. Robertz [19] extends Henriksson's approach by time-scheduling the GC based on a period/deadline computed from the worst case allocation requirements of the mutator.

Joint scheduling of the GC alongside hard real-time processes is explored by Kim [12]. A sporadic server approach is utilized to provide sufficient GC bandwidth to service the memory requirements of the co-scheduled periodic mutators. Bacon [2] describes an efficient collector with very low overhead and real-time properties. Implicitly triggered incremental GC is used by Siebert [20] and Nilsen [18]. The incremental real-time GC amortizes its work throughout the execution of the task and reclaims sufficient memory ahead of time. The interleaved GC execution overhead is taken into account for load calculations. Nilsen's collector is adaptively tuned to keep up with a dynamic workload. These time-triggered and allocation-triggered GC scheduling approaches aim to ensure sustained memory availability for all tasks during underload conditions and do not take the specific utility of tasks into account.

Real-time Java [4] proposes the use of scoped and immortal memory regions to avoid garbage collection. These scopes are comparable to CADUS's logical heaps, although Sun's proposal does not address the question of how to link the availability of scoped memory to the system's scheduler.

6. CONCLUSIONS

We have designed, prototyped, and evaluated the first garbage collection-aware scheduler for utility accrual systems. Initial results indicate such awareness can be crucial in certain situations. We believe that the inclusion of GC scheduling into the overall scheduling framework is necessary to make automatic memory management more acceptable for real-time systems.

7. REFERENCES

[1] Godmar Back and Wilson C Hsieh. The KaffeOS java runtime system. *ACM Transactions on Programming Languages and Systems*, 27(4):583–630, 2005.

[2] David F. Bacon, Perry Cheng, and V.T. Rajan. A real-time garbage collecor with low overhead and consistent utilization. In *Conference Record of the Thirtieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, New Orleans, LA, January 2003. ACM Press.

[3] Henry G. Baker. The Treadmill, real-time garbage collection without motion sickness. *ACM SIGPLAN Notices*, 27(3):66–70, March 1992.

[4] Gregory Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. The Java Series. Addison-Wesley, Reading, MA, first edition, January 2000.

[5] R. K. Clark. *Scheduling Dependent Real-Time Activities*. PhD thesis, Carnegie Mellon University, 1990. CMU-CS-90-155.

[6] S. Feizabadi, W. Beebee Jr., B. Ravindran, P. Li, and M. Rinard. Utility accrual scheduling with real-time java. In *Proceedings of JTRES*, pages 550–563, November 2003.

[7] Roger Henriksson. Scheduling real-time garbage collection. Licentiate thesis, Department of Computer Science, Lund University, 1996. Lund technical report LU-CS-TR:96-161.

[8] Roger Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund Institute of Technology, July 1998.

[9] E. D. Jensen. Asynchronous decentralized real-time computer systems. In W. A. Halang and A. D. Stoyenko, editors, *Real-Time Computing*, Proc. of the NATO Advanced Study Inst. Verlag, Oct 1992.

[10] E. D. Jensen and B. Ravindran. Guest editor's introduction to special section on asynchronous real-time distributed systems. In *Proc. of The IEEE Trans. on Comp.*, pages 881–882, Aug. 2002.

[11] Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[12] Taehyoun Kim, Naehyuck Chang, and Heonshik Shin. Joint scheduling of garbage collector and hard real-time tasks for embedded applications. *Journal of Systems and Software*, 58(3):247–260, September 2001.

[13] Peng Li. *Utility Accrual Real-Time Scheduling: Models and Algorithms*. PhD thesis, Virginia Tech, 2004.

[14] Tian F. Lim, Przemyslaw Pardyak, and Brian N. Bershad. A memory-efficient real-time non-copying garbage collector. In Richard Jones, editor, *ISMM'98 Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*, pages 118–129, Vancouver, October 1998. ACM Press.

[15] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[16] C. D. Locke. *Best-Effort Decision Making for Real-Time Scheduling*. PhD thesis, Carnegie Mellon University, 1986. CMU-CS-86-134.

[17] Tobias Mann, Morgan Deters, Rob LeGrand, and Ron K. Cytron. Static determination of allocation rates to support real-time garbage collection. In *LCTES'05: Proceedings of the 2005 ACM*

SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems, pages 193–202, New York, NY, USA, 2005. ACM Press.

[18] Kelvin Nilsen. Starting to PERC. *Java Developer's Journal*, 1(2):11, July 1996.

[19] Sven Gestegård Robertz and Roger Henriksson. Time-triggered garbage collection—robust and adaptive real-time gc scheduling for embedded systems. In *Proc. ACM SIGPLAN LCTES*, San Diego, CA, June 2003.

[20] Fridtjof Siebert. Hard real-time garbage collection in the Jamaica Virtual Machine. In *Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, Hong Kong, 1999.