ABSTRACT
Whereas fairness can be the basis for general-purpose operating system scheduling policies, timeliness is the primary concern for real-time systems. As such, real-time schedulers permit uninterrupted, exclusive access to the CPU by a specific task to ensure its timely completion of execution. Only a subset of tasks, however, can satisfy their timing constraints during processor overload conditions in a real-time system. Utility accrual (UA) scheduling disciplines assure scalability and graceful performance degradation by identifying the subset of tasks to be granted the heavily contended system resources.

Furthermore, whereas general-purpose operating systems treat memory monolithically and indiscriminately service dynamic allocation requests, UA-schedulers can benefit from special memory management considerations during memory overload conditions. MSA, the scheduling algorithm here presented is the first of its kind to treat memory as a UA-scheduler-managed resource. The scheduler is made aware of memory allocation requirements of each task throughout runtime and accordingly makes appropriate CPU and resource scheduling decisions. The algorithm is well-suited for use in resource-constrained embedded systems in a soft real-time environment.

We have implemented MSA in a POSIX real-time operating environment and measured its performance under various load conditions. Our experimental results show overall performance gains over other memory-unaware UA scheduling algorithms during memory overload.

Categories and Subject Descriptors

Keywords
Scheduling, real-time systems, soft real-time, utility accrual scheduling, embedded systems, resource-constrained systems

1. INTRODUCTION
Real-time systems are, by definition, those designed to predictably adhere to a predefined set of timeliness constraints. Hard real-time systems, for instance, must necessarily always satisfy all timing requirements. Soft real-time systems, by contrast, operate in environments where it is known that all timing constraints cannot or need not always be satisfied. Real-time systems enter overload conditions as the aggregate demand for the CPU exceeds the processor’s bandwidth making it impossible to satisfy all timing constraints. Soft real-time scheduling disciplines are often invoked during such overload conditions with the broadly defined objective of “graceful degradation.” The utility accrual model is one method of defining graceful degradation by specifying customizable and predictable temporal system behavior during overload.

The existing UA algorithms [5], however, do not consider the mutual impact of memory management and scheduling. Under the current practice, memory is allocated statically and a task is denied admission into the system if sufficient memory is not available at load-time. This coarse-grain constraint prevents, at the outset, the entry of potentially valuable tasks. It thereby excludes them from competition for system resources in favor of those already memory-resident. MSA addresses this problem by allowing finer-grain explicit memory management at run-time. All tasks are now admitted into the system and scheduling decisions are made in light of their respective memory requirements as they dynamically arise.

2. UTILITY ACCRUAL SCHEDULING
Various aspects of the utility accrual model, central to our overload scheduling scheme, are elaborated below.

2.1 Timeliness
A deadline is the most widely utilized form of a timing constraint. It offers a binary view of the usefulness of a task’s completion with respect to a singular point in time:
the hard deadline. The completion of a task is of no value beyond the deadline, and conversely, would yield full benefit any time prior to the deadline. Such deadline-based systems have a wide range of application, and sufficiently address the requirements of a large sector of the real-time industry. The notion of deadline is particularly well-suited for hard real-time environments where the modes of system operations are mutually exclusive and likewise binary in nature: the system operates correctly if all deadlines are always met, incorrectly otherwise; a task succeeds if it meets its deadline and fails otherwise [4, 1]. For example, the task of deploying parachutes of the Martian Lander by its controller systems must be completed by a hard deadline, else the entire system catastrophically fails. Throughout this paper the term “task” refers to a CPU-schedulable single flow of execution: a process, a thread, or a job.

2.2 Time/Utility Functions

There exist application domains where the binary vision of a deadline-based system does not offer the wider range of expression required to describe the semantics of specific temporal system behavior. For instance, the latest appearance of a radar blip due to delayed computations in a refresh cycle is preferable to no displayed data during that cycle. A Time/Utility Function (TUF) [5], accommodates the shades of gray absent from the hard deadline paradigm. The completion of a task is assumed to yield a precisely quantifiable amount of utility over a continuous range of possible values. The utility curve is predefined for each task and is highly application-specific.

The TUF expresses the utility of a task as a function of its completion time [5, 4]. The semantics of a hard deadline can be expressed using TUFs as yielding full utility prior to the deadline, and zero afterwards. Figure 1(a) depicts the rectangular TUF for one such hard deadline. The gray blocks in the figure represent the execution time of the task starting at time and completing execution at , and can be anywhere between the task arrival time and the task deadline .

We denote as optimal completion time, the time at which a task yields maximal utility upon completion. Figure 1(b) represents an application-specific TUF example. Test flights of unmanned aircraft, for instance, must include a remote self-destruct mechanism in case of a catastrophic systems malfunction. Should the craft’s flight path cross two population centers with an unpopulated expanse in between, the optimal completion time for the self-destruct task would be the halfway point. The timeliness criteria of such a task can be expressed in the form of the depicted TUF. Another commonly used TUFs is illustrated in Figure 1(c) where the task’s utility diminishes beyond the optimal completion time, yet is still of some value even though tardy. Figure 1(d) depicts two execution segments of a single task. The task arrives at , starts execution at , is preempted at , resumes execution at , and completes its execution at .

2.3 System Examples

TUFs are often found in an emerging class of real-time control systems known as supervisory systems. Such systems are typically deployed in environments where there exist significant runtime uncertainty and frequent overload conditions, making it difficult to provide firm guarantees of deterministic temporal system behavior. Soft real-time algorithms are increasingly incorporated into these complex systems to provide resource scheduling decision support. For example, under such conditions the system may not meet all deadlines, but some degree of predictability can be expected, given a pre-defined set of criteria. Crafting TUFs is, therefore, a highly application-specific endeavor [4, 5].

Figure 2 illustrates one such TUF from an operational BM/C2 supervisory system [7]. The system is designed to address a problem that can be generically described as dynamic interception of moving objects. Assume we wish to intercept a set of moving objects prior to their crossing a specific boundary. Further, assume our secondary objective is for the interceptions to occur as far away from the boundary as possible.

Remote sensors track the location of the moving objects and continuously communicate updated coordinates to interceptors. The distance between the interceptors and the objects is greatest at interceptor launch time. It initially suffices the interceptors to aperiodically receive object coordinates for the purposes of plotting a proximity intercept course. As the interceptors close range with the moving objects, more frequent and timely updates are required to make appropriate course corrections. The interceptors require the finest-grain update periodicity just prior to contact. The interceptors must complete course calculations based on received sensor data by . Arrow 1 in Figure 2 reflects the inverse proportionality of range and update frequency. Furthermore, should an interceptor receive fresh sensor data in mid-calculations, it must drop currently incomplete calculations of now-stale data in favor of more accurate results to be derived from the newly communicated update. The rapidly decreasing utility of completing calculations based
on old data is reflected by arrow 2. Lastly, arrow 3 indicates the urgency of the intercept as the objects close their distance with their intended destinations. Interception, for instance, of an object that has penetrated the boundary would be of higher utility than one that is considerably farther out. Figure 2(b) illustrates the dynamically changing TUF during various phases of interceptor operations [7].

3. MEMORY AND REAL-TIME

Unaware of timing constraints, general-purpose OS schedulers extend the time horizon of task execution during overload: all tasks eventually complete [9]. Similarly, general-purpose memory managers expand the limits of main memory into secondary store during memory overload [9, 10]. Virtual memory, however, is often impractical for real-time systems. Resource-constrained embedded systems may be limited to main memory only, while others may not be able to bear the high cost and unpredictability of demand paging. As such, real-time systems often rely solely on main memory, static allocation, and dedicated fixed-sized partitions to increase predictability. This approach is well-suited for the deterministic environment of a hard real-time system. Conversely, the less predictable environment of a soft real-time system would be conducive to a more space-efficient and dynamic storage management scheme. Furthermore, general-purpose memory managers indiscriminately service memory requests unaware of other attributes of the requesting task.

UA tasks currently allocate memory statically at load-time to ensure sustained availability. Unaware of its memory footprint, the scheduler may admit a task into the system, soon to preempt it by another. The preempted task, however, retains its allocated memory while waiting in the ready queue. Not unlike the priority inversion problem, this can lead to low utility tasks holding disproportionately large amounts of memory, while preventing admission of newly arrived high utility tasks.

4. THE MSA ALGORITHM

The scheduling problem addressed by MSA is \(\mathcal{NP}\)-hard along both independent dimensions of time and memory. Optimal sequencing of TUF-described independent tasks is shown to be \(\mathcal{NP}\)-hard in [2], and optimal value-based storage mapping reduces to the classic \(\mathcal{NP}\)-hard 0/1 knapsack problem [8].

As MSA is an on-line scheduler in a dynamic real-time environment, it cannot afford an exhaustive search to produce the optimal solution. A heuristic approach must therefore be employed to produce an approximate solution in polynomially bounded time. MSA utilizes a hybrid greedy, geometric, and partial-combinatorial approximation heuristic.

MSA is invoked at the scheduling points of: task arrival, task completion, memory allocation request (static and dynamic), and memory de-allocation. The scheduler is preemptive and operates in a uni-processor, single address-space environment. Preempted and blocked tasks stay in the system until completion or time-infeasibility, whichever comes first. The scheduler may also elect to “terminate” a task in which case, the task is discarded from the system and all its held resources reclaimed.

Furthermore, memory allocation requests are wrapped to be blocking calls managed by the scheduler. The allocation request may not be serviced immediately, but upon a state transition, the scheduler may grant the request if the requesting task is still time-feasible. The notion of a blocking memory allocation request is similar to FreeBSD’s kernel implementation of a blocking malloc that ensures eventual memory availability for specific device drivers.

4.1 Framework and Notation

Throughout this paper, we make the assumption that there are \(n\) independent tasks (interchangeably, jobs) in the system: \(J = \{j_1, j_2, \ldots, j_n\}\), each characterized by the triplet \(\langle e_i, c_i, u_i \rangle\). The task’s WCE (worst case execution) time is denoted as \(c_i\), its critical time (the time beyond which the task’s utility is non-positive) is \(c_i\), and \(u_i\) is the task’s TUF.

We adapt the notion of processor load at time \(t\), \(\rho(t)\), as defined in [1]:

\[
\rho(t) = \frac{\sum_{i=1}^{\sigma} e_i(t)}{\sigma} \quad , \quad \rho_j(t) = MAX_{i} [\rho_i(t)]
\]

Denoting as \(m_i\) the memory requirements of the \(i^{th}\) task, we calculate memory load of \(J\) at time \(t\) as:

\[
\mu_J(t) = \frac{\sum_{i=1}^{n} m_i(t)}{M} \quad , \quad M = \text{system memory}
\]

Denoting as \(S(J)\) all possible subsequences of tasks in \(J\), \(\sigma \in S(J)\) is therefore an ordered subset of tasks representing a schedule [2]. The memory load for a specific task sequence is therefore:

\[
\mu_{\sigma}(t) = \frac{\sum_{i=1}^{\sigma} m_i(t)}{M} \quad \text{where} \quad |\sigma| \quad \text{is the number of tasks in} \quad \sigma, \quad \text{and} \quad m_i(t) \quad \text{is the memory requirements, at time} \quad t, \quad \text{of the task at position} \quad i \quad \text{within the sequence. The schedule} \quad \sigma \quad \text{is considered memory-feasible if} \quad \mu_{\sigma}(t) \leq 1, \quad \text{i.e., there is enough memory for all tasks in} \quad \sigma \quad \text{at time} \quad t.
\]

Furthermore, we define the maximal theoretical utility, \(U_{MAX}\), to be:

\[
U_{MAX} = \sum_{i=1}^{J} u_i(t_i) \quad \text{where} \quad \forall j_i, i \in \{1, \ldots, n\}: \quad t_j = t_0. \quad \text{That is, the aggregate system-wide utility if all tasks in} \quad J \quad \text{were to complete execution at their respective optimal completion time. We define the aggregate utility of} \quad \sigma \quad \text{to be the cumulative utility, at completion time, of its member tasks:} \quad U_{\sigma} = \sum_{i=1}^{\sigma} u_i(t_i). \quad \text{Our objective can then be defined as:} \quad \text{MAXIMIZE} \quad _{\sigma \in S(J)} U_{\sigma}, \quad \text{given the constraints:} \quad \rho(\sigma) \leq 1 \quad \text{and} \quad \mu(\sigma) \leq 1. \quad \text{Otherwise stated, we wish to find a time-and-memory-feasible subsequence of tasks that would yield the greatest aggregate utility.}
\]

4.2 Algorithm Description

The algorithm’s first step is to determine, based on its
TUF, each task’s start time such that it completes yielding maximal utility: \( t_c = t_{\text{opt}} \) as depicted in Figure 3(a). The figure illustrates 4 individual tasks with their respective TUFs and remaining execution times. The scheduler’s composite view of the 4 contending tasks is depicted in Figure 3(b). The areas of overlap in the figure illustrate the CPU overload conditions and it can be observed that not all tasks may complete execution while gaining non-zero utility. This step greedily seeks to extract maximal utility values from each task.

Figure 4(a) depicts a set of tasks, corresponding to 187% CPU demand, arranged according to the outlined procedure. This establishes the initial ordering of tasks prior to application of any constraints. PUD (Potential Utility Density) for each task is calculated next. PUD is defined as a task’s utility-gain at completion, divided by its remaining execution time. This is depicted as the slope of the execution block in Figure 4.

The algorithm’s next step of “drop-and-shift” identifies a time-feasible subset of the ready tasks. We employ a linear scan technique for this phase. Starting at \( t_0 \) (now), the scan line sweeps forward along the time axis (Figure 4(b)). It: (1) Does nothing if it only crosses one task; (2) Dropping all others, leaves only the highest PUD task if it crosses an overlapped area of two or more tasks; (3) Shifts left the first task it encounters if it crosses no task, or if a gap along the time line is created as a consequence of dropping a task in a contentuous overlap area. The left shift of a task stops at its left neighbor’s boundary, or at \( t_0 \) if no left neighbor exists. Figure 4(c) illustrates, at scan conclusion, the final non-overlapping sequence of tasks against the backdrop of the original skyline. Please note that “dropping” a task does not imply discarding or terminating the task from the system. It merely denotes its no longer being considered for contention during the limited scope of the scan phase. We then check the remaining task sequence against the memory constraint and successively terminate tasks in ascending PUD order until the remaining subset is memory-feasible.

We designate this ordering of tasks as our initial sequence \( \sigma_t \), and its corresponding aggregate utility \( U_t \). Algorithm 1 outlines the scan phase.

Given the infeasibility of exhaustively obtaining the optimal solution, greedy algorithms are often found to produce good approximations. However, this may not always be the case as certain task combinations can be overlooked. Various techniques, however, can be utilized to improve the quality of the initial greedy solution. One such method is outlined in [8] for the classic 0/1 knapsack problem where an initial greedy solution is evaluated for possible improvements using a partial combinatoric technique. For each \( p \)-combination of items, \( \binom{\sigma_t}{p} \), \( p \in \{0, \ldots, k\} \), \( k \leq n \), the combination is indiscriminately placed in the knapsack (if it fits), while the remaining \( n - p \) items are successively fit-tested for inclusion, in descending profit density order. Compared to a greedy placement strategy, the solution quality generally improves with increasing \( k \) as more sets of combinations are iteratively evaluated. As \( k \) approaches \( n \), this method becomes an exhaustive search, unsuitable for our purposes of on-line scheduling. It is, however, demonstrated in [8] that improved results can be obtained even for small \( k \). We adapt this technique for use in our schedule construction.
shown in the next section that MSA has a computational complexity of $O(kn^{k+1})$. We limit $k$ to 3 in our adaptation as an $O(n^4)$ algorithm approaches the limit of suitability for on-line scheduling, even for small $n$.

There are $\sum_{p=0}^{k} \binom{n}{p}$ $p$-combinations for any $k$. For each instance of the $p$-combinations, we designate the corresponding $p$ member tasks as “persistent.” The $p$-combination is then checked for time and memory feasibility and rejected if found infeasible. We then proceed normally with the scan phase of the algorithm. A persistent task, in this context, is defined to be one that always survives contention with a non-persistent task (regardless of PUD) during the scan phase. Contention amongst persistent tasks, should they arise, are resolved greedily based on PUD. For each of the 1 through $k$-permutations, the algorithm proceeds to produce a schedule, taking task persistency into account. Of the resulting set of schedules, the one with the highest aggregate utility is selected as the final schedule. MSA’s high-level pseudo code is outlined in Algorithm 2.

Algorithm 2: The MSA algorithm

Input: $J$: Unordered set of all tasks in the system
Output: $\sigma_{final}$: Task sequence (schedule) to be dispatched in order

begin

Initialization: $\sigma_{final} = \emptyset$

1. $\forall j_i \in J$: Determine $t_{opt}$

2. Sort $J$ by $t_s$ (such that $t_s = t_{opt}$):
   
   $\sigma_{MAX} = j_i \in J \mid \forall (j > i) : t_s \leq t_j$

3. Sort $J$ by PUD:
   
   $\sigma_{PUD} = j_i \in J \mid \forall (j > i) : PUD_i \geq PUD_j$

4. for (all $p$-combinations, $p = 0, 1, \ldots, k$) do
   
   Designate the $p$ tasks persistent:
   
   $\forall i \in p : (j_i \in U_{MAX}) \implies \text{persistent}$
   
   $\sigma_{out} = \text{LinearScan}(U_{MAX})$

   while $\mu_{out} > 1$ do
     
     Terminate the lowest PUD task:
     
     $\sigma_{out} = \sigma_{out} - \{j_i\}$
     
     $\forall j_i \in \sigma_{out} : PUD_{j_i} \geq PUD_{j_{opt}}$

   $\sigma_{final} = \text{MAX}[\sigma_{out}, \sigma_{final}]$

end

4.3 Algorithm Complexity

The TUF of a task provides a closed-form description of its utility as a function of time. Gained utility at completion time, as well as $t_{opt}$, is gained in constant time for each task. The initial step of the algorithm, determination of $t_{opt}$, dominates all $t_{opt}$ for each task (step 1 of Algorithm 2) is therefore $O(n)$ in time.

Steps 2 and 3 of MSA are to sort by $t_s$ and PUD respectively. Each sort is done in $O(n \log n)$ complexity. There are $\binom{n}{p}$ combinations for each $p \in \{0, 1, \ldots, k\}$. Step 4 of MSA is therefore executed $\sum_{p=0}^{k} \binom{n}{p}$ times. The summation is dominated by the term $\binom{n}{2}$ with a time complexity of $O(kn^k)$. The for loop of step 4, therefore, executes $O(kn^k)$ times. The LinearScan step inside the for loop executes in $O(n)$ time as it moves across the sorted list of tasks. The composite complexity of step 4 is thus $O(kn^{k+1})$. As step 4 dominates the entire algorithm, the overall time complexity of MSA is $O(kn^{k+1})$.

4.4 Adaptability

The scheduler invariably incurs some overhead as it requires CPU cycles to construct a schedule, update the appropriate system level data structures, perform context switches, etc. The overhead is insignificant if the tasks’ timing constraints are orders of magnitude higher than the scheduler’s time requirements. However, under relatively tight timing constraints and CPU overload conditions with no cycles to spare, the adverse effect of the scheduler overhead is more pronounced.

MSA calculates the system load, $\rho$, at the time of invocation and can dynamically adjust its overhead accordingly. Given its $O(kn^{k+1})$ time complexity, the algorithm’s overhead can be adjusted with $k$. We limit $k$ to 3 so that an upper bound of $O(n^3)$ is imposed on the scheduler. Also, a lower bound of $O(n \log n)$ (dominated by the sort steps) can be achieved at $k = 0$. As $k$ ranges in value between 0 and 3, the quality of the resulting schedule is affected.

5. PERFORMANCE

MSA was implemented over a middleware scheduling framework designed to accommodate UA schedulers in a standard POSIX real-time environment — in this case, over QNX RTOS. Furthermore, we utilized an instrumented kernel to measure performance and scheduler overhead. The instrumented kernel generates a post-processed trace delineating kernel activity for a specific time period. As such, thread execution times were measured at the same level of precision seen by the kernel.

Figure 5 illustrates the system-wide performance degradation (measured in terms of accrued utility ratio) with increasing CPU and memory load. Note that performance degradation begins as CPU load approaches 100%, as expected. Degradation under the same scheduler and CPU load profile, however, begins near 50% memory load in this case. The observed overhead is due to internal and external fragmentation of memory. The buddy system policy for memory allocation is often used in real-time systems due to its high performance [10]. Such fragmentation, however, is inevitable regardless of the allocation policy. External fragmentation can be alleviated by deferred heap compaction when CPU load drops below 100%.

Figure 6(a) illustrates the comparative performance of
decisions. This offers the UA application programmer the

Figure 6: MSA Performance

MSA and LBESA [6], the only other UA scheduling algo-

Figure 7: MSA Performance (Memory Constraints)

6. CONCLUSIONS

We believe MSA makes the following contributions:

Explicit Dynamic UA Memory Allocation — In absence of a memory-aware UA-scheduler, tasks are con-

surance that a request will be serviced if it results in higher overall utility accrual.

Scalability — As a soft real-time UA scheduling algo-

Pseudo-Greedy Approach — Whereas nearly all other UA-schedulers adopt a strictly greedy sequencing strategy, MSA’s use of the 0/1 knapsack techniques introduces a par-

7. REFERENCES


Predictable Scheduling Algorithms and Applications.


for tasks described by time value function. Journal of


Activities. PhD thesis, Carnegie Mellon University,


computer systems. In W. A. Halang and A. D.

Savenko, editors, Real-Time Computing. Proc. of the


duction to special section on asynchronous real-time
distributed systems. In Proc. of The IEEE


University, 1986. CMU-CS-86-134.

Northcutt, R. B. Kegley, B. A. Zimmerman, and P. J.
Keleher. An example real-time command, control, and
battle management application for alpha. Technical
report, CMU Computer Science Dept., December

[8] S. Sahni. Approximate algorithms for the 0/1

knapsack problem. Journal of the ACM (JACM),


Dynamic storage allocation: A survey and critical

Kinross Scotland (UK), 1995.