

**Dynamic Memory Management
In A Resource-Constrained Real-Time
Utility Accrual Environment**

Shahrooz Feizabadi

Research proposal submitted to the Faculty of the
Virginia Polytechnic Institute & State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science

Binoy Ravindran
Edward E. Fox
E. Douglas Jensen
James D. Arthur
Richard E. Nance

April 25, 2004
Blacksburg, Virginia

TABLE OF CONTENTS

1	Introduction and Overview	7
1.1	Present State of This Research	8
1.2	Summary of Proposed Work	8
1.3	Organization	9
2	Utility Accrual Scheduling	10
2.1	Timeliness	10
2.2	Time/Utility Functions	10
2.3	System Examples	12
3	Memory And Real-Time	14
3.1	Virtual Memory	14
3.2	Static Storage Allocation	14
3.3	Explicit Dynamic Memory Management	15
3.4	Automatic Dynamic Memory Management	15
3.4.1	Reference Counting	16
3.4.2	Mark-Sweep Collection	17
3.4.3	Copying Garbage Collectors	17
3.4.4	Real-Time Garbage Collection	18
3.5	Utility Accrual and Memory Management	19

4	The MSA Algorithm	21
4.1	Framework and Notation	21
4.2	Algorithm Description	24
4.2.1	Initial Pre-Processing	24
4.2.2	Drop-and-Shift	25
4.2.3	Partial Combinatorial Approximation	26
4.3	Algorithm Complexity	28
4.4	Adaptability	29
5	Task/Resource Dependencies	30
5.1	Dependencies and Deadlock	30
5.2	Task Clusters	30
5.3	Persistent Clusters	32
6	Performance	33
6.1	Experimental Setup	33
6.2	Performance Degradation	34
6.3	An Interesting Special Case	37
7	Conclusions And Contributions	38
7.1	Explicit Dynamic UA Memory Allocation	38
7.2	Scalability	38

7.3	Pseudo-Greedy Approach	39
7.4	Adaptability	39
7.5	UA Features	39
8	Future Work	40
8.1	Memory Management Granularity	40
8.2	Implicit Dynamic UA Memory Management	41
8.3	System Properties	41
A	Additional Performance	48
B	Kernel Trace Sample	49
C	TUF in AWACS	50
C.1	Adaptive Timeliness Performance Measurements	51

LIST OF FIGURES

1	Examples of Time/Utility Functions.	11
2	TUF from GD-CMU's BM/C2 system.	12
3	System Snapshot at a Scheduling Point	23
4	Drop-and-Shift	24
5	Dependent Task Clustering	31
6	MSA Performance (No Memory Constraints)	33
7	Cumulative Performance Degradation	34
8	Comparative MSA Performance (No Memory Constraints) . .	35
9	Comparative MSA Performance (With Memory Constraints) .	36
10	Comparative Performance Degradation (With Memory Con- straints)	36
11	Memory Allocation Profile of a Task	40
12	Cumulative Performance under Bursty Traffic with Memory Constraints	48
13	Performance of MSA and DASA with Dependencies and no Memory Constraints	48
14	Track Association TUF	50
15	Average Number of Dropped Tracks vs. Association Capacity	51
16	Track Quality vs. Association Capacity	52

ABSTRACT

This proposal presents MSA, a uniprocessor, single address space resource scheduling algorithm. MSA addresses the open research problem of explicit dynamic memory management in a resource-constrained real-time utility accrual environment. As an overload scheduler, the algorithm ensures graceful performance degradation as defined by a set of optimality criteria. MSA extends the scope of resource scheduling to include main memory, and dynamically evaluates the system-wide effect on utility of biased memory management during memory overload conditions. Furthermore, preliminary work has been performed to enable MSA to perform scheduling in an environment where task/resource dependencies can dynamically develop. A task clustering scheme is devised to preserve the requisite precedence order while honoring timeliness criteria. MSA has been implemented in a POSIX real-time operating environment and its performance profiled under various load conditions. The experimental results show overall performance gains compared to other, memory-unaware UA scheduling algorithms during memory overload. As such, MSA is well-suited for use in a UA governed real-time embedded system.

A possible future enhancement to MSA is outlined to produce finer-grain scheduling decisions. Task WCE (Worst Case Execution) times are assumed to be known for UA scheduling. Task WCAR (Worst Case Allocation Requirements) can likewise be determined and incorporated into the scheduling decision. Furthermore, the memory allocation patterns of a task can be profiled as a function of time and the task's execution divided into storage-based segments, and each segment scheduled individually. The problem of automatic dynamic memory management in a UA environment remains open. This research will next focus on investigating the feasibility, schedulability, utility, and effectiveness of real-time garbage collectors in a UA environment.

1 INTRODUCTION AND OVERVIEW

Whereas fairness-based policies are a consideration for general-purpose OS schedulers, timeliness is the primary concern of real-time systems. As such, real-time schedulers permit CPU monopolization by a specific task to ensure timely completion of execution [2, 30].

Optimal solutions, such as EDF, exist for deterministic hard deadline scheduling: all tasks always meet their deadlines. Such deterministic scheduling schemes, however, do not scale well. EDF catastrophically fails as CPU load approaches, and surpasses 100% [20, 2].

During overload, only a subset of tasks can satisfy their timing requirements. Soft real-time is the discipline concerned with identification of that subset which can, and/or should satisfy their timing requirements in such situations. The desired effect is graceful degradation and scalability [13].

The utility accrual (UA) scheme is well-suited for use as a metric to gauge the fitness of the resource-contentious tasks for survival. Time utility functions (TUFs) are an integral part of UA scheduling. A TUF describes the utility of a task as a function of its completion time. The TUF/UA system generally seeks to maximize the aggregate system-wide utility [13, 14].

The problem of utility-based CPU overload scheduling is shown to be \mathcal{NP} -hard. While an exhaustive search may be possible for a static or clairvoyant scheduler, it is not feasible for an on-line dynamic scheduler. A heuristic solution must therefore be devised [3].

Aggregate task memory requirements can also exceed system memory capacity, creating a memory overload condition. Unlike general-purpose operating systems, real-time systems often cannot afford the overhead and non-determinism of demand paging. As extending the boundaries of main memory into a virtual memory backing store is not practical, all tasks must compete for, and reside in main memory [15].

1.1 Present State of This Research

Analogous to fair scheduling, standard memory managers indiscriminately and monolithically allocate memory to the requesting tasks on a first-come, first-serve basis. UA scheduling, however, would benefit from a level of discernment and robustness built into the allocation decision.

The Memory-aware Scheduling Algorithm (MSA) presented here, is the first UA scheduling discipline designed to address overload along both independent dimensions of time and memory. In addition to the CPU, explicit memory management is relegated to the scheduler. Application programmers can now be provided with the assurance that a task's memory allocation requests will be satisfied if it leads to greater overall utility.

Utility-based preferential memory allocation reduces directly to the classic \mathcal{NP} -hard 0/1 knapsack problem. MSA uses a hybrid geometric/greedy/partial-combinatoric approach to construct time and memory feasible schedules. Furthermore, MSA utilizes a task clustering scheme to address task/reousce dependencies as they arise at runtime.

MSA exhibits higher performance than other UA scheduling algorithms during memory overload conditions. In absence of memory overload, its CPU scheduling performance is comparable to those of other UA schedulers.

No assumptions are made by MSA about a task's memory allocation profile, though such information can be determined. The next possible enhancement for MSA is to utilize such information to produce finger-grain schedules taking into account allocation pattern variations as a function of execution time.

1.2 Summary of Proposed Work

The ultimate objective of this research is to investigate the mutual effects of implicit dynamic memory management (garbage collection) and UA scheduling. Incremental real-time garbage collectors can conceivably be utilized in a UA environment to work on behalf of the executing thread. The consequent indirect utility gain of such collection action, however, requires proper

consideration in the global context of system-wide resource contention.

The mutual effects of memory management and UA scheduling are further complicated in presence of task/resource dependencies and the added precedence constraint relationships. An investigation of the timeliness properties of the GC is required to make appropriate scheduling decisions in such environments.

1.3 Organization

The remainder of this proposal is organized as follows: Section 2 presents an outline of time utility functions and utility accrual scheduling. Memory management, with an emphasis on real-time computing is presented in section 3. Section 4 presents the MSA algorithm and its features. Section 5 outlines the MSA mechanism for handling task/resource dependencies. Section 6 includes the performance characteristics of MSA. The conclusions and contributions of MSA are listed in Section 7. The future work of this research is outlined in section 8.

2 UTILITY ACCRUAL SCHEDULING

Central to our method of overload scheduling is the concept of utility accrual. Various aspects of the UA model are outlined 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 [13, 2]. For example, the task of deploying parachutes of the Martian Lander by its on-board control systems must be completed by a hard deadline, else the entire system catastrophically fails. Please note that throughout this proposal we use the term "task" to refer to a CPU-scheduler's single flow of execution: a process, a thread, or a job.

Deadlines may also be utilized in soft real-time systems where missed deadlines are to be expected. The operational optimality criteria for these systems can consequently be defined in terms of met or missed deadlines. Operational objectives of such systems can be, for instance, minimizing the number of missed deadlines [14].

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. Time/Utility Func-

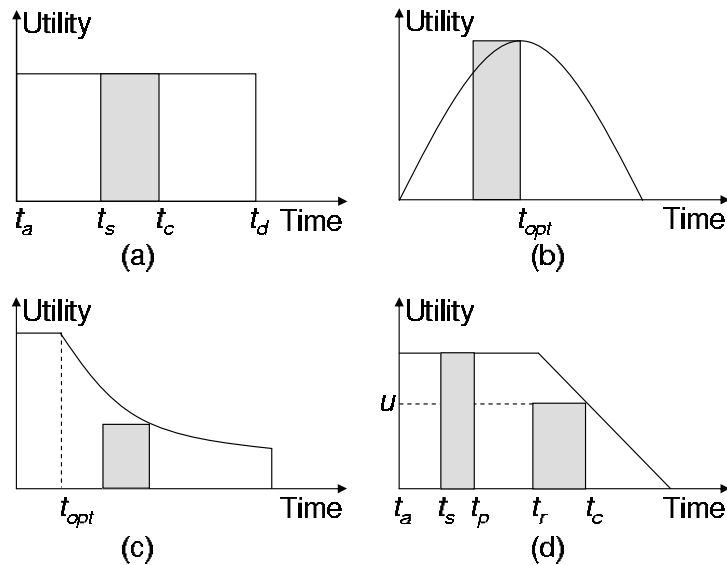


Figure 1: Examples of Time/Utility Functions.

tions [14], accommodate the various 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 range is predefined for each task and is highly application-specific. For instance, the late appearance of a blip on the radar screen due to delayed computations during a refresh cycle is preferable to going a full cycle with no data displayed at all.

The Time/Utility Function (TUF), therefore, expresses the utility of a task as a function of its completion time [14, 13]. Note that 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 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 .

We denote as optimal completion time, t_{opt} , 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

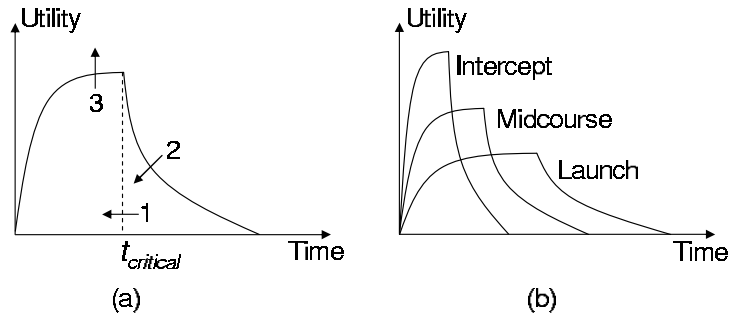


Figure 2: TUF from GD-CMU's BM/C2 system.

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 t_a , starts execution at t_s , is preempted at t_p , resumes execution at t_r and completes its execution at t_c .

2.3 System Examples

TUFs are often found in an emerging class of real-time control systems collectively 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 set of pre-defined optimality criteria. Crafting TUFs is, therefore, a highly application-specific endeavor [13, 14].

Figure 2 illustrates one such TUF from an operational BM/C^2 supervisory system jointly developed and deployed by General Dynamics and Carnegie

Mellon University [21]. 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 correction calculations based on received sensor data by $t_{critical}$. 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 stages of interceptor operations [21]. Another prominent example of a highly sophisticated operational TUF/UA system is the Airborne Warning and Control System (AWACS) tracker [4]. A more detailed case study of this application is included in Appendix C.

Highly versatile, TUF/UA scheduling is not limited to special-purpose embedded systems. Its application domain includes telecommunications, industrial automation, and multimedia QoS. TUFs can also be applied in environments where task timing constraints are on the order of hours or days, rather than microseconds: during a catastrophic natural disaster, for instance, an overloaded emergency response center can temporarily delay dispatching of supplies to a remote location in favor of first servicing a nearby high population density area.

3 MEMORY AND REAL-TIME

3.1 Virtual Memory

Unaware of timing constraints, general-purpose OS schedulers extend the time horizon of task execution during overload: all tasks eventually complete [30]. Similarly, general-purpose memory managers merely expand the boundaries of main memory into secondary store during memory overload [30, 32]. 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.

3.2 Static Storage Allocation

Providing predictable temporal behavior is the objective of real-time systems. Hard real-time systems must guarantee fully deterministic behavior while soft real-time systems offer specific timeliness properties. Both systems benefit from minimizing all possible sources of non-deterministic temporal operations. Memory allocation can be one such source of temporal non-determinism. The difficulty of ascertaining an upper-bound on the overhead incurred by the allocator is often an impediment to the use of dynamic storage allocation. Programmers of time-critical applications often circumvent the inherent risk of unforeseen allocation delays by utilizing only static allocation. This approach has the dual advantages of first, not jeopardizing unpredictable delays, and second, guaranteeing exclusive availability of the required memory resources for the life-time of the task.

Such approach of static partitioning and fixed memory segmentation is well-suited for the deterministic environment of a hard real-time system where the allocation and execution requirements of all tasks are assumed known in advance. Given such *a priori* knowledge, appropriate capacity planning and resource allocation patterns can confidently be accomplished at design-time.

The determinism of static allocation, however, comes at the cost of efficiency and flexibility. Static, front-loaded allocation of memory generates large vacuous object files leaving little room for run-time flexibility.

3.3 Explicit Dynamic Memory Management

In contrast to the highly controlled and precisely defined operational parameters of hard real-time, the less predictable environment of a soft real-time system would be conducive to a more space-efficient and dynamic storage management scheme. Given the inherent uncertainties of a soft real-time operating environment, *a priori* load planning is infeasible as resource demand overload is now a possibility. In a memory-constrained environment, the space efficiency offered by the dynamic allocation model could well offset its associated cost of possible allocation latency. Furthermore, efficient allocation strategies, such as Knuth's Buddy System, offer known latency upper bounds [7].

In addition to allocation latency, a task now faces the risk of a failed mid-execution memory allocation request. This, however, need not be detrimental. So long as the task remains time-feasible, it can be held in a blocked state until a subsequent system state transition yields enough memory for the blocked task to continue. This is a fundamental concept for this work as MSA exploits this fact to bring explicit allocation and de-allocation requests under the supervision of the scheduler. Along with other task attributes, MSA then makes appropriate allocation decisions.

3.4 Automatic Dynamic Memory Management

The intersection of real-time computing and automatic dynamic memory allocation (also known as garbage collection) is an active area of research. Garbage collection (GC) can significantly improve code quality and efficiency as the application programmer is relieved from the nuances of explicit memory management bookkeeping and often complex and cumbersome debugging of dangling references and leak detection. Another GC benefit is a simplified software engineering process. Memory management details no longer need to

cross functional module boundaries and can be implicitly handled within the program's global scope of execution. A garbage collection entity, typically running in a separate thread, is responsible for run-time heap maintenance for the executing application.

Programming languages such as Java and ML rely exclusively on GC and provide no mechanism for fine-grain explicit manipulation of memory. In the notable case of Java, a standard JVM (Java Virtual Machine) exposes no low-level interfaces for directly accessing memory and manages all references on a program's behalf. This is consistent with Java's design objectives of type-safety and managed security.

Multiple garbage collection techniques, each with its own specific properties have been proposed and implemented. Fundamental approaches to GC are outlined below:

3.4.1 Reference Counting

Memory objects under the control of the collector are augmented with an additional counter field which keeps track of all the references made to the object. The counter is incremented when a pointer is assigned to point to the object. Conversely, the counter is decremented when a pointer referencing it is reassigned to another memory object or destroyed. Once the object's reference counter reaches zero, the memory occupied by the object can then be returned to the free pool as the object is no longer reachable from the mutator [32].

Reference counting, while simple, also offer the advantage that the overhead of memory management is distributed throughout the execution [15]. This fine-grain method of immediately freeing individual objects upon unreadability has the effect of a smoother execution profile in comparison to other collection strategies where entire groups of objects are simultaneously processed and freed, leading to long pauses.

The most significant drawback of the reference counting method, in its simple form, is its inability to handle cyclic references. Data structures can form a cycle in memory as a consequence of specific execution patterns. Even if

all external references to the objects in the cycle are removed, the reference counter for each object will have a value of at least one by the virtue of being in the cycle. The objects in the cycle are thus left unreclaimed as their respective counter values never reaches zero [32]. A hybrid technique suggested by Knuth [16] offers methods of addressing this issue. Furthermore, continuous maintenance and update of object reference counters incurs significant overhead [10]. Reference counter of both objects must be adjusted as a pointer is reassigned from one object to another.

3.4.2 Mark-Sweep Collection

Under this garbage collection strategy, memory objects immediately visible to a program (such as its static variables, global variables, machine registers, and those in the activation stack) are designated as its *root set*. The objects in the root set are *marked*. Each object in the root set is then recursively examined to identify any pointers they may contain. Objects referenced by the newly identified pointers are marked in turn. All reachable objects within the pointer reference graph are marked in this manner. What remains, must then necessarily be unreachable and can be freed [22, 32].

Unlike reference counting, memory objects are not freed immediately as they become unreachable from the root set. These object remain in memory until the garbage collector is triggered by an event such as a failed allocation request. Once the GC is triggered, the execution of the mutator is suspended until the sweep phase is completed. Such mechanisms are referred to as *stop-the-world* garbage collectors. The collector's latency is proportional to the number of objects in memory, and the pointers to the objects. Given the uncertainty of the collector's activation time and its unpredictable latency once triggered, mark-sweep collectors cannot be used in a time-critical application [23, 15].

3.4.3 Copying Garbage Collectors

This class of collectors move objects in memory by copying them from one location to another. The heap is typically divided into two equal-sized *semi-*

spaces. The executing program has access to only one of the semi-spaces at any point during execution. Memory from the semi-space is linearly allocated to the mutator and an allocation request failure triggers the collector. Once activated, the collector finds all reachable objects and copies them from the current semi-space (*fromspace*) to the other semi-space (*tospace*). At the first subsequent failed allocation request, the *fromspace* and *tospace* are flipped and the copying performed as before [32].

Copying collectors incur comparatively little overhead. As the semi-space memory is linearly allocated, the cost of allocation is merely that of incrementing the free space pointer. Out-of-space checks also cost only one pointer comparison operation [32]. The copying phase inherently compacts the heap and eliminates external fragmentation in the process. Besides the immediate 50% space overhead, the tracing and copying phase requires time proportional to the number of objects in memory and introduces non-deterministic mutator delays.

3.4.4 Real-Time Garbage Collection

Timeliness is clearly the single most important aspect of any real-time program execution. As such, from a real-time perspective, a collector's most significant property is its guaranteed worst case execution bound — if any. Of secondary importance, is the collector's space bound — if any [32, 24]. Garbage collection schemes based on an atomic two-phase identification and reclamation action are typically unsuitable in a real-time environment. Such atomic collection requires halting mutator execution for undesirably long periods. Furthermore, collector can be activated at random points if triggered by a failed allocation request [23].

The garbage identification/reclamation step need not necessarily be atomic and can be performed in discrete increments [6]. The execution of such an *incremental* collector can then be interleaved with that of the running program. Furthermore, the execution time of each incremental step can be deterministically bounded [32, 23]. Two immediate difficulties arise: first, maintaining heap reference consistency between increments, and second, ensuring sufficient GC progress at each step.

The incremental collector can be suspended in mid-trace, only to discover an altered object reachability graph at its subsequent invocation. The mutated graph (hence the name *mutator* for the application program) must contain consistent references, and the changes introduced during the mutation must be tracked by the collector. This can be accomplished by utilizing a *write barrier*. All pointer assignments are intercepted by the write barrier and manipulated to reflect the most up-to-date location of an object as known by the collector [24].

The second difficulty of this method of garbage collection is determination of the length of time for an incremental step. The collector must have enough time to make reclamation progress proportional to the demands of the mutator [12, 11]. This approach, however, requires *a priori* knowledge of the mutator's worst case allocation requirements [26, 27].

Hardware-assisted real-time garbage collection has been shown to provide significant performance enhancements [25]. The GC hardware has the same interface as a standard memory module and logically appears as another DIMM to the CPU and is transparently integrated into the host machine. The requisite specialized hardware, however, is not commonly available nor cost effective.

3.5 Utility Accrual and Memory Management

Analogous to the notion of fair scheduling, general-purpose memory managers indiscriminately service memory requests unaware of other attributes of the requesting task. This model, while effective, does not best serve a utility accrual environment. TUF/UA tasks inherently encapsulate a rich set of attributes which can be utilized to provide special considerations during memory allocation.

Priority-based dynamic memory management in a hard real-time environment has been explored in [27]. In this approach, a real-time incremental garbage collector (as outlined above) is periodically and deterministically scheduled to keep ahead of the known allocation requirements of a high-priority periodic task. The GC is assigned a deadline, and scheduled as any other task under the EDF (Earliest Deadline First) [19] policy.

Given the application space of UA schedulers, however, no such detailed assumptions can be made about the CPU or memory load at any specific point in time. Furthermore, overload is a highly likely possibility in such an environment. EDF's optimality of meeting *all* task deadlines up to a theoretical 100% CPU load has been demonstrated in [19]. During overload, however, EDF suffers catastrophic failure due to a "domino effect" described in [20]. Once one task falls behind in meeting its deadline, the scheduler frantically attempts to successively schedule tasks that have no chance of meeting their respective deadlines.

TUF/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 [17], 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 [3], and optimal value-based storage mapping reduces to the classic \mathcal{NP} -hard 0/1 knapsack problem [28].

As MSA is an on-line scheduler in a dynamic real-time environment, it cannot afford an exhaustive search of trying all possibilities 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-combinatoric 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 “abort” 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 somewhat similar to FreeBSD’s kernel implementation of a blocking `malloc` to ensure eventual memory availability for specific device drivers [1].

4.1 Framework and Notation

Throughout this paper, we make the assumption that there are n tasks (interchangeably, *jobs*) in the system: $J = \{j_1, j_2, \dots, j_n\}$, each characterized by the triplet $\langle e_i, c_i, u_i \rangle$. The task’s WCE (worst case execution) time is denoted as e_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 [2]:

$$\rho_i(t) = \frac{\sum_{c_k \leq c_i} e_k(t)}{c_i - t} \quad , \quad \rho_J(t) = \text{MAX}_i [\rho_i(t)].$$

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 [3]. The memory load for a specific task sequence is therefore:

$$\mu_\sigma(t) = \frac{\sum_{i=1}^{|\sigma|} m_i(t)}{M}$$

where $|\sigma|$ is the number of tasks in σ , and $m_i(t)$ is the memory requirements of task at position i within the sequence. The schedule σ is considered memory-feasible if $\mu_\sigma(t) \leq 1$, i.e., there is enough memory for all tasks in σ at time t .

The TUF of a task is of the form:

$$U_J = \begin{cases} 0 & t < t_c \\ f(t_c) & t < c_i \\ \leq 0 & t \geq c_i \end{cases}$$

A task accrues no partial utility unless it fully completes execution.

Furthermore, we define U_{MAX} to be:

$$U_{MAX} = \sum_{i=1}^{|J|} u_i(t_o)$$

where:

$$\forall j_i, i \in \{1, \dots, n\} : t_{c_i} = t_{o_i}$$

That is, the aggregate system-wide utility if *all* tasks in J were to complete execution at their respective optimal completion time. We define the ag-

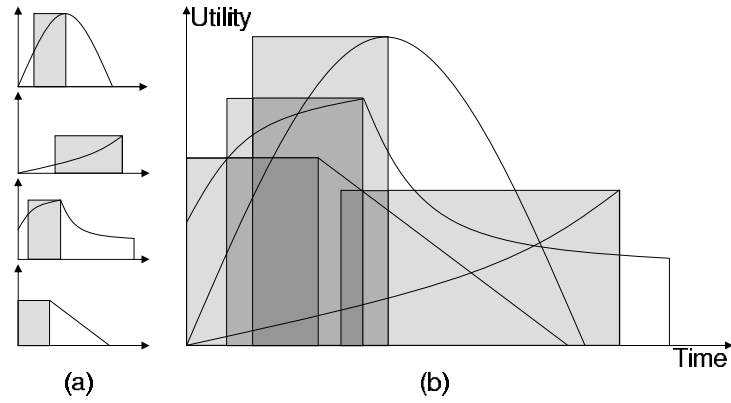


Figure 3: System Snapshot at a Scheduling Point

Aggregate utility of σ to be the cumulative utility, at completion time, of its member tasks:

$$U_\sigma = \sum_{i=1}^{|\sigma|} u_i(t_c)$$

Our objective can then be defined as:

$$\text{MAXIMIZE}_{\sigma \in S(J)} U_\sigma$$

given the constraints:

$$\rho(\sigma) \leq 1 \quad \text{and} \quad \mu(\sigma) \leq 1.$$

Otherwise stated, we wish to find a time-*and*-memory-feasible sequence of tasks that would yield the greatest aggregate utility.

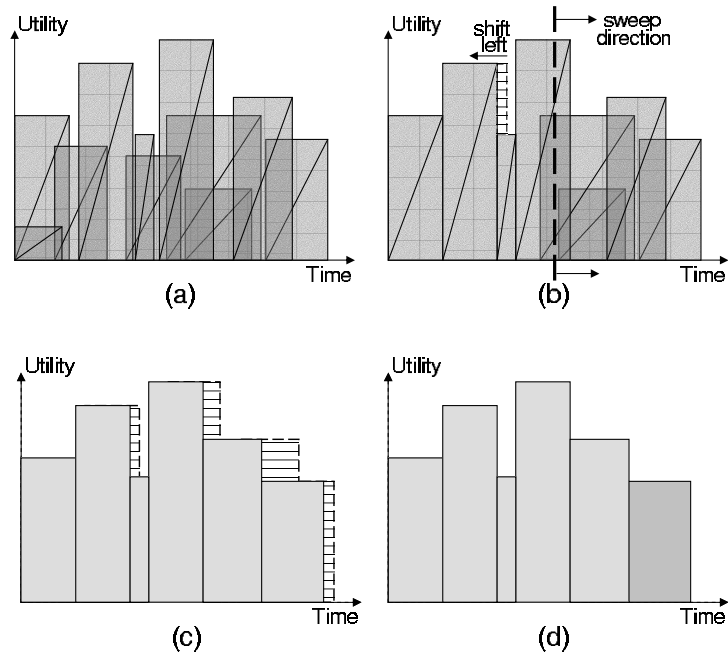


Figure 4: Drop-and-Shift

4.2 Algorithm Description

4.2.1 Initial Pre-Processing

The algorithm's first step is to determine, based on its TUF, each task's start time such that it completes yielding maximal utility: $t_c = t_{opt}$ 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.

4.2.2 Drop-and-Shift

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 graphically represented 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 contentious 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 aborting 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 *abort* tasks in ascending PUD order until the remaining subset is memory-feasible. We designate this ordering of tasks as our initial sequence σ_i , and its corresponding aggregate utility U_i . Algorithm 1 outlines the scan phase.

Shifting the execution time of a task changes its time of completion, t_c , and hence its utility yield. As t_c is only shifted left, a task may complete execution sooner than expected, but never later, i.e., no deadlines misses. Shifting t_c , however, does affect the task’s utility gain at completion time. Left shifting a t_c in an interval where the TUF’s time rate of change is positive, yields lower than original utility. For example, should the execution block of the parabolic TUF in Figure 3(a) be shifted left, the task’s utility yield is lowered. The adverse impact of the left shift in this situation is a function of the degree of locality of the TUF’s maxima. In cases of moderate positive TUF time

rate of change, the utility loss is likewise moderate. Most TUFs from the UA application domain, however, are non-increasing in time, hence unaffected by a left shift.

4.2.3 Partial Combinatorial Approximation

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 [28] 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, $\binom{k}{p}$, $p \in \{0, \dots, k\}$, ($k \leq n$), 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 [28] that improved results can be obtained even for small k . We adapt this technique for use in our schedule construction. It is 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

Algorithm 1: Linear Scan (Drop-and-Shift)

Input: σ_{MAX} : Sequence of *all* tasks in J
yielding U_{MAX} sorted by t_s
Output: σ_{out} : Time-feasible task subsequence

```
CurrentTask  $\leftarrow J_i \in J \mid t_{s_j} = t_{s_{MIN}}$ 
while CurrentTask  $\neq \emptyset$  do
  Next  $\leftarrow (CurrentTask \rightarrow Next)$ 
  if Next =  $\emptyset$  then break
  if CurrentTask. $t_c > Next.t_s$  then
    /* overlap */
    if CurrentTask.Persistent then
      /*Persistent tasks always stay */
      drop Next
      continue
    if CurrentTask.PUD  $> Next.PUD$  then
      drop Next
    else
      /* drop & shift */
       $\delta_t = Next.t_s - CurrentTask.t_s$ 
      drop CurrentTask
      Next. $t_s = Next.t_s - \delta_t$ 
      Next. $t_c = Next.t_c - \delta_t$ 
      CurrentTask  $\leftarrow Next$ 
  else
    /* No overlap, shift */
     $\delta_t = Next.t_s - CurrentTask.t_c$ 
    Next. $t_s = Next.t_s - \delta_t$ 
    Next. $t_c = Next.t_c - \delta_t$ 
    CurrentTask  $\leftarrow Next$ 
```

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: σ_{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 $t_c = t_{opt}$):
 $\sigma_{MAX} = j_i \in J, \forall(j > i) : t_{s_i} \leq t_{s_j}$

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

4 **for** (all p -combinations, $p = 0, 1, \dots, k$) **do**
 Designate the p tasks persistent:
 $\forall i \in p : (j_i \in U_{MAX}) \leftarrow persistent$
 $\sigma_{out} = \text{LinearScan}(U_{MAX})$
 while $\mu_{\sigma_{out}} > 1$ **do**
 Abort the lowest PUD task:
 $\sigma_{out} = \sigma_{out} - \{j_z\} :$
 $\forall j_i \in \sigma_{out} : PUD_{j_i} \geq PUD_{j_z}$
 $\sigma_{final} = \text{MAX}[\sigma_{out}, \sigma_{final}]$

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 derived in constant time for each task. The initial step of the algorithm, determination of 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 $p \in \{0, 1, \dots, k\}$. Step 4 of MSA is therefore executed $\sum_{p=0}^k \binom{n}{p}$ times. The summation is dominated by the term $\binom{n}{k}$ 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, ρ , 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 $O(n^4)$ 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 scheduler's quality of service (QoS) is also affected.

5 TASK/RESOURCE DEPENDENCIES

5.1 Dependencies and Deadlock

Analogous to the priority inversion problem[17], situations may arise where a task currently holds a dedicated resource requested by another, higher PUD task. A possible resolution, similar to the priority inheritance protocol[29], is to allow the task holding the resource to finish execution whereupon it releases the resource. Alternatively, as advocated by DASA[5], the task holding a *preemptible* resource can be marked for abortion to release the resource if a higher PUD task requesting that resource could then execute to completion. Though the cost of aborting a CPU-bound task is little more than the cost of the context switch, releasing other types of held resources may take considerably longer. Some resources may require immediate execution of clean-up handlers to properly restore the resource to a consistent state: a robot arm must be returned to a safe position even if the task holding its controller is aborted.

Resource deadlock, however, may arise in such systems. Given the run-time uncertainties of the UA application domain, deadlock prevention and deadlock avoidance are considered impractical. Deadlock handling must therefore be reactive. In an environment with resource dependencies, the scheduler first resolves deadlock if one is detected. The single-unit resource model has been widely used to model resource acquisition[30]. A task may acquire and hold multiple resources, but must request them one at a time. Adopting this model, we detect a deadlock by searching for a cycle in the resource request (a.k.a wait-for) graph. Should one exist, the cycle is broken by aborting the lowest PUD task.

5.2 Task Clusters

MSA is augmented to handle dependent tasks by implementing a *task clustering* scheme. During the initial placement phase, as dependent tasks are encountered, the dependency chain is treated as a cluster: a monolithic block of sequential execution of tasks in order of dependency precedence. Figure 5(a)

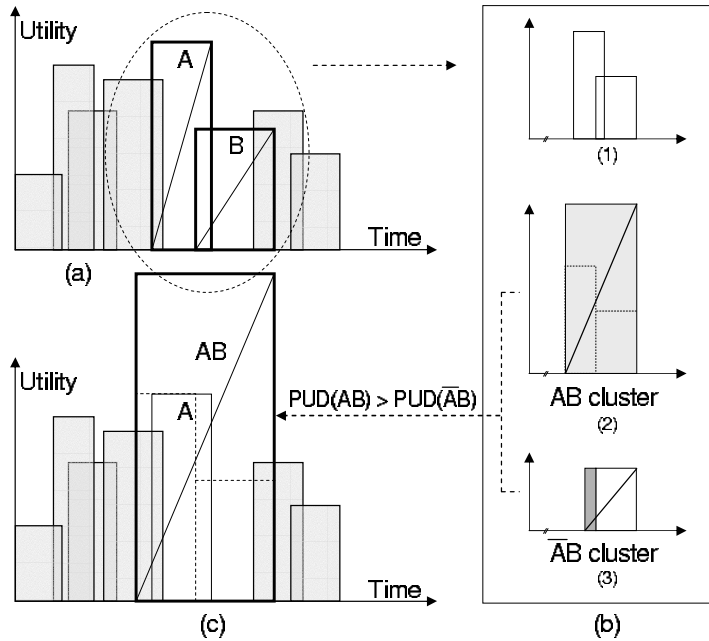


Figure 5: Dependent Task Clustering

illustrates two tasks with a dependency $B \rightarrow A$ (B requires a resource held by A). Figure 5(b-2) illustrates the construction of the cluster from the two tasks. Note that as with shift-and-select, task A is shifted left to ensure it does not push task B beyond its $t_{critical}$. The PUD of the cluster, is then the aggregate utility gained by execution of all its member tasks, divided by the *total* remaining execution time.

Note that in the single-unit resource model, all nodes in the resource request graph have, by definition, an out degree of one. As dependent tasks are encountered, the dependency can therefore be broken by aborting only the task holding the requested resource. Suppose, for example: $D \rightarrow C \rightarrow B \rightarrow A$. Task D can then be relieved of its dependency by aborting only C . The $B \rightarrow A$ dependency remains unchanged, and is considered in turn, once task B is encountered. The cost of aborting a task is the execution time of its clean-up handler. In the example of Figure 5, B 's dependency on A can be broken only if B 's execution is preceded by the execution of A 's clean-up handler (Figure 5(b-3)).

A cluster assumes voluntary yielding of held resources as tasks along the dependency chain execute to completion. We do, however, simultaneously explore the possibility of aborting a task to acquire the resources it holds, i.e., preempting the resource. As a dependent task is encountered, its individual execution (after abortion of the task on which it depends) is weighed against the execution of the entire dependency chain. The subsequence offering the higher PUD is selected. As illustrated in Figure 5(b-3), the PUD of the individual task is adjusted to reflect the execution time of the clean-up handler. Should a two-member cluster of a clean-up handler followed by a task be selected, the task is placed back on the timeline at its original position immediately preceded by the clean-up handler. Conversely, if the complete (or partial) dependency chain is selected, it is placed along the timeline at the earliest deadline of any task in the cluster. This would ensure a left-shift-only repositioning, thereby avoiding missed deadlines.

5.3 Persistent Clusters

Once a task cluster has been constructed, the algorithm proceeds normally with the drop-and-shift phase, taking into account an entire cluster's aggregate PUD as depicted in Figure 5(c). Similarly, should a dependent task be included in a p -permutation, its cluster (whether clean-up handler or dependency chain) is marked as persistent. Suppose, for instance, we have $B \rightarrow A$, a 2-permutation of $\langle BC \rangle$, and $PUD(AB) < PUD(\overline{AB})$. We would then modify the 2-permutation to be $\langle \overline{AB}C \rangle$ and mark the cluster as persistent.

There could conceivably arise an unlikely situation where all the tasks in the system form one long dependency chain. Determination of abortion or cluster execution, in this case, would require $\Omega(n^2)$ time. The overall algorithm complexity is therefore increased to $O(kn^{k+2})$. Given the single-resource request model, however, there would need to be n resources in the system for this to hold. As $m \ll n$ in a typical system, the overall complexity remains at $O(kn^{k+1})$ as only a chain of maximum length m could ever develop.

6 PERFORMANCE

This section summarizes MSA’s performance in comparison with other UA scheduling algorithms in an environment with no effective memory limitations, as well as a memory-constrained system.

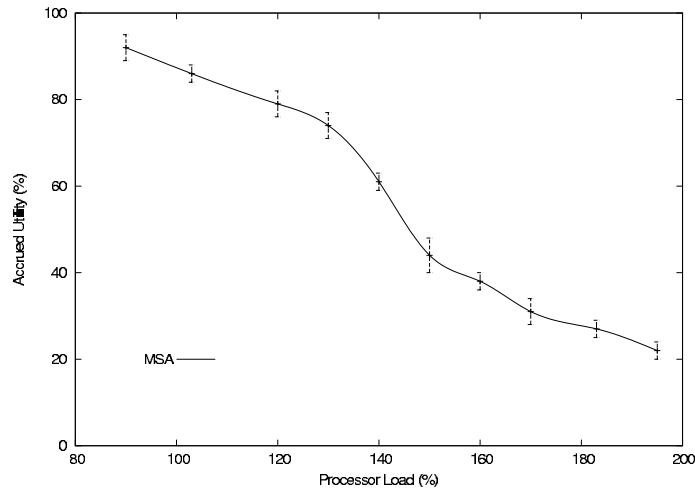


Figure 6: MSA Performance (No Memory Constraints)

6.1 Experimental Setup

MSA was implemented inside a meta-scheduler [18] — a middleware scheduling framework designed to accommodate UA schedulers in a standard POSIX real-time environment, in this case, the QNX Neutrino RTOS. Furthermore, we utilized an instrumented kernel to measure performance and scheduler overhead. The instrumented kernel generates a post-processed trace delineating all kernel activity during a specific time period. A sample kernel trace is provided in Appendix B. As such, thread execution times were measured at the same high level of precision seen by the kernel. A hard limit was externally imposed on the heap size of the executing tasks to produce a memory barrier. Each experiment was conducted with an average of 500 tasks per run, and the results averaged.

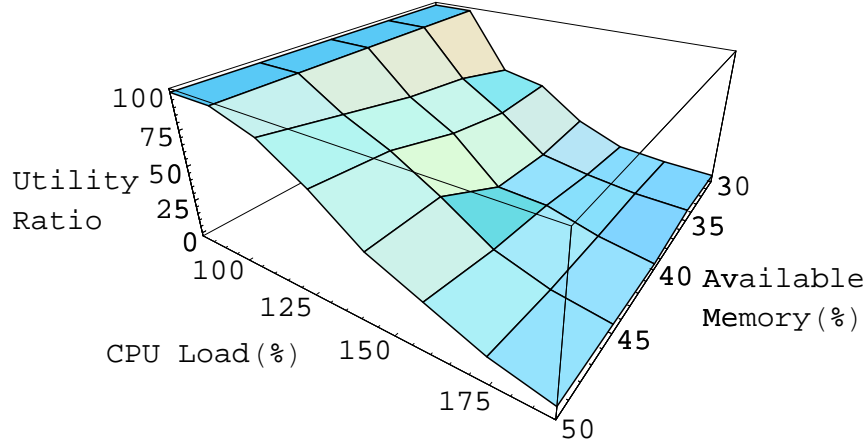


Figure 7: Cumulative Performance Degradation

6.2 Performance Degradation

Figure 6 illustrates MSA’s performance (measured in terms of accrued utility ratio) with only CPU overload. Figure 7 illustrates the system-wide performance degradation 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 (binary or Fibonacci) policy for memory allocation is often used in real-time systems due to its relatively high performance [32]. Such fragmentation, however, is inevitable regardless of the allocation policy. Sub-allocators, though highly space efficient, can only allocate memory in uniform blocks and require *a priori* knowledge of the application’s requirements — typically unavailable in a UA application domain. External fragmentation can be alleviated by deferred execution of a heap compaction routine when the CPU load drops below 100%. MSA’s value of total available memory, M , must be heuristi-

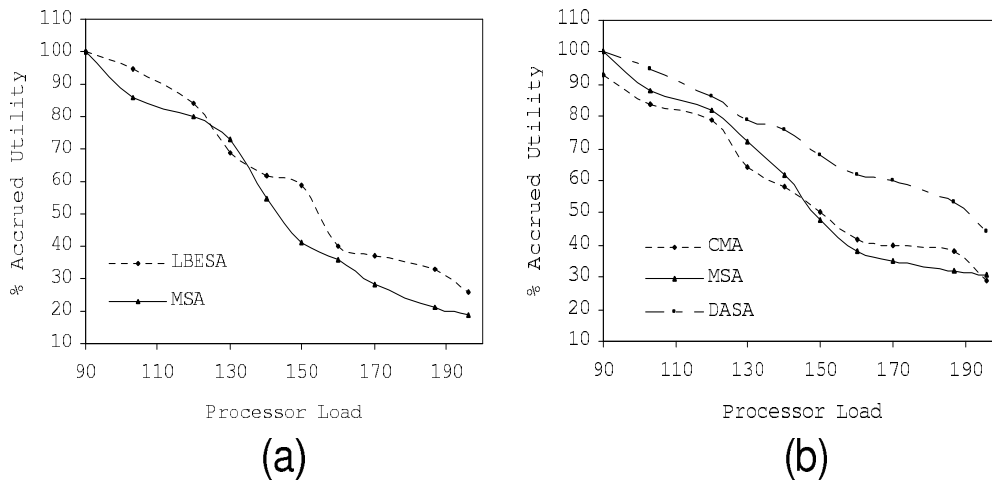


Figure 8: Comparative MSA Performance (No Memory Constraints)

cally adjusted to appropriately compensate for the fragmentation overhead.

Figure 8(a) illustrates the comparative performance of MSA and LBESA [20], another UA scheduling algorithm capable of handling arbitrary TUFs. Performance of MSA is compared to DASA [5], and CMA [3] using only rectangular TUFs in Figure 8(b). DASA, the overall highest performing UA scheduling algorithm is restricted to rectangular TUFs (hard deadlines) only. No memory constraints were imposed on the system for the performance measurements depicted in Figure 8. During memory overload, however, MSA exhibits higher performance than the other UA scheduling algorithms. Figure 9 depicts MSA’s comparative performance for the same set of tasks under 20% memory overload.

Furthermore, our experiments indicate that for a given CPU load, MSA performs better for a higher number of tasks, e.g., 50 tasks constituting an aggregate 140% CPU load, versus 20 tasks generating the same load level. The drop-and-shift mechanism is more likely to produce a better initial sequence under these conditions given the increased number of choices. Figure 10 illustrates the overall cumulative performance of MSA as compared to LBESA during CPU and memory overload. Appendix A includes an additional cumulative performance illustration for MSA under 30% more tasks and less stringent memory constraints. The appendix also includes the comparative

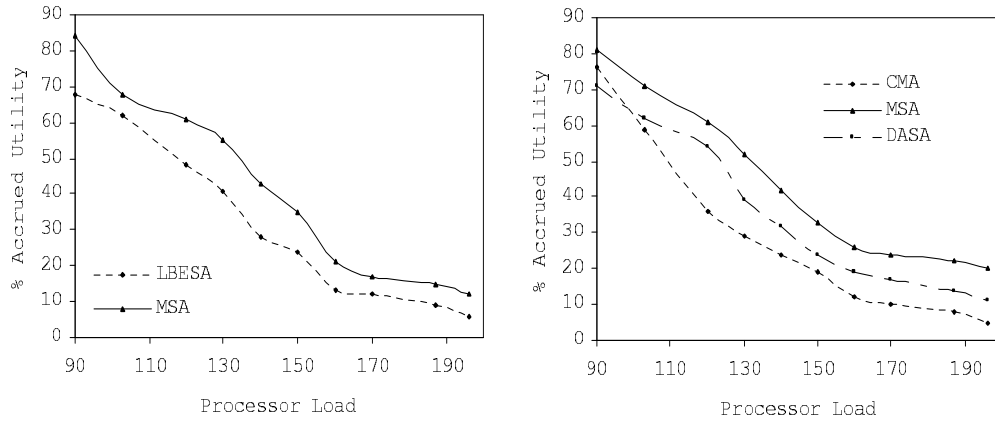


Figure 9: Comparative MSA Performance (With Memory Constraints)

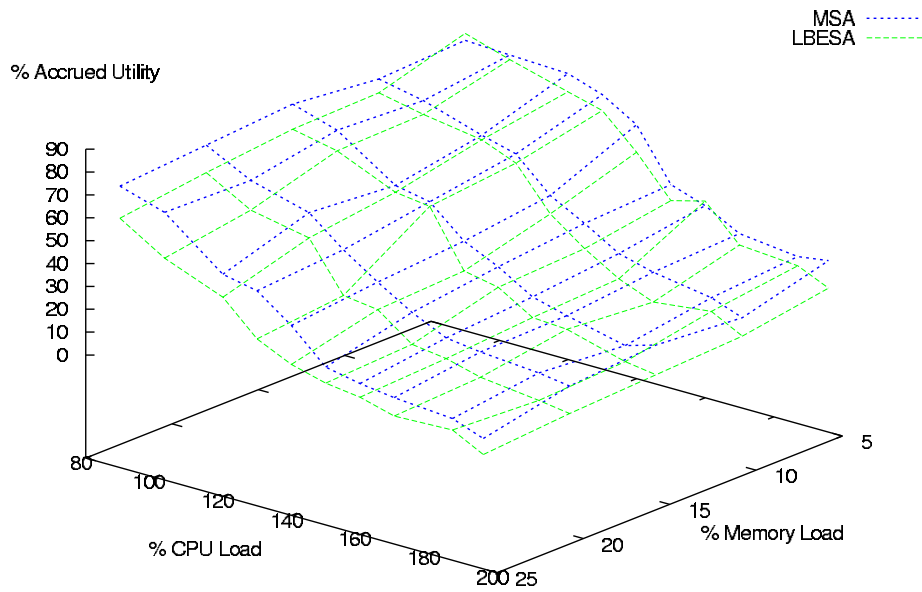


Figure 10: Comparative Performance Degradation (With Memory Constraints)

performance measurements of MSA’s task clustering initial prototype with DASA.

6.3 An Interesting Special Case

At $k = 0$, an early prototype of MSA turned into a strictly myopic highest-PUD-first scheduler for non-increasing TUFs. Under bursty traffic, MSA’s performance was surprisingly high. The cause of this phenomenon is likely the construction of complex schedules by the DASA-like family of UA algorithms. DASA may defer execution of a high PUD task if its EDF order is beyond a feasible lower PUD task with a nearer deadline. This mechanism indeed extracts higher utility from the sequence *if* the entire sequence is allowed to execute to completion. During rapidly changing conditions caused by a “spraying” of the system with numerous light-weight tasks, the $O(n \log n)$ myopic dispatching approach of MSA appears effective.

7 CONCLUSIONS AND CONTRIBUTIONS

The MSA algorithm makes the following contributions:

7.1 Explicit Dynamic UA Memory Allocation

In absence of a memory-aware UA-scheduler, tasks are limited to static memory allocation to ensure acquiring sufficient storage, or else risk a detrimental mid-execution memory allocation failure. Static allocation, though definitive, is inefficient use of memory by large, initially mostly zero-filled object files. Indiscriminate servicing of allocation requests can lead to disproportionately large areas of memory assigned to low PUD tasks, leaving little or no space for others. Taking into account system-wide knowledge of the tasks' TUFs, their current memory footprint, CPU demands, and allocation requests, MSA can make more sophisticated memory allocation decisions. This enables the application programmer to use explicit dynamic memory allocation with the assurance that a request will be serviced *if* it results in higher overall utility accrual. Furthermore, if the allocation request cannot be immediately granted, the requesting task can now be blocked in the hope of memory becoming available later (or until the task becomes time-infeasible).

7.2 Scalability

As a soft real-time UA scheduling algorithm, MSA degrades gracefully under increasing CPU load, thereby allowing the system to scale up. Making more efficient use of memory, MSA is well-suited for use in memory-constrained UA-governed embedded systems, thus allowing hardware requirements to scale down.

7.3 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 partial combinatoric element into an otherwise greedy approach.

7.4 Adaptability

Sensitive to CPU load and the number of tasks in the system, MSA can dynamically adjust k to produce schedules of varying granularity corresponding to 4 QoS levels.

7.5 UA Features

— Other than its memory awareness, MSA is a viable UA-scheduler with competitive performance. It is capable of scheduling arbitrary-shaped TUF/UA tasks, and provides limited support for resource dependencies.

8 FUTURE WORK

8.1 Memory Management Granularity

During schedule construction, we make the assumption that worst case execution times (WCE) of tasks are known in advance.

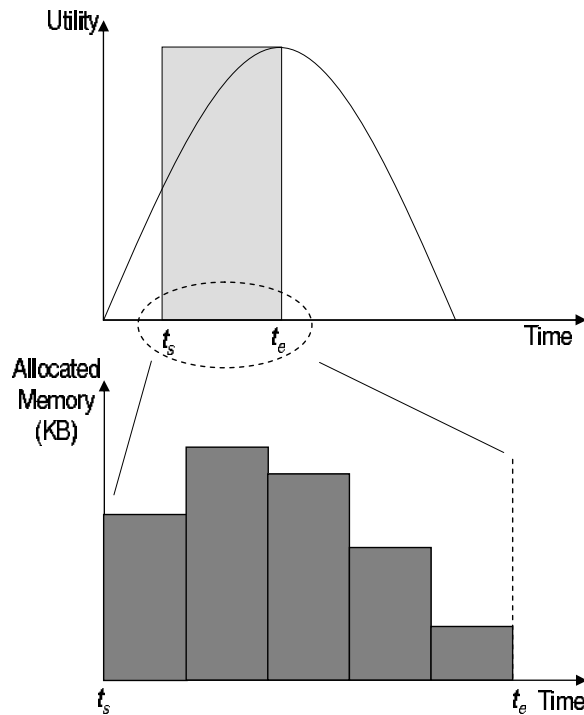


Figure 11: Memory Allocation Profile of a Task

It is then reasonable to assume that worst case allocation requirements (WCAR) for the tasks can also be determined ahead of time. Memory allocation patterns of the task can consequently be profiled as well. Figure 11 illustrates one such memory allocation profile.

Awareness of a task's WCAR would then allow making finer-grain memory-aware scheduling decisions. Given the fixed memory bandwidth, each WCAR-based task *segment* can then be individually scheduled such that at any point in time the segments are accommodated within the fixed memory boundaries,

and simultaneously maximize aggregate utility.

An additional *yield* scheduling point is required for the implementation as this would involve voluntary yielding of the CPU by the currently executing task in order to accomplish a “greater good.” In contrast to preemption of a task due to external events (e.g., arrival of a new, higher PUD task), the yielding of the CPU by an uninterrupted executing task would be implemented as an internally scheduled planned preemption.

8.2 Implicit Dynamic UA Memory Management

The simplicity of transparent garbage collection is highly desirable from an application programming and software engineering perspective. In the context of UA scheduling, the open research problem is determination of the mutual impact of the GC and the mutator during CPU overload conditions. At what point is it in the interest of the mutator to yield the CPU to the collector? How long can the mutator afford to wait for the collector prior to substantial utility loss? How much progress can the collector make during its execution, and will it suffice the mutator’s requirements?

The viability of TUF/UA scheduling in a Real-Time Java environment has been demonstrated in [8]. Java, however, exposes no memory manipulation details and the JVM handles all references on behalf of the executing program. Given the added complexities of Java’s security policies and strict type checking, garbage collection has been the single most complex issue faced by the Real-Time Java community.

8.3 System Properties

Predictability is of paramount import in real-time systems. As such, the timeliness properties of a UA memory management scheme must be established. Stochastic analysis can enhance the research by considering stochastic task execution times, memory allocation patterns, and inter-arrival rates.

Possibility of providing specific timeliness assurances, such as bounds on

system-wide or task-level utility, need to be investigated. Currently, UA algorithms attempt to maximize global utility. This, however, may not be optimal. Specific low-utility tasks possessing other, desirable properties can be overlooked. Such properties need to be considered at UA scheduling time.

Implementation of the multi-unit resource request model would enhance the flexibility of resource request and allocation.

ACKNOWLEDGEMENTS

This work was made possible in part by a grant from The MITRE Corporation (Grant 52917), a software grant by QNX Software Systems, Inc., and The U.S. Office of Naval Research (Grant N00014-00-1-0549).

REFERENCES

- [1] FreeBSD architecture handbook. <http://www.freebsd.org/Architecture>.
- [2] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, 1997.
- [3] K. Chen and P. Muhlethaler. A scheduling algorithm for tasks described by time value function. *Journal of Real-Time Systems*, 10(3):293–312, 1996.
- [4] R. Clark, E. D. Jensen, A. Kanevsky, J. Maurer, P. Wallace, T. Wheeler, Y. Zhang, D. Wells, T. Lawrence, and P. Hurley. An adaptive, distributed airborne tracking system. In *Proceedings of The Seventh IEEE International Workshop on Parallel and Distributed Real-Time Systems*, volume 1586 of *Lecture Notes in Computer Science*, pages 353–362. Springer-Verlag, April 1999.
- [5] R. K. Clark. *Scheduling Dependent Real-Time Activities*. PhD thesis, Carnegie Mellon University, 1990. CMU-CS-90-155.
- [6] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. In *Lecture Notes in Computer Science, No. 46*. New York, 1976.
- [7] Steven M. Donahue, Matthew P. Hampton, Morgan Deters, Jonathan M. Nye, Ron K. Cytron, and Krishna M. Kavi. Storage allocation for real-time, embedded systems. In Thomas A. Henzinger and Christoph Meyer Kirsch, editors, *Embedded Software: Proceedings of the First International Workshop, EMSOFT 2001, Tahoe City, CA, USA, October, 8-10, 2001*, volume 2211 of *Lecture Notes in Computer Science*, pages 131–147, Tahoe City, CA, October 2001. Springer Verlag.
- [8] S. Feizabadi, W. Beebee Jr., B. Ravindran, P. Li, and M. Rinard. Utility accrual scheduling with real-time java. In *Proceedings of The First Workshop on Java Technologies for Real-Time and Embedded Systems*, November 2003.

- [9] The Open Group Research Institute's Real-Time Group. *MK7.3a Release Notes*. The Open Group Research Institute, Cambridge, Massachusetts, October 1998.
- [10] Pieter H. Hartel, Marc Feeley, Martin Alt, and et. al. Pseudoknot: A float-intensive benchmark for functional compilers. In J. R. W. Glauert, editor, *6th Implementation of Functional Languages*, pages 13.1–13.34. School of Information Systems, University of East Anglia, Norwich, UK, 1994.
- [11] Roger Henriksson. Predictable automatic memory management for embedded systems. <http://citeseer.ist.psu.edu/henriksson97predictable.html>.
- [12] Roger Henriksson. Adaptive scheduling of incremental copying garbage collection for interactive applications. <http://citeseer.ist.psu.edu/henriksson96adaptive.html>, 1996.
- [13] 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.
- [14] 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.
- [15] 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.
- [16] Donald E. Knuth. *The Art of Computer Programming*. Four volumes. Addison-Wesley, 1968. Seven volumes planned (this is a cross-referenced set of BOOKs).
- [17] H. Lauer and E. Satterwaite. The impact of mesa on systems design. In *IEEE Int'l Conf. on Software Engineering*, pages 174–182, 1979.
- [18] P. Li, B. Ravindran, S. Suhaib, and S. Feizabadi. Utility accrual scheduling on off-the-shelf posix real-time operating systems: A formally verified framework. *IEEE Trans. on Software Engineering*, 2003. Submitted July 2003 (under review).

- [19] 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.
- [20] C. D. Locke. *Best-Effort Decision Making for Real-Time Scheduling*. PhD thesis, Carnegie Mellon University, 1986. CMU-CS-86-134.
- [21] D. P. Maynard, S. E. Shipman, R. K. Clark, J. D. 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 1988. Archons Project Technical Report 88121.
- [22] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3:184–195, 1960.
- [23] Kelvin Nilsen. A high-performance architecture for real-time garbage collection. <http://citeseer.ist.psu.edu/234276.html>.
- [24] Kelvin Nilsen. Taking out the garbage. <http://www.embedded.com/articleID=9900531>.
- [25] Kelvin D. Nilsen and William J. Schmidt. Hardware support for garbage collection of linked objects and arrays in real-time.
- [26] Patrik Persson. Predicting time and memory demands of object-oriented programs. <http://citeseer.ist.psu.edu/303988.html>.
- [27] 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.
- [28] Sartaj Sahni. Approximate algorithms for the 0/1 knapsack problem. *Journal of the ACM (JACM)*, 22(1):115–124, 1975.
- [29] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. on Computers*, 39(9):1175–1185, 1990.
- [30] Mukesh Singhal and Niranjana G. Shivaratri. *Advanced concepts in operating systems*. McGraw-Hill Inc., 1994.

- [31] D. Wells. A trusted, scalable, real-time operating system environment. In *Proceedings of The Dual-Use Technologies and Applications Conference*, pages 262–270, 1994.
- [32] P. Wilson, M. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *Proc. Int. Workshop on Mem. Mgmt.*, Kinross Scotland (UK), 1995.

APPENDIX

A ADDITIONAL PERFORMANCE

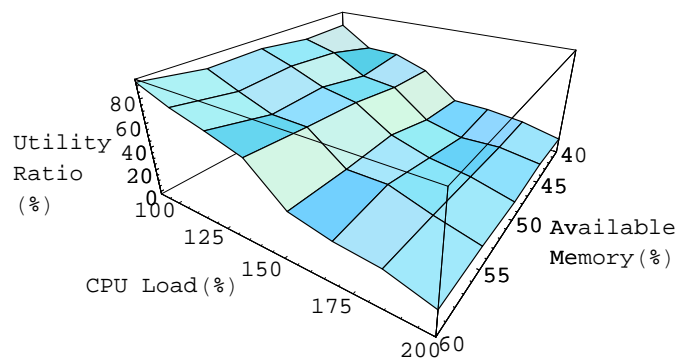


Figure 12: Cumulative Performance under Bursty Traffic with Memory Constraints

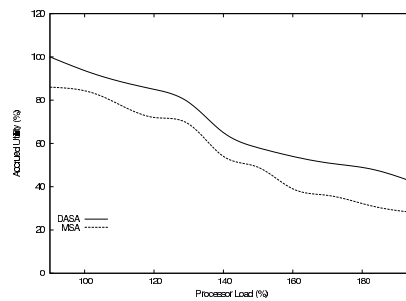


Figure 13: Performance of MSA and DASA with Dependencies and no Memory Constraints

B KERNEL TRACE SAMPLE

```
t:0x9acadb64 CPU:00 THREAD :THREADY          pid:12295 tid:204
t:0x9acaed10 CPU:00 THREAD :THMUTEX         pid:12295 tid:3
t:0x9acaee4 CPU:00 THREAD :THRUNNING       pid:12295 tid:204
t:0x9acaf23c CPU:00 KER_EXIT:SYNC_CONDVAR_SIG/83 ret_val:0
empty:0x00000000
t:0x9acaf820 CPU:00 KER_CALL:SYNC_MUTEX_UNLOCK/81 sync_p:805292c owner:0
t:0x9acb0760 CPU:00 THREAD :THREADY         pid:12295 tid:3
t:0x9acb08f0 CPU:00 THREAD :THREADY         pid:12295 tid:204
t:0x9acb0a78 CPU:00 THREAD :THRUNNING       pid:12295 tid:3
t:0x9acb1d6c CPU:00 KER_EXIT:SYNC_CONDVAR_WAIT/82 ret_val:0
empty:0x00000000
t:0x9acb239c CPU:00 KER_CALL:CLOCK_TIME/65 id:0 new(sec):0
t:0x9acb273c CPU:00 KER_EXIT:CLOCK_TIME/65 ret_val:0 old(sec):1078600408
t:0x9acb2d7c CPU:00 KER_CALL:SCHED_GET/88 pid:0 tid:204
t:0x9acb3034 CPU:00 KER_EXIT:SCHED_GET/88 ret_val:1 sched_priority:12
t:0x9acb4100 CPU:00 KER_CALL:TRACE_EVENT/01 mode:0x4000001e
class[header]:0x0000029a
t:0x9acb4270 CPU:00 USREVENT:EVENT:666, d0:0x00000000 d1:0x000000c8
t:0x9acb4490 CPU:00 KER_EXIT:TRACE_EVENT/01 ret_val:0x00000000
empty:0x00000000
t:0x9acb49c4 CPU:00 KER_CALL:TRACE_EVENT/01 mode:0x4000001e
class[header]:0x0000029a
t:0x9acb4b0c CPU:00 USREVENT:EVENT:666, d0:0x00000001 d1:0x000000c8
t:0x9acb4ca0 CPU:00 KER_EXIT:TRACE_EVENT/01 ret_val:0x00000000
empty:0x00000000
t:0x9acb52b8 CPU:00 KER_CALL:SCHED_SET/89 pid:0 sched_priority:12
t:0x9acb5e24 CPU:00 COMM :SND_PULSE_EXE scoid:0x40000003 pid:4102
t:0x9acb61bc CPU:00 THREAD :THREADY         pid:4102 tid:1
t:0x9acb652c CPU:00 KER_EXIT:SCHED_SET/89 ret_val:0 empty:0x00000000
t:0x9acb6d18 CPU:00 KER_CALL:CLOCK_TIME/65 id:0 new(sec):0
t:0x9acb709c CPU:00 KER_EXIT:CLOCK_TIME/65 ret_val:0 old(sec):1078600408
t:0x9acb793c CPU:00 KER_CALL:SYNC_CONDVAR_SIGNAL/83 sync_p:8052948 all:1
t:0x9acb896c CPU:00 KER_EXIT:SYNC_CONDVAR_SIG/83 ret_val:0
empty:0x00000000
t:0x9acb92f0 CPU:00 KER_CALL:SYNC_CONDVAR_WAIT/82 sync_p:8052934
mutex_p:805292ct:0x9acba70 CPU:00 THREAD :THCONDVAR pid:12295 tid:3
t:0x9acbb1cc CPU:00 THREAD :THRUNNING       pid:4102 tid:1
t:0x9acbbb18 CPU:00 COMM :REC_PULSE        scoid:0x40000003 pid:4102
t:0x9acbbd78 CPU:00 KER_EXIT:MSG_RECEIVEV/14 rcvid:0x00000000 rmsg:""
(0x00000000)
t:0x9acbc6a8 CPU:00 KER_CALL:MSG_SENDV/11 coid:0x00000003 msg:""
(0x00100102)
```

C TUF IN AWACS

By Binoy Ravindran

AWACS is an airborne radar system with many missions, including air surveillance. Surveillance missions generate aircraft tracks for command and control. The tracker’s most demanding computation called *association*, associates sensor reports to aircraft tracks. The tracker employs two sensors that sweep 180 degrees out of phase with a ten second period. Thus, association has a “critical time” at the period length. If the computation can process a sensor report for a track in under five seconds (half the sweep), that will provide better data for the corresponding report from the out-of-phase sensor. Thus, prior to critical time, utility of association decreases as critical time nears.

After the critical time, the utility of association is zero, because newer sensor data has probably arrived. Thus, if the processing load in one sensor sweep period is so heavy such that it cannot be completed, probably the load will be about the same in the next period. Thus, there will not be any resources to also process sensor data from the previous sweep.

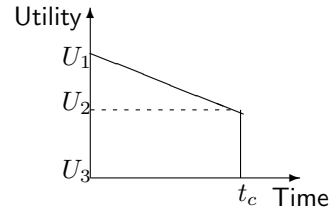


Figure 14: Track Association TUF

These semantics establish association’s TUF shape: a critical time t_c at sweep period; utility that decreases from a value U_1 to a value U_2 until t_c ; and an utility value U_3 after t_c . U_1 , U_2 , and U_3 are determined using metrics such as: (1) track quality, which is a measure of the amount of sensor data incorporated in a track record; (2) track accuracy, which is a measure of the uncertainty in the estimate of a track’s position and velocity; and (3) track importance, which is measure of track attributes such as threat. Figure 14 shows the association’s TUF.

The tracker creates threads for each airborne object that it tracks. The threads then perform the association computation. The TUFs of all threads have the same basic shape shown in Figure 14, but use different values for U_1 , U_2 , and U_3 . The tracker’s UA scheduling algorithm resolves the resource contention between the threads and schedules system resources to maximize

the total summed utility.

C.1 Adaptive Timeliness Performance Measurements

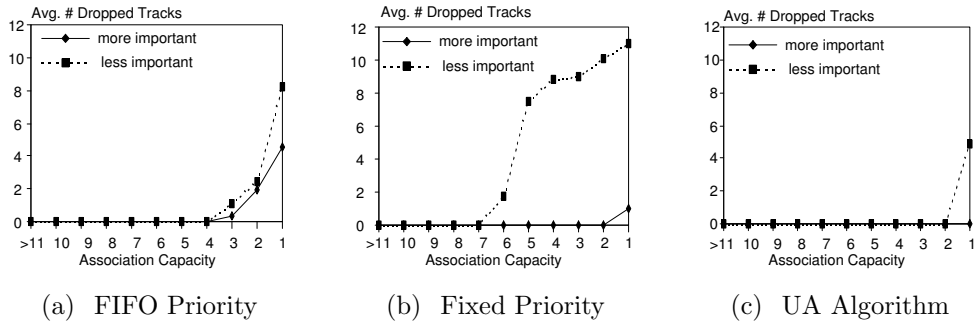


Figure 15: Average Number of Dropped Tracks vs. Association Capacity

The implementation of the AWACS was done using The Open Group’s MK7 operating system [9, 31], which contains the UA scheduling algorithm described in [20]. To understand how well MK7’s UA algorithm is able to provide adaptive timeliness, significant performance measurements and comparisons were made from the implementation. Different scheduling algorithms including FIFO priority and fixed priority were used and compared with the UA algorithm.

Figures 15 and 16 show a snapshot of the measurements. The figures show the average number of dropped tracks and the track quality versus association capacity for the three scheduling policies. The x -axes in the figures represent increasingly constrained system in terms of association capacity. The figures illustrate that the UA algorithm sacrifices a little quality of more important tracks to maintain higher quality for less important ones, thereby illustrating the adaptivity of the TUF/UA paradigm.

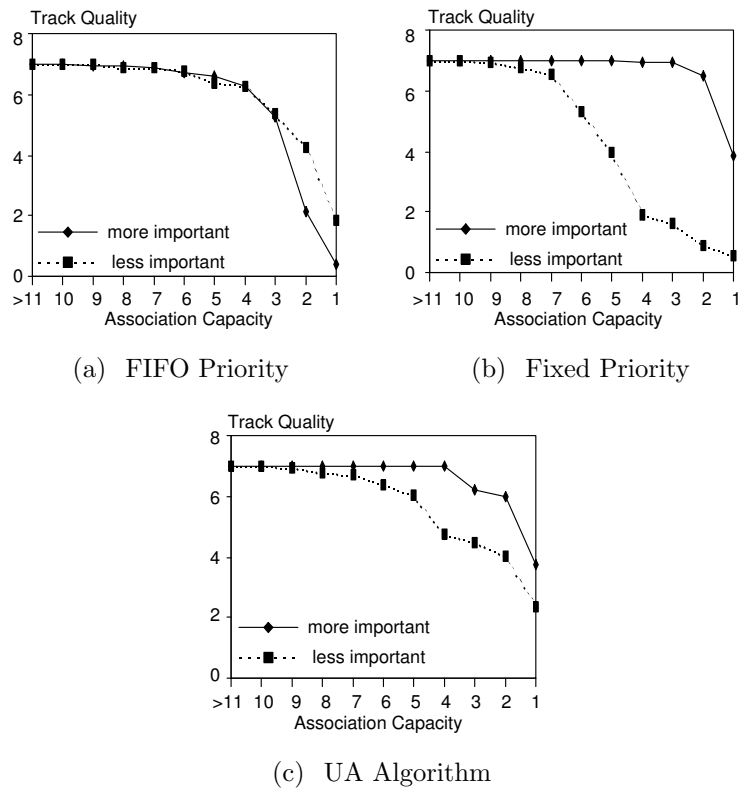


Figure 16: Track Quality vs. Association Capacity