Utility Accrual Scheduling With Real-Time Java

Shahrooz Feizabadi
Real-Time Systems Laboratory
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061
shahrooz@vt.edu

William Beebee, Jr.
MIT Laboratory for Computer Science
200 Technology Square, Cambridge, MA 02139
wbeebee@mit.edu

Binoy Ravindran and Peng Li
Real-Time Systems Laboratory
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061
{binoy,peli2@vt.edu}

Martin Rinard
MIT Laboratory for Computer Science
200 Technology Square, Cambridge, MA 02139
rinard@mit.edu

Abstract

Graceful performance degradation during overload conditions is the primary objective of soft real-time systems. Utility accrual soft real-time scheduling algorithms allow specification of highly customized temporal system behavior during overload. Such algorithms are typically found in real-time *supervisory* systems where significant run-time uncertainty exists. This paper outlines an investigation of several utility accrual scheduling algorithms implemented in a Real-Time Java (RTJ) environment. These alternate schedulers are constructed, tested, and evaluated under the MIT FLEX/RTJ Compiler Infrastructure. The scheduling framework for this environment and its associated scheduling primitives are described and the corresponding performance characteristics are profiled. Furthermore, we outline the architecture of an experimental distributed Real-time Java scheduler.

Keywords

Real-time Java, Utility Accrual Scheduling, Overload Scheduling, FLEX/RTJ, Distributed Real-time Java, Soft Real-time, Embedded Systems Tools, Real-time Systems

1. Introduction

The Real-Time Specification for Java (RTSJ) mandates a strict fixed-priority preemptive scheduler with 28 unique priority levels [30]. This is consistent with core facilities traditionally provided by real-time operating systems. The properties of this scheduling model are well-understood and widely utilized [11, 6]. Augmented with built-in support for priority inversion [30, 15], RTSJ provides the essential tools for hard real-time programming. This *base scheduler* is the minimal RTJ scheduler implementation requirement.

Beyond the base scheduler, the specifications anticipate construction of alternate schedulers and provide the requisite API's to accommodate them: real-time threads now enjoy language-level support for such dynamic scheduling notions as deadline, cost, and cost enforcement.

The motivation for this paper is to investigate the viability of construction of such dynamic schedulers in general, and Utility Accrual (UA) schedulers in particular.

1.1 Contributions

Below is an outline of contributions of this paper:

- **Utility Accrual RTJ scheduling:** We demonstrate the viability of constructing UA schedulers in Real-time Java. By corollary, we demonstrate the implementation of complex dynamic deadline schedulers. We present the performance measurements for these algorithms in a FLEX/RTJ environment, compare the results with their counterparts in a QNX RTOS [18] environment, and note the associated performance characteristics.

- **User-Level RTJ Thread multiplexor:** The FLEX/RTJ `chooseThread` is designed to offer fine-grain thread control at the user-level: any thread can be arbitrarily chosen for execution at any point for an arbitrary period of time. Presenting a single point-of-entry into the system, the thread multiplexor is simple and effective.

- **Separation of policy and mechanism:** Inherent complexities of interactions between the JVM and the OS kernel present significant technical challenges for implementation of alternate scheduling policies at the user level. JVM's typically implement a thin scheduling layer over the existing OS scheduler, map Java threads onto native threads, and thereby leverage the properties of the underlying scheduler.

  The FLEX/RTJ scheduling framework abstracts away implementation complexities by providing a high-level construct for transparent thread manipulation. It allows the programmer to focus on policy rather than implementation mechanisms.

This research was supported by DARPA/AFRL Contract F33615-00-C-1692, and the Singapore-MIT Alliance.

- **Experimental Distributed RTJ Thread Scheduler:** We outline a framework for scheduling of distributed threads in a real-time Java environment. This is a `chooseThread` implementation over sockets and includes the basic scheduling API's required for a distributed system.

- **Implementation:** Concurrency on a uniprocessor system is merely a high-level abstraction – albeit an effective one. The thread multiplexor is in essence a serializer. To utilize it, the programmer needs to adopt a "systems-level perspective" in place of the traditional higher level concurrency constructs. We discuss insights and experience gained from implementation of our schedulers in this environment.

1.2 Organization

The remainder of this paper is organized as follows: section 2 outlines the basic concepts of utility accrual scheduling and provides an example. Section 3 is a discussion of the scheduling framework, its design objectives, provided system facilities, outline of the distributed threads management scheme, and implementation details. Section 4 outlines the experimental results. The discussion of experiment design is followed by a description of each implemented algorithm and the corresponding performance results and observations. The performance of the scheduling framework itself is discussed in section 5. Section 6 presents a discussion of possible future enhancements and issues, and we conclude the paper in section 7.

2. Utility Accrual Scheduling

Real-time systems, by definition, are those designed to predictably adhere to a predefined set of timeliness constraints [20, 21]. *Hard* real-time systems must always satisfy all predefined timeliness requirements. *Soft* real-time systems, by contrast, are those designed to exhibit a specific temporal behavior when the system is unable to meet all its timing constrains.

The most commonly used notion of a timing constraint is the *deadline*. A deadline is a point in time prior to which, the completion of an activity yields the most utility; and conversely, less utility is derived by completion of the task after that specific point in time. In the context of hard real-time systems, this is the binary choice of completing a task by the deadline and gaining its full utility, or gaining no utility for task completion after the deadline [20]. It is of no utility, for example, to service a network request after the connection has timed out.

In soft real-time systems, the utility of a task is a function of its completion time. Completion of a task after the deadline may yet be of some utility — a radar "ghost" resulting from delayed processing of residual data is preferable to a blank screen.

An emerging class of real-time control systems, collectively known as *supervisory* systems [3], are increasingly incorporating soft real-time algorithms for resource scheduling decision support. Supervisory systems are typically comprised of a hierarchy of lower level real-time systems, and function in complex environments with significant run-time uncertainty.

In the absence of a priori knowledge of upper bounds on task execution times, the hard optimality criterion of always meeting all deadlines is difficult to guarantee at desing time. As soft real-time systems enter overload conditions, the "sequencing optimality" can then be defined in terms of producing a specific system-wide behavior such as minimizing missed deadlines, maximizing aggregate utility, etc.

The timing constraints of such supervisory systems can be accurately expressed in terms of Time/Utility Functions (TUF's) [20]. Figure 1 illustrates several examples of TUF's. This generalization offers a great degree of versatility for representing timeliness requirements. Deadline timing constraints can be viewed as a special case, and modeled using rectangular TUF's – Figure 1(b).

Supervisory systems are specifically designed to operate in environments where there exists a potential for overload. TUF-based algorithms can be found in a wide range of application domains such as telecommunications, defense, and industrial automation. [23] Utility accrual models can also aide management and decision support systems: during a catastrophic natural disaster, an overloaded emergency response center can temporarily delay dispatching of emergency supplies to a remote location in favor of first servicing a nearby high population density area.

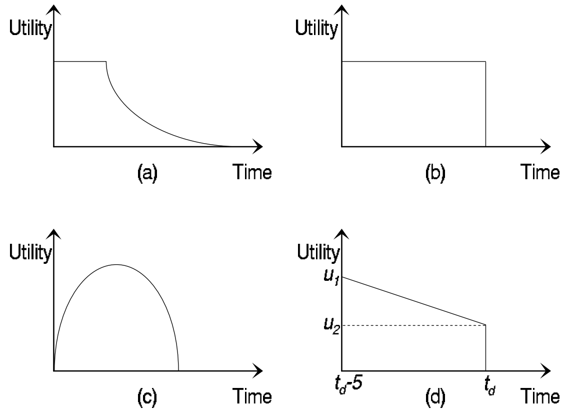

Figure 1: TUF examples

Supervisory system design is a significant software (and systems) engineering effort and requires global knowledge of dynamically varying conditions to appropriately compose the corresponding TUF's. Examples of highly sophisticated UA-based supervisory systems in complex environments include MITRE Corporation's AWACS tracking system [22], and the BM/C^2 System developed by General Dynamics and Carnegie Mellon University [27].

Figure 1(d) illustrates the data processing TUF for two sensors 180 degrees out of phase, each with a 10 second period. The sensor data processing deadline is therefore 10 seconds. If, however, the data from one sensor is processed within the first 5 seconds of the period, it can improve the quality of the data received from the out of phase sensor. Completion of data processing beyond the first 5 seconds can still partially strengthen the incoming data from the out of phase sensor. The utility of the data, however, linearly diminishes to zero with the approaching 10 second deadline as new sensor data will be collected [22]. A more detailed UA algorithm design and implementation process is outline in [27].

3. The Scheduling Framework

Although the RTSJ explicitly supports scheduling policies other than fixed-priority preemptive, it does not elaborate the interfaces to allow the creation of such schedulers:

> *If the RTSJ required a deadline scheduler, its implementation would require kernel work on any of the real-time platforms (such as VxWorks, OS-9,QNX, Neutrino, LynxOS, PSOS, and Linux). Instead, the RTSJ has scheduler API's that make an effort to anticipate scheduling paradigms that are unlike anything known today. The goal was to let OS/JVM vendors build new schedulers and make them accessible through the RTSJ API's. Unfortunately, the RTSJ does not attempt to expose the interfaces that would let a programmer create a new scheduler. That is a job for someone with access to the internals of the JVM and the supporting system software.* [8]

The design of the FLEX/RTJ scheduling framework bridges this gap and alleviates the need for detailed knowledge of, and access to, the JVM or the OS.

3.1 FLEX/RTJ

We implemented the general real-time scheduler framework and user threads package using the MIT FLEX compiler infrastructure. FLEX is an ahead-of-time compiler for Java that generates both native code and C.

Real-Time Java is a superset of the Java language (RTSJ) which provides a framework for building real-time systems. FLEX has an implementation of Real-Time Java which includes both region-based memory management [28, 29, 32, 33] and a scheduler framework.

Separation of policy from mechanism to achieve flexibility in real-time user threads packages is not a new idea [34].

Our threads package exports thread multiplexing and lock handling (which influences scheduling policies) to the user scheduler. User schedulers can be written entirely in Java. Since the `chooseThread` abstraction can implement any interleaving of threads, many policies can be implemented simply by providing a single method. Performance of the system is often dependent on the performance of the `chooseThread` method. The system restricts interaction with the scheduler to a minimal interface, enhancing productivity by reducing the scope of the interactions which must be considered when debugging or optimizing an implementation of a new scheduling algorithm. For example, a simple preempting round-robin policy can be expressed in only four lines of code using `chooseThread`.

3.2 Design Benefits

The primary design benefit of the scheduler framework is to provide a single, central point of control for all timing aspects of the Real-Time Java system. Without centralized information and control, a single event can invalidate any hard real-time guarantees of the system. For instance, a scheduler may choose to defer handling an asynchronous event or asynchronous interrupt in favor of running a higher-priority task. The scheduler may need to know about all timers in the system in order to perform adequate admission control. Priority inheritance relies on knowledge of locks held in the system.

A second design benefit is to provide a single point where knowledge and control of the operating system, Real-Time Java libraries, the Java runtime, the user program, middleware, and information from static program analysis can be brought together. For instance, the user program or middleware may inform a specialized user scheduler of unique scheduling characteristics of an asynchronous event (which may prove useful to an implementation of the *Distributed Real-Time Specification for Java.* [16]

A third design benefit is to promote concise and flexible scheduler design through the use of minimal interfaces with expressive power. Since the scheduler may potentially need to manage information about many different aspects of the system simultaneously, complex, sophisticated schedulers which take advantage of the full flexibility of the system are often hard to write. Concise schedulers are easier for programmers to write and debug. Furthermore, the system provides a debugging interface which logs every event occurring in the system, every choice made by the scheduler, and the state maintained by the scheduler which led to a particular choice. A utility provides a graphical summary of the choices made by the scheduler and the time interval which has elapsed between each scheduling point, a simple overview of scheduler behavior. Since the scheduler is simply a Java class, a scheduler can even be tested outside of the system during initial development.

3.3 System Facilities

The system provides several primitives to facilitate the implementation of hard and soft real-time scheduling policies as described in Figure 2. Work and system load can be estimated using `clock`. A reservation interface allows schedulers to interact with the TimeSys kernel to provide hard real-time guarantees when running under a TimeSys RTOS. `setQuanta` provides preemption capabilities and `sleep` can yield time to the kernel. `contextSwitch` can force a context switch in response to handling an event.

The system can generate many events which can affect policy decisions (Figure 3). The scheduler can choose to either handle or ignore them. POSIX thread events are generated by native methods or by the runtime in response to Java synchronization. All POSIX thread events can be handled by the scheduler. RTSJ events are generated by our RTSJ implementation and can inform the scheduler of all aspects of the RTSJ which influence scheduling. Events generated by the user program, compiler, or middleware are all policy-specific. Thread entry and exit, blocking and unblocking

Provided by user scheduler:

`chooseThread(time)` → { thread, quanta }
Chooses a user `thread` from the currently available threads to run for the next `quanta` of time or until blocked, whichever occurs first. The scheduler multiplexes user threads into a heavy thread. The `time` provided to `chooseThread` is the start of the context switch.

feasibility
Standard RTSJ interface to determine feasibility of a set of tasks.

Utilities provided by system:

`getDefaultScheduler()` → { scheduler }
To install a scheduler, simply point this to your `scheduler` (a Java object allocated in immortal memory).

`contextSwitch()`
Force a context switch at the earliest available time.

`setQuanta(time)`
Reset the time until the next context switch. `setQuanta` is provided to adjust the current schedule in response to events.

`clock()` → time
Return process CPU `time` used by the process for calculating work done by a thread.

`sleep(time)`
Yield control to the operating system scheduler for `time`.

Optional RTOS-specific utilities provided by System:

`reserveCPU/NET` → success flag
Interface to TimeSys linux to provide CPU/NET reservation access to scheduler. The returned `flag` indicates whether the reservation was successful.
Successive calls to `reserveCPU/NET` simply modify the current reservation.

Figure 2: Scheduler interface

Mandatory event handlers provided by user scheduler:

`addThreadremoveThread(thread)`
Called by the system to give the scheduler an opportunity to update internal data when a Java or C thread starts or ends.

`enableThreaddisableThread(thread)`
Called by the system to inform scheduler of threads unblocking or blocking on locks.

Optional event handlers for making policy decisions:

pthread events
Handlers are provided to allow the scheduler to be notified of all pthread events handled by our user-threads package.

RTSJ events
Handlers are provided to allow the scheduler to be notified of all events generated by our RTSJ implementation.

Policy-specific events support:

user program events
The scheduler can provide its own API to the user program or middleware for generating events directly (via a simple method call).

compiler generated events
The compiler can weave code into the user program to call methods on the scheduler directly based on program analysis simply by inserting `CALL` instructions into the program during compilation.

Figure 3: Scheduler event handler interface

Utilities provided by the system:

`bind(scheduler name)` → thread
Bind the current scheduler to `scheduler name` in the name service and start a server to handle events. Returns the thread which handles incoming events.
Network timing can be managed through `chooseThread`, `disableThread`, and `enableThread`

`resolve(scheduler name)` → scheduler stub
Resolve the `scheduler name` in the name service to a stub which represents the destination scheduler.

`generateDistributedEvent(destination, message ID, data)` → thread
Create an event on the `destination` scheduler using a network call.
Returns the thread which handles outgoing events.

Optional event handler for distributed real-time scheduling:

`handleDistributedEvent(name, message ID, data)`
Allow the scheduler to respond to an event generated by another scheduler.

Figure 4: Distributed real-time thread and event support

cannot be ignored by the scheduler since they impact the active thread list.

3.4 Distributed Real-time Java Threads

A socket-based implementation provides distributed event support (Figure 4). `generateDistributedEvent` can generate an event on another scheduler across the network. The event can be handled by `handleDistributedEvent`. In the socket-based implementation, a client thread and a server thread are scheduled by the scheduler. The server thread causes `disableThread` to be called on the scheduler when no pending request is available. `enableThread` informs the scheduler that an event has arrived. The client thread causes `disableThread` to be called when a request has been sent. The scheduler can then use `chooseThread` and `setQuanta` to provide hard real-time bounds on the processor time given to servicing the network. We have implemented an RMI-based mechanism that allows communication between multiple RTJ schedulers each running on remote node.

3.5 Implementation

The thread multiplexor uses a standard, signal-based, `sigsetjmp` and `siglongjmp` implementation. The system tells the kernel to use signal-interruptable kernel calls. The system saves the thread context using `sigsetjmp`. The system then sets a timer at the end of a context switch to cause a `SIGALRM` to be generated after an interval set by `setQuanta`. The signal handler calls `chooseThread` to determine the next thread to be run, retrieves the saved registers associated with the thread and uses `siglongjmp` to simultaneously unblock `SIGALRM` and restore the program counter to that saved by the context switch.

Thread start involves setting up a jump buffer and environment that appears like a `sigsetjmp` from the thread's start. Blocking on a lock calls `disableThread` to inform the scheduler of the blocked status and then forces a context switch by signalling `SIGALRM`. Unblocking calls `enableThread`. `SIGALRM` is blocked during scheduler calls to prevent reentry, facilitate local reasoning about the correctness of the policy implementation, and provide bounds on stack usage. Therefore, the scheduler implementation should provide bounds on method calls to ensure bounded preemption latency.

4. Scheduler Performance

In this section we present an overview of the implemented scheduling algorithms followed by the associated performance results.

4.1 Experiment Design

The following points regarding our experimental setup are of significance:

- The algorithms we investigated were initially coded and tested in a simulated environment. The simulated scheduler performance depicts ideal conditions under various processor loads as the simulator assumes instantaneous preemption, and no scheduler or OS overhead.
- Each scheduler was tested with a set of concurrent periodic tasks and its behavior at each scheduling point was verified against a manual trace for various execution hyper-periods and load conditions.
- The tasks in the system are vacuous threads executing busy wait to generate artificial CPU load.
- The experiments were performed with garbage collection disabled.
- We tested the schedulers with the highest possible priority under the TimeSys Linux RTOS [19] to minimize OS overhead and eliminate unanticipated CPU contention from other processes — alternatively, it is possible to use the TimeSys CPU reservations mechanism to guarantee each scheduler a dedicated share of the CPU as in [9].
- We then tested the same scheduling policies with identical task/load conditions in a QNX Neutrino RTOS [24] environment using C/POSIX implementations.

- The performance of each scheduler is then compared to its simulated, and QNX counterparts.
- In addition to the utility accrual schedulers, we implement the well-known Rate Monotonic, and Earliest Deadline First (EDF) schedulers to establish a baseline.

4.2 RMA

Rate Monotonic Analysis (RMA) provides a verifiably optimal priority assignment algorithm for systems with fixed-priority preemptive schedulers and independent periodic tasks [10]. Priorities, in descending order, are assigned to tasks in ascending order of period — i.e., the highest priority goes to the task with the lowest period (highest frequency), etc.

Rate Monotonic (RM) is a static scheduling algorithm as the corresponding priorities are assigned at design time and remain constant throughout run-time. RMA has been proven to be optimal: if a task set cannot be scheduled using this policy, it is not possible to schedule the same task set using any other fixed-priority scheduling algorithm. Furthermore, Liu and Layland [10] derived a test that guarantees the schedulability of a set of N independent periodic tasks solely based on their CPU utilization:

$$\sum_{i=1}^{N}\left(\frac{C_i}{T_i}\right) \leq N(2^{1/N}-1)$$

Where C_i and T_i are a task's execution time and period respectively. As the lower bound asymptotically approaches 69.3%, any arbitrary set of N tasks with a lower total utilization demand is guaranteed to be schedulable. This test is considered to be sufficient, but not necessary [21] — it does not imply that a task set with higher CPU utilization demand is *not* schedulable. Studies have shown that a *random* set of tasks can effectively be scheduled according to rate monotonic assignment up to approximately 88% CPU utilization [1].

Response time analysis [4] provides the sufficient *and* necessary schedulability test for RMA (as well as other scheduling policies). Figure 5 shows the typical performance of our implementation of an RM scheduler.

Though rate monotonic assignment continues to provide some level of deadline satisfaction during overload, its scope is limited since it merely services the highest frequency task without regard to any other selection criteria. As the CPU utilization (C_i/T_i) for the highest frequency task approaches 100%, the overall deadline satisfaction ratio approaches that of the highest frequency task — all other tasks are ignored.

4.3 EDF

The Earliest Deadline First (EDF) scheduling algorithm merely executes tasks in ascending order of deadline at any scheduling point [10]. This is a dynamic scheduling policy as the tasks' absolute deadlines are determined at run-time and the execution order of tasks are arranged accordingly.

EDF is shown to be optimal (up to 100% CPU utilization) for uniprocessor systems: if a task misses a deadline under an EDF scheduler, no other policy can arrange a schedule where the task meets its deadline [10]:

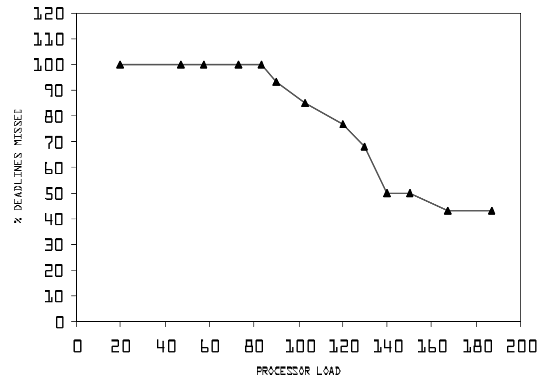

Figure 5: RMA Scheduler Performance

$$\sum_{i=1}^{N}\left(\frac{C_i}{T_i}\right) \leq 1$$

EDF suffers from a "domino effect" [26] at or near overload. The point at which EDF fails, however, can be used as a performance indicator of system overhead. RED (Robust Earliest Deadline Scheduling) [2] addresses the instability of EDF during overload and uses the mechanism of "deadline tolerance" for enhanced scheduling decision making.

The performance of our EDF implementation is depicted in Figure 6.

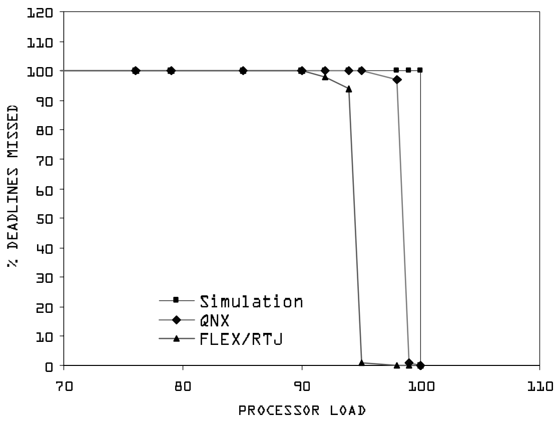

Figure 6: Performance of EDF scheduler

4.4 LBESA

LBESA (Locke's Best Effort Scheduling Algorithm) [26] was the first publicly available utility accrual real-time scheduler intended for supervisory control systems. LBESA was implemented in the Alpha distributed real-time OS kernel [7]. The BM/C^2 system [27] was developed using Alpha.

```
1. Create an empty schedule.
2. Sort all tasks by deadline.
3. For each task (in increasing order of deadline):
    3.1. Insert task in schedule at deadline position.
    3.2. Check schedule feasibility.
    3.3. While (schedule not feasible and schedule
         not empty):
        3.3.1 Remove task with lowest potential
              utility density from schedule.
4. Execute task in schedule with earliest deadline.
```

Figure 7: Sketch of LBESA algorithm

LBESA uses non-convex functions (values cannot increase after a decrease) to describe time utility. Furthermore, it stochastically characterizes task execution times using probability distribution functions. Task interarrival times need not necessarily be periodic or deterministic. Tasks are assumed to be independent. A general overview of the algorithm is outlined in Figure 7.

The algorithm produces an EDF-based absolute deadline ordering and determines feasibility. If the system is in overload, the algorithm continually rejects the task with the lowest potential utility density until a feasible schedule is achieved.

The concept of potential utility density used by LBESA is analogous to the notion of "return on investment." At any arbitrary point, the time invested thus far in the task's execution can be weighed against the potential return on investment at maturity - utility yield at task completion.

Reflecting the nature of its application domain, a full LBESA implementation is proportionally complex and requires significant system resources. We implemented the basic algorithm for the special case of rectangular TUF's and known worst case execution times. The scheduler performance is illustrated in Figure 8.

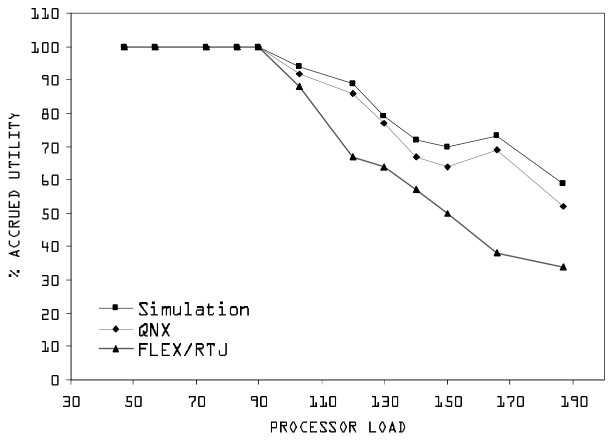

Figure 8: Performance of LBESA scheduler

In this experiment, the performance of the QNX implementation closely follows the simulation results, whereas the RTJ implementation somewhat lags behind. Due to higher aggregate system overhead and lower time resolution, the RTJ implementation "slips" early-on and determines a tentative schedule infeasible, whereas the QNX implementation is able to make the deadline. The subsequent scheduling decisions by each system therefore vary, resulting in the performance disparity.

4.5 DASA

DASA (Dependent Activity Scheduling Algorithm) [14] is another example of a utility accrual resource scheduler used in supervisory control systems. The second generation of the Alpha OS includes an implementation of DASA. The Open Group's OSF.1 MK7.3a [25] distributed RTOS also incorporates DASA.

DASA has the twofold objectives of maximizing system-wide accrued utility while minimizing the number of missed deadlines.

Concurrent tasks can develop relative dependencies as they serially access devices, channels, and other exclusively shared system resources: a task requesting a resource is considered dependent on the task currently holding that resource. Given its objectives, DASA makes appropriate scheduling decisions while taking into account such dynamic dependencies, and the corresponding precedence relationships. The algorithm assumes that possible deadlock resulting from cycles in the dependency graphs can be detected and resolved. A simplified version of the algorithm, DASA/ND [13], can be used in systems where tasks are known to be independent. The performance of the DASA/ND scheduler is illustrated in Figure 9.

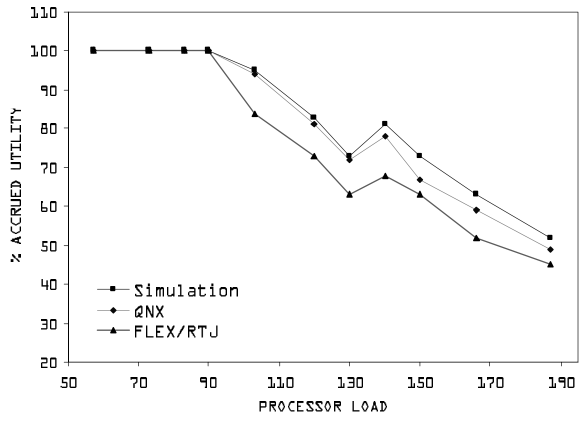

Figure 9: Performance of DASA/ND scheduler

DASA considers all deadlines as hard, and exclusively relies upon rectangular utility functions. Utility maximization at any scheduling point is achieved by dynamically determining the task that would yield the most utility if chosen to continue. The remaining competing tasks may then be

```
1. Create an empty schedule.
2. Determine dependencies among tasks.
3. Calculate potential utility density for
   each task.
4. Resolve deadlocks if detected.
5. Sort tasks by potential utility densities.
6. Examine each task in decreasing order of
   potential utility density:
   6.1. Tentatively add the task and its dependent
        tasks to the schedule in deadline order.
   6.2. Test schedule feasibility.
   6.3. If not feasible, remove task from schedule.
   6.4. Optimize schedule if possible.
7. Execute task in schedule with earliest deadline.
```

Figure 10: Sketch of DASA algorithm

preempted or aborted (during overload) in favor of the chosen task. DASA is equivalent to EDF (i.e., is optimal) up to the theoretical bound of 100% CPU utilization [14]. Figure 10 is a generalized sketch of the algorithm.

Tasks in DASA are assumed to contain the static deterministic attributes of worst case execution time, absolute deadline, and utility. Furthermore, the timinig requirements of the shared resources need to be known in advance: should a task be selected for abortion, the scheduler must know how soon the held resources can be released and reassigned.

Figure 11 shows the performance of a limited DASA implementation. Only one shared system resource is simulated to eliminate the possibility of deadlock, and tasks are assumed to implicitly release the resource at the conclusion of their execution.

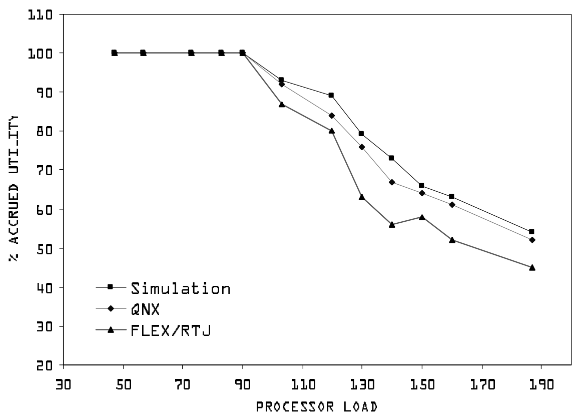

Figure 11: Performance of DASA scheduler

As with LBESA, the performance disparity between the RTJ and QNX implementations is due to the relative overhead incurred by each system. The schedulers make identical decisions up to the point of nearing system performance limits. Though the task attributes (e.g., release time, execution profile, potential utility, etc.) are the same for each implementation, the schedulers make differing decisions after the initial "slip" caused by reaching performance boundaries. The RTJ implementation does, however, continue to accrue utility as best it can, given the circumstances of its environment.

Furthermore, it should be noted that the aggregate accrued utility of each system is highly dependent on the present task-sets and their associated attributes — it is possible (though not likely) for a task-set to produce identical ordering of scheduling decisions by each implementation during overload.

5. Scheduling Framework Performance

The two metrics which drive the performance of the scheduling system are the preemption latency and context switch service time. The difference in time from the time set by `setQuanta` and the time the signal handler runs is measured as preemption latency. The scheduling system blocking `SIGALRM` and the operating system clearing interrupts during timing critical sections influences preemption latency. The difference in time from the time of the beginning of the signal handler to the actual context switch is the context switch service time. Context switch service time is primarily influenced by the performance of `chooseThread`. Therefore, the performance of the system in terms of preemption latency and context switch service time is dependent on the policy implementation. The typical context switch service time for the complex AU schedulers are measured to be on the order of 1000 micro-seconds whereas the simpler scheduling algorithms require approximately 50 micro-seconds. The preemption latency, however, is less affected by the `chooseThread` implementation. The typical preemption latency for UA schedulers is approximately 20 micro-seconds whereas their simple scheduler counterparts stay below 10.

6. Future Work

While well-suited to their application environment, both LBESA and DASA have inherent limitations: LBESA does not account for task dependencies and takes non-convex TUF's; DASA only takes rectangular TUF's and requires deterministic parameters. Generic Benefit Scheduling algorithm (GBS) [17] is the next iteration of UA algorithms and is capable of handling arbitrary TUF's while allowing for task dependencies. A full RTJ implementation of GBS would further explore UA scheduling in a RTJ and DRTJ environment.

RTSJ provides the language-level API's for alternate schedulers. FLEX/RTJ thread multiplexor provides a high-level abstraction for low-level thread manipulation (and therefore, arbitrary scheduling). An intermediate layer can be constructed to formally specify and map the semantic requirements of real-time threads onto the `chooseThread` abstraction.

The runtime system allows each RTJ thread to choose its own scheduler therefore making it possible for the simultaneous existence of multiple concurrent threads each under a different scheduling policy. It is also possible to construct a hierarchical scheduling environment. A framework such as HLS (Hierarchical Loadable Schedulers) [12] can be adapted to Real-time Java in order to analyze CPU allocation *guarantees* in the case of multiple scheduling policies.

7. Conclusions

Our implementations and subsequent performance analyses demonstrate the viability of complex utility accrual scheduling in a Real-time Java environment. We outline the implemented algorithms and compare their respective performance measures.

The FLEX/RTJ thread multiplexor scheduling primitive provides an effective and flexible tool for construction of arbitrary schedulers. This abstraction now allows RTJ scheduler construction to be a user-level activity rather than a static JVM/OS component.

The scheduling framework has been extended to accommodate distributed thread management. The corresponding high-level scheduling abstractions are designed to parallel the simplicity and effectiveness of the stand-alone framework. Preliminary tests of the experimental system indicate the viability of extending the existing scheduling model.

FLEX has been ported to a variety of architectures such as x86, StrongARM, and PowerPC under operating systems such as RedHat, Debian, Familiar, and TimeSys Linux. Users can compile to alternate targets using C cross-compilers. The FLEX/RTJ framework provides an effective array of tools for embedded systems design and implementation.

8. Acknowledgements

The UA scheduling-related effort in this paper owes much to the seminal work of E. Douglas Jensen, C. Douglas Locke, and Raymond Clark. We are grateful to C. Scott Ananian, Brian Demsky, Catalin Francu, Brian Fink, and Dumitru Daniliuc for contributing code which influenced this work.

9. References

[1] J. Lehoczky, L. Sha, and Y. Ding. *The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior* In Proceedings of IEEE Real-Time Systems Symposium, December 1992.

[2] G. Buttazzo, and J. Stankovic. *RED: Robust earliest deadline scheduling* In Proceedings of the 3rd International Workshop on Responsive Computing Systems pages 100-111, September 1993.

[3] E. D. Jensen *Asynchronous Decentralized Real-Time Computer Systems* In *Real-Time Computing* Proc. of NATO Advanced Study Institute, Springer Verlag, October 1992.

[4] P. Harter *Response times in level-structured systems ACM Trans. Computer Systems* vol. 5, pp. 232–248, Aug. 1987

[5] L. Sha, M. Klein, and J. Goodenough. *Rate Monotonic Analysis for Real-Time Systems* Software Engineering Institute, Carnegie-Mellon University, USA. CMU/SEI-91-TR-6

[6] G. Buttazzo. *Hard Real-time Computing Systems : Predictable Scheduling Algorithms and Applications* Boston, Massachusetts: Kluwer Academic Publishers, 1997.

[7] E. Jensen and, J. Northcutt. *Alpha: A Non-Proprietary Operating System for Large, Complex, Distributed Real-Time Systems* In *Proceedings of The IEEE Workshop on Experimental Distributed Systems* Pages 35–41, 1990.

[8] P. Dibble. *Real-Time Jave Platform Programming* Paulo Alto, California: Sun Microsystems Press, 2002.

[9] D. de Niz, and R. Rajkumar *Chocolate: A Reservation-Based Real-Time Java Environment on Windows NT* In Proceedings of the IEEE Real-time Technology and Applications Symposium Washington D.C., June 2000

[10] C. Liu, and J. Layland. "Scheduling Algorithms for Multiprocessing in a Hard-Real-Time Environment". JACM, vol. 20, pp. 46-61, January 1973.

[11] M. Klein, T. Raylya, B. Pollak, R. Obenza, and M. Harbour. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems.* Norwell, Massachusetts: Kluwer Academic Publishers, 1993.

[12] J. Regehr, and J. Stankovic *HLS: A Framework for Composing Soft Real-Time Schedulers* Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS 2001), 3–14, London, UK, December 2001.

[13] P. Li, B. Ravindran, and T. Hegazy. "Implementation and Evaluation of a Best-Effort Scheduling Algorithm in an Embedded Real-Time System". In *Proc. 2001 IEEE International Symposium on Performance Analysis of Systems and Software* (ISPASS), pp.22-29, Tucson, Arizona, November 4-6, 2001.

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

[15] L. Sha and R. Rajkumar and J. P. Lehoczky *Priority Inheritance Protocols: An Approach to Real-Time Synchronization* In *IEEE Transactions on Computers* Vol. 39, Number 9, Pages 1175–1185, 1990.

[16] *Distributed Real-Time Specification for Java* http://www.drtsj.org

[17] P. Li, B. Ravindran, and H. Wu. *GBS: A Utility Accrual Scheduling Algorithm for Real-Time Activities With Mutual Exclusion Resource Constraints* Submitted to RTSS 2003 http://nile.ece.vt.edu/submissions/RTSS03-lrw.pdf

[18] P. Li, B. Ravindran, J. Wang and, G. Konowicz *Peng Li and Binoy Ravindran and Jinggang Wang and Glenn Konowicz* In *Peng Li and Binoy Ravindran and Jinggang Wang and Glenn Konowicz* May 1990.

[19] TimeSys Inc. http://www.timesys.com

[20] E. Douglas Jensen *Real-Time for the Real World* http://www.real-time.org/deadlines.htm

[21] A. Burns, and A Wellings *Real-Time Systems and Programming languages* London, UK: Addison Wesley, 2001

[22] R. Clark and E. D. Jensen and A. Kanevsky and J. Maurer and P. Wallace and T. Wheeler and Y. Zhang and D. Wells and 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 Springer-Verlag, April 1999

[23] E. D. Jensen, and B. Ravindran *Guest Editor's Introduction to Special Section on Asynchronous Real-Time Distributed Systems* In *IEEE Transactions on Computers, IEEE Computer Society* August, 2002.

[24] QNX Real-Time Operating System http://www.qnx.com

[25] MK7.3a Relase Notes *The Open Group Research Institute* http://www.real-time.org/docs/RelNotes7.Book.pdf

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

[27] D. Maynard, S. Shipman, R. Clark, J. Northcutt, R. Kegley, B. Zimmerman, and P. 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.

[28] W. Beebee, Jr. Region-based memory management for Real-Time Java. MEng thesis, Massachusetts Institute of Technology, September 2001.

[29] W. Beebee, Jr. and M. Rinard. An implementation of scoped memory for Real-Time Java. In *First International Workshop on Embedded Software (EMSOFT)*, October 2001.

[30] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java.* Addison-Wesley, 2000. Latest version available from http://www.rtj.org.

[31] A. Corsaro and D. Schmidt. Evaluating Real-Time Java features and performance for real-time embedded systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, September 2002.

[32] A. Sălcianu, C. Boyapati, W. Beebee, Jr., and M. Rinard. A type system for safe region-based memory management in Real-Time Java. Technical Report TR-869, MIT Laboratory for Computer Science, November 2002.

[33] C. Boyapati, A. Sălcianu, W. Beebee, Jr., and M. Rinard. Ownership types for safe region-based memory management in Real-Time Java. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, June 2003.

[34] Thorsten Kramp. FREE JAZZ: An User-Level Real-Time Threads Package Designed for Flexibility SFB 501 Report, October 1998.