Design and Implementation of Statistical Rate Monotonic Scheduling in KURT Linux*

Alia K. Atlas and Azer Bestavros

Computer Science Department
Boston University
Boston, MA 02215
{*akatlas, best*}*@cs.bu.edu*

Abstract

Statistical Rate Monotonic Scheduling (SRMS) is a generalization of the classical RMS results of Liu and Layland [LL73] for periodic tasks with highly variable execution times and statistical QoS requirements. The main tenet of SRMS is that the variability in task resource requirements could be smoothed through aggregation to yield guaranteed QoS. This aggregation is done over time for a given task and across multiple tasks for a given period of time. Similar to RMS, SRMS has two components: a feasibility test and a scheduling algorithm. SRMS feasibility test ensures that it is possible for a given periodic task set to share a given resource without violating any of the statistical QoS constraints imposed on each task in the set. The SRMS scheduling algorithm consists of two parts: a job admission controller and a scheduler. The SRMS scheduler is a simple, preemptive, fixed-priority scheduler. The SRMS job admission controller manages the QoS delivered to the various tasks through admit/reject and priority assignment decisions. In particular, it ensures the important property of task isolation, whereby tasks do not infringe on each other.

In this paper we present the design and implementation of SRMS within the KURT Linux Operating System [HSPN98, SPH+98, Sri98]. KURT Linux supports conventional tasks as well as real-time tasks. It provides a mechanism for transitioning from normal Linux scheduling to a mixed scheduling of conventional and real-time tasks, and to a focused mode where only real-time tasks are scheduled. We overview the technical issues that we had to overcome in order to integrate SRMS into KURT Linux and present the API we have developed for scheduling periodic real-time tasks using SRMS.

Keywords: real-time computing and communication; scheduling algorithms and analysis; admission control; probabilistic analysis; Quality of Service (QoS) management; operating systems; Linux.

*This work was partially supported by NSF research grant CCR-9706685.

1 Introduction

The increasing use of multimedia, live streams, and real-time data feeds in general desktop systems necessitate that the operating systems of such systems have native support for real-time scheduling. This paper presents the design and implementation of a novel real-time scheduling infrastructure that is particularly suited for both real-time and non-real-time applications on the desktop.

Motivation: Despite the leaping advances in real-time scheduling theory over the last two decades, OS support for real-time applications on the desktop is—to put it mildly—lacking. The main reason behind this lack of support is in the very nature of the real-time applications on the desktop. Unlike conventional real-time systems, (1) a desktop computer may go into overload arbitrarily, due to the unpredictable nature of user requests; (2) real-time tasks must be scheduled along with conventional non-real-time tasks; and (3) the nature of most desktop periodic real-time tasks is different from that of traditional hard real-time tasks considered in the literature.

Traditional scheduling and resource management algorithms devised for periodic real-time task systems have focused on the strict "hard" deadline semantics. Under such semantics, a set of periodic tasks is deemed *schedulable* if every instance of every task in the set is guaranteed to meet its deadline. An optimal fixed-priority algorithm is the classical Rate Monotonic Scheduling (RMS) algorithm of Liu and Layland[LL73]. To ensure the satisfaction of the hard deadlines imposed on periodic tasks, RMS requires that either the periodic resource requirement of each task be constant, or the periodic worst-case resource requirement of each task be known *a priori*. Given such knowledge, RMS guarantees the satisfaction of all deadlines, provided that a simple schedulability condition is satisfied. Using RMS on an unschedulable task system will improve utilization, but will not provide clear predictability of which tasks will miss their deadlines. Indeed, because RMS couples period and priority, tasks with longer periods will miss deadlines more frequently than tasks with shorter periods—the criti-

cality of the tasks is ignored.

There are many real-time periodic applications in which (1) tasks have highly variable utilization requirements and (2) deadlines are firm. For such applications, RMS is too restrictive in assuming a constant resource requirement, and it provides a more stringent guarantee on deadlines than is necessary. In particular, for such applications missing a deadline may be acceptable, as long as (say) a specified percentage of the deadlines are met. This flexibility—coupled with the fact that resource utilization for periodic tasks in such application is typically highly variable—suggests that the worst-case resource requirement need not be planned for.

An important class of such applications is the scheduling of real-time multimedia applications. For such an application, it is obvious that (1) the individual applications may have highly variable execution times , and (2) missing deadlines, while not desirable, is not fatal. Using RMS for scheduling multiple multimedia applications is impractical, as it would result in very poor utilization.

Paper Scope and Outline: This paper presents the design for an implementation of Statistical Rate Monotonic Scheduling (SRMS) [AB98c] in KURT Linux [HSPN98, SPH+98, Sri98]. SRMS allows the scheduling of periodic tasks with highly variable execution times and statistical QoS requirements. It enforces task isolation (a.k.a. the *firewall* property) so that tasks cannot interfere with each other. SRMS wastes no resource bandwidth on jobs that will miss their deadlines due to overload conditions, resulting from excessive variability in execution times. SRMS is cognizant of the value of the various tasks in the system. Thus it ensures that under overload conditions, the deterioration in QoS suffered by the various tasks is inversely proportional to their value. Last but not least, both the SRMS scheduling algorithm and schedulability analysis are computationally efficient.

The remainder of this paper is organized as follows. In section 2, we present previous work related to SRMS and to implementations of real-time or multimedia scheduling algorithms in operating systems. In section 3, we present the details of our SRMS paradigm, including the task model, basic algorithms and schedulability analysis. In section 4, we discuss the design decisions that we faced when implementing SRMS in KURT Linux. In section 5, we present our application programming interface (API) for periodic task scheduling using SRMS on KURT Linux. We conclude in section 6 with a summary of on-going research.

2 Related Work

SRMS relaxes the pivotal assumption of RMS—namely, that the resource requirement of a periodic task is fixed and known *a priori*. Several other relaxations of this assumption have been explored in the literature. Woodbury examined the execution time of real-time tasks in [Woo86]. In [CLL90], Chung, Liu and Lin defined incremental tasks, where the value to the system increases with the amount of time given to the task, until the deadline occurs. In [MC96], Mok and Chen presented the multiframe model, where each task has a sequence of resource requirements which it iterates through.

When a system has variable resource requirements, overload is expected to occur. When a system is in overload, the goal of the scheduling algorithm must be revisited since meeting *all* deadlines becomes impossible. Possible system goals include maximizing the number of deadlines met [BHS94], maximizing the effective processor utilization [BHS94], and completing all critical work [TH97, KS95]. In [MS95], Marucheck and Strosnider provided a taxonomy of scheduling algorithms with varying levels of overload and criticality cognizance. To deal with overload, Koren and Shasha introduced the skip factor in [KS95], where occasionally a job can be skipped. This was expanded to (n m)-hard deadlines by Bernat and Burns in [BB97], where the relaxed deadline requirement allowed increased responsiveness for aperiodic tasks.[1]

Dealing with variable execution requirements introduces an unpredictability akin to that introduced when aperiodic tasks are to be executed along with RMS-scheduled periodic tasks. This latter problem has been examined in a number of studies. Proposed solutions include the polling server [SC95], the deferrable server [Str88], the sporadic server [Spr90], the extended priority exchange algorithm [SLS88], and slack stealing [LRT92, RTL93, RTL94, Thu93]. The latter keeps exact track of the slack available in the system at every priority and reclaims unused execution time.

The work of Tia *et al.* [TDS+95] is most closely related to SRMS in that it considered the problem of scheduling periodic tasks with variable resource requirements and soft deadlines. In their study, Tia *et al.* presented the transform-task method, which uses a threshold value to separate jobs guaranteed under the RMS schedulability condition from those which would require additional work. Jobs that fall under the threshold are guaranteed to meet their deadlines by RMS. The other jobs are split into two parts. The first part is considered as a periodic job with a resource requirement equal to the threshold; the second part is considered to be an aperiodic job and is scheduled via the sporadic server when the periodic part has completed. In [TDS+95], an analysis was given for the probability that the aperiodic job would meet its deadline. However, the aperiodic jobs are served in FIFO order, disregarding any sort of inter-task fairness. Finally, no jobs are ever rejected, because the deadlines are soft and all work must be completed.

Motivated by the work in [TDS+95], we considered a similar approach, Slack Stealing Job Admission Control (SSJAC) [AB98b], where tasks have firm deadlines and

[1] Out of any consecutive m jobs, at least n must meet their deadlines.

slack stealing was used to admit or reject jobs. Associated with each task is a threshold. Jobs with resource requirements below the threshold were automatically admitted. Jobs with resource requirements above that threshold were considered for admittance based upon the slack in the system at their priority level.

SRMS uses a schedulability analysis similar to that of RMS. This makes many of the schedulability results obtained for RMS applicable to SRMS as well. Examples of such results include the less restrictive, though more complex, schedulability test by Lehoczky, Sha and Ding [LSD89] and the improved polynomial-time schedulability test by Han and Tyan [HyT97].

Implementations of Scheduling Algorithms: The actual implementation of a scheduling algorithm requires significant design decisions. A key problem with priority-based algorithms is that there is no temporal protection; a misbehaving task can damage the performance of other tasks [MRZ94]. RMS was implemented in RT-Mach using *procesor capacity reserves*, which reserves, for a given task, a requested amount of CPU time during each of its periods [MST94]. A policing mechanism was implemented, so that tasks couldn't use more high priority time than was guaranteed. In the Rialto OS, a graph-based scheduling algorithm was implemented to provide guaranteed periodic CPU reservations as well as to guarantee aperiodic tasks [JRR97]. The CPU schedule is precalculated; this calculation involves reducing all periods so that each is equal to the product of the minimum period and a power of two.

Other implementation efforts have focused on guaranteeing CPU fairness, as defined by a proportional share discipline. In [YL96], Yau and Lam introduced Adaptive Rate-Controlled Scheduling (RCS) and discussed an implementation in the Solaris UNIX operating system. Essentially, RCS schedules the task with the earliest virtual deadline; when work is done, the virtual deadline is advanced based upon the amount of that work and the task's guaranteed rate. The adaptiveness of the algorithm results from feedback provided by the scheduling algorithm to the task, based upon how much of its reserved rate the task is using in actuality.

A similar approach—also based on a proportional share algorithm, but with more concern for timeliness—was given by Nieh and Lam. In [NL97], they developed SMART to schedule both conventional and real-time applications and implemented it in Solaris. Essentially, SMART uses priorities and Weighted Fair Queueing (WFQ), based upon task shares, to develop a task ordering for importance. The calculation for a task's virtual time is very similar to that used in [YL96]; the virtual time advances at a rate proportional to the amount of processing time the task consumes divided by its share of the processor. Once the importance ordering is established, a working schedule can be constructed. All real-time tasks, which have an importance higher than the most important conventional task, are scheduled according to EDF. If there are no such real-time tasks, then the most important conventional task is scheduled. A real-time task will not be added to the schedule if it will cause a more important task to miss its deadline. SMART also allows graceful shedding of work under overload; real-time tasks will miss occassional deadlines and conventional tasks will take longer. Within the same priority level, tasks are given resources based upon shares, when the system is in overload. However, it is unclear how a given task can be guaranteed a QoS regardless of future task entries. This is the major drawback of proportional share systems; *the value of a share can be changed without warning*.

Real-Time Linux Systems: Linux is an increasingly popular free Unix-clone OS. Due to its popularity and freely available source code, research has been done on making a version of Linux which can support real-time tasks. In [YB, Yod], Yodaiken and Barabanov introduce Real-Time Linux, which supports hard real-time applications. The implementation inserts a small hard real-time kernel into the system, with the normal Linux kernel running as the lowest priority task. Interrupts are caught by the real-time kernel and are only passed to the Linux kernel when the kernel is scheduled to run. This insulates the real-time tasks from timing uncertainties caused by the Linux kernel. However, it also means that the real-time tasks have no access to any Linux kernel services. The real-time tasks can only communicate via a FIFO pipe to a Linux process. In RT-Linux, there is no temporal protection nor can multimedia applications, which require kernel services, be run.

A more general approach has been explored by Hill, Srinivasan, Pather, Ansari, and Niehaus in [HSPN98, SPH+98, Sri98]. Their version of real-time Linux (called KURT Linux) supports firm and soft deadlines. To support real-time tasks, the authors added microsecond resolution to the system, using a timer chip available in the hardware and an event scheduling structure. The kernel can smoothly transition from (1) standard Linux scheduling to (2) scheduling only real-time tasks or to (3) scheduling all tasks. The scheduling of real-time tasks is done via a table of time events. KURT Linux allows real-time tasks to access kernel functions and resources, but it provides no support for scheduling, admission control, or QoS specification/negotiation for periodic task sets. All of these are capabilities we have implemented for KURT Linux as described later in this paper.

3 Statistical Rate Monotonic Scheduling

3.1 SRMS Task Model

The SRMS task model we use in this paper extends the RMS's task model and the semiperiodic task model given by Tia *et al.* [TDS+95]. We start with the following basic definitions.

Definition 1 *A periodic task, τ_i, is a three-tuple, (P_i, $f_i(x)$, Q_i), where P_i is the task's period, $f_i(x)$ is the probability density function (PDF) for the task's periodic resource utilization requirement, and Q_i is the task's requested Quality of Service (QoS).*

Without loss of generality, we assume that tasks are ordered rate monotonically. Task 1, τ_1, is the task with the shortest period, P_1. The task with the longest period is τ_n, where n is the total number of tasks in the system. The shorter the period, the higher the task's priority. [2] At the start of every P_i units of time, a new instance of task τ_i (a *job* of task τ_i) is available and has a firm deadline at the end of that period. Thus, the j^{th} job of task i—denoted by $\tau_{i,j}$—is released and ready at time $(j-1) * P_i$ and its firm deadline is at time $j * P_i$. Its ready time is denoted by $r_{i,j}$ and its deadline is denoted by $d_{i,j}$. We assume that all tasks have a phase of zero.

Definition 2 *The superperiod of τ_i is P_{i+1}, the period of the next lower priority task, τ_{i+1}.*

We assume that the resource requirements for all jobs of a given task are independent and identically distributed (iid) random variables. The distribution is characterized using the probability density function (PDF), $f(x)$. Obviously, it is impossible for a job to require more than 100% of the resource. Thus, $x > P \Rightarrow f(x) = 0$. We assume that the resource requirement for a job is known when the job is released and that such a requirement is accurate. The resource requirement for the j^{th} job of the i^{th} task is denoted by $e_{i,j}$.

The third element of a task specification under the SRMS paradigm is its requested Quality of Service (QoS). For the purpose of this paper, we restrict QoS to the following definition.[3]

Definition 3 *The quality of service $Q_i = QoS(\tau_i)$ for a task τ_i is defined as the probability that in an arbitrarily long execution history, a randomly selected job of τ_i will meet its deadline.*

To enable tasks to meet their requested QoS, SRMS assigns to each task τ_i an allowance, which is replenished periodically (every superperiod) to a preset value a_i. Task allowances are set through the QoS negotiation process (*i.e.* SRMS schedulability analysis). In particular, there is a one-to-one correspondence between the allowance extended to a task and the QoS it achieves. A task set is schedulable under SRMS if the QoS of every task in the task set is satisfied through a feasible assignment of allowances.

[2] It is important to note that the "priority" of a task is not (and should not) be mistaken for the "value" (or importance) of a task. In particular, the manner in which a resource is allotted to various tasks depends on both task priority and value.

[3] Other definitions which allow for closed-form schedulability analysis include (for example) Restricting the execution history to a finite window.

Definition 4 *A set of tasks $\tau_1, \tau_2, ..., \tau_n$ is said to be schedulable under SRMS, if every task τ_i is guaranteed to receive its allowance a_i at the beginning of every one of its superperiods. Thus, a schedulable task set is one in which every task achieves its specified/negotiated QoS.*

3.2 SRMS Overview

The SRMS algorithm consists of two parts: a job admission controller and a scheduler. Like RMS, the SRMS scheduler is a simple, preemptive, fixed-priority scheduler, which assigns the resource to the job with the highest priority that is in need of the resource. The SRMS job admission controller is responsible for maintaining the QoS requirements of the various tasks through admit/reject and priority assignment decisions. In particular, it ensures the important property of *task isolation* (temporal protection), whereby tasks do not infringe upon each other's guaranteed allowances. Job admission control occurs at a job's release time. All admitted jobs are guaranteed to meet their deadlines through a priority assignment that is rate monotonic (similar to RMS). Jobs that are not admitted may be either discarded, or allowed to execute at a priority lower than that of all admitted jobs.[4]

SRMS consists of an analyzable core and several extensions to optimize performance. In the remainder of this section we consider each of these components, starting with the SRMS core, which we henceforth term *Basic SRMS*.

3.3 Basic SRMS with Harmonic Task Sets

One of the main tenets of SRMS is that *the variability in task resource requirements could be smoothed through aggregation*. To simplify the analysis of the gains possible through such aggregation, we start with an examination of Basic SRMS for harmonic task sets. We consider non-harmonic task sets later in subsection 3.4.

Definition 5 *A task set is harmonic if, for any two tasks τ_i and τ_j, $P_i < P_j \Rightarrow P_i | P_j$.*

Basic SRMS is based upon the following task transformation. A task, τ_i, with period, P_i, is transformed into a task with a longer period, P_{i+1}. If the original task was assumed to have a fixed resource requirement, t_i, then the new resource requirement is $t_i * \frac{P_{i+1}}{P_i} = a_i$.

Lemma 1 *If a task system, $((P_1, t_1), ..., (P_i, t_i), (P_{i+1}, t_{i+1}), ..., (P_n, t_n))$, is schedulable according to RMS, then the transformed task system $((P_1, t_1), ..., (P_{i+1}, t_i * \frac{P_{i+1}}{P_i}), (P_{i+1}, t_{i+1}), ...(P_n, t_n))$ is also schedulable.*

This task transformation is based upon a detail of the task ordering defined by RMS. If two tasks have the same period, then the tie is broken arbitrarily, so that either can be

[4] This is an extension briefly discussed in section 3.5.

given the higher priority. Therefore, it is possible to transform task τ_i to have the same period as task τ_{i+1} and still maintain a higher priority. If task τ_i were transformed to have a period longer than P_{i+1}, then either it would have a lower priority than τ_{i+1} and miss deadlines, or it would have a higher priority and could cause τ_{i+1} to miss deadlines. Therefore, the maximum interval over which jobs can be aggregrated is the period of the next lowest priority task.

Obviously, the task transformation above is meaningless for the last task in the system, τ_n. The goal of the task transformation is to aggregate as many jobs as possible without causing lower priority jobs to miss their deadlines. Task τ_n has no lower priority jobs to be concerned about, and can, therefore, have an arbitrarily large superperiod. To visualize this, imagine that there is a task τ_{n+1} with no resource requirement and an arbitrarily large period. For task τ_n, the budget available does not impose a serious barrier to job admission. The important constraint on whether a job $\tau_{n,j}$ is admitted is whether $e_{n,j}$ is less than the time remaining in the period after all higher priority tasks have claimed their allowance.

In SRMS job admission control is used to ensure that: (1) no task is using more of the resource than it has been guaranteed, and (2) no task is admitted if it cannot be guaranteed to meet its deadline. The first of the above two goals prevents higher priority tasks from infringing on the QoS promised to lower priority ones. Recall that in SRMS, the notions of "priority" and "value" (or criticality, importance, etc.) are divorced from each other. Thus a lower priority task may be more valuable to the system than a higher priority one—hence the necessity of ensuring that higher priority tasks do not infringe on lower priority ones. The second of the above two goals maximizes the useful utilization of the resource by disallowing the use of the resource by any job that cannot be guaranteed to finish by its firm deadline.

SRMS job admission control works as follows. At the beginning of each superperiod, a task τ_i has its budget b_i replenished up to its allowance a_i. A job $\tau_{i,j}$ released at time $r_{i,j}$ and requesting $e_{i,j}$ units of resource time is admitted if the following two conditions (corresponding respectively to the two goals explained above) hold: (1) $e_{i,j}$ is less than b_i, and (2) $e_{i,j}$ is less than the time remaining in the period after all higher priority tasks have claimed their allowances. This leads to the following admissibility condition for a job $\tau_{i,j}$:

$$(e_{i,j} \leq b_i) \wedge (e_{i,j} \leq P_i - \sum_{j=1}^{i-1} \frac{a_j * P_i}{P_{j+1}})$$

Schedulability Analysis: In SRMS, each task is assigned an allowance, a_i, which is the amount of time the resource is assigned to that task during its superperiod.[5] For schedulability analysis purposes, the allowance takes the place of the constant resource requirement in RMS. Thus, under SRMS, a necessary and sufficient condition for a *harmonic* task set to be schedulable is that:

$$\sum_{i=1}^{n} \frac{a_i}{P_{i+1}} \quad \leq \quad 1$$

Moreover, according to RMS and Lemma 1, a transformed task is guaranteed to receive at least its allowance every superperiod. To be able to relate the QoS achieved by a given allowance, it is necessary to determine how many jobs available during a superperiod can be completed, given that allowance. Recall that under Basic SRMS, periods are harmonic. For our calculations, we will assume that the probability distribution function is truncated, so that no impossible jobs are submitted to the system.[6]

As illustrated in figure 1, a job $\tau_{i,j}$ can fall into $\frac{P_{i+1}}{P_i}$ different *phases* within the superperiod P_{i+1}. The probability that $\tau_{i,j}$ will be admitted is dependent on the phase in which it falls. To explain this, it suffices to observe that the first job in the superperiod has a replenished budget and has the best chance of making its deadline, while the last job in the superperiod has a smaller chance, because the budget is likely to have been depleted.

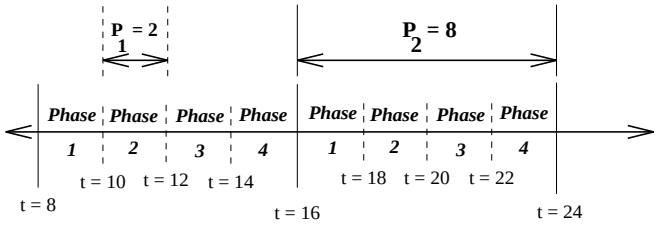

Figure 1: Sample Task with Four Phases

An arbitrary job $\tau_{i,j}$ has an equal probability of being in any given phase out of the possible $\frac{P_{i+1}}{P_i}$ phases within the superperiod P_{i+1}. To explain this, it suffices to note that in an infinite execution of task τ_i, there will be an equal number of jobs in each phase, and thus a uniform distribution for the phase of a randomly selected job is reasonable.

Let $S_{i,k} = 1$ ($S_{i,k} = 0$) denote the event that a job $\tau_{i,j}$ released at the beginning of phase k of a superperiod of task τ_i is admitted (not admitted) to the system. Now, we proceed to compute $P(S_{i,k} = 1)$—the probability of admitting a job in the k^{th} phase of a superperiod of task τ_i (i.e. the probability of success).

[5] The superperiod of the last task, which would be P_{n+1}, is not defined. It can be specified by the user. In practice, we have used $5 * P_n$ successfully. If all tasks in the system are expected to be in overload, then the superperiod of the last task should be shorter.

[6] In practice, if a job with an infeasible resource requirement is submitted, it must automatically be rejected.

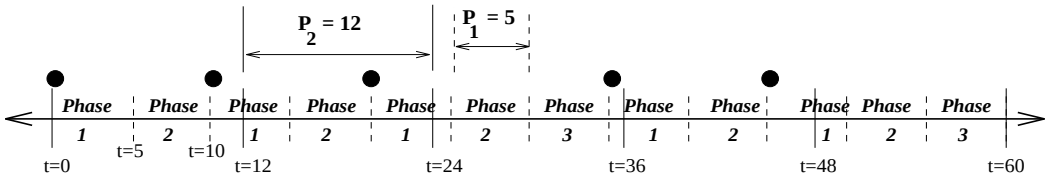

Figure 2: Phases for Task with Overlap Jobs

Recall that a_i is the allowance made available to task τ_i at the start of its superperiod P_{i+1}, which is the start of the first phase. Obviously, a job $\tau_{i,j}$ released in this first phase (i.e. $k = 1$) will be admitted only if its requested utilization is less than or equal to a_i. This leads to the following relationship.

$$P(S_{i,1} = 1) \quad = \quad P(e_{i,j} \leq a_i)$$

For a job $\tau_{i,j}$ released in the second phase (i.e. $k = 2$), two possibilities exist, depending on whether the job released in the first phase was admitted or not admitted. This leads to the following relationship.

$$\begin{aligned} P(S_{i,2} = 1) \quad &= \quad P(e_{i,j-1} \leq a_i) * P(e_{i,j-1} + e_{i,j} \leq a_i) \\ &\quad + P(e_{i,j-1} > a_i) * P(e_{i,j} \leq a_i) \\ \ldots \quad &= \quad \ldots \end{aligned}$$

Obviously, each $P(S_{i,k} = 1)$ can be calculated as the sum of 2^{k-1} different terms, where each term expresses a particular history of previous jobs being admitted and/or rejected (i.e. deadlines met and/or missed). Thus, to calculate $P(S_{i,3} = 1)$, the sum of the probabilities of all possible histories, where the job in the third phase meets its deadline, must be calculated. The set of possible histories are ((1,1,1), (1,0,1), (0,1,1), (0,0,1)), where 1 represents a met deadline and 0 represents a missed deadline.

We are now ready to define the QoS guarantee that SRMS is able to extend to an arbitrary set of tasks with harmonic periods.

Theorem 1 *Given a task set with harmonic periods, the probability that an arbitrary job $\tau_{i,j}$ of task τ_i will be admitted is the QoS function of τ_i.*

$$Q_i = QoS(\tau_i) = \frac{P_i}{P_{i+1}} * \sum_{k=1}^{\frac{P_{i+1}}{P_i}} P(S_{i,k} = 1)$$

Theorem 1 follows from the assumption that an arbitrary job has an equal probability of being in any given phase. The value thus calculated, $QoS(\tau_i)$, is the statistical guarantee which harmonic RMS provides on the probability that an arbitrary job will not miss its deadline.

3.4 Basic SRMS with Arbitrary (non-harmonic) Periods

In the previous section, we assumed that the task set is harmonic. When task periods are harmonic, it is impossible for the release time and deadline of a job to be in different superperiods. When task periods are not harmonic, this situation is possible—a job could overlap two superperiods. To generalize Basic SRMS to schedule task systems with arbitrary periods, we must determine how *overlap jobs* should be treated.

Definition 6 *A job $\tau_{i,j}$ whose release time is in one superperiod and whose deadline is in the next superperiod is called an* overlap job.

First, we explain the subtlety involved in dealing with overlap jobs. The primary purpose of job admission control in Basic SRMS is to prevent variability in resource utilization by a high priority task from disturbing other lower priority tasks. This is done by ensuring that the high priority task does not consume more than its allocated budget within each of its superperiods. Now consider the advent of an overlap job. By definition, an overlap job is one that is released in one superperiod (the release superperiod) and whose deadline is in the next (the deadline superperiod). Figure 2 shows a task which has overlap jobs. The difficulty in making admission decisions for overlap jobs is due to the simple fact that any resource use charged to a given budget *must* be completed within the superperiod of that budget. The fact that overlap jobs span two superperiods complicates that process. There are three possibilities for admitting an overlap job, which we consider below.

If the overlap job is to be admitted based on the available budget in the release superperiod, then (in order not to disturb lower priority tasks) the overlap job must complete its execution before the end of the release superperiod. This may or may not be possible. If possible, the overlap job is admitted and the budget of the release superperiod is debited.

If the overlap job is to be admitted based on the available budget in the deadline superperiod, then (in order not to disturb lower priority tasks) the execution of the overlap job must be delayed until the beginning of the deadline superperiod, or at least until the job of the next lower priority task has finished its execution and thus is not subject to being infringed upon by the overlap job. Again, this may or may not be possible. If possible, the overlap job is admitted, but not permitted to run until after some delay, and the budget of the deadline superperiod is debited.

Finally, for the purpose of admission control and debiting the appropriate budgets, it would be possible to com-

bine the above two possibilities by splitting the overlap job into two components. The first would be admitted at release time and allowed to execute against the budget available in the release superperiod. The second would be delayed until the beginning of the deadline superperiod and allowed to execute against the budget available in that deadline superperiod. Again, this may or may not be possible. If possible, the overlap job would be admitted, otherwise it would be rejected. We did not implement this in SRMS due to the additional complexity required in the scheduler.[7]

Schedulability Analysis: The evaluation of the feasibility of achieving the requested QoS for a SRMS task system with arbitrary periods is an elaboration of the schedulability analysis for a harmonic task system presented in subsection 3.3. The additional complexity is caused by an analysis of the behavior for overlap jobs. Due to space limitations, we do not include this analysis here. Interested readers are referred to the derivations and formulae in [AB98a].

3.5 Extensions to Basic SRMS

There are a number of extensions to optimize the performance of the Basic SRMS algorithm described above. These extensions include the use of *time inheritance* and *second chance priorities*. In the remainder of this paper, we use "SRMS" to refer to the Basic SRMS algorithm augmented with these two extensions. The following is a brief overview of these extensions. Interested readers are refered to [AB98c] for more details.

Time Inheritance: Time inheritance is another instance of the SRMS concept of "smoothing the variability in resource usage through aggregation". In Basic SRMS, this aggregation was done over time for a single task (see Lemma 1). Using the time inheritance extension of SRMS, this aggregation is done across tasks, whereby the unused budget of a higher priority task is percolated down to lower priority tasks.

Second Chance Priorities: It is possible that a job may fail its admission test, but *still* be able to complete on time *without* jeopardizing the task isolation property of SRMS. This is so because the admission controller operates under the pessimistic assumption that other tasks in the system will use their maximum allowances. If this is not the case, then "idle times" may be available to complete the job *despite* the job's failure to pass the admission test. Rather than simply discarding rejected jobs, the second chance priority extension assigns a lower "second chance" priority to the jobs, which are then allowed to execute.

[7] We are permitting this in the implementation in KURT Linux, because the policing mechanisms necessary are already incorporated to enforce task isolation.

4 Implementation Decisions

In this section we discuss the features of KURT Linux which were instrumental for the purposes of our work. Also, we discuss a number of challenges that we had to overcome throughout the design and implementation process. In the following section, we present our SRMS scheduling and QoS management API.

Features of KURT Linux: KURT Linux is designed for non-hard-deadline (*i.e.* soft or firm deadline) real-time tasks, which may require use of kernel functions. KURT Linux provides microsecond time resolution for event scheduling. It has an API for transitioning into and out of real-time mode. To support real-time tasks, it requires that real-time tasks register and that periodic tasks undergo an admission test. The extant of real-time scheduling in KURT Linux is table-based. A file with a list of events is supplied and used for scheduling; each event consists of the time it should occur and the function which should be called at that time.

Interrupt Handling in KURT Linux: KURT Linux presents some challenges to an implementation of any real-time scheduling algorithm. While KURT Linux reduces the work done in an interrupt,[8] it does not isolate tasks from the timing uncertainties caused by such an interrupt. This is acceptable, since it is targetted to support soft/firm real-time tasks. In KURT Linux, interrupts are not delayed; they can occur at any point.

When an interrupt occurs, most interrupt service routines set a flag, indicating that work needs to be done; the system must schedule it. This remaining work is known as the *bottom half* of the ISR. In normal Linux, the *bottom halves* are completed every time the scheduler is run. Clearly, intelligent scheduling of the *bottom halves* of interrupts is necessary to minimize priority inversion. This presented a serious implementation challenge that we had to address, as described later in this section.

4.1 Assigning Overhead Costs to Tasks

SRMS (as described in the previous sections) does not consider any operating system or scheduling overheads. Operating system overheads are due primarily to the management of interrupts. As discussed above, interrupts consist of two parts—the ISR and the *bottom half*. The overheads for each one of these two parts must be treated in a different manner due to the asynchronous nature of ISR overheads versus the synchronous nature of bottom-half overheads. Scheduling overheads are due to the need of SRMS to determine which task should be scheduled next and to swap that task into the CPU.

[8] Interrupt overhead averages 7 μseconds.

Accounting for Scheduling Overhead: First, we considered the scheduling overhead. We assumed that a task cannot voluntarily suspend execution.[9] Each job preempts the CPU exactly once and voluntarily releases it once.

This observation provides a convenient method to upper bound the scheduling overhead of each task. When a task τ_i preempts a lower priority task, which it does once, the task τ_i is charged with the scheduling overhead. Similarly, when the task τ_i releases the CPU voluntarily to a lower priority task, task τ_i is charged with the scheduling overhead. Thus, the time it takes to run the scheduler and to swap processes is always charged to the higher priority process. If a task needs to suspend (waiting for an interrupt) the extra preemption overheads must be considered for calculation of the job's resource requirement.

Accounting for Interrupt Overheads: Certain types of real-time tasks may require that interrupts be used[10], whether it be for disk I/O or network traffic. While it is possible to mask off interrupts, it is not desirable for long periods, since meaningful interrupts may be missed. Therefore, a task will suffer overhead from interrupts. To give some perspective on the size of this overhead, the overhead of having an event timer go off and call the correct process takes over 50 μseconds, while an interrupt takes an average of 7 μseconds. The OS overhead due to interrupts can only be estimated as a function of a given job's expected execution time and the system of tasks' usage of interrupt-driven kernel services.

Dealing with Priority Inversion Due to Scheduling Interrupt Bottom Halves: When an ISR is run, it may set a flag indicating the kernel should complete some specific work, known as the interrupt's *bottom half*. Ideally, the *bottom half* of each interrupt would be run by the task which required the services supplied by that interrupt. We assume that each interrupt will wake up a given task. When the scheduler determines if a task is ready to be scheduled, it can also check if the task is waiting on an interrupt whose ISR has been run, but whose *bottom half* has not been scheduled. If so, then the scheduler could run the appropriate *bottom half*, which would wake up the task. That task would then be charged with the overhead of running the *bottom half*.

Even with this ideal situation, the problems of *priority inversion* would not be eliminated. If a higher priority task's interrupt occurs immediately after the scheduler has swapped in a lower priority task, then the higher priority task must wait for the next scheduling event. One could modify all ISRs such that the scheduler is called if the ISR is associated with a higher priority task. However, this would require modifications to all possible drivers, and would still not eliminate all priority inversion. Priority inversion is inevitable, because the higher priority task is not scheduled while it is awaiting the interrupt.

The above situation would be ideal in that each task would be charged for the execution of the interrupt *bottom halves* which it required. Unfortunately, there is no support in the kernel to permit associating interrupts with the tasks which are waiting upon them. Instead, we chose a compromise design as follows.

Normally, the scheduler selects the highest priority task with work to do. To do this, the scheduler checks each task sequentially, from highest priority to lowest. In this check, if the scheduler finds a task which is waiting on an interrupt before it finds a task with work to do, then the scheduler runs the *bottom halves* of the interrupts. If the task which was waiting is awaken, then the scheduler has its selection; otherwise it proceeds. The *bottom halves* are run at most once every scheduling event.

This solution bounds the potential priority inversion to be the length of the shortest period in the system. The time to run the *bottom halves* is considered to be part of the scheduling overhead and is charged to the higher priority task of those swapped out and in. The waiting high priority task will frequently be charged the cost of running the *bottom halves*.

4.2 Task Management and Control

Enforcing Resource Requirements through Policing: An assumption of SRMS is that the scheduler has knowledge of a job's resource requirement as soon as it is released. In an actual operating system, this assumption is usually false and potentially dangerous. Therefore, rather than subtracting the job's execution time from its task's budget when the job is released and admitted, the actual execution time is subtracted from the budget once it is spent. An accurate calculation of a job's execution time will require execution time to complete, and therefore this knowledge will not be available when the job is released. The overhead to calculate the execution time, if known, can be taken into account in calculating the quality of service for the task. Malicious tasks may also lie about a job's expected execution time, in an attempt to acquire more CPU time. To protect against malicious tasks and tasks which cannot accurately compute their execution times, a policing mechanism is necessary.

The policing mechanism we employ consists of setting a scheduler event to occur immediately before the task can spend more time than is available in its budget. The delay from when the task is scheduled to this event is the task's budget minus the one scheduling overhead for releasing the CPU to a lower priority task.

[9] Not permitting a task to voluntarily suspend is a common requirement in real-time scheduling; at the ready-time, the entire job must be ready.

[10] Every 10 ms a heartbeat event interrupt is scheduled to maintain any kernel services which depend on that timer.

```
/* returns the period of the server for conventional tasks */
unsigned long get_server_period(void);

/* returns the utilization of the server for conventional tasks */
float get_server_util(void);

/* returns old period if successful and -1 otherwise */
long set_server_period(unsigned long new_period);

/* returns old utilization if successful and -1 otherwise */
float set_server_util(float new_util);
```

Figure 3: Calls to access and modify conventional task server

Scheduler Events: The job of the scheduler is to swap in the chosen task and to set an interrupt for the next time at which the scheduler should be run. It is only necessary to specify the time of the next scheduling event. The time of the next event can be determined simultaneously when deciding which task should be given the processor. This decision is quite simple, namely the highest priority task with work to do is scheduled and the next scheduling event should occur at the earliest release time of any equal- or higher-priority task.

That calculation for the scheduling event does not consider the policing required. The policing mechanism determines the time of the budget-constrained scheduling event, as described above. The earlier of the budget-constrained scheduling event and the release time scheduling event is selected and set to trigger the scheduler at that time, using KURT Linux's built-in event timing mechanism.

Recovery from Missed Deadlines: Some jobs will not be allowed to run to completion under SRMS. This may happen either because the job was rejected or because the job attempted to use more time than was available in its remaining budget. In both cases, the process must be cleaned up so it is ready to run its next job. This clean-up could occur either at the end of the missed job or at the beginning of the next accepted job. If the clean-up occurs at the end of the missed job, then the execution time of the clean-up routine must be known. Moreover, a job may fail due to the need to guarantee the overhead to clean it up, if it fails. Therefore, we choose to consider the time required to clean-up a previously failed job as part of the execution time of the next job.

For a newly-released job, when it is scheduled for the first time, if the previous job failed, the scheduler will send the process a signal. The process will catch this signal and clean up to prepare to run the next job. When the signal handler exits, the process is manipulated so that the signal handler exits to the beginning of its periodic loop, where it will start executing the next job.

5 SRMS API

KURT Linux provides three different modes—normal Linux, focused real-time, and mixed conventional and real-time. In this API, we assume that the mixed scheduling mode is in effect. All real-time tasks are assumed to be periodic;[11] they may appear (*i.e.* be released) in the system at any time and they may be removed (*i.e.* be terminated) from the system at any time.

To ensure that conventional tasks are not starved, a periodic server is created to service conventional tasks. By default, the server's period is set to five times the maximum real-time task period. The utilization of the server is adjustable. The functions shown in Figure 3 allow access and modification of the period and utilization for the conventional task server.

5.1 Life Cycle of a Real-Time Task

A real-time task has three basic stages. First, it must register as a real-time task and request admission with a given minimum QoS. Once admitted, the task must execute periodically, as expected. Finally, the task must unregister when it has completed execution. The timeline of a task's existence is illustrated in Figure 4.

Registration: The registration of a task as a real-time task is a straightforward extension of what is supplied by KURT Linux. As seen in Figure 6, the `rtparams`

[11] Dealing with aperiodic real-time tasks is possible by modeling the aperiodic task as a periodic one and terminating it at the end of its first period.

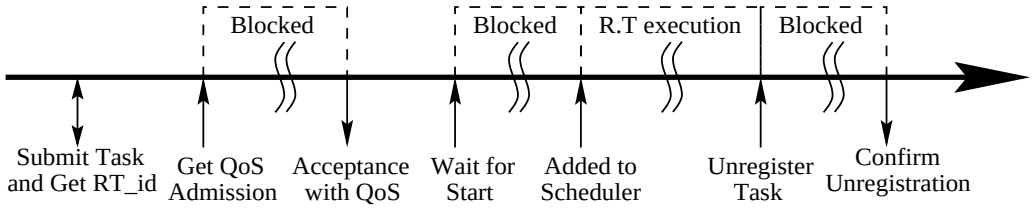

Figure 4: Timeline of a Periodic Task

```
/* Returns granted QoS, which is 0 if minimum_QoS can't be guaranteed */
float request_QoS_admission( /* float */ desired_QoS, /* float */ minimum_QoS);

/* Returns granted QoS.  Blocks until at least minimum_QoS is guaranteed. */
float await_QoS_admission( /* int */ RT_id, /* float */ desired_QoS,
                           /* float */ minimum_QoS);
```

Figure 5: Calls for Admission with QoS

structure is increased to include the task's importance, a pointer to an array of sample execution times, and the number of samples in that array. Once a task has called `set_rtparams`, its information is stored with the kernel.

Admission with QoS: No resources are given to a task until it has been admitted to the system. To request admission, there is a choice of a blocking call and a non-blocking call, as shown in Figure 5. An example using the non-blocking call is shown in Figure 6. The non-blocking call `request_QoS_admission` checks if the task can gain admittance with (at least) the specified minimum QoS. If this cannot be immediately guaranteed, then no resources are allocated to the task, and a QoS of 0 is returned. If a QoS between the minimum and the requested QoS can be guaranteed, then the allowance is set so as to allocate the appropriate resources to the task and the promised QoS is returned. The blocking function `await_QoS_admission` does not return until at least the minimum QoS has been guaranteed. If this is not possible when the function is first called, then the task is blocked. Whenever an admitted periodic task leaves the system (or decreases its QoS), an effort is made to admit such blocked tasks in order of importance. The `await_QoS_admission` function depends on the assumption that some tasks will eventually complete and unregister themselves.

Periodic Execution: Once a task has registered and been admitted, it is ready to be scheduled. To support periodic execution of a task, we designed a function call `await_scheduling` which blocks the task until a new job of that task is released and available for scheduling. The number of the newly released job is returned. This function is used before any jobs are released and between the completion of one job and the release time of the next.

In addition to the proper use of `await_scheduling`, a task must either catch an `RT_JOB_FAILED` signal or use the `void ignore_jobfail_signal(/* int */ TRUE)` function to report to the scheduler that job failures should be ignored and not reported. This option is useful for a task with a *soft deadline*, which needs to complete the work of a job even after its deadline.[12] The signal handler should clean up any remnants of the failed job and restore a pristine state, as expected by a newly released job of that task.

There are three different possible task models which we have designed APIs for. The first API is to support tasks which have no method of determining what the execution time of a job will be. A sample loop for the periodic execution is given in Figure 7. The second API is for a task which has accurate knowledge of its jobs' execution times and of the time it will take to compute those execution times. An example of this *default* API is shown in Figure 6. The third API is for design-to-time tasks; such tasks can select which procedure to use depending upon the time available for the execution [CLL90, Bin97]. An example task is shown in Figure 8.

Unregistration: Once a task has completed its execution, the task must notify the system. To do so, `void unregisterRT(int myRT_id)` is used. It recalcu-

[12] The `ignore_jobfail_signal` is also used by the kernel during the `unregisterRT` function.

```
void failedJob_handler(int jobnum)
  {
   /* Clean up from failed job and prepare for new one
    * On return from this signal handler, process will wake up
    * after await_scheduling() call.
    */
  }

int main(int argc, char * argv[])
{
  int num_samples = NUM_SAMPLE_EXEC_TIMES;
  unsigned long sample_execs[NUM_SAMPLE_EXEC_TIMES];
  struct rtparams myRTparams;
  int myRT_id;
  float myQoS;

  /* Fill in sample_execs from a file or memory. */

  myRTparams = {
    /* RT id to refer to this process */ ASSIGN_RT_ID,
    /* RT priority, assigned rate-monotonically */ ASSIGN_RT_PRIORITY,
    /* importance of task (1 - 99) */ 1,
    /* array of sample execution times */ sample_execs,
    /* length of sample exec array */  num_samples,
    /* period in microseconds */ 33000
  };

  /* Now, register with the kernel as a real-time process */
  myRT_id = set_rtparams( /* pid, 0 if current process */ 0,
                          /* process type */ SCHED_KURT,
                          /* RT parameter info */ &myRTparams);
  signal(RT_JOB_FAILED, failedJob_handler);
  myQoS = await_QoS_admission(myRT_id, 80.0, 50.0);

  while (haveWork) {
    jobnum = await_scheduling();
    /* The budget returned has scheduling and OS overheads subtracted
     * for this job of the task.
     */
    budget = get_RTbudget();
    if (budget > CALCULATE_TIME)  /* time to calculate execution time */
      execTime = myCalculateExec(jobnum);
    /* Allow job admission control to set priority of job properly.
     * admit_RTjob returns 1 if the job is admitted and 0 otherwise.
     */
    admitted = admit_RTjob(execTime);
    /* If job was rejected, it runs at low priority */
    haveWork = do_work();
  }
  unregisterRT(myRT_id);
  exit(0);
}
```

Figure 6: Example real-time user process

```
while (haveWork) {
   jobnum = await_scheduling();
   haveWork = do_work();
}
```

Figure 7: Periodic loop for a task ignorant of its execution time

```
while (haveWork) {
  jobnum = await_scheduling();
  /* The budget returned has scheduling and OS overheads subtracted
   * for this job of the task.
   */
  budget = get_RTbudget();
  /* myPickAlgorithm selects which algorithm should be used and
   * returns its required execution time.  PICK_TIME is the time
   * needed to run the myPickAlgorithm function.
   */
  execTime = myPickAlgorithm(budget - PICK_TIME, jobnum, &alg);
  /* Allow job admission control to set priority of job properly */
  admitted = admit_RTjob(execTime);
  do_work(alg); /* If job was rejected, the fastest algorithm was
                 * picked, so try at the low priority. */
}
```

Figure 8: Periodic loop for a design-to-time task

lates the allowance of the next higher priority task and waits until it is safe to have that allowance changed. Then it changes the allowance of that next higher priority task and removes the time allocation of the task which is unregistering. Finally, it removes all information about the task. Once `unregisterRT` returns, the task is no longer considered a real-time task and is free to exit or continue executing, as its application demands. A simple example process, illustrating the registrating, QoS admission, signal handling, periodic execution, and unregistering is shown in Figure 6.

6 Conclusion

In this paper, we have presented Statistical Rate Monotonic Scheduling—an algorithm that schedules firm-deadline periodic tasks with variable resource requirements—and discussed its implementation in KURT Linux. SRMS is predictable, configurable, overload-cognizant, value-cognizant, and enforces task isolation. This temporal protection provided by SRMS is vital and the implementation we have discussed preserves this protection regardless of malicious tasks. SRMS is overload-cognizant on an individual task basis; the responses caused by overload only affect the misbehaving task.

SRMS is ideal for an operating system which desires to support firm-deadline periodic tasks. Because SRMS anticipates variable resource requirements, it can tolerate variations caused by interrupts. SRMS also provides adjustable quality, so that a user can specify the necessary QoS of different tasks.

Our current work focuses on evaluating SRMS in KURT Linux and on improving its usability. For instance, currently a new task can only be entered into the schedule at very limited points.[13] We are examining a framework where the delay in scheduling new tasks is much less.

[13] These points are the beginning of a new hyperperiod, the least common multiple of all periods in the system. Thus, only when all tasks in the system are starting at the same time can a new task be entered.

References

[AB98a] Alia K. Atlas and Azer Bestavros. Maintaining quality of service for multimedia systems using statistical rate monotonic scheduling. Technical Report BUCS-TR-98-011, Boston University, Computer Science Department, 1998.

[AB98b] Alia K. Atlas and Azer Bestavros. Slack stealing job admission control. Technical Report BUCS-TR-98-009, Boston University, Computer Science Department, 1998.

[AB98c] Alia K. Atlas and Azer Bestavros. Statistical rate monotonic scheduling. Technical Report BUCS-TR-98-010, Boston University, Computer Science Department, May 1998. Accepted for publication in IEEE Real-Time Systems Symposium, December 1998, Madrid, Spain.

[BB97] Guillem Bernat and Alan Burns. Combining (n m)-hard deadlines and dual priority scheduling. In *Real-Time Systems Symposium*, pages 46–57, 1997.

[BHS94] Sanjoy Baruah, Jayant Haritsa, and Nitin Sharma. On line scheduling to maximize task completions. In *Real-Time Systems Symposium*, pages 228–237, December 1994. URL is http://www.emba.uvm.edu/ sanjoy/Papers/cc-jnl.ps.

[Bin97] Pam Binns. Incremental rate monotonic scheduling for improved control system performance. In *Real-Time Technology and Applications Symposium*, June 1997.

[CLL90] Jen-Yao Chung, Jane W. S. Liu, and Kwei-Jay Lin. Scheduling periodic jobs that allow imprecise results. *IEEE Transactions on Computers*, 39(9):1156–1174, September 1990.

[HSPN98] Robert Hill, Balaji Srinivasan, Shyamalan Pather, and Douglas Niehaus. Temporal resolution and real-time extensions to linux. Technical Report ITTC-FY98-TR-11510-03, Information and Telecommunication Technology Center, Department of Electrical Engineering and Computer Sciences, University of Kansas, June 1998.

[HyT97] Ching-Chih Han and Hung ying Tyan. A better polynomial-time schedulability test for real-time fixed-priority scheduling algorithms. In *Real-Time Systems Symposium*, pages 36–45, 1997.

[JRR97] Michael B. Jones, Daniela Rosu, and Marcel-Catalin Rosu. Cpu reservations and time constraints: Efficient, predictable scheduling of independent activities. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, October 1997.

[KS95] Gilad Koren and Dennis Shasha. Skip-over: Algorithms and complexity for overloaded systems that allow skips. In *Real-Time Systems Symposium*, 1995.

[LL73] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1), 1973.

[LRT92] John P. Lehoczky and Sandra Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In *Real-Time Systems Symposium*. IEEE Computer Society Press, December 1992.

[LSD89] John Lehoczky, Lui Sha, and Ye Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Real-Time Systems Symposium*, pages 166–171, 1989.

[MC96] Aloysius K. Mok and Deji Chen. A multiframe model for real-time tasks. In *Real-Time Systems Symposium*. IEEE Computer Society Press, December 1996.

[MRZ94] C. W. Mercer, R. Rajkumar, and J. Zelenka. Temporal Protection in Real-Time Operating Systems. In *Proceedings of the 11th IEEE Workshop on Real-Time Operating Systems and Software*, pages 79–83, May 1994.

[MS95] M. Marucheck and J.K. Strosnider. An evaluation of the graceful degradation properties of real-time schedulers. In *The Twenty Fifth Annual International Symposium on Fault-Tolerant Computing*, June 1995.

[MST94] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves for Multimedia Operating Systems. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[NL97] Jason Nieh and Monica S. Lam. The design, implementation and evaluation of smart: A scheduler for multimedia applications. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, October 1997.

[RTL93] Sandra Ramos-Thuel and John P. Lehoczky. On-line scheduling of hard deadline aperiodic tasks in fixed-priority systems. In *Real-Time Systems Symposium*. IEEE Computer Society Press, December 1993.

[RTL94] Sandra Ramos-Thuel and John P. Lehoczky. Algorithms for scheduling hard aperiodic tasks in fixed-priority systems using slack stealing. In *Real-Time Systems Symposium*. IEEE Computer Society Press, December 1994.

[SC95] Kang G. Shin and Yi-Chieh Chang. A reservation-based algorithm for scheduling both periodic and aperiodic real-time tasks. *IEEE Transactions on Computers*, 44:1405–1419, December 1995.

[SLS88] Brinkly Sprunt, John Lehoczky, and Lui Sha. Exploiting unused periodic time for aperiodic service using the extended priority exchange algorithm. In *Real-Time Systems Symposium*, 1988.

[SPH+98] Balaji Srinivasan, Shyamalan Pather, Robert Hill, Furquan Ansari, and Douglas Niehaus. A firm real-time system implementation using commercial off-the-shelf hardware and free software. In *Real-time Technology and Applications Symposium*, pages 112–119, June 1998.

[Spr90] B. Sprunt. *Aperiodic task scheduling for real-time systems*. PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, August 1990.

[Sri98] Balaji Srinivasan. A firm real-time system implementation using commercial off-the-shelf hardware and free software. Master's thesis, Department of Electrical Engineering and Computer Science, University of Kansas, June 1998.

[Str88] J. K. Strosnider. *Highly responsive real-time token rings*. PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, August 1988.

[TDS+95] T.S. Tia, Z. Deng, M. Shankar, M. Storch, J. Sun, L.-C. Wu, and J. W.-S. Liu. Probabilistic performance guarantees for real-time tasks with varying computation times. In *Real-Time Technology and Applications Symposium*, pages 164–173, May 1995.

[TH97] Teik Guan Tan and Wynne Hsu. Scheduling multimedia applications under overload and non-deterministic conditions. In *Real-Time Technology and Applications Symposium*, June 1997.

[Thu93] Sandra Ramos Thuel. *Enhancing Fault Tolerance of Real-Time Systems through Time Redundancy*. PhD thesis, Carnegie Mellon University, May 1993.

[Woo86] M. Woodbury. Analysis of the execution time of real-time tasks. In *Real-Time Systems Symposium*, pages 89–96, 1986.

[YB] Victor Yodaiken and Michael Barabanov. A real-time linux. Online at http://rtlinux.cs.nmt.edu/ rtlinx/u.pdf.

[YL96] David K. Y. Yau and Simon S. Lam. Adaptive rate-controlled scheduling for multimedia applications. In *ACM Multimedia*, 1996.

[Yod] Victor Yodaiken. The rt-linux approach to hard real-time. Online at http://rtlinux.cs.nmt.edu/ rtlinux/whitepaper/short.html.