Statistical Rate Monotonic Scheduling*

Alia Atlas†
Dept of Internetwork Research
BBN Technologies
Cambridge, MA 02138

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 [10] 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.

1. Introduction

Traditional scheduling and resource management algorithms devised for periodic real-time task systems have focused on strict "hard" deadline semantics, whereby 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[10]. 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 criticality 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 variability in resource utilization—suggests that the worst-case resource requirement need not be planned for. An important class of such applications is the multiplexing of real-time multimedia streams on a shared fixed-bandwidth channel. For such an application, it is obvious that (1) the individual streams may have highly variable bandwidth requirements, and (2) missing deadlines, while not desirable, is not fatal. Using RMS for arbitrating a shared communication channel amongst the various streams is impractical, as it would result in very poor utilization.

This paper presents Statistical Rate Monotonic Scheduling (SRMS), a generalization of RMS that allows the scheduling of periodic tasks with highly variable execution times and statistical QoS requirements. SRMS maximizes the utilization of the resources being managed. In particular, it 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.

2. Related Work

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, ex-

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

†Research completed while co-author was at Boston University.

act schedulability test by Lehoczky, Sha and Ding [9] and the less accurate but faster polynomial-time schedulability test by Han and Tyan [7].

SRMS relaxes the pivotal assumption of RMS—namely that the resource requirement of a periodic task is fixed. Several other relaxations of this assumption have been explored in the literature. The execution time of real-time tasks has been examined and modeled [21, 5]. In [6], 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 [12], Mok and Chen presented the multiframe model, where each task has a sequence of resource requirements which it iterates through. In [18], Tan and Hsu used feedback to control the resource requirements of tasks and admission control to prevent overload.

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 [3], maximizing the effective processor utilization [3], and completing all critical work [18, 8]. In [11], 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 [8], where occasionally a job can be skipped. This was expanded to (n m)-hard deadlines by Bernat and Burns in [4], 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 [14], the deferrable server [17], the sporadic server [15], the extended priority exchange algorithm [16], and slack stealing [13]. 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.* [20] 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 a sporadic job and is scheduled via the sporadic server when the periodic part has completed. In [20], an analysis was given for the probability that the sporadic job would meet its deadline. However, the sporadic jobs are served in FIFO order, disregarding any sort of intertask fairness. Finally, no jobs are ever rejected, because the deadlines are soft and all work must be completed.

Motivated by the work in [20], we considered a similar approach, Slack Stealing Job Admission Control (SSJAC) [2], where tasks have firm deadlines and 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. SSJAC is discussed in more detail in section 5.2 as we pit it against SRMS for performance comparison purposes.

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.* [20]. 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 $d_{i,j}$.[3]

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, $x > P \rightsquigarrow f(x) = 0$. We assume that the resource requirement for a job is known when the job is released and that such a requirement is accurate.[4] The resource requirement for the j^{th} job of the i^{th} task is denoted by $e_{i,j}$.

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

[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] While this model does not include non-zero task phases, the small modification necessary to support them is described in subsection 3.3.

[4] If this assumption cannot be ensured, then a policing mechanism could be employed, whereby when a task is given the resource, an interrupt is set so that the task is interrupted at the end of its "requested" time to ensure that it does not use more than what it had requested upon its release.

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.[5]

Definition 3 *The quality of service $QoS(\tau_i)$ 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, as we will show later in this section, 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.

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. Overview of SRMS Scheduling Algorithm

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*, 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.[6]

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 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.

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

[6] This is discussed in more details in section 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 depends upon the ability to arbitrarily break ties between two tasks with the same period, so that either can be 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 aggregated is the period of the next lowest priority task.

In the SRMS model presented, task phases are not considered, but little modification is necessary to support them. The task transformation is valid with non-zero task phases. The sole difference is that with phases, the start of a task's superperiod does not align with the release of a job of the next lower priority task. That is the start of the first superperiod for task τ_i occurs at $r_{i,1}$, not at $r_{i+1,1}$.

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 over 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.

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. The second goal 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. When a job $\tau_{i,j}$ is admitted to the system, its resource requirement $e_{i,j}$ is debited from the task's current budget. 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 $\tau_{i,j}$:

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

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.[7] 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 and thus no *overlap* jobs exist. For our calculations, we will assume that the probability distribution function is truncated, so that no impossible jobs are submitted to the system.[8]

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.

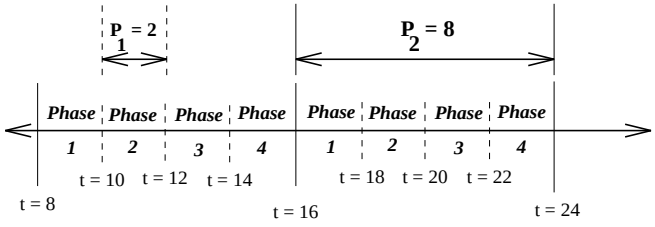

Figure 1. Sample Task with Four Phases

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).

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.*

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

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 Periods

Previously, 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

[7] 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.

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

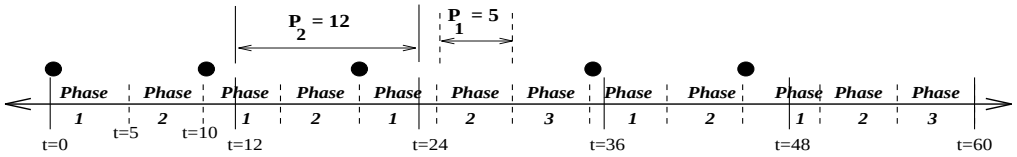

Figure 2. Phases for Task with Overlap Jobs

job could overlap two superperiods.[9] 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 the 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 superperiod. 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 combine 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.

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 [1].

4. Extensions to Basic SRMS

In this section we examine a number of extensions that optimize the performance of the Basic SRMS algorithms presented in the previous section. For the remainder of this paper, we use SRMS to refer to the Basic SRMS algorithm (whether or not the task set is harmonic) when augmented with all of the extensions presented in this section.

Time Inheritance: At the start of each superperiod, a task's budget is replenished. However, that task (say τ_i) may have time leftover in the budget of its previous superperiod. In Basic SRMS, this unused budget is simply discarded. But, does it have to be? To answer this question, we first note that such leftover time can only be spent by a task with priority lower than that of τ_i. Task τ_i can't use the leftover allowance because such use may adversely affect τ_{i+1}. In particular, using this leftover time by τ_i may result in tasks of priority i and higher getting more than their fair share (i.e. reserved percentage) of the resource during the superperiod P_{i+1}. However, if τ_{i+1} is not also ending a superperiod, then τ_{i+1} *can* spend this leftover time. Such use won't affect τ_{i+2} because it will not result in exceeding the percentage of the resource reserved for tasks with priority higher than that of τ_{i+2}.[10]

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. Figure 3 shows an example where time inheritance occurs twice.

[9] Again, if task τ_i has a non-zero phase, the start of the first superperiod is aligned with the release time of the first job $\tau_{i,1}$, not with the release of job $\tau_{i+1,1}$.

[10] Clearly, the last task in the system merely discards any unused allowance when it replenishes its budget.

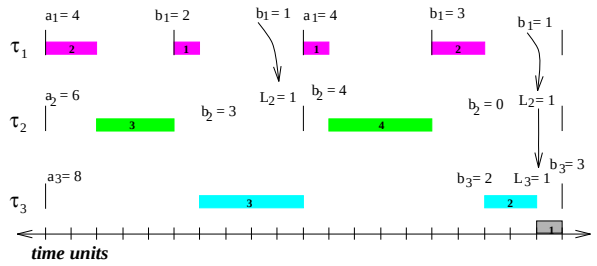

Figure 3. Illustration of Time Inheritance

Second Chance Priorities: In Basic SRMS, it is possible to reject a job (1) because its budget is depleted, or (2) because the admission controller cannot guarantee that such job (if scheduled) will have enough time (leftover from higher priority tasks) to meet its deadline. The above two conditions—while sufficient to satisfy the task isolation and efficient resource utilization properties of SRMS—may be unnecessarily stringent. Namely, it is possible that a job may be rejected and still be allowed to use the resource *without* jeopardizing the task isolation and efficient resource utilization properties of SRMS. The admission controller pessimistically assumes that other tasks in the system will use their maximum allowances. If the other tasks do not, then "idle times" may be available to complete the job *despite* the job's failure to satisfy one (or even both) of the above conditions. Thus, rather than simply discarding rejected jobs, it would may be advantageous to give those jobs a *second chance*. This is the motivation for the following extension.

Each task has two possible priorities, either HIGH (and admitted) or LOW (and rejected). If a job is admitted, then the priority is set to HIGH and the allowance is debited. Otherwise, the priority is set to LOW and the allowance is unchanged. This splits the tasks into two RM-ordered priority bands. First, the HIGH priority tasks are scheduled; then, if there is time, the LOW priority tasks are scheduled. This gives guaranteed jobs highest priority, and still permits a best-effort attempt on the rejected jobs. For example, given two tasks with periods of 5 and 8 respectively, a HIGH priority job with period 8 is scheduled after a HIGH priority job with period 5 and before a LOW priority job with period 5.

5. Performance Evaluation of SRMS

To evaluate the performance of SRMS, we developed a simulator to run a periodic task system subject to the model and assumptions discussed in section 3.1.

5.1. Simulation Model and Performance Metrics

In our experiments, we made a number of simplifying assumptions. These assumptions were necessary to allow for a more straightforward interpretation of the simulation results, by eliminating conditions or factors that are not of paramount interest to the subject matter of this paper (e.g. effects of task criticality). First, we assumed that all tasks demand the same average percentage utilization of the resource being managed. In other words, the ratio $\frac{E(e_{i,k})}{P_i}$ for all tasks is constant. Second, the probability distributions used to generate the resource requirements were of the same type[11] (but with different parameters) for each task in the system. Also, these distributions were truncated so that no infeasible jobs were submitted to the system. Third, we assumed that all tasks were of equal criticality/importance, which implies that the assignment of allowances $(a_1, a_2, \ldots)$ to the tasks in the system should not reflect any preferability due to the task's "value" to the system.

To compare algorithms and discuss their characteristics, we define a few performance measures. In the following definitions, the number of tasks in the system is n.

Definition 7 *The job failure rate (JFR) is the average percentage of missed deadlines.*[12]

$$JFR = \frac{1}{n} * \sum_{i=1}^{n} \frac{\tau_i \; missed \; jobs}{\tau_i \; jobs}$$

We chose to use the job failure rate because it gives all tasks equal priority. Using a completion count gives unfair importance to tasks with shorter periods, because in any time interval, those tasks will release more jobs than tasks with longer periods. Naturally, this job failure rate assumes that all tasks are of equal criticality and require the same QoS.

With the assumption that all tasks require the same performance, there is a need to describe how fair the system is. For example, in RMS it is quite possible that the highest priority task meets all its deadlines and the lowest priority task meets none. Intertask unfairness describes how unfair the scheduling algorithm is.

Definition 8 *The intertask unfairness is a measure of how unfair the scheduling algorithm is to the different tasks. It is the standard deviation of the percent of missed jobs.*

$$Intertask\,Unfairness = \sqrt{\frac{\sum_{i=1}^{n} (\frac{\tau_i \; missed \; jobs}{\tau_i \; jobs} - JFR)^2}{n}}$$

Finally, we consider the average utilization requested of the system and the average useful utilization achievable by a scheduling algorithm. Note that the achievable utilization is an average, and some overloaded intervals may occur even when the requested utilization is within the schedulability requirement of RMS.

Definition 9 *The requested utilization is the sum of all jobs' resource requirements divided by the time interval during which scheduling occurs.*

Definition 10 *The achievable utilization is the sum of all successful jobs' resource requirements divided by the time interval during which scheduling occurs.*

[11] We considered a variety of such distributions as will be evident later in this section.

[12] This is the opposite of the **job completion rate** used in [11], which is the average percentage of met deadlines.

5.2. Algorithms Considered for Comparison

To evaluate the performance of SRMS, it was necessary to identify algorithms against which SRMS should be compared. This was challenging, as there are no algorithms in the literature addressing the problem of scheduling periodic tasks with highly-variable resource requirements under firm-deadline semantics, subject to minimal QoS requirements. We decided to use three algorithms: RMS, SSJAC, and an Oracle. We justify these choices below.

Rate Monotonic Scheduling: SRMS and RMS are alike in many aspects. Both employ a fixed priority preemptive scheduler, with priorities being assigned in a rate monotonic fashion. Despite the fact that RMS was designed for hard deadlines (as opposed to firm) and constant (as opposed to highly variable) resource requirements, we decided to use it to provide a baseline (a performance lower bound) of what is readily achievable using RMS.

Slack Stealing Job Admission Control: As described in section 2, SSJAC [2] uses slack-stealing to determine whether to admit jobs with resource requirements above a set threshold. Like SRMS, when a job is released, it must undergo admission control. If the job's resource requirement is below the threshold, then it is automatically admitted. Otherwise, it is conceptually "split" into two parts. The first has a resource requirement equal to the allowance and the second part has a resource requirement equal to difference between the originally requested resource requirement and the threshold. The second part is treated as a sporadic task with the same priority, release time, and deadline; it is considered for admittance using the slack in the system. If there is adequate slack to admit such a sporadic task, then the job (with both of its parts) is admitted to the system. Otherwise, the job is rejected. For SSJAC, we chose to calculate the available slack myopically so that no aperiodic servers are necessary. Once a job is admitted to the system, it runs completely at its original priority. To reclaim unspent resource time, we used Thuel's slack reclaimer [19].

SSJAC could be considered as an evolution of the *transform-task* method introduced by Tia *et al.* in [20]. For this problem, the performance of SSJAC subsumes that of the transform-task method. In SSJAC, any job which is not guaranteed to meet its deadline is discarded. This is the correct approach when dealing with firm deadlines—in contrast to the transform-task method's approach of completing all jobs, even if the deadline is missed, which is useful for soft (but not firm) deadlines. Second, rather than using the sporadic server, which has no guarantees, SSJAC uses slack stealing enabling the use of accurate job admission control with immediate results at a job's release time. The main drawback of SSJAC (when compared to the transform-task method) is the high overhead of slack stealing. However, in our experiments, we completely neglected overhead, thus giving SSJAC (as a representative of competing algorithms) a tremendous advantage over SRMS which has a constant overhead.

Oracles for Establishing Performance Upper Bounds: We found it interesting to consider, not merely how SRMS performed against RMS and SSJAC, but also how close is SRMS' performance to the "best possible" performance. To this end, we developed an omniscient oracle for systems with harmonic periods. The oracle accepts different value functions for each job, and will optimize the schedule accordingly. Three value functions are particularly useful. First, the optimal completion count is determined by assigning an equal value to each job of each task. We denote by OPT-J the oracle under this "all-jobs-are-equal" value function. Second, the optimal JFR is determined by using a function that values tasks equally by assigning to each job a value equal to its period. Thus, in any interval of time, each task has the same total value assigned to its jobs. We denote by OPT-T the oracle under this "all-tasks-are-equal" value function. Finally, the optimal effective processor utilization is determined by setting a job's value equal to its resource requirement. We denote by OPT-U the oracle under this "all-resource-cycles-are-equal" value function.

5.3. Simulation Experiments:

We will discuss two of the sets of simulation experiments that we conducted. The first set, *harmonic 5-Tasks*, contained five periodic tasks with harmonic periods.[13] The first period was fixed, and the remaining periods were chosen randomly, so that the ratio between adjacent periods was an *integer* uniformly distributed between two and four. The second set, *arbitrary 5-Tasks*, contained five periodic tasks with arbitrary (i.e. non-harmonic) periods. The first period was fixed, and the remaining periods were randomly chosen, with the ratio between adjacent periods being a *real number* uniformly distributed between two and six.

For our experiments, we pre-determined the resource requirement of each job, so that all algorithms were run on the identical scheduling problem. While we ran sets of different random systems, the results presented below show one run of a given set of randomly generated systems and are representative. We have also run experiments for significantly longer and shorter simulation periods, with comparable results.

Our experiments were run with different probability distributions used to generate the variable resource requests. We considered exponential, gamma, poisson, normal, uniform, and pareto distributions, as well as constant resource requirements, to determine if the gross behavior of the algorithms changed. We found that it did not. In this paper we restrict our presentation to the results we obtained for the poisson distribution. The poisson distribution was chosen because it is frequently used to model data arrivals. In real-time systems, a periodic task may well be responsible for processing all events that arrive within a period of time (hence the variability in execution requirements).

[13] The small size of our task sets was chosen to permit comparison against the *optimal oracles* discussed earlier.

Experiments with Harmonic Task Sets: First, we compare the performance of the various algorithms to those of the oracles we developed for harmonic task sets. Figure 4 shows that OPT-J forms a clear performance upper bound for RMS. This is expected since OPT-J maximizes the completion count. RMS attempts to maximize the completion count by giving preference to tasks with shorter periods (i.e. those likely to contribute "more" to the completion count due to their frequent jobs).

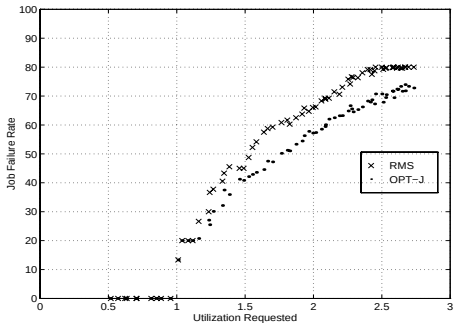

Figure 4. JFR of RMS vs OPT-J for *harmonic 5-Tasks* **with poisson PDFs.**

While RMS attempts to optimize the completion count, both SRMS and SSJAC do not. With all tasks given the same percentage utilization (and requesting the same percentage utilization), both SRMS and SSJAC attempt to fairly distribute the resource among all tasks. This is similar to the function maximized by OPT-T, which gives each task equal value. Figure 5 shows that OPT-T forms a clear performance upper bound for both SRMS and SSJAC.

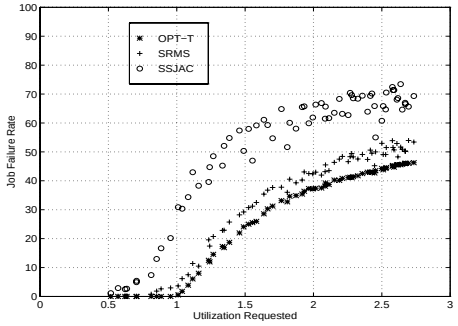

Figure 5. JFR of SRMS/SSJAC vs OPT-T for *harmonic 5-Tasks* **with poisson PDFs.**

We also compared the performance of SRMS, RMS and SSJAC, as shown in Figures 4 and 5. As expected, SRMS outperformed both RMS and SSJAC by a wide margin. Two factors contribute to SRMS superiority. First, SRMS attempts to assign the resource fairly to all tasks. Thus, no outlier tasks significantly reduce the job failure rate, and deadlines are missed fairly by the different tasks. Second, the ratio between adjacent periods is guaranteed to be at least two. This permits aggregation of at least two jobs, which increases the smoothing gained (we will discuss the significance of the ratio between adjacent periods at the end of this section).

As mentioned previously, RMS attempts to maximize the completion count which does not result in the maximization of the job failure rate. However, when the system is not in overload RMS may have performance superior to SRMS. This is because SRMS is pessimistic and more reactive to potential overload than RMS; SRMS may reject a job that could actually make its deadline without damaging effects if it were scheduled at an accepted priority. This can occur if one task in the system is in overload, but the others are not, and the overall system is not in overload.

SSJAC gains most of its advantage by scheduling the extra one third of the utilization which SRMS and RMS cannot guarantee. However, with harmonic periods, full utilization can be auctioned by both RMS and SRMS. Therefore, SSJAC edge is not likely to be evident when the task set is harmonic. Additionally, SSJAC can only acquire extra slack from the past. It cannot schedule an extra long job with the assumption that it can steal that time from the future, as SRMS does. Nonetheless, it does succeed in preventing one task from harming a guaranteed job of another task. In serious overload, SSJAC does perform better than RMS.

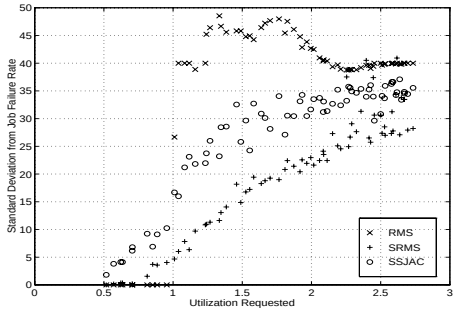

Figure 6. Intertask Unfairness for *harmonic 5-Tasks* **with poisson PDFs.**

Looking at the intertask unfairness shown in Figure 6, similar patterns apply. RMS is completely fair until it becomes overloaded. This is the case because no jobs miss their deadlines! However, as soon as deadlines are missed, the intertask unfairness for RMS rises rapidly, because RMS is extremely unfair in penalizing lower priority tasks. As the system becomes more overloaded, SRMS' intertask unfairness increases, but still manages to be the least of all three algorithms. SRMS' unfairness increases because of time inheritance; tasks with shorter periods will have jobs rejected. The unspent budgets of those tasks are added to the budgets of lower-priority (longer-period) tasks, thus improving the outlook for those tasks, and hence increasing SRMS' intertask unfairness. SSJAC exhibits an intertask unfairness between RMS and SRMS. SSJAC is better than RMS, because a set percentage of jobs for every task are admitted, since their requirements are below the threshold. However, SSJAC performs worse than SRMS because it distributes its slack on a FCFS basis.

Experiments with Arbitrary (non-harmonic) Task Sets: The results for task sets with arbitrary (non-harmonic) pe-

riods were similar to those obtained for harmonic task sets. As evident in Figure 7, RMS performs best before overload. As soon as overload occurs, SRMS has the best job failure rate throughout most of the overloaded area. However, when the overload becomes severe, SSJAC occasionally does better than SRMS. We believe that this is due to two factors. First, SSJAC usually has significant slack to distribute, which is the unguaranteed time, nearly a third of the resource. Second, in overload, SSJAC will reclaim even more time to redistribute, because more jobs will be rejected and not take any of their guaranteed allowances.

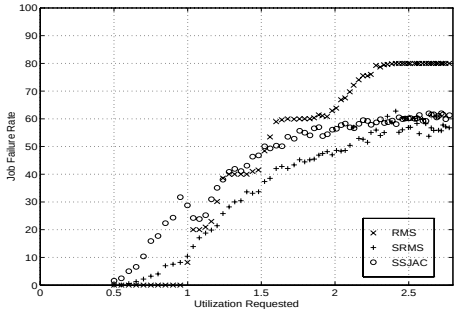

Figure 7. JFR for *arbitrary 5-Tasks* with poisson PDFs.

Figure 8 shows that the intertask unfairness observed for task sets with arbitrary periods is also similar to that observed for harmonic task sets. The main difference is that SSJAC may have better performance than SRMS under serious overload. For a few experiments, even RMS achieved lower unfairness.

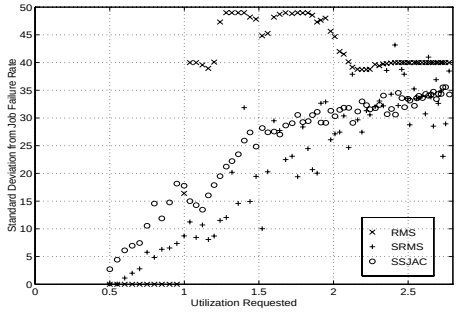

Figure 8. Intertask Unfairness for *arbitrary 5-Tasks* with poisson PDFs.

Figure 9 shows the achievable utilization for arbitrary task sets. Until overload is reached, RMS (again) delivers the best utilization with SRMS a close second. In overload, SRMS is a clear winner. This result is somewhat surprising. Although SSJAC can distribute nearly an extra one third of the utilization, it does not do better than SRMS.

5.4. Effect of Aggregation

As we iterated several times in this paper, one of the main tenets of SRMS is that the variability in periodic resource utilization for a given task can be smoothed through aggregation over time (see Lemma 1) and across tasks (see the extensions in section 4). Such an aggregation is most (least) effective when the ratio of adjacent periods (i.e. P_{i+1}/P_i) is large (small).

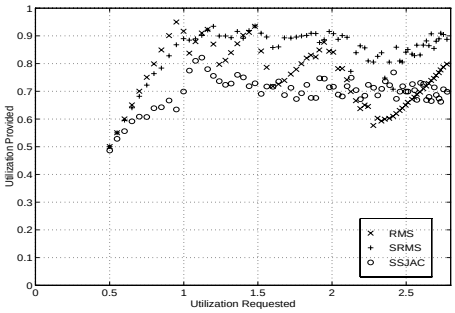

Figure 9. Resource Utilization for *arbitrary 5-Tasks* with poisson PDFs.

Although we could not include them in this paper (for space limitations), we have conducted extensive experiments to study the sensitivity of SRMS to the ratio of adjacent periods. We found that SSJAC does slightly better than SRMS in overload as long as the ratio of adjacent periods is less than two. The regions and shape of the intersection varies depending upon the probability distribution examined. While this is interesting for characterizing the algorithms, the overhead needed for SSJAC is *significantly* higher than that of SRMS, making SRMS much more attractive even when the ratio of adjacent periods is less than two.

6. The SRMS Workbench

For demonstration purposes, we have packaged: (1) the SRMS schedulability analyzer (QoS negotiator), and (2) our SRMS simulator (Basic SRMS + all extensions) into a Java Applet that can be executed remotely on any Java-capable Internet browser. For comparison, a RMS simulator and a SSJAC simulator are included.

Through a simple GUI, the *SRMS Workbench* allows users to specify a set of periodic tasks, each with (a) its own period, (b) the distributional characteristics of its periodic resource requirements (e.g. Poisson, Pareto, Normal, Exponential, Gamma, etc.), (c) its desired QoS as a lower bound on the percentage of deadlines to be met, and (d) a criticality/importance index indicating the value of the task (relative to other tasks in the task set). Once the task set is specified, the SRMS Workbench allows the user to check for schedulability under SRMS. If the task set is schedulable, the SRMS Workbench generates the appropriate allowance for each task and allows the user to create an animated simulation of the task system, which can be executed and profiled. If the task set is not schedulable, the SRMS Workbench informs the user of that fact and suggest (as part of the QoS negotiation) an alternative set of *feasible* QoS requirements that reflects the specified criticality/importance index of the tasks in the task set.

The SRMS Workbench is available at: `http://www.cs.bu.edu/groups/realtime/SRMSworkbench`

7. Conclusion and Future Work

In this paper, we have introduced Statistical Rate Monotonic Scheduling (SRMS)—an algorithm that schedules firm-deadline periodic tasks with variable resource requirements. In addition to providing a predictable scheduling algorithm for this type of periodic task, SRMS is value-cognizant, overload-cognizant, predictable, configurable and enforces task isolation. The job admission control used in SRMS introduces a low overhead of constant complexity. SRMS maximizes useful system utilization by not wasting resources on jobs which will fail. The SRMS enforces task isolation, so that no task can adversely affect another task. This permits SRMS to be overload-cognizant on an individual task basis; the responses caused by the overload only affect the misbehaving task. Additionally, quality of service (QoS) guarantees can be specified for each task[1]. SRMS also permits intratask fairness; a job with a large resource requirement can still be admitted, and a job with a small resource requirement can be rejected.

Our current work focuses on deploying SRMS in working real-time environments. In particular, we are examing a framework where the task set is allowed to change dynamically (i.e. new periodic tasks can enter the system and old ones can leave). To that end, we are designing an API suitable for SRMS, which would allow for QoS specification, negotiation, and for on-line task admission control, including notification of job admission or rejection decisions.

We plan to analyze the case of tasks where the deadlines are either shorter or longer than their periods. The former case for hard deadlines has been solved using deadline monotonic scheduling. We believe that a similar approach will work for SRMS; the key detail is defining the correct task transformation. The case of deadlines longer than their deadlines is more easily solved; the solution for RMS involves different priority bands of rate-monotonically ordered tasks. SRMS already has such bands for accepted and rejected jobs. To transform SRMS to have more bands would not be difficult. The complexity will come from determining when and which budgets should be debited to support a given job.

References

[1] A. K. Atlas and A. Bestavros. Multiplexing vbr traffic flows with guaranteed application-level qos using statistical rate monotonic scheduling. Technical Report BUCS-TR-98-011, Boston University, Computer Science Department, 1998.

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

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

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

[5] K. Bradley and J. K. Strosnider. An application of complex task modeling. In *Real-Time Technology and Applications Symposium*, pages 85–90, June 1998.

[6] J.-Y. Chung, J. W. S. Liu, and K.-J. Lin. Scheduling periodic jobs that allow imprecise results. *IEEE Transactions on Computers*, 39(9):1156–1174, Sept. 1990.

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

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

[9] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Real-Time Systems Symposium*, pages 166–171, 1989.

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

[11] M. Marucheck and J. 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.

[12] A. K. Mok and D. Chen. A multiframe model for real-time tasks. In *Real-Time Systems Symposium*. IEEE Computer Society Press, Dec. 1996.

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

[14] K. G. Shin and Y.-C. Chang. A reservation-based algorithm for scheduling both periodic and aperiodic real-time tasks. *IEEE Transactions on Computers*, 44:1405–1419, Dec. 1995.

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

[16] B. Sprunt, J. Lehoczky, and L. Sha. Exploiting unused periodic time for aperiodic service using the extended priority exchange algorithm. In *Real-Time Systems Symposium*, 1988.

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

[18] T. G. Tan and W. Hsu. Scheduling multimedia applications under overload and non-deterministic conditions. In *Real-Time Technology and Applications Symposium*, June 1997.

[19] S. R. Thuel. *Enhancing Fault Tolerance of Real-Time Systems through Time Redundancy*. PhD thesis, Carnegie Mellon University, May 1993.

[20] T. 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.

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