1998-05-02

Slack Stealing Job Admission Control Scheduling

Atlas, Alia

Boston University Computer Science Department

Slack Stealing Job Admission Control[*]

Alia K. Atlas and Azer Bestavros

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

Abstract

In this paper, we present Slack Stealing Job Admission Control (SSJAC)—a methodology for scheduling periodic firm-deadline tasks with variable resource requirements, subject to controllable Quality of Service (QoS) constraints. In a system that uses Rate Monotonic Scheduling, SSJAC augments the slack stealing algorithm of Thuel et al with an admission control policy to manage the variability in the resource requirements of the periodic tasks. This enables SSJAC to take advantage of the 31% of utilization that RMS cannot use, as well as any utilization unclaimed by jobs that are not admitted into the system.

Using SSJAC, each task in the system is assigned a resource utilization threshold that guarantees the minimal acceptable QoS for that task (expressed as an upper bound on the rate of missed deadlines). Job admission control is used to ensure that (1) only those jobs that will complete by their deadlines are admitted, and (2) tasks do not interfere with each other, thus a job can only monopolize the slack in the system, but not the time guaranteed to jobs of other tasks.

We have evaluated SSJAC against RMS and Statistical RMS (SRMS). Ignoring overhead issues, SSJAC consistently provides better performance than RMS in overload, and, in certain conditions, better performance than SRMS. In addition, to evaluate optimality of SSJAC in an absolute sense, we have characterized the performance of SSJAC by comparing it to an inefficient, yet optimal scheduler for task sets with harmonic periods.

Keywords: real-time computing and communication; scheduling algorithms and analysis; admission control; operating systems.

1   Introduction

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.

Motivation 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, for instance, 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 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 scheduling the use of the shared communication channel amongst the various streams is impractical, as it would result in very poor utilization.

Paper Scope and Outline: This paper presents Slack Stealing Job Admission Control (SSJAC), a generalization of RMS [LL73] which uses slack stealing [LRT92, RTL93, RTL94, Thu93] to allow the scheduling of periodic tasks with highly variable resource requirements and statistical QoS requirements. SSJAC maximizes the utilization of the resource 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 resource requirements.

The remainder of this paper is organized as follows. In section 2, we present previous work related to SSJAC. In section 3, we present the details of the SSJAC model and algorithm. In section 4, we present the results of extensive simulations to evaluate the performance of SSJAC against that of other algorithms. We conclude in section 5 with a summary of on-going research.

2 Related Work

SSJAC is a generalization of Rate Monotonic Scheduling. Therefore, all the schedulability results obtained for RMS are applicable to SSJAC. 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].

SSJAC is based upon slack stealing, introduced by Thuel and Lehoczky[LRT92, RTL93, RTL94, Thu93]. Slack stealing provides a method for guaranteeing and scheduling aperiodic tasks with periodic tasks, which are scheduled by RMS. Slack stealing does this by keeping track of exactly upon which task each time unit is spent. With complete knowledge of the system's execution and future periodic requirements, slack stealing can determine whether there is adequate time in the system to admit an aperiodic task at a given priority level. Slack stealing also has a mechanism for reclaiming unused resource time, known as the slack reclaimer (given by Thuel in her thesis [Thu93]), which credits slack when a job doesn't use all its resource requirement. This occurs with RMS, when the worst case resource requirement must be used.

Significant previous work was done on scheduling aperiodic tasks with rate-monotonic scheduled periodic tasks. Much of this work considered a periodic task which functioned as a server. Sin and Chang considered a polling server with a fixed budget and preset periority[SC95]. Strosnider improved upon the polling server with the deferrable server, which permits the server budget to be spent at any time during its current period [[Str88] in [vTK91]]. The Sporadic Server (SS), presented by Sprunt in [[Spr90] in [vTK91]], has its execution budget replenished based upon how much was consumed since the server last became active. The Extended Priority Exchange (EPE), described by Sprunt, Lehoczky and Sha in [SLS88], exploits the actual variability of task resource requirements to gain more budget for serving aperiodic tasks and exchanges high priority aperiodic time to lower priority periodic time, when no aperiodic works exists. This preserves the high priority of the aperiodic budget.

Binns used slack stealing in [Bin97] to provide scheduling for incremental tasks[CLL90] and design-to-time tasks, where the time needed by the task can be decided at release time based upon the system availability. This use is

similar to that considered in SSJAC. Other work[TH97, KS95] has also explored relaxing the pivotal assumption of RMS—namely that the resource requirement of a periodic task is fixed. Woodbury examined the execution time of real-time tasks in [Woo86]. 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. In [BHS94], Baruah, Haritsa and Sharma considered the theoretically possible performance of an on-line algorithm versus a perfect knowledge optimal algorithm in the presence of overload. In [MS95], Marucheck and Strosnider provided a taxonomy of scheduling algorithms with varying levels of overload and criticality cognizance. In [TH97, KS95], algorithms were given for a scheduler to dicard unnecessary, optional work in the presence of overload.

Work other than SSJAC has considered alternatives to hard deadlines. For overloaded systems, Koren and Shasha considered tasks where some portion of the jobs can just be skipped [KS95]. In [BB97], Bernat and Burns expanded the idea of a skip factor to create the idea of (^n_m)-Hard deadlines, where in any consecutive m jobs, at least n deadlines must be met. They used the ability to skip all non-mandatory jobs to enhance the system's responsiveness to aperiodic tasks.

The work of Tia $et\ al.$ [TDS$^+$95] 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 [TDS$^+$95], 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.

In [AB98a], we developed Statistical Rate Monotonic Scheduling (SRMS), which also considers the problem of scheduling periodic tasks with variable resource requirements and firm deadlines. SRMS provides statistical Quality of Service (QoS)[AB98b] and task isolation. Overloads are dealt with by punishing the task which is in overload. SRMS is an alternative solution to the problem addressed by SSJAC, but it cannot guarantee the efficient use of the extra utilization that exists in the system (as mandated by RMS to ensure schedulability).

3 Slack Stealing Job Admission Control

3.1 Task Model

The SSJAC 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, $(P_i, f_i(x), T_i)$, 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 T_i is the task's resource threshold.*

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.[1] 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 jth job of task i—denoted by

---

[1] 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.

$\tau_{i,j}$—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 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.[2] The resource requirement for the jth job of the ith task is denoted by $e_{i,j}$.

The third element of a task specification under SSJAC is a resource threshold. A task is guaranteed at least its resource threshold every period. To specify a specific QoS, the resource threshold can be set so that the value of the cummulative distribution function (CDF) is at least that desired QoS.

Definition 2 *A set of tasks* $\tau_1, \tau_2, ..., \tau_n$ *is said to be schedulable under SSJAC, if every task* τ_i *is guaranteed to receive its resource threshold* T_i *at the beginning of every one of its periods. Thus, a schedulable task set is one in which every task achieves its specified/negotiated QoS.*

3.2   Algorithm

As mentioned in 2, SSJAC is based upon the slack stealing work presented by Thuel. She applied it to to admitting soft and hard aperiodic tasks into an RMS periodic task system. Binns used slack stealing for scheduling incremental and design-to-time tasks. We use slack stealing to admit periodic jobs of variable length.

The slack stealing algorithm was used to supply budgets to aperiodic task servers, which ran at every priority where aperiodic tasks might run. Thus, in Thuel's algorithm for admitting hard aperiodic tasks, the total amount of slack available before that aperiodic task's deadline is calculated. The aperiodic server recalculates its budget at specified times. This allows the server to dole out the slack to the aperiodic tasks so that no periodic tasks will miss their deadlines.

For SSJAC, we do not use any aperiodic servers; only a job admission test is necessary when a job is released. To allow for no aperiodic servers, our job admission control algorithm only considers slack that is immediately available; any slack which must be waited for is not considered. These slack calculations are similar to those used in the Myopic Slack Manager and help decrease the algorithm's overhead.

Maintaining Slack Information   Before job admission can be used, the state of the system's slack must be updated and stored. Briefly, each job of every task has a slack value associated with it, which is the amount of slack available from time 0 until the job's deadline; this slack is the inactivity time for that task's priority level. The inactivity time can either be precalculated and stored in a table or calculated whenever a new job is released. Thuel in [RTL94, Thu93] described the algorithms for calculating the level-i inactivity time for every $\tau_{i,j}$. If the values are stored in a table, the table must cover a hyperperiod of jobs, where a hyperperiod is the least common multiple of all the task periods. In our implementation, we calculate each job slack when that job is released.

In addition to the calculated inactivity time, run-time counters must be maintained, which describe how and where time has been spent. Slack time could be spent either on a higher or equal priority job or on lower priority jobs. Extra slack time could also be reclaimed from jobs which are shorter than their resource threshold. Using the run-time counters and the inactivity time, the actual slack available for a given job can be determined. With this information, job admission control can take place.

---

[2]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.

Job Admission Control   SSJAC uses RMS to actually schedule all periodic tasks. It differs by requiring jobs to undergo a test before they can enter the system and run. A job is only admitted if it is guaranteed to meet its deadline; no work is done on rejected jobs.

The admission test has two phases. Admission is based upon the job's individual resource requirement and the slack available. As mentioned in 3.1, each task is assigned a guaranteed resource threshold, T. For a given job, if $e_{i,j} \leq T_i$, then the job is automatically accepted. Otherwise, the job is *long*, and must attempt the second phase.

A *long* job conceptually consists of a guaranteed portion and an extra resource requirement. The extra resource required is $e_{i,j} - T_i$. Using the system slack information, admission control determines whether there is slack to support the extra required resource. If there is adequate slack, then the job is admitted. Otherwise it is rejected.

To determine if there is adequate slack, a check is done with the assumption that no other tasks are currently running *long* jobs. The minimum of the slack associated with the job $\tau_{i,j}$ and the slack associated with the next job to complete of each lower priority task is determined and stored as the *available slack*. If the *available slack* is at least equal to the extra resource requirement, then the effects of previously admitted *long* jobs must be considered. Otherwise, the job is rejected.

Previously guaranteed *long* jobs can have two effects. If the job's priority is higher than that of $\tau_{i,j}$, then the job will require some of the available slack. If the job's priority is lower than that of $\tau_{i,j}$, then th job must have adequate slack to permit the extra resource requirement. First, the extra resource required by all guaranteed higher priority jobs is subtracted from the *available slack*. If the resulting *available slack* is smaller than the extra required resource, then the job is rejected. Finally, the minimum slack of all lower priority guaranteed *long* jobs is determined. If that minimum slack is no smaller than the extra resource required, then the *long* job, $\tau_{i,j}$, is admitted and the slack of all lower priority guaranteed *long* jobs is reduced appropriately.

This algorithm is detailed in figure 1.

Fairness Considerations   The slack in the system is given away on a FCFS basis. Therefore the intertask unfairness may be high, as tasks with short periods require admittance more frequently, and may thus acquire more of the slack in the system. In the other hand, the amount of slack the short tasks will require may be less than tasks with longer periods.

Given this FCFS distribution of slack, there are no analytical results for determining each task's Quality of Service using this algorithm. A trivial lower bound on a task's QoS is the probability that $e_{i,j} \leq T_i$, because all jobs shorter than the threshold are automatically admitted.

Task Criticality   Our algorithm for Slack Stealing Job Admission Control does not directly schedule based upon task criticality. However, the value of T_i can be specified to take this information into account. For instance, a task which actually has a hard deadline can still be scheduled, but its T_i must correspond to the worst case resource requirement. If all the tasks require the same processor utilization, but the tasks have different criticalities associated with them, then T_i can be assigned as follows:

$$T_i = \frac{w_i}{\sum_{\forall \tau_j} w_i}$$

More complex relationships can also be reflected in the assignment of the T_i.

Overhead   Any slack stealing algorithm requires considerable overhead, because it is necessary to keep track of how time is spent. Thus, each time a task is scheduled for time, O(n) counters need to be updated. In addition, whenever a job is released, the calculation of that level's inactivity is also O(n). Actually admitting a long job requires O(n), since all lower priority tasks must be checked.

e_i^c = resource requirement of current job of task i
$spent_i$ = time spent on current job of task i
$Inactivity_i$ = inactivity time for next uncompleted job of task i
$slack_i$ = available slack for current long job of task i

SlackAdmitJob(i, e_i^c)
   if ($e_i^c \leq T_i$)
     return ADMIT
   minSlack $\leftarrow Inactivity_i - SpentAperiodicTime_i - SpentIdleTime_i$
   $\forall j > i$, if ($Inactivity_j - SpentAperiodicTime_j - SpentIdleTime_j <$minSlack)
        minSlack $\leftarrow Inactivity_j - SpentAperiodicTime_j - SpentIdleTime_j$
   aperiodicLength $\leftarrow e_{i,k} - T_i$
   if (minSlack < aperiodicLength)
        return REJECT
   minSlack $\leftarrow$ minSlack - aperiodicLength
   promisedAperiodics $\leftarrow$ 0
   $\forall j < i$, if ($e_j^c > T_j$ AND $spent_j < e_j^c - T_j$)
        promisedAperiodics $\leftarrow$ promisedAperiodics $+ e_j^c - T_j - spent_j$
   if (minSlack < promisedAperiodics)
        return REJECT
   jobSlack $\leftarrow$ minSlack - promisedAperiodics
   minSlack $\leftarrow \infty$
   $\forall j > i$, if ($e_j^c > T_j$ AND $spent_j < e_j^c$ AND $slack_j < minSlack$)
        minSlack $\leftarrow slack_j$
   if (minSlack < aperiodicLength)
        return REJECT
   $\forall j > i$,
        $slack_j \leftarrow slack_j - aperiodicLength$
   return ADMIT

Figure 1: Job Admission Test Using Slack Stealing

4   Performance Evaluation of SSJAC

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

4.1   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[3] (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 thresholds $(T_1, T_2, \ldots)$ 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, we assume that the number of tasks in the system is n.

Definition 3 *The job failure rate (JFR) is the average percentage of missed deadlines.*[4]

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

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 4 *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 5 *The requested utilization is the sum of all jobs' resource requirements divided by the time interval during which scheduling occurs.*

Definition 6 *The achievable utilization is the sum of all successful jobs' resource requirements divided by the time interval during which scheduling occurs.*

---

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

[4]This is the opposite of the **job completion rate** used in [MS95], which is the average percentage of met deadlines.

4.2 Algorithms Considered for Comparison Purposes

To evaluate the performance of SSJAC, it was necessary to identify algorithms against which SSJAC should be compared. We decided to compare SSJAC against SRMS, RMS, and an Oracle. We discuss these algorithms below.

Rate Monotonic Scheduling: SSJAC 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.

Statistical Rate Monotonic Scheduling: As described in section 2, SRMS[AB98a, AB98b] uses job admission control and guaranteed time budgets to provide statistical QoS to tasks with firm deadlines and variable resource requirements. SRMS has a constant overhead. Task transformation is used to conceptually aggregate multiple jobs of a task to smooth variability and to increase the jobs which can be guaranteed to meet their deadlines. SRMS does an excellent job of this, as well as providing controllable QoS. However, SRMS is based on RMS, and thus cannot auction away the 31% of utilization that RMS requires as slack for schedulability purposes.

Oracles for Establishing Performance Upper Bounds: We found it interesting to consider, not merely how SSJAC performed against RMS and SRMS, but also how close is SSJAC's performance to the "best possible" performance. To this end, we developed a pseudo-polynomial time, perfect knowledge, oracle for systems with harmonic periods. The oracle accepts different value functions for each job, and will optimize the schedule accordingly. Three value functions that are particularly useful. First, it is possible to determine the optimal completion count 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, it is possible to determine the optimal JFR 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, a third possible value function is one which assumes that the value of a job is proportional to its resource utilization. We denote by OPT-U the Oracle under this "all-resource-cycles-are-equal" value function.

4.3 Simulation Experiments:

We will discuss three of the sets of simulation experiments that we conducted. The first set, *harmonic 5-Tasks*, contained five periodic tasks with harmonic periods.[5] 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. The third set, *arbitrary 10-Tasks*, contained ten periodic tasks with arbitrary (i.e. non-harmonic) periods. The task periods were picked uniformly from the range (100, 100,000]; therefore the adjacent period ratio varied and was, on average, less than two.

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

---

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

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 exponential and normal distributions. The exponential distribution represents the amount of time until an arrival of data, and may be of use in modeling certain tasks. The normal distribution is of more general use due to the Central Limit Theorem, which states that the distribution of a sum of i.i.d. random variables will approach a normal distribution. The execution time of a job may depend upon many identical random variables, such as cache hits and misses[LMW96]. Therefore, the normal distribuiton is of general interest.

Experiments with Harmonic Task Sets: First, we compare the performance of the SSJAC and SRMS to the oracle OPT-T. With all tasks given the same percentage utilization (and requesting the same percentage utilization), both SSJAC and SRMS attempt to distribute the resource among all tasks. This is similar to the function maximized by OPT-T, which gives each task equal value. Figure 2 shows that OPT-T forms a clear performance upper bound for both SRMS and SSJAC.
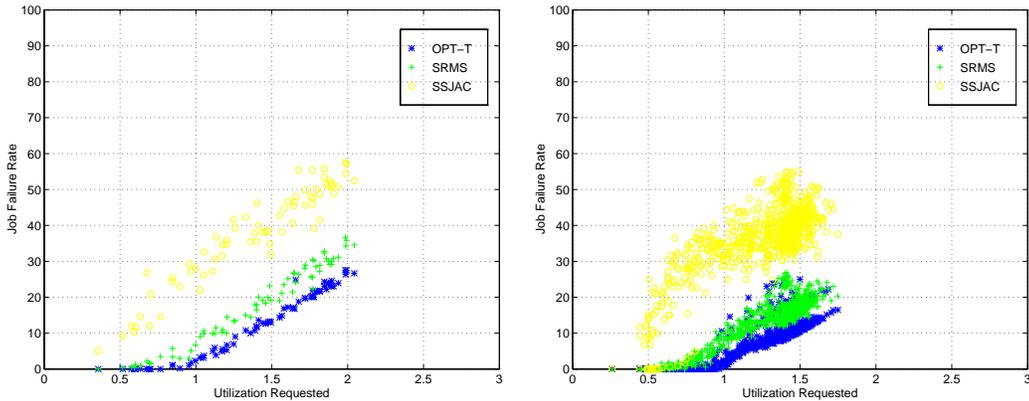

Figure 2: JFR of SRMS/SSJAC vs OPT-T for *harmonic 5-Tasks* with Exponential (left) or Normal (right) PDFs.

We also compared the performance of SRMS, RMS and SSJAC, as shown in Figure 3. We found that SSJAC performed poorly, except in serious overload. This performance is due to the limited system slack available for SSJAC to distribute; SSJAC can only distribute reclaimed slack, and cannot borrow against future short resource requirements. On the other hand, SRMS can aggregate at least two jobs in each time interval, because of a minimum adjacent period ratio of two; therefore, SRMS will perform better.

Experiments with Arbitrary (non-harmonic) Task Sets: The results for task sets with arbitrary (non-harmonic) periods were significantly different. First, we will discuss *arbitrary 5-Tasks* which maintains a minimum adjacent period ratio of two. Next, we will consider *arbitrary 10-Tasks*, where the periods are uniformly picked from a fixed interval.

As can be seen in Figures 4 and 5, SSJAC performs significantly better with arbitrary periods. This is because the unguaranteed 31% utilization can be intelligently distributed. However, SSJAC performs very poorly before overload; it can only acquire slack from previous underload to deal with current overload, rather than counting on future underload. Once the system requests 120% utilization, the behavior of SSJAC is only slightly worse than that of SRMS.

In Figure 6, a surprising result is seen. Although SSJAC can distribute the full utilization of the resource, its provided useful utilization is actually lower than that of SRMS. This result occurs because SSJAC will only schedule against known slack in the system. SSJAC has access to more resource utilization, but cannot always distribute it effectively. Essentially, SSJAC can be too pessimistic in its assumption of available slack and, thus, the jobs which it
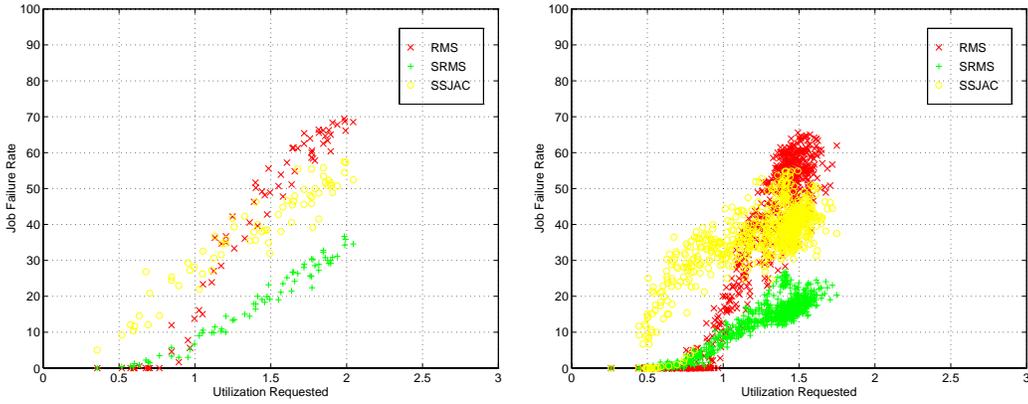
Figure 3: JFR for *harmonic 5-Tasks* with exponential PDFs (left) or normal PDFs (right).
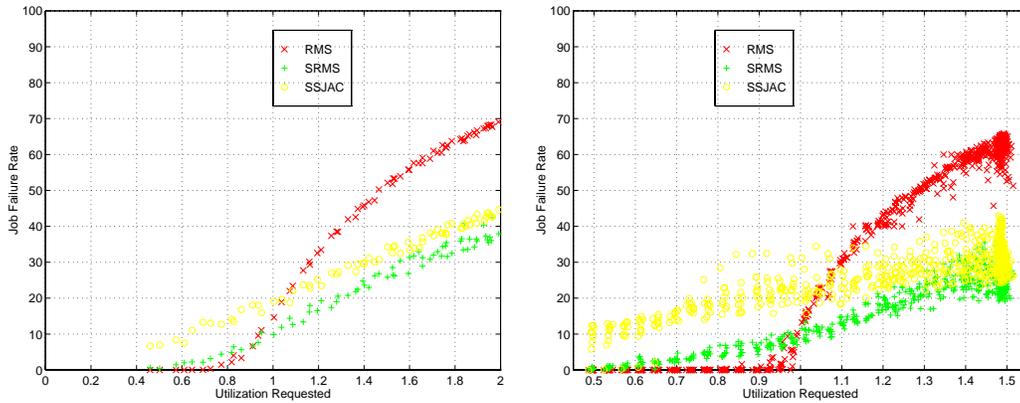

Figure 4: JFR for *arbitrary 5-Tasks* with exponential PDFs (left) or normal PDFs (right).

admits. While no resource is wasted on a job which is guaranteed to fail, neither are any rejected jobs given a best-effort chance to run. Therefore, utilization may be wasted because of lack of future knowledge or possible assumptions.

The results of SSJAC in *arbitrary 10-Tasks* are very different. In Figures 7 and 8, SSJAC clearly outperforms SRMS and RMS in overload. SSJAC is expected to do better in overload than RMS, because SSJAC can reject jobs which cannot meet their deadlines. Therefore, SSJAC controls the overload and directs the resource time towards feasible jobs. In addition, in *arbitrary 10-Tasks*, the adjacent period ratio is expected to usually be less then two. Therefore, SRMS cannot effectively aggregate even two jobs together; this reduces its effectiveness.

The comparatively better performance of SSJAC can be seen as well in Figure 9, where the provided utilization of SSJAC is better than that of SRMS or RMS in overload. Essentially, SSJAC can outperform SRMS because job aggregation is not effective.

As can be seen if Figures 7, 8, and 9, SSJAC can outperform other algorithms during overload under certain circumstances. In our comparisons, SSJAC did the best when periods were not harmonic and the expected adjacent period ratio was less than two.
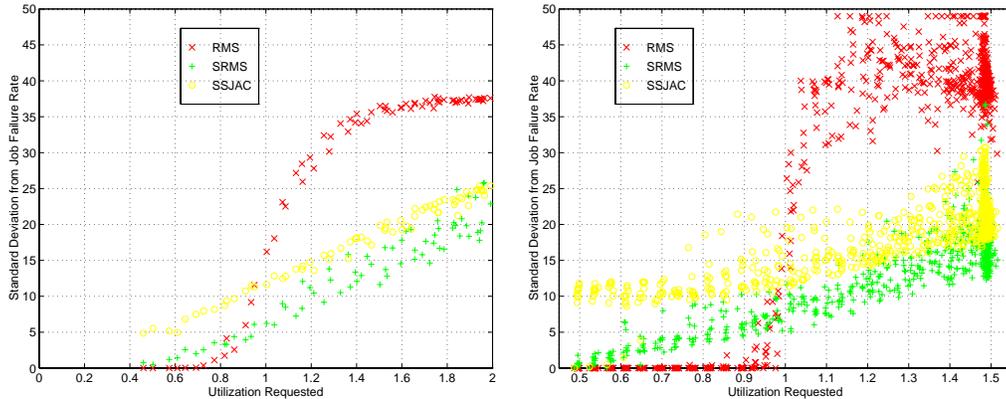
Figure 5: Intertask Unfairness for *arbitrary 5-Tasks* with exponential PDFs (left) or normal PDFs (right).
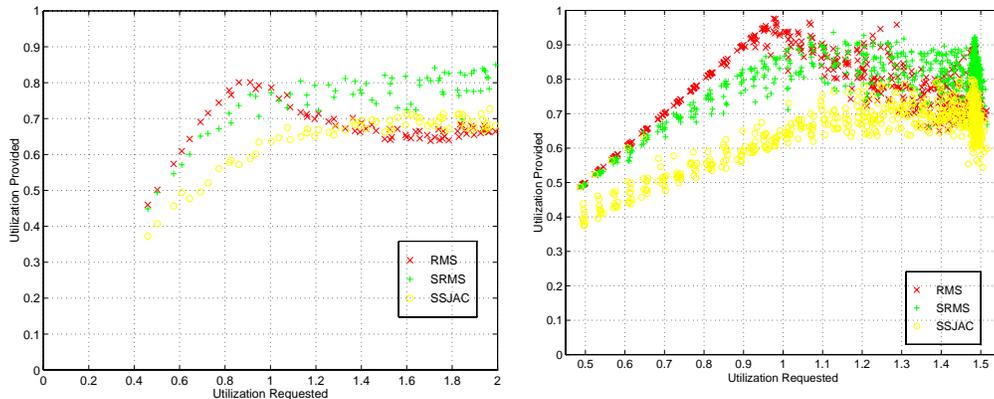

Figure 6: Useful Resource Utilization for *arbitrary 5-Tasks* with exponential PDFs (left) or normal PDFs (right).

5 Conclusion

In this paper, we introduced Slack Stealing Job Admission Control (SSJAC)—an algorithm that schedules firm-deadline periodic tasks with variable resource requirements. SSJAC is overload-cognizant and prevents inter-task intrusion. All tasks have an equal chance to utilize the slack in the system, but no task can prevent another from achieving a user-specified minimum performance. Task criticality can be considered by adjusting the different task thresholds. In short, SSJAC provides a schedule with configurable performance results.

Our performance simulations have explored different possible scenarios, and found that SSJAC performs worst with harmonic task sets. SSJAC is, comparatively, best for task sets with arbitrary periods which are relatively close together. These results do ignore the overhead involved in the slack stealing algorithm. Because of this overhead, SSJAC may not be the correct algorithm for many circumstances. Nevertheless, despite its overhead, SSJAC succeeds in dealing with the problem of scheduling firm-deadline, variable, periodic real-time tasks—a problem with eminent applications, which has not been addressed adequately in the literature.
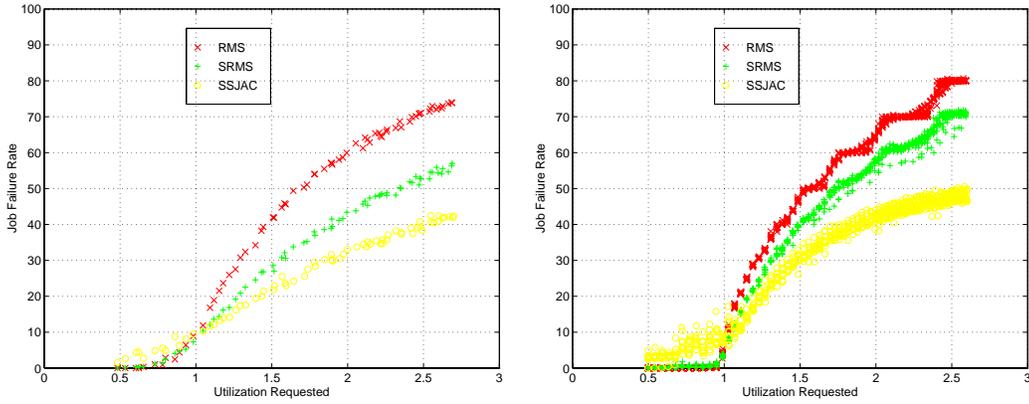
Figure 7: JFR for *arbitrary 10-Tasks* with exponential PDFs (left) or normal PDFs (right).
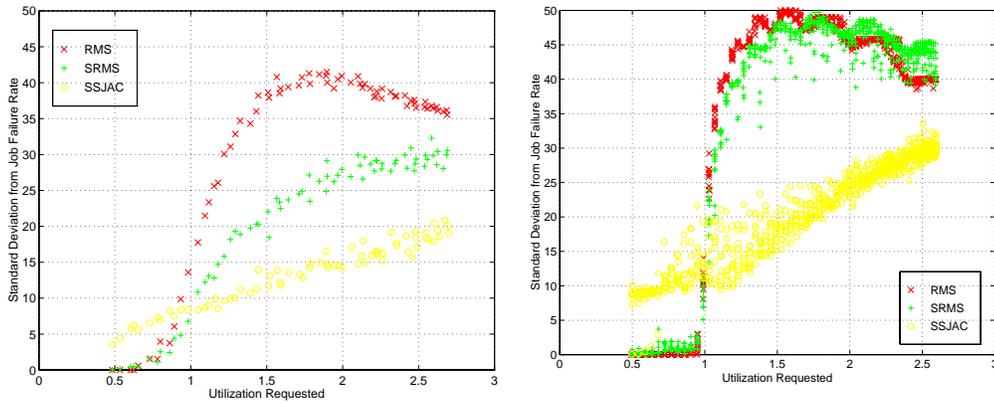

Figure 8: Intertask Unfairness for *arbitrary 10-Tasks* with exponential PDFs (left) or normal PDFs (right).

A   SRMS Workbench

For demonstration purposes, we have packaged a SSJAC simulator with a SRMS simulator and a RMS simulator. Also included is the SRMS schedulability analyzer. The different simulators allow comparison of the schedulers on the same task sets.

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 resource threshold if the scheduler is SSJAC, its desired QoS if the scheduler is SRMS, 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, SSJAC, or RMS. If the task set is schedulable, the SRMS Workbench allows the user to create an animated simulation of the task system, which can be executed and profiled. Further help and system specification is available if the scheduler is SRMS.
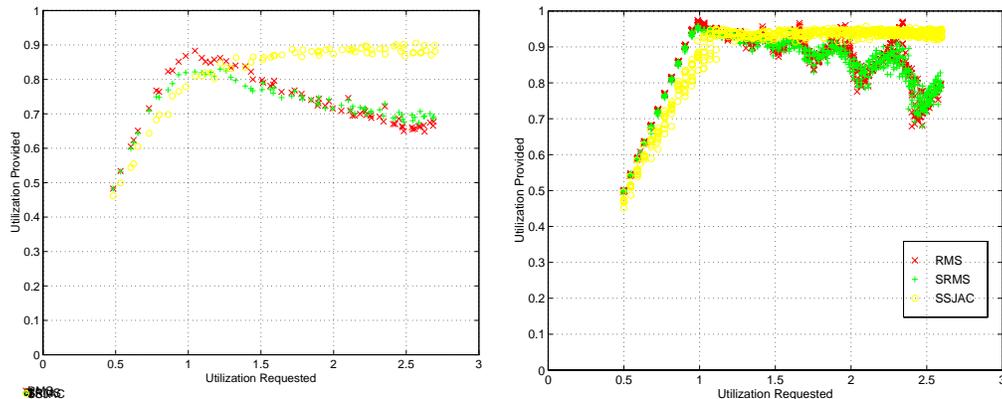
The SRMS Workbench is available at: `http://www.cs.bu.edu/groups/realtime/SRMSworkbench`

Figure 9: Useful Resource Utilization for *arbitrary 10-Tasks* with exponential PDFs (left) or normal PDFs (right).

References

[AB98a]   Alia 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.

[AB98b]   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.

[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, Dec 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.

[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.

[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.

[LMW96]   Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *Real-Time Systems Symposium*, pages 254–263, December 1996.

[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, Dec 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, Dec 1996.

[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.

[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, Dec 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, Dec 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.

[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.

[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.

[vTK91]   Andre M. van Tilborg and Gary M. Koob. *Foundations of Real-Time Computing Scheduling and Resource Management*. Kluwer Academic Publishers, 1991.

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