Co-scheduling Real-time Tasks and Non Real-time Tasks Using Empirical Probability Distribution of Execution Time Requirements

Abhishek Singh

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2009

Approved by:

Kevin Jeffay, Advisor
Sanjoy Baruah, Reader
Ketan Mayer-Patel, Reader
Alan Burns, Reader
Don Smith, Reader

ABSTRACT

ABHISHEK SINGH: Co-scheduling Real-time Tasks and Non Real-time Tasks Using
Empirical Probability Distribution of Execution Time Requirements.
(Under the direction of Kevin Jeffay)

We present a novel co-scheduling algorithm for real-time (RT) and non real-time response time sensitive (TS) tasks. Previous co-scheduling algorithms focussed on providing isolation to the tasks without considering the impact of scheduling of the RT tasks on the response times of the TS tasks. To best utilize the available processing capacity, the number of jobs qualifying for acceptable performance should be maximized. A good scheduling algorithm would reduce the deadline overrun times for soft real-time tasks and the response times for the TS tasks, while meeting deadline guarantees for the RT tasks. We present a formulation of optimal co-scheduling algorithm and show that such an algorithm would minimize the expected processor share of RT tasks at any instant. We propose Stochastic Processor Sharing (SPS) algorithm that uses the empirical probability distribution of execution times of the RT tasks to schedule the RT tasks such that their maximum expected processor share at any instant is minimized. We show theoretically and empirically that SPS provideds significant performance benefits in terms of reducing response times of TS jobs over current co-scheduling algorithms.

Table of Contents

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
Introduction

Scheduling real-time (RT) tasks is a well studied problem. For example, Earliest Deadline First (EDF) is known to be optimal scheduling algorithm for uniprocessor systems, for the Liu and Layland periodic task model where task deadlines same as their period. If the cumulative worst case utilization of the task set is not greater than 1 then the task set is schedulable [LL02]. While this scheduling model is suitable for task sets composed of just hard RT tasks, many practical task sets are composed of tasks with varying timeliness requirements. For example, a General Purpose Operating System (GPOS) runs a wide variety of tasks with different response time sensitivities –

- Interrupts that require "instant" service (a response time of a few μs to a few ms)
- Media playback, computer games, interactive tasks like document editing that require response times in the range of 30-200ms.
- Web servers or databases servicing a large number of concurrent clients, whose performance depend on the response time of the service.
- Large response time tasks that are not sensitive to slight variation in response times. These tasks include tasks submitted to grid systems, media encoding, scientific problem solving tasks, downloading large files etc.

The scheduling problem for such task sets is neither purely deadline based nor one of response time minimization. The goal is to schedule tasks so that the response times are reduced by a factor of 2-3 times as compared to current predominant scheduling algorithms over a wide range of practical scenarios.

1.1 Motivation

A solution to the problem of GPOS scheduling would require a task model where the tasks may have deadlines (RT tasks) or response time constraints. (time sensitive or TS tasks). The goal of the scheduler would then be to

provide deadline guarantees to RT tasks while minimizing response times of TS tasks.

However, realizing such a scheduler in a GPOS is difficult. Most current GPOS are based on multilevel feedback queue schedulers which give preference to shorter jobs and i/o jobs over compute intensive jobs. Multilevel feedback queues fall in the category of best effort scheduling, where there are no scheduling guarantees provided, but the average response times for the jobs are better than using scheduling schemes like FIFO or pure round-robin.

Most GPOS provide mechanisms to assign priorities to tasks. In this setup, RT tasks can be given priority over the non-RT tasks. But this is the worst possible way to co-schedule RT and non-RT tasks because the non-RT tasks are blocked whenever any RT task is active, hence may experience unnecessary long response times.

This raises an important question why current GPOS support priority based schedulers, even though it is the worst possible way to co-schedule RT and non-RT TS tasks. There are two main reasons for this –

- A lack of suitable schedulers for co-scheduling RT and response time sensitive non-RT tasks
- No pressing need for new schedulers

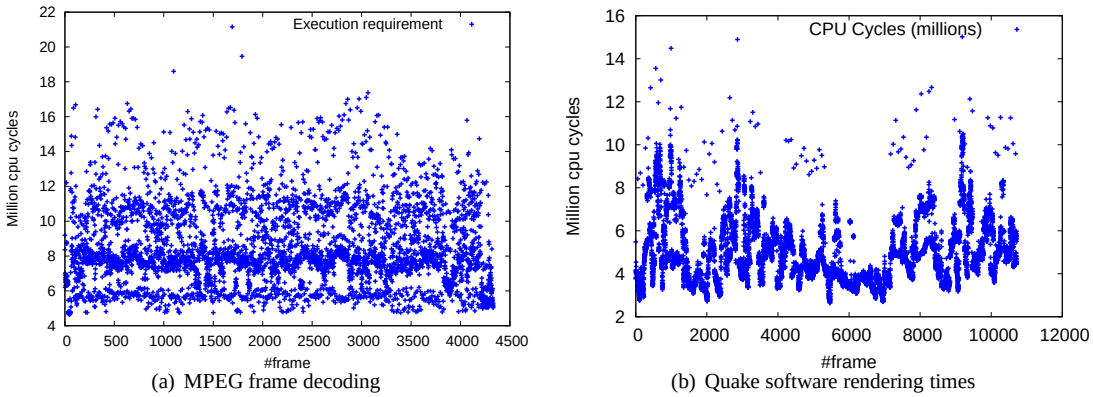
For underutilized systems, the scheduling algorithm has little impact on the response times because there are enough computing resources for all the tasks to finish within reasonable time. Problems arise when there is resource contention. When the processor utilization of the task set is high, inefficient scheduling may lead to RT tasks missing their deadlines and non-RT tasks getting delayed unnecessarily.

1.2 Execution Time Variability

Current real-time schedulers, though appropriate for scheduling tasks with constant execution time requirements, do not handle variable requirement RT tasks well. In fact, they schedule variable requirement RT tasks by assuming that each job may require its worst case execution time requirement, which is not efficient.

For example, consider a MPEG playback task. The frame decoding times have large variation. For example, it is not uncommon for the maximum decoding time to be more than 3 times the minimum decoding time). Even the variation between the decoding time from one frame to another is very large (Fig 1.1).

The deadlines for decoding a frame are not hard. Thus, if a frame misses its deadline by a small amount then it may not lead to any performance degradation at all. Therefore, each job of the MPEG decoding task can be



(a) MPEG frame decoding (b) Quake software rendering times

Figure 1.1: Execution time variation for an MPEG frame decoding task and Quake I software rendering task. The frame decoding times for a StarWars trailer using mpeg_play are reported. Note that the minimum execution time frame requires around 5 million cycles, some frames requirement more than 20 million cycles. Some frames require more than 20 million cycles. Even from frame to frame, the execution time variation is significant. For Quake I software rendering task, though frame to frame execution time variation is not as substantial as MPEG decoding task and the execution time between adjacent frames is similar, the overall execution time variation is large, with mean requirement of around 5M cycles and maximum of more than 15 M cycles.

allocated less than its worst case execution time requirement, and if a job requires more than what it is allocated, then its the duty of the scheduler to keep the deadline overrun time small to avoid performance degradation.

For task sets composed of many soft real-time (SRT) tasks, even if the cumulative worst case utilization is greater than 1, the task set may be schedulable with acceptable performance. For example, consider a task set with 3 tasks, each with worst case utilization of 0.5 and mean utilization of 0.2. Now, while the cumulative worst case utilization is 1.5, the cumulative mean utilization is just 0.6. Also, the probability of ever reaching the theoretical worst case utilization may be very low. Current scheduling algorithms would call for reserving less than the worst case utilization for each job of the tasks ([AB98a], [APLW02], [RH95]), but in that case some jobs may miss their deadlines and in this case the deadline overrun time determines the performance of the algorithm.

Broadly, the scheduling algorithm's responsibilities are –

- Determining the amount of execution time that is guaranteed to each job (the reservation), which may be less than the worst case execution time requirement
- Minimizing deadline overrun times in case of deadline miss.

While many solutions have been proposed for the first problem (like feedback control [APLW02] [LSST02]), few solutions, if any, exists for the second problem.

As mentioned earlier, for underutilized systems, any reasonable scheduling algorithm (like multilevel feedback queues) provides decent performance. This is because there is no actual resource contention and all tasks can finish within reasonable time frame. Though a scheduler giving priority to RT tasks over non-RT tasks may be highly inefficient even for underloaded systems. For example, consider a task set with single RT task with period 100 ms and constant execution time requirement of 50 ms. The utilization is 0.5. Now, suppose a very small requirement non-RT job arrives every 100 ms. The RT jobs also arrive every 100 ms. So each of the non-RT job has to wait for 50 ms, before it is serviced. The response time of each of the non-RT job is at least 50 ms, even though the processor utilization is just 50%. This problem becomes more serious as the processor utilization increases, and the non-RT jobs are delayed even more due to the RT jobs.

In the following sections, we introduce a co-scheduling algorithm that provides hard deadline guarantees to the RT tasks while minimizing response times of the non-RT tasks.

1.3 A Novel Approach - Smart, Adaptive and Learning Scheduler

We focus on the problem of co-scheduling RT and TS tasks such that RT tasks meet their deadlines while reducing response time of TS tasks. This is an important problem with significant practical implications. For example, for a task set with a large number of variable requirement SRT tasks, the cumulative utilization of the tasks at any instant is not very variable (due to statistical multiplexing), and hence the reservation required is significantly less than the cumulative worst case requirement of the task system. But if the task set is allocated less processing time as compared to its worst case requirement, then there may be deadline overruns in some cases when the cumulative processing time requirement of the task set is greater than what is allocated to them. For example, suppose the cumulative mean utilization of SRT tasks is 0.8, cumulative worst case utilization is 3 and suppose one of the constituent tasks has mean utilization of 0.1 and worst case utilization of 0.2. Suppose the task gets a reservation of 0.12. Then any job of this task with greater utilization than 0.12 may potentially miss its deadline. But as mentioned earlier, the cumulative mean utilization of all tasks is 0.8, so on average there is 0.2 processor utilization which is free to be used by any of the tasks. If this free utilization is distributed properly and in a timely manner to the overrun jobs then the potential deadline misses may be avoided. Also, if there are non-RT tasks in the system, then the RT jobs can be delayed such that they do not miss their deadline, but this delaying of RT jobs significantly improves the non-RT response times.

The RT jobs can only be delayed by certain amount without missing their deadlines. Whenever RT jobs

are active they compete for processor time with the other tasks. But if the processor has enough idle time such that all the RT jobs can finish using just the idle processor time, then the RT jobs can be scheduled at lowest priority and still they would meet all their deadlines without interfering with the execution of other tasks. If the processor is busy on the other hand, then the RT jobs would interfere and compete for resources with the jobs of other tasks in the system. For practical task sets, the RT jobs may get some part of their allocation as idle allocation and the remaining because of the scheduling algorithm. Also, the actual execution time requirements of the RT jobs may be variable, so some jobs may require very less execution time, while other jobs may require their worst case execution time. An algorithm can proactively take into account this execution time variability of RT jobs and ensure that the lesser requirement jobs are treated differently from the jobs that require greater execution time, can provide performance benefits to the other tasks in the system. But the execution time of the RT jobs may not be available beforehand, and guessing the execution time is not an option because that may lead to deadline misses. To gracefully handle the variability in execution time requirement of the RT jobs we propose varying the processor share of RT jobs such that the RT jobs get a lower processor share in the beginning but this processor share progressively increases, but the intervals of higher processor share are rarely reached because most jobs would finish before reaching that phase of execution. To achieve this kind of behavior, the probability distribution information of the RT requirements would be needed to ensure that the higher processor share phase of RT jobs is reached only rarely.

Note that our goal is not to find what reservation utilization is required for attaining certain performance level, instead, the goal is to minimize the TS response times while providing reservation guarantees to the RT jobs, whatever they may be. This formulation of the problem is useful as well as practical because the problem of finding the reservation requirement for SRT task has been addressed before but the problem of co-scheduling RT and non-RT tasks efficiently for given reservations for the SRT tasks has largely been unaddressed.

The way our work fits in practical systems is as follows. Algorithms like feedback-control focus on determining the minimum reservation required for SRT tasks to attain acceptable performance. Using our scheduling algorithm would yield the minimum reservation (calculated through feedback-control) required for the SRT tasks to attain given performance in terms of jobs finishing within certain response time. So while the feedback-control loop remains the same, the internal scheduling is changed to use our algorithm that provides RT guarantees while minimizing response times. In the next section we present a formal definition of the problem we are addressing along with key contributions and results.

Notation	Meaning
RT Task	Real-time task with a deadline.
non-RT Task	Non real-time task.
SRT	Soft real-time, task with non-critical deadline. In case of deadline miss, deadline overrun time determines performance.
TS	Time sensitive task, response time determines performance.
EDL	Earliest Deadline as Late as Possible
GPS	Generalized Processor Sharing
GPOS	General Purpose Operating System
SPS	Stochastic Processor Sharing
WCET	Worst-Case Execution Time Requirement for a job of a task
Worst-case utilization	WCET divided by task period
Mean utilization	Mean execution time requirement divided by task period
Cumulative worst-case utilization	Sum of worst-case utilizations of all the tasks
Cumulative mean utilization	Sum of mean utilizations of all the tasks. This may be considerably less than the cumulative worst case requirement.
Task reservation utilization	Execution time requested by each job of the task divided by its period. The scheduler has to guarantee this execution time to each job of the task. The reservation utilization is usually between the mean utilization and the worst case utilization of the task.
Cumulative reservation utilization	Sum of reservation utilizations of all the tasks.
Scaled response-time	Response time of a job of a task divided by its period.
Scaled overrun-time	Overrun time of a job of a task divided by its period.

Table 1.1: Notation summary

1.4 The Co-Scheduling Problem

Table 1.1 briefly describes the notation.

1.4.1 Task Model

A task is represented as the tuple (P_i, χ_i, C_i, R_i), where P_i is the period (jobs arrive every P_i time units), χ_i is the random variable representing the execution time requirement of a job of this task, C_i is the worst case execution time requirement of this task (WCET) and any job of the SRT task should receive R_i execution time before its deadline. We call R_i the reservation of the SRT task. Our scheduling algorithm schedules the SRT tasks in the best possible manner such that the each SRT tasks is guaranteed its allocation of R_i execution time by its deadline (P_i time units), while minimizing the response times of TS tasks.

A task with $R_i < C_i$, is a SRT task and some jobs of this task may miss their deadline and the goal of the scheduling algorithm is to minimize the deadline overrun time for jobs missing their deadline. A task with

$R_i = C_i$ is RT task where no job should miss its deadline. A task with $R_i = 0$ is a TS task with no deadline, rather the performance is determined by its response time. Response time is defined as the difference between the job finish time and its arrival time. Overrun time for a SRT job is the difference between its response time and its period. If a SRT job finishes before its deadline, then its overrun time is considered to be 0.

A SRT job of task i arriving at time t is guaranteed R_i execution time during the interval $[t, t + P_i]$. If the job does not finish after it has received its allocation share of R_i, then this job is scheduled as a TS job for the remaining duration of its execution.

The goal of scheduler is to allocate R_i execution time to each job of task i before its deadline P_i, and do so while minimizing the response times of TS jobs.

We assume that the only information available to scheduler is the period P_i and reservation R_i for each SRT task in the system. The execution time of any job of a task is a random variable, denoted as χ_i, that is the scheduler has no information about the actual execution time requirement of any job (non clairvoyance). For a clairvoyant scheduler, the optimal schedule would be different than that given by proposed approach. Assuming non clairvoyance is very useful , because for most practical systems, the actual execution time requirements of a jobs are neither known nor easily available. The usual way to get around this problem is that the scheduler fixes the reservation value R_i for each SRT task, and the scheduling is done assuming each SRT job requires R_i execution time units, neglecting the variability in execution time requirement. This is a serious flaw with current schedulers, and we will show through this work that substantial performance gains are possible if the execution time distribution is accounted for by the scheduling algorithm.

This brings us to the thesis statement.

The empirical probability distribution of execution time requirements of tasks can be effectively used by an online scheduling algorithm to improve response-times of the non real-time tasks while meeting deadlines for the real-time tasks.

1.4.2 $s(t)$ and $A(t)$

The goal of our scheduling algorithm is to minimize the TS response times. For doing that, we first need to know what are the best possible response times for the TS jobs. If the TS jobs are scheduled in FIFO order, then the scheduling algorithm that provides optimal response times is known [RCGF97]. But FIFO is not a good policy to schedule TS jobs if the job sizes are variable. Shortest Job First (SJF) minimizes the mean response time if the execution time requirements for the jobs are known beforehand, Least Attained Service first (LAS) minimizes

the mean response time if the execution time requirements are not known beforehand.

Now the TS jobs share the processor with RT jobs. Let $A(t)$ denote the cumulative allocation to the TS jobs as a function of time. Now depending upon the co-scheduling algorithm the value of $A(t)$ may vary. The algorithm that schedules the TS jobs gets a cumulative allocation of $A(t)$ by time t, and this allocation is dependent upon the co-scheduling algorithm and independent of the algorithm used to schedule the TS jobs.

A co-scheduling algorithm that gives greater $A(t)$ by time t, in general can provide better response times to the TS jobs.

But $A(t)$ alone does not characterize the TS response times fully. For example, consider a task set with a single RT task with period 100 ms and constant execution time requirement of 50 ms. Now suppose there is a very large execution time requirement TS job that arrives at time 0, and a very small execution time requirement TS job arrives every 100 time units starting at time $t = 50$. Now the co-scheduling algorithm that maximizes $A(t)$ is one that maximally delays the RT jobs. So the RT job arriving at time 0, is scheduled at time $t = 50ms$, and RT job arriving at time $t = 100ms$ is scheduled at time $t = 150ms$ and so on. The problem with this co-scheduling algorithm is that any TS job arriving at time $t = 50ms$ or $t = 150ms$ is delayed for $50ms$ before it gets any service. This is because, the RT jobs needs to execute from time $t = 50$ till time $t = 100$ and from time $t = 150$ to $t = 200$ in order to meet their deadlines. On the other hand, giving RT jobs priority over the TS jobs would give a schedule where the RT jobs are scheduled from time $t = 0$ till time $t = 50$, and from $t = 100$ till time $t = 150$ and so on. So the small requirement TS jobs arriving at time $t = 50$, $t = 150$ and so on, get serviced as soon as they arrive.

This example shows that to provide better TS response times, the small TS jobs should not be made to wait. To model this requirement we introduce the measure $s(t)$ which represents the processor share of RT jobs at time t. So, at time t, the TS jobs get a processor share of $(1 - s(t))$. The value $E[s(t)]$ represents the expected processor share of RT jobs at time t, and keeping this value small rather than $s(t)$ is more useful. This is because if the intervals when $s(t)$ is high are rare, then their performance impact is less. And this is particularly useful, if having $s(t)$ high (though rarely), reduces the value of $s(t)$ for more probable cases.

Formally,

- $s(t)$ is a number in the range $[0, 1]$ and denotes the cumulative RT processor share at time t. Correspondingly $(1 - s(t))$ denotes the cumulative processor share available to TS tasks at time t.
- While $(1 - s(t))$ is the instantaneous processor share available to TS tasks, $A(t) = \int_0^t (1 - s(t))\mathrm{d}t$ is the

cumulative allocation to TS tasks in the interval $[0, t]$.

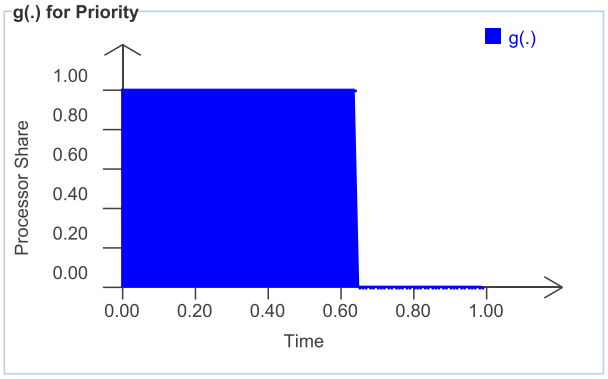


Figure 1.2: $s(t)$ for a job with requirement equal to the worst case execution time requirement under Priority scheduling. Note that the RT job gets the full processor during the time it is active. We label this curve as $g(.)$. The reasons for this will be explained in the later chapters, but for now understanding the shape of the curve is sufficient.

To understand the meaning of $s(t)$, let us work through some simple examples. Consider a task system with a single unit period RT task with mean requirement of 0.3, worst case requirement of 0.65. Suppose all the other tasks in the system are non-RT. The goal of the scheduling algorithm is to finish each job of the RT task before its deadline (which is same as period and equal to one time unit). There are many ways to achieve this. What we are looking for is the way that best benefits the non-RT tasks in the system.

Now the most straightforward way to guarantee no deadline misses to the RT task is to give the RT jobs priority over the non-RT jobs in the system. That is, whenever there is an RT job in the system, it preempts all other non-RT jobs and executes on the processor until it finishes. Clearly, such a schedule guarantees that all the RT jobs meet their deadlines. Figure 1.2 shows the processor allocation to the RT job requiring its worst case execution time requirement as function of time since its arrival. We call this the *Priority* scheduling approach. The height of the shaded region is the value of $s(t)$, and the area of the shaded region is the allocation to the RT job.

Another way to schedule the RT job is to give the RT job a constant processor share of 0.65. Clearly, even in this case no RT job misses its deadline. But this is a better processor allocation scheme than before, because in this case the non-RT jobs may get some processor allocation even when the RT job is active, which gives better $A(t)$. This schedule is shown in the Fig 1.3. We call this the *GPS* scheduling approach, where GPS is short form for Generalized Processor Sharing model [PG93].

Still another way to schedule the RT job is to delay its execution until the latest time such that it still meets

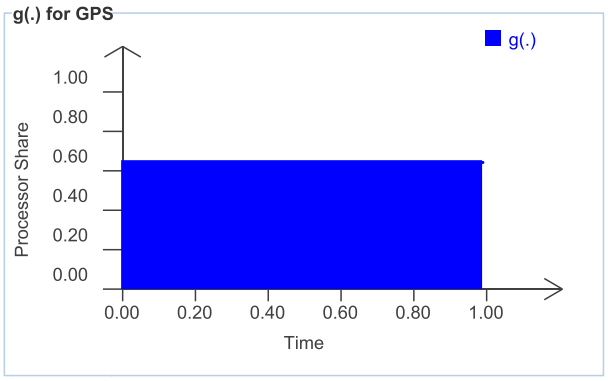


Figure 1.3: $s(t)$ for a job with requirement equal to the worst case execution time requirement under GPS scheduling. Note that the RT job gets the processor share equal to its worst case utilization during the time it is active.

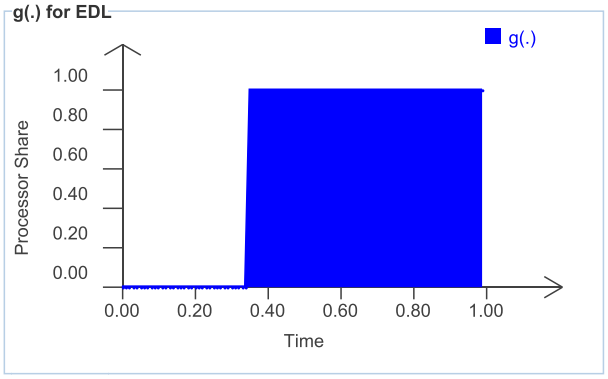


Figure 1.4: $s(t)$ for a job with requirement equal to the worst case execution time requirement under EDL scheduling. Note that the RT job gets the full processor share starting from time 0.35 till its deadline of 1 time unit. Since the RT job is delayed maximally, so $A(t)$ is maximum for this scheme for any value of t.

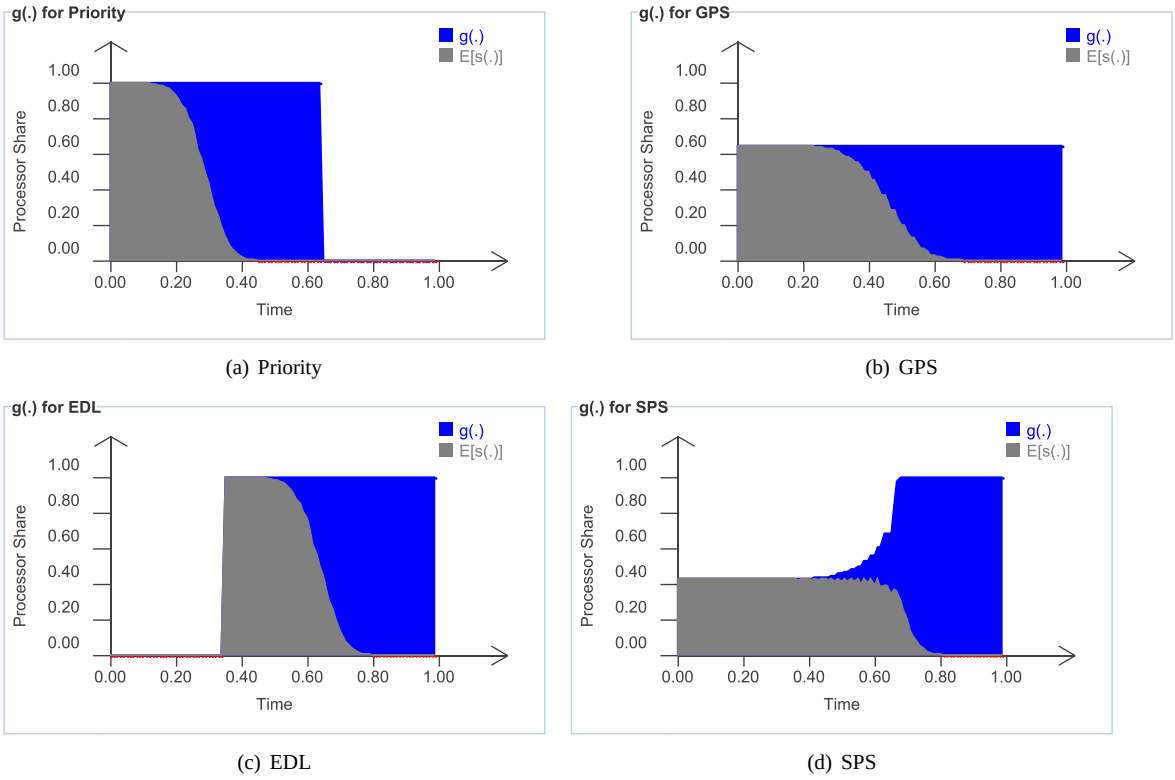
its deadline in the worst case execution time requirement case. In particular, a RT job arriving at time 0, is not scheduled until time 0.35. And once it is scheduled at time 0.35, it alone gets the entire processor. Also, it can be seen that no RT job misses its deadline in this case also. And this schedule has the added advantage of giving the best possible $A(t)$ function, because delaying the RT job any more may lead to deadline overruns. Fig 1.4 shows such a schedule. We call this the EDL scheduling approach, where EDL stands for the *Earliest Deadline as Late as possible* scheduling algorithm, because this algorithm is based on the notion of delaying the execution of the RT job by as much as possible, thereby giving better timely allocation to the non-RT jobs. In fact, this will be the optimal algorithm if the TS jobs are scheduled in FIFO order.

So is EDL the best solution ? No! This is because of two main reasons. First, note that a non-RT job arriving

when the RT job is scheduled is blocked for the entire duration during which the RT job is active under EDL. In this scenario, EDL performs as badly as Priority scheduling, even though EDL has better $A(t)$ than Priority. In fact, EDL may lead to arbitrary long intervals when non-RT jobs are blocked. For example, consider a task set with a single RT task with period 100 seconds and constant execution time requirement of 50 seconds. Suppose there is a constant streams of TS jobs arriving, then under EDL class of algorithms, from time $t = 0$ till $t = 50$ seconds non-RT jobs are scheduled while RT job waits. From time $t = 50$ seconds till $t = 100$ seconds, the RT job is scheduled and non-RT jobs arriving during this time are blocked. In worst case, a very small requirement (and probably highly response time sensitive) non-RT job arriving at time $t = 50$ seconds is blocked for 50 seconds (from $t = 50$ until $t = 100$). In such a scenario, a GPS based schedule would provide better response times to small TS jobs. Second, note that even though the worst case execution time requirement of the job is 0.65, as we mentioned before the mean requirement is just 0.3. The above three scheduling algorithms plainly neglect this information, and schedule irrespective of it.

This is where we enter. We propose a scheduling algorithm, which we call Stochastic Processor Sharing or SPS, and this scheduling algorithm provides guarantees to the RT tasks while taking into account their execution time requirement variability to provide better response times to the non-RT tasks in the system. How can it do that ? Figures (Fig 1.5) give a general idea on what actually SPS does, and in the following chapters, SPS is explained in detail. Note that SPS continuously varies the processor share allocated to the RT job with progress (the shape of the function is determined by the probability distribution of the execution time requirement of the RT job). Under SPS, the RT job starts with a lesser processor share as compared to GPS, and gets the full processor share near its deadline. And the shape of the function $s(t)$ is such that the maximum expected processor share of the RT jobs at any time t, represented as $E[s(t)]$ is minimized.

In terms of the $A(t)$ functions, the algorithms fare as shown in the figure Fig 1.6. As can be seen from the figure, EDL performs the best in terms of $A(t)$, followed by SPS, GPS and Priority in that order. In fact, giving priority to the RT jobs over the non-RT jobs in the system is the worst possible way to schedule RT jobs from the point of view of non-RT jobs, and still it remains the most widely implemented scheduling algorithm in current General Purpose Operating Systems (GPOS).



(a) Priority

(b) GPS

(c) EDL

(d) SPS

Figure 1.5: This figure shows the $s(t)$ function for the worst case requirement job (as before) for each of the scheduling algorithms (Priority, GPS, EDL and SPS), and it also shows the expected processor share as a function of time. Note that under EDL and Priority, the maximum expected processor share of RT job may be 1, while under SPS it is 0.4 (mean is 0.3 and worst case is 0.65). This also gives insight into the shape of $s(.)$ for SPS. Basically, for SPS, the shape of $s(.)$ is such that the maximum expected processor share of RT job is minimized.

1.5 Performance Gains

All that said and done, the performance gains should be tangible or quantifiable. Now, what we are doing is to improve response times of non-RT or TS jobs while meeting deadlines for the RT jobs. So the obvious measure of performance is the response times of the non-RT jobs.

We started with the definition of $s(t)$ and $A(t)$ which give a good theoretical model to compare performance of various algorithms in terms of these measures without worrying about the actual non-RT workload and its scheduling. In the following chapters we present theoretical proofs that SPS performs better than GPS and Priority in terms of the measure $A(t)$, and better than all three (EDL, GPS and Priority) in terms of the maximum value of $E[s(t)]$.

To translate these results, which are in terms of $s(t)$ and $A(t)$, into actual measurements, more groundwork

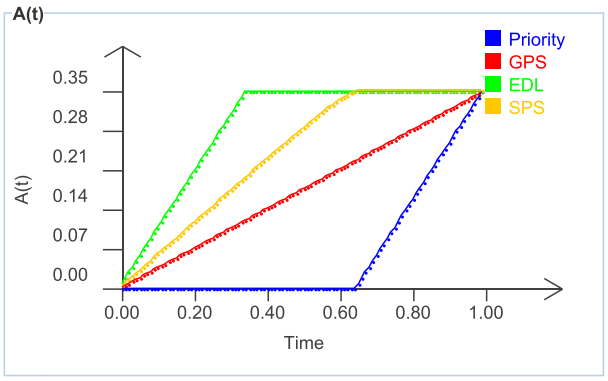


Figure 1.6: This figure shows $A(t)$ for a job with requirement equal to the worst case execution time requirement under the four scheduling schemes. Note that EDL $A(t)$ is the maximum , followed by SPS, GPS and Priority in that order. In fact, Priority gives the worst possible $A(t)$ for any scheduling schemes because under Priority the non-RT jobs are blocked by RT jobs.

needs to be done. First, note that measuring improvement in response times of non-RT jobs is a very open problem. For example, what kind of non-RT workload should be used to measure performance upon. Once the workload is decided, what should be measured – the mean response time perhaps. But the mean response time may be strongly biased in the favor of jobs with large response times. Also, the mean response time is just a number and it does not give information about how actually the response times are distributed.

To understand this consider a MPEG decoding application playing at 30 frames per second. Now, a frame needs to be decoded every 33 ms. If a frame takes 35 or 38 ms to decode, the impact on perceived performance is minimal. For frames taking say 50 ms to 100 ms, the is some perceived delay but not serious performance degradation. For frames taking more than say 200 ms to decode, the impact is visible and there is little difference if the decoding time is 150 ms or 170 ms. In particular, jobs with response time close to the period have little performance impact, while jobs with considerably greater response time all have negative impact on the performance.

Thus, there is a need to formulate a measure that can quantify and compare performance of scheduling algorithms that combine hard deadlines with softer ones. Though works like Time Utility Functions (TUFs) [JLT85] tie response time to performance. We use a much simpler measure, which is easy to understand and can provide important understanding and insights into the scheduler performance. Basically, to properly express the behavior of scheduling algorithms, we propose the Φ measure, where $\Phi(x)$ denotes the number of jobs whose response time scaled by their period is greater than x. And instead of reporting one value, a curve $\Phi(x)$ for $x > 0$ is reported, which gives the distribution of response times for all the TS jobs.

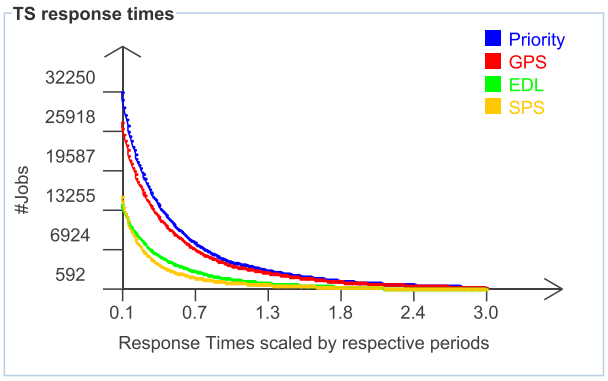


Figure 1.7: This figure shows $\Phi(.)$ function for a task set with 50 SRT tasks and 50 TS tasks. The cumulative mean utilization of SRT tasks is 0.50 and their cumulative reservation is 0.65, and the cumulative mean utilization of the TS tasks is 0.35. The execution time requirement for all the tasks are normally distributed. Note that EDL/SPS perform nearly equally while outperforming GPS by a factor of 2 and Priority by a factor of 3.

Fig 1.7 shows the response time $\Phi(.)$ function for a task set with 50 SRT tasks with cumulative mean utilization of 0.50 and cumulative reservation of 0.65. There are 50 TS tasks in the system, with cumulative mean utilization of 0.35. Also, all the tasks have normally distributed execution time requirements. To understand this graph, note that the lesser the value of $\Phi(x)$ for given value of x, the better the performance of scheduling algorithm in terms of TS response times. Also $\Phi(x)$ is a decreasing function. Because $\Phi(x)$ denotes the number of jobs with scaled response time greater than x. Also, note that we plot the values for x from 0.1 to 3.0, this is because, all TS jobs will have their scaled response times greater than 0, so $\Phi(0)$ would report the number of TS jobs for all the scheduling algorithms.

For SRT tasks, $\Phi(x)$ represents the scaled overrun time. So $\Phi(0.4)$ represents the number of SRT jobs with their deadline overrun times greater than 0.4 times their respective periods. The usefulness of the $\Phi(.)$ measure is that even if $\Phi(x)$ is high for small values of x (say $0 \leq x \leq 0.4$), the performance may be acceptable if few RT job have their deadline overrun time greater than say 0.4 times their period ($\Phi(0.4)$ is the number of such RT jobs).

1.6 Scheduler Evolution - From Simple Deadline Driven EDF to Intelligent SPS

EDF has been known to be optimal RT scheduling algorithm for a long time now. But current workloads are complex and their timeliness requirements are not plain deadline based but a mixture of fuzzy deadline and response time sensitiveness. These workloads require scheduling algorithms that specifically address these issues. SPS not only handles these issues, but does so intelligently with minimal information about the task workload. The only information it requires is the periods and the reservations for the SRT jobs, and it finds the best schedule automatically by constantly profiling the execution time requirements of the SRT jobs. It does not rely on schemes predicting execution time requirement or fancy feedback mechanism to adapt to changing execution times of the application.

Most current GPOS are based on quantum based schedulers, and as we will show in later chapters, SPS can be easily mapped to quantum based schedulers. But the quantum size on current GPOS is large (around 10 ms). Though over the years, the scheduling quantum has come down from 100ms to 10ms or less, still for an efficient SPS implementation the quantum size should be in the range of few ms if the minimum SRT periods are in the range of 20ms. A positive development in this regard is the arrival of parallel hyper-threaded processors. Hyper-threaded processors can service multiple tasks concurrently, and this model of processor service is closer to GPS and hence more suited to SPS than the sequential single task processing model. SPS formulation requires just two tasks to be run concurrently on the processor (one RT job with processor share $s(t)$ and one TS job with the remaining processor share). While current processors can run multiple tasks concurrently (as in hyper-threading), this is neither true parallel service model nor does it allow tasks to given a fraction of processor. Still, it is a positive development, and probably soon in future processors may support weighted concurrent sharing by multiple tasks.

1.6.1 Scheduling Network Flows with Bandwidth Guarantees

Though the analysis done in this document is based on operating system task model, it can be easily applied to the case of scheduling variable requirement network flows with bandwidth guarantees while providing smaller response times to other flows. This scenario may arise in cases like booking network bandwidth for video conferencing application, or supporting massively multi-player online video games. In such a setup, the bandwidth booking would be useful on the bottleneck network node. The constituent TCP flows can be considered as SRT

tasks with their periods depending on the RTT (actually the timeout value for the individual connections). If we define the response time for a packet as the time spent by the packet on the network node then the network node should limit the response time of packets on it belonging to the reserved bandwidth flows, while keeping the response times for packets of other flows small. It should be noted that GPS as a concept was introduced in the domain of network bandwidth sharing, and SPS is closely related to GPS, barring the fact that it does not assign fixed share to the tasks (or flows), instead their share may vary with progress/time. In the following chapters we present examples on how SPS can be used to schedule variable bandwidth requirement flows with bandwidth guarantees on network nodes.

1.7 Organization

The organization of this thesis is as follows. We start with a broad literature survey of RT scheduling algorithms and SRT scheduling algorithms. The scheduling approach we follow is significantly different from the current RT scheduling algorithms, so that makes direct comparison difficult. Still, the problem of co-scheduling has been frequently addressed, and we hope to provide the reader with a broad perspective on the approaches used. In the next chapter we present the theoretical framework, and use this as the basis for proofs that SPS has good performance in terms of the measures $s(t)$ and $A(t)$, as compared to current scheduling algorithms. In this chapter, we show how a quantum based scheduler can support SPS. In the next chapter we expand the domain of SPS to SRT tasks and also bring about the importance of profiling the RT requirements correctly. Since the $s(t)$, function is directly dependent upon the RT requirement distribution, hence measuring them reasonably well is the key factor impacting the performance. In particular, we show that taking into account the processor idle time is very useful to get good performance using SPS. In this chapter we also formally introduce and explain the notion of $\Phi(x)$ measure and present some preliminary results to understand how to interpret it. Finally, the conclusions, limitations and future work are presented.

CHAPTER 2
Literature Survey

2.1 Real-Time Scheduling

In real-time scheduling, tasks are required to finish before deadline. For a non clairvoyant scheduler, this implies that every job is allocated for Worst Case requirement. The scheduling is concerned with allocating such that all jobs of all the real time tasks meet their deadline.

For periodic task systems, where jobs arrive periodically, the cumulative system utilization should be less than one for schedulability. Consider P_i represents the period of i^th task and W_i its worst case requirement, then, for schedulability,

$$\sum \frac{W_i}{P_i} \leq 1$$

Static priority scheduling like Rate Monotonic [LL02], provide deadline guarantees if the cumulative worst case system utilization is less than 0.69.

Commercial real-time systems often use frame based scheduling [Hor74], which is again based on worst case requirements.

Though allocating for worst case is essential for real-time tasks, this may cause serious system underutilization. For example consider a two task system, where both tasks have unit period, and the mean utilization of the tasks is 0.2 and 0.3 respectively, while the worst case utilization is 0.5 and 0.5. Thus, on average only 50% of computing resources are used, but since the worst case system utilization is 1, hence no more real-time tasks can be supported.

In classic real-time scheduling theory, the remaining 50% of system utilization goes to waste on average. Note that this is not problem of knowing the execution requirement, but a more basic problem due to requirement variability. That is even if the exact execution requirement are known, no more tasks can be added to this system because the worst case utilization with the two tasks is 100%. Hence, if more workload is added to this system then there is a risk of missing deadline.

Now, often missing deadlines may not be serious if the deadline miss time is small. It may be the case that the task does not have a clear deadline but a range of acceptable values, so while some jobs may miss deadline, they may still finish within acceptable time, and hence not counting as deadline miss. Another scenario is when real-time tasks are composed of multiple steps. If the deadline is missed by a small amount in one step, the task may still finish on time if it finishes early enough in following steps or stages. Thus alongwith providing deadline guarantees, minimizing response time in case of deadline miss is critical in minimizing resources required to build real-time systems. For general tasks like media decoding etc, the deadline is not a clear value instead it is a range of values which may be viewer dependent. That is some people may notice even slight frame decode time variation, while others may not notice small variations at all. Thus keeping response time close to deadline leads to better perceived performance for these tasks, while meeting all deadlines is not necessary. Through this work we address these issues and provide solutions for them.

And the goal of scheduling is to minimize the deadline overrun time, so that even in case of deadline miss, the likelihood of having response time within acceptable range is maximized. Thus task reservations may be significantly smaller than the worst case values, with jobs missing deadline having high likelihood of not causing performance degradation.

2.2 Earliest Deadline as Late as Possible

The problem of response time minimization while maintaining deadline guarantees for RT tasks have been addressed before. If the response time sensitive tasks are scheduled in FIFO order, then either Earliest Deadline as Late as possible (EDL) [CC89] or aperiodic deadline assignment [BS99] provides optimal solution. Both are based on the notion of delaying RT jobs to the latest possible instant such that they can still be scheduled without missing their deadlines. Dual priority [DW95] extends this notion of delaying RT jobs to static priority systems. By delaying RT jobs, the other jobs (TS jobs) in the system get allocated earlier thereby reducing their response times. But the drawback with this approach is that if the FIFO restriction on response time sensitive tasks is removed, then delaying RT jobs for any response time sensitive job is no longer optimal. In particular, it may happen that a relatively response time insensitive task may delay the RT jobs, and a more sensitive task may be blocked by RT jobs which cannot be preempted because they are already maximally delayed. In this scenario, the more sensitive TS job is blocked for the duration in which RT jobs are active. Also this blocking time may be arbitrarily long. While actual EDL schedule is calculated for hyper-period based on worst case execution time

requirement estimate, we use a coarse approximation to EDL schedule, which would be described in the next chapter. From hereon, EDL would refer to this approximation.

In the following sections we look at how current work in real-time scheduling handles requirement variability. But first we begin by giving a brief introduction to Generalize Processor Sharing (GPS) [PG93] model, because in the following sections we would use this model of processor sharing to analytically analyze our proposals.

2.3 Generalized Processor Sharing

GPS [PG93] is resource sharing model which is frequently used in network flow scheduling. The basic idea is that the tasks which are referred to as flows, can be concurrently allocated a fraction of processing capacity as if they were executing on a separate processor with that capacity. So for example, consider two tasks. Each is given half fraction of the processor. So it is as if, both of them were executing on a different processor with processing capacity half of the original processor.

On sequential processors, GPS can be optimally invoked using quantum based algorithms like Earliest Eligible Virtual Deadline First (EEVDF) algorithm [SAWJ$^+$96]. In EEVDF, each task is characterized by a positive integer w_i, called its weight, and gets $\frac{w_i}{\sum w_j}$ fraction of processing capacity at any time. $\sum w_j$ represents the sum of weights of all active tasks. EEVDF guarantees that the allocation to task with weight w_i, lies within $\int_0^t \frac{w_i}{\sum_{j \in \mathcal{A}(t)} w_j} \mathrm{d}t \pm q$, where q is the quantum size. Here $\mathcal{A}(t)$ represents the set of active tasks at time t, and it is assumed that task i was active in the interval from 0 till t.

If the set of active tasks is constant, and the sum of weights is one, then the relation can be expressed simply by saying that allocation till time t is between $w_i * t \pm q$. Each task is allocated a quantum q units of computation at a time, and EEVDF determines the order in which the jobs are executed, giving the allocation guarantees mentioned above.

2.3.1 Constant Bandwidth Server and Total Bandwidth Server

Constant bandwidth server (CBS) proposed by Abeni et al [AB98a], is characterized by budget c_i, and a tuple (Q_i, T_i), where Q_i represents the maximum budget and T_i represents the time period. The server utilization U_i is given by Q_i/T_i.

Whenever a job arrives it is assigned deadline equal to the current server deadline. And is run till its budget does not run out. When the server runs out of its budget, it is recharged to the maximum value Q_i, and its

deadline is advanced by T_i, in effect maintaining utilization of U_i over any interval. At any moment the server with the earliest deadline is executed.

A constant bandwidth server decouples the notion of jobs and reservation. That is, the task is provided a constant reservation of U_i, and the allocation granularity is determined by the server period P_i. If the task period is equal to the server period, then CBS is similar to a periodic task with period P_i and worst case requirement Q_i. In case of mismatch, the allocation guarantees are provided at every P_i time units, that is during first P_i time units the task is guaranteed cumulative allocation of Q_i computation units and so on. The smaller the server period P_i, the better the allocation guarantee.

A Total Bandwidth server (TBS) proposed by [SBS95] is different from a constant bandwidth server in the following way. Suppose a job arrives for a TBS, then it is assigned deadline which is maximum of previous job response time and previous server deadline plus the term c_i/U_i, where c_i is the worst case execution requirement of arriving job and U_i is the TBS bandwidth. Note that if a job requires greater than c_i computation units than it gets allocated at rate higher than U_i. This is because, it misses deadline and then becomes the earliest deadline task with its deadline in the past. CBS overcomes this problem by allocating the task at a constant rate irrespective of other factors.

But both of them suffer from the problem that they allocate for the Worst case requirement. That is, even though most of jobs may require substantially less computation, they are allocated at rate which is determined by the worst case requirement. So for example, consider a task which requires x units of computation 99% of time and $2x$ units of computation for the remaining 1% of jobs, still it is allocated at rate $2x/P$, where P is the period. Thus, over-allocating by twice for 99% of the jobs.

2.3.2 Slack Reclamation

If the task's requirement are variable then frequently jobs would finish without consuming the full allocation provided to them, leading to slack. Then, it is the job of the scheduler to distribute this extra available computation time amongst the other jobs in the system. Many slack reclamation schemes have been proposed [LB00], [BBB04] etc. Slack reclamation is a passive scheme, that is, once the slack appears the scheduler distributes it. In this work we do not address the issue of handling slack, instead we address the issue of pro-actively adjusting task schedule according to the probability distribution of its execution requirement such that other tasks in the system get greater processor share while the variable requirement real-time task is still active.

The goals of our approach are similar to that of dual priority scheduling discussed in next section.

2.4 Modeling Task Criticalness

The prevalent measure of criticalness of a task is deadline miss ratio. For example, a task with acceptable deadline miss ratio ρ of 0.01 implies 1% of deadline may be missed. Note that usually there is no specification of the scheduling behavior in case of deadline miss. More elaborate guarantees, like in any k consecutive deadlines atmost m deadline may be missed have been proposed [RH95].

While these schemes reduce the utilization requirement of tasks, by allowing some jobs to miss deadline. The scheduling is still done for the boundary case value. That is, consider a task with 90% probability of requiring x computation units and 9% probability of requiring $2x$ computation units and 1% probability of requiring $3x$ computation units.

Now to attain a deadline miss ratio of 1%, each job would have to be allocated atleast $2x$ units of computation. Thus even though 90% of the time the computation requirement is x units, each job is allocated for $2x$ units of computation. Thus even though the reservation requirement for the task is decreased, jobs are still overallocated in most cases.

Usually, the response time of jobs missing deadline are not considered. Why is response time important ? As discussed earlier, to increase average utilization of a system containing variable requirement tasks, implies overloading. That is there may be intervals where jobs miss deadline. But by keeping these intervals rare, the likelihood of any serious performance loss can be minimized.

Consider a sensor network, where information of certain event needs to reach a monitoring station. Now there may be a critical deadline, which gives the monitoring station enough time to provide the best response for the event. But what if the event notification process misses deadline. In that scenario, the earliest the notification reaches the monitoring system, the better.

Or consider a media playback application, there is no clear value of deadline, instead a range of response times which may be acceptable. That is a frame missing deadline by 1-5 ms does not deteriorate performance, and may not even be noticed by a user. Note that at frame rate of 25fps, frames are shown at 40ms interval. So if a frame is shown 45ms after previous frame there is little performance loss if any. Also note that even if 50% of frames miss deadline and finish by say 41ms, there is little performance loss. So deadline miss ratio is a useful performance indicator but the response time in case of deadline miss is also an important factor.

2.4.1 TUFs and Imprecise Computation

While for hard real-time tasks, even a single deadline miss may be critical, there is a large class of tasks for which occasional deadline misses are not critical. A general way to cover the task deadline characteristics is to use Time Utility Functions (TUF) proposed by Jensen et al [JLT85]. The basic idea behind TUF is to specify the utility of finishing by certain time for a job for all possible response times. Hard real-time task can then be represented as having maximum utility for response times less than deadline and maximum negative utility otherwise. Non critical tasks may have different utility functions where there may be some positive utility for response times greater than deadline.

While this is a general approach, practically it is not easy to associate utility with tasks. Consider for example a multimedia task. Specifying a utility function for frame decoding task deadline is not easy. Even if some such function is specified, it is not clear how this will compare with utility function of rest of the tasks. For example, consider a media decoding task and a web-server task. There is no single right way to assign TUF to these two different tasks. In some cases, the user might desire good media playback at expense of web-server task. In other cases, the user might want to strike a balance between the two tasks, but expressing this in terms of utility functions is not straightforward.

Jane Liu et al [LLS$^+$91] considered imprecise computation or reward based scheduling model. A task is characterized by a mandatory part denoted by m_i and optional part denoted by o_i. The scheduling goal is to finish the mandatory part before deadline and finish maximum part of the optional computation. Reward is associated with the amount of optional computation done. That is, the optional computation done for a job may be any value between 0 and o_i, where o_i is the maximum computation requirement of optional part. This model is different from Time Utility Functions (TUF) in the sense that the optional part has no concept of response time associated with it. That is the reward is just a function of how much of the optional part is done. This simplifies the analysis for maximizing rewards and for some reward functions it can be easily calculated by solving a linear optimization problem. This solution was proposed by Aydin et al [AMMMA01]. The solution requires knowledge of amount of slack available in hyper period (LCM of all task periods) interval. For variable requirement tasks this value is not known. Because the slack in a hyperperiod is not a discrete value but a random variable, thus probability is attached to each value.

The other and more significant difference of our work from imprecise computation model is that, for us the response time of job is important and not the amount of computation done for the optional part. That is, in

Task	Worst Case Utilization	Average Case Utilization	Boundary value	ρ at Boundary value
A	0.7	0.3	0.5	5%
B	0.7	0.3	0.5	5%
Cumulative	1.4	0.6	1.0	10%

Table 2.1: Task setup

imprecise model of computation the optional part can be done anytime to obtain reward, but in our model the goal is to minimize response time of optional computation. Also in our model the optional computation is not exactly optional rather it has low priority than the computation of jobs whose deadline is in future.

The following discussion explains our model. To maximize average processor utilization, variable requirement resilient tasks would often have worst case processor utilization greater than 1. Now the performance of resilient tasks is not only determined by number of deadlines missed but also by the amount of time by which deadlines are missed. So minimizing the deadline miss time becomes an unimportant performance criterion and indicator. Current scheduling schemes focus on providing deadline guarantees and no action is taken to minimize response time in case of deadline misses. The following example illustrates this problem.

Consider two tasks tasks A and B with variable execution requirement and unit period. Let both have worst case requirement of 0.7 and mean requirement of 0.3. This implies cumulative worst case utilization is $0.7/1 + 0.7/1 = 1.4$ (since period is 1). And cumulative mean utilization is 0.6. Clearly, running a single task on the processor leads to gross underutilization of resources. So what can be done ?

Suppose the minimum requirement of these tasks 0.5 computation units, which gives a deadline miss ratio of 5%. So both these tasks can be scheduled concurrently while each incurring 5% deadline misses.

Let j_i^A represent the i^{th} job of A and j_i^B represent the i^{th} job of B. Consider the following scenario. At time 0, j_1^A and j_1^B arrive with requirement 0.6 and 0.5 respectively. Since the period is 1 time unit, so on unit speed processor only 1 unit of computation is finished till time 1. Since both tasks are allocated 0.5 fraction of processing power hence j_1^B has 0.1 unit of computation remaining and suffers a deadline miss.

Now at time 1, j_2^A and j_2^B arrive say with execution requirement 0.3 each. Now the two tasks together reserve $0.5 + 0.5 = 1$ full fraction of processor capacity so the pending 0.1 units of j_1^B would have to wait for either of j_2^A or j_2^B to finish. This is because if it is serviced before the new jobs then the new jobs may miss deadline even if both the new jobs required less than 0.5 computation, violating the deadline guarantee. Now at 0.5 processor share, 0.3 requirement job finishes in $0.3/0.5 = 0.6$ time units. Thus, j_1^B misses deadline by 0.7 time units.

But if the job requirements j_2^A and j_2^B are known in advance then two things can be done. One approach is

the dual priority approach which would run the second jobs of the tasks starting at time 1.4 with full priority, which would let them finish on deadline, and from time 1.0 till time 1.4 the jobs are run at lowest priority, so the pending computation of j_1^B gets serviced between time 1.0 and 1.1 giving deadline miss time of 0.1 units.

Another approach is to allocate 0.6 fraction of processor to the real-time tasks, thereby giving the pending computation of job j_1^B 0.4 fraction of processor. This implies j_1^B misses deadline by $0.1/0.4 = 0.25$ time units.

While the dual priority approach gives the best solution it necessitates a priori information of exact execution requirement information, which may not be available. The other issues with dual priority scheduling have also been pointed out. Basically, in dual priority scheduling there are periods when real-time tasks run at lowest priority and periods when they run at highest priority, thus the service time is dependent upon arrival time. Also in case of unaligned periods and large number of tasks, calculating the time at which real-time tasks have to be promoted becomes complicated and would require timing analysis. This leaves us with the solution of running the real-time tasks at reservation rate such that they finish on deadline. Using this approach the real-time tasks can be treated independently and simple utilization based feasibility analysis determines schedulability.

But to attain this solution, the knowledge of execution requirement is required beforehand, which is hard to come by. In the following sections we look at problems with prediction execution requirement.

2.5 Predicting Execution Requirement

Predicting execution requirement of a piece of code is a difficult problem. First, for arbitrary piece of code it is not possible (Halting problem). Second, even for relatively simple code, the actual execution path information (if conditions), and time spent in loops may not be predictable beforehand. Though, upper bounds may be used, but they give the worst case values, which is not what we set out to find out.

Even for straight line piece of code, the actual execution time on modern processor depends on a host of factors like cache state, pipeline state, branch predictions etc. This makes the problem of predicting execution requirement difficult for applications.

Here it is important to mention that there exists some heuristics for specific applications, specifically MPG decoding. MPEG stream is composed of Intra (I), Predicted (P), Bidirectional (B) frames. I frames are most computationally intensive, P frames lesser and B frames the least. These frames usually occur in a pattern like IBBPBB called Group of Pictures (GOP). Bavier et. al. [BMP98] showed that there is a high degree of correlation between frame size and decoding time for each frame types. That is, the frame size and computation

time required for I frames have a linear relation and same holds for P and B frames too. But this approach is highly encoding dependent and may not hold for different encoding standards. Also, to implement such a scheme, the applications would need to be changed to appropriately communicate the execution requirement to the operating system.

But the prime problem is that for most other applications, there are no such heuristics available. This leads us to explore other ways to handle requirement variability. One of the more discussed about ways is feedback-control scheduling, which is discussed in next section.

2.6 Feedback Scheduling

As the name suggests feedback scheduling is a reactive mechanism that responds to changes in *observed value* through *actuator*, which permits to apply feedback action. So for example to minimize reservation for any job, the *observed value* is actual reservation required, which is available after job completes execution and the *actuator* controls the reservation provided for the next job. Most feedback theory is based on linear relation between actuator and observed value. Non linear feedback-control is not a well understood problem. Thus the feedback-control mechanism relies on **trends** in execution cost. That is, if there is an increase in execution requirement for the latest job, then feedback would predict increase in execution requirement for the next job and so on. How much increase is predicted is dependent upon choice of feedback function. The underlying theory relies on convergence (usually exponential), that is successive approximations yield better results. But this underlying assumption does not hold for predicting a noisy signal like execution cost, where there is little linear trend in execution cost variation.

This approach has the following major drawbacks. First, there is an assumption that the job execution requirements are dependent. Second, for feedback control, it is assumed that the dependence can be expressed in terms of linear transfer functions. This is because non-linear feedback control is cumbersome, and closed form solutions may not exist. Third, even if the job requirements are assumed to be dependent, feedback control does not provide a way to capture that dependence, but relies on ad hoc selection of feedback control function to model the relation.

In MPEG decoding there are frequent computation spikes due to the MPEG encoding structure containing I,B and P frames, occurring in $IB \ldots BPB \ldots B$ order, i.e. I and P frames are surrounded by multiple B frames. I frames have substantially larger computation requirement than B or P frames. To decode I frames

before deadline, a reactive mechanism like feedback would have to allocate for them always, because linear feedback-control cannot predict/accommodate spikes in computation requirement.

In this regard, Abeni et al [APLW02] analyzed performance of feedback-control scheduling for MPEG task. They came up with the notion of fast and slow feedback-controller which basically determines the reaction speed of feedback. Thus a fast controller reacts quickly to changes while a slow controller reacts slowly. As pointed out earlier, reacting fast does not help in case of noisy signal. A slow controller works much less same as worst case reservation, but over reserves for low requirement B frames and under-allocates for I frames. In fact there is a trade-off, that is if the feedback controller tries to optimize for B frames (i.e. it needs to react fast to decrease in execution cost), it loses out on I frames, the execution spike following B frames. And a feedback controller that optimizes for I frame (i.e. react slowly to the multiple low requirement contiguous B frames), ends up over-allocating for the B frames.

In this work, we show that without requiring dependence between execution requirement of jobs, the reservation rate requirement can be minimized nearly to that achieved by an ideal clairvoyant scheduler, by just using the probability distribution of execution requirements.

2.7 Probabilistic Analysis vs Probabilistic Scheduling

But before discussing that, we bring to attention the field of probabilistic analysis of schedule of variable requirement tasks. The probabilistic analysis approach entails using well known algorithms like Rate Monotonic (RM) or EDF and estimate system performance like deadline miss probability and the response time distribution of variable requirement tasks under the given algorithm. The representative example of this approach is Real Time Queuing Theory [Leh97]. The goal of this approach is to probabilistically quantify performance of algorithms like FIFO and EDF. In this work, we use the probability distribution information to construct schedule that satisfies certain constraints, like guaranteeing bounded deadline miss ratio and minimizing response time in case of deadline miss.

Tia et al [TDS+95] proposed semi-periodic task model for scheduling tasks whose jobs have highly varying execution times. They extended time demand analysis [LSD89] for static priority systems to consider execution times as random variables instead of fixed values. In such a scenario, the cumulative execution requirement of random variables in not their sum but a convolution. This is because the probability distribution of sum of random variables is their convolution. By calculating this convolution, the distribution of response time of jobs can be

calculated and using this probability distribution, the probability that a job will miss deadline can be calculated. Diaz et al [DGK+02] proposed using Discrete Time Markov Chains (DTMC), to model priority based systems with variable requirement tasks. They showed that the response time distribution can be determined by analyzing this DTMC. Stochastic Rate Monotonic Scheduling (SRMS) proposed by Atlas and Bestavros (1998) [AB98b] provides statistical deadline guarantees to schedule periodic tasks using Rate Monotonic (RM) algorithm. It uses approach similar to timing analysis to bound the amount of interference that a task at priority i can receive from all the higher priority tasks. By finding the cumulative demand distribution of higher priority tasks and task i, the probability that demand exceeds the capacity can be calculated. This probability gives the probability that task i misses deadline. To provide statistical guarantees, the value of allocation needed to limit the deadline miss probability to a given value can be calculated. For every task the minimum such value is calculated. Thus each task reserves for minimum utilization that would guarantee given deadline miss probability.

The goal of these approaches is again to probabilistically quantify performance of algorithms like Rate Monotonic (RM) which are essentially based on boundary or worst case values. That is the reservation or the utilization value chosen for a task is basically the boundary value that would give guarantee that deadline miss probability is not greater than the specified value. But choosing this boundary value still leaves space for improvement. For example, the average requirement of a job may be substantially less than the chosen boundary value, and scheduling should use this fact to improve performance like minimizing response time of tasks.

2.8 Requirement Variability and Dynamic Voltage Scaling

Processor power consumption is proportional to square of voltage multiplied by frequency i.e. $E \propto V^2 * f$. Also the processor frequency is proportional to voltage, giving $V \propto f$. So the energy consumption rate of a processor can be written as $E = K * f^3$, where K is a constant of proportionality. To finish x units of computation, the processor consumes $E * x/f$ energy, which from the derived relation is equal to $f^2 * x$.

For variable requirement tasks, running the processor at a constant speed such that the task finishes on deadline in worst case, is not efficient. For example, if the job required half of its worst case requirement, then running it at half the worst case speed is sufficient to meet the deadline and in that case, energy consumption is reduced by 1/4.

The problem of processor energy consumption has received much attention but we focus on a specific approach proposed by Lorch et al [LS04] called Processor Acceleration to Conserve Energy (PACE). The basic

idea behind this approach is that instead of predicting execution requirement of job, the schedule is calculated based on the probability distribution of execution requirement such that the expected energy consumption is minimized. Though this approach was proposed for a system containing a single task, subsequently heuristics were proposed for extending this approach to multiple task systems [YN04] [YN03]. The extension to multiple task system is done by dividing the execution time amongst the tasks in the ratio of their worst case requirement. And then schedule is calculated for each task independently as in the single task case proposed by Lorch et al [LS04].

For this work, we do not go into details of Dynamic Voltage Scaling and processor energy considerations. Instead our focus is on the approach of using probability distribution to minimize the expected value of energy consumed. Note that the nothing is assumed about the actual per job execution requirement except for the fact that the jobs' execution requirement has the same distribution, which is a pretty general assumption. So this approach does not suffer from the drawbacks of predictive schemes like feedback which rely on knowledge of interdependence between execution requirement of jobs.

This is an important concept. The probability distribution of execution requirement is a information which can be easily made available in practical systems. Its usefulness has already been shown in processor energy conservation schemes. Through this work, we hope to bring out the usefulness of probability distribution of execution requirement in real-time scheduling of variable requirement tasks.

Specifically, we assume a task model where deadline misses may not lead to total performance loss, instead if the response time is withing some acceptable range, then the there is little or no performance loss. This task model allows for specifying the worst performance level, and then optimizing for the average case system performance. The importance of our work lies in the fact that the real-time scheduling of variable requirement tasks has largely been based on worst case or boundary values, without taking the actual variability into account. So while the system is designed for worst case guarantees its average case performance is not optimized for. But this was not required in task models which assumed that missing deadlines is critical. As we move into task model, where deadlines are no longer concrete instead fuzzy values, that is under some circumstances, deadline miss by small amount may be accommodated by the system, factors like deadline miss time become important performance criterion. And using the probability distribution information of execution requirement is a practical and effective choice to guide efficient scheduling of variable requirement tasks. In the following sections the approach is explained.

CHAPTER 3
Coscheduling Real-time and Response Time Sensitive Tasks

3.1 Motivation - Single RT Media Decoding Task System

We start our discussion of SPS with an example. Consider a system with one variable requirement periodic RT task, and other non-RT tasks. Assume that at any instant at least one non-RT task is active.

A periodic RT task τ is characterized by period P and a new job arrives every P time units. Any job may have a worst case execution time requirement of C, which is referred to as its WCET. Let χ be a random variable denoting the execution time of a job.

For example, for a MPEG decoding task, the period P is 40ms (25 fps), the WCET C is 24ms and the average execution time requirement $E[\chi]$ is 10ms. These values were obtained by counting CPU cycles required to decode the frames in Star Wars movie trailer using $mpeg\_play$. As can be seen the execution time requirement for the media decoding task is highly variable, with the worst case requirement C being more than twice the mean requirement of $E[\chi]$.

We look at following scheduling approaches to schedule this RT task.

- Priority - The RT task gets priority over non-RT tasks.
- GPS - This algorithm is derived from Generalized Processor Sharing (GPS) [PG93]. The RT task gets a constant processor share given by its worst case utilization
- EDL - Earliest Deadline as Late as possible (EDL) [CC89]. The RT task is delayed as much as possible such that it still finishes by its deadline.

We compare these algorithms using the measures $A(t)$ and $s(t)$.

- $s(t)$ denotes the processor share of a RT task as a function of time
- $A(t)$ is the cumulative allocation to other tasks in the system until time t (in the interval [0,t], assuming RT tasks arrived at time 0), and is given by $(t - \int_0^t s(t)\mathrm{d}t)$.

3.1.1 Defining $s(t)$

The $s(t)$ function is defined in terms of a function $g(.)$ which represents the processor share for a job of the corresponding RT task as a function of time duration since the job's arrival. So function $g(x)$ is defined for $0 \leq x \leq P$. Also since $g(.)$ represents processor share, its value lies between 0 and 1.

Formally,

Definition

$$s(t) = \begin{cases} g(t - \lfloor \frac{t}{P} \rfloor P) & if\ task\ active \\ 0 & otherwise \end{cases}$$

Note that for a RT task arriving at time 0 and with period P, $(t - \lfloor \frac{t}{P} \rfloor P)$ represents the time duration since arrival of the job active at time t. From hereon we would represent the arrival time of job active at time t as $a(t)$ and $a(t) = \lfloor \frac{t}{P} \rfloor P$. Furthermore function $g(.)$ should satisfy the following property,

$$\int_0^P g(x)\mathrm{d}x \geq C$$

That is, the job should finish C execution time units on or before its deadline which is P time units after its arrival.

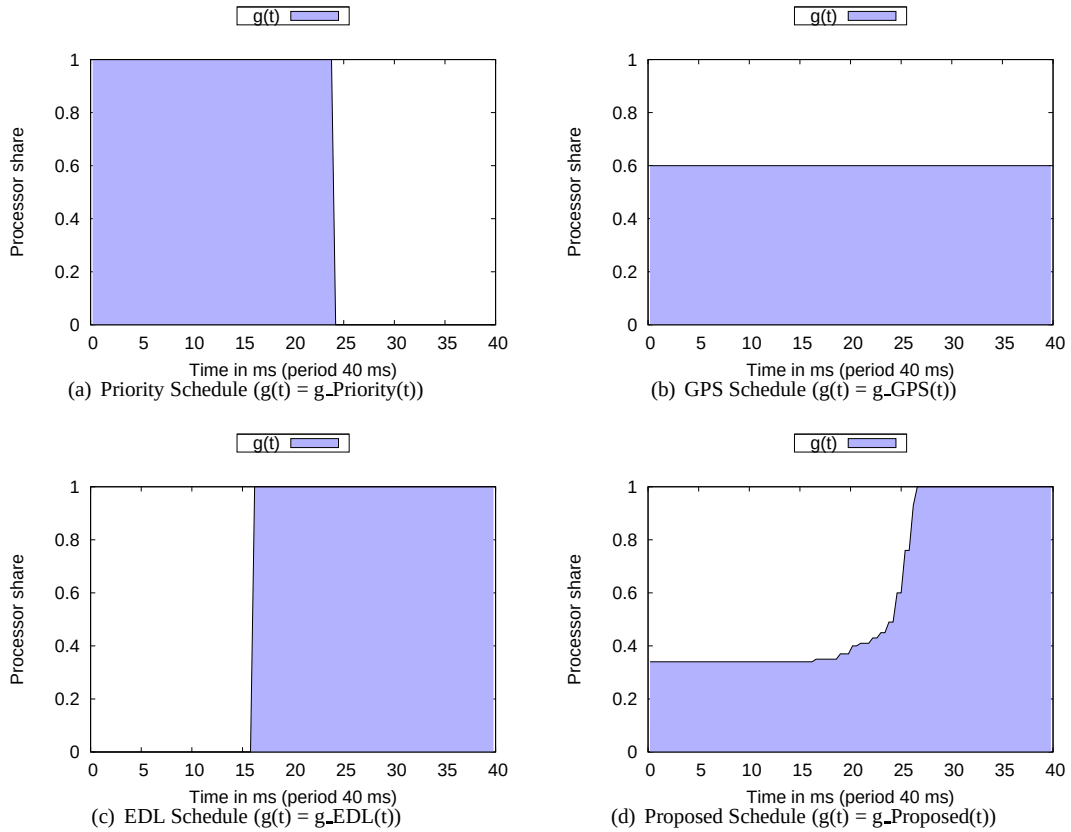
The function g(t) where (t is between 0 and P) for the above mentioned scheduling algorithms can be written as follows. We represent the $g(.)$ function for a particular scheduling algorithm as $g\_$ followed by the scheduling algorithm name to differentiate between various scheduling algorithms. Figure 3.1 illustrates the $g(.)$ functions for Priority GPS and EDL algorithms as well as the Proposed algorithm. The shaded area represent the allocation to the RT job, while the height of the shaded area represents the actual processor share of the RT job at that time. The curve $g\_Proposed(.)$ is unique in the sense that it is continuously varying and in the following sections we see how to determine the shape of curve $g\_Proposed(.)$ based upon given optimization criterion.

- Priority

$$g\_Priority(t) = 1$$

- GPS

$$g\_GPS(t) = C/P$$



(a) Priority Schedule (g(t) = g_Priority(t))
(b) GPS Schedule (g(t) = g_GPS(t))
(c) EDL Schedule (g(t) = g_EDL(t))
(d) Proposed Schedule (g(t) = g_Proposed(t))

Figure 3.1: *These figures show a job schedule for MPEG task with period 40ms, WCET 24 ms and mean requirement of 10ms. The filled curves g(t) shows the processor share of any RT job. The shaded area represents the allocation to the RT job as function of the time duration since its arrival. The height of curve $g(t)$ represents the actual processor share of the RT job as a function of its time since arrival. The total work done under each scheduling policy is equal to the WCET (24ms). While the Priority, GPS and EDL g(t) functions are constant, the proposed g(.) function ($g\_Proposed(.)$) varies with job progress. In the following sections we describe how $g\_Proposed(.)$ is calculated and its properties.*

- EDL

$$g\_EDL(t) = \begin{cases} 1 & if\ t \geq P - C \\ 0 & otherwise \end{cases}$$

Lemma 3.1.1 *For any given t between 0 and P,*

$$\int_0^t g\_EDL(t)\mathrm{d}t \leq \int_0^t g\_GPS(t)\mathrm{d}t \leq \int_0^t g\_Priority(t)\mathrm{d}t$$

Proof Under Priority, the RT job is scheduled as soon as it arrives and is given the full processor share (g_Priority(t)=1)

until it finishes. Under GPS, the RT job is given a constant processor share of C/P (≤ 1) from its arrival until it finishes. Hence $\int_0^t g\_GPS(t)\mathrm{d}t \leq \int_0^t g\_Priority(t)\mathrm{d}t$. Under EDL, the RT job starts getting allocation only when the time remaining is just enough to finish the job on deadline under its worst case execution time requirement. Hence, $\int_0^t g\_EDL(t)\mathrm{d}t \leq \int_0^t g\_GPS(t)\mathrm{d}t$ □

Lemma 3.1.2 *For any t,*

$$\begin{aligned} Priority\ A(t) &\leq\ GPS\ A(t) \\ &\leq\ EDL\ A(t) \end{aligned}$$

Proof By definition, for a single RT media task and a set of non-RT tasks,

$$A(t) = t - \int_0^t s(t)\mathrm{d}t$$

Break this integral until the arrival time of latest job (or the deadline of the latest job finished),

$$\begin{aligned} A(t) = (a(t) - \int_0^{a(t)} s(t)\mathrm{d}t) \\ +((t - a(t)) - \int_{a(t)}^t s(t)\mathrm{d}t) \end{aligned}$$

Now the cumulative allocation to the RT task until the arrival time of latest job $a(t)$ is same under any scheduling algorithm where all jobs meet their deadline, hence the first term is same for Priority, GPS and EDL.

The difference is caused by the second term. Now note the second term can be written as

$$((t - a(t)) - \int_{a(t)}^t s(t)\mathrm{d}t)$$

The negative term $\int_{a(t)}^t s(t)\mathrm{d}t$ represents the allocation to the RT job as function of time. From Lemma 3.1.1, the allocation to RT job as a function of time since its arrival is greatest under Priority, lesser under GPS and least under EDL. So the cumulative term $((t - a(t)) - \int_{a(t)}^t s(t)\mathrm{d}t)$ is least under Priority, greater under GPS and greatest under EDL. □

Lemma 3.1.3 *For any RT job, let max s(.) denote the maximum value of s(t).*

$$GPS \ \max s(.) \leq \ Priority \ \max s(.), \ EDL \ \max s(.)$$

Proof Note that under GPS, the maximum value of $s(t)$ is ($C/P \leq 1$) , while it is 1 under Priority and EDL. This is because under EDL and Priority scheduling, the RT job is assigned full processor share when it is scheduled. □

3.1.2 Understanding A(t), s(t) and g(.)

Both $A(t)$ and $s(t)$ are important because while $A(t)$ denotes the cumulative allocation to other tasks in the system in the interval $[0, t]$, $(1 - s(t))$ denotes the instantaneous processor share available to other tasks in the system at time t.

As pointed out earlier, $A(t)$ is same for all scheduling algorithms on job deadlines because the allocation to RT task by any the deadline is same under any scheduling algorithm. The variation is caused when the job is active. EDL delays the RT job such that the other tasks in the system get scheduled before the RT task, and hence maximizes $A(t)$.

$s(t)$ is the processor share of RT task with time. A good scheduling algorithm would be one for which the value of $s(t)$ is small for all values of t. GPS keeps the value of $s(t)$ constant at the worst case utilization of the RT task whenever it is active. Under EDL and Priority, $s(t)$ is 1 while the RT job is active.

$s(t)$ is based on function $g(.)$ which gives the processor share for a RT job as a function of time duration since its arrival (for RT task arriving at time 0 and job arriving every P time units, $t - a(t)$). The g(.) function is dependent upon the scheduling algorithm used.

3.1.3 Problem with Priority

If an RT task is given priority over other tasks in the system, then the other tasks are blocked whenever the RT task is active. This gives the worst value of $A(t)$ (minimum) amongst all algorithms guaranteeing that the RT task does not miss any deadline. Also, the value of $(1 - s(t))$ is 0 while the RT task is active and hence this algorithm does not perform well on both measures.

3.1.4 Problem with EDL

Even though $A(t)$ is maximized by EDL, the value of $(1 - s(t))$ may be 0 while the RT job is active, that is the other tasks arriving when RT job is active are blocked until RT job finishes as in Priority. Furthermore, this blocking time may be arbitrarily long.

For example, consider a RT task with period 1000ms and execution time requirement of 500ms. Let there be non-RT tasks active at any time. In such a scenario, under Priority and EDL, the RT task is active for 500ms, thereby blocking all non-RT tasks for the entire duration of 500ms. The period of the RT task can be chosen arbitrarily long, thereby leading to blocking of non-RT tasks for arbitrarily long intervals.

3.1.5 Problem with GPS

The advantage of GPS is that the processor share available to other tasks in the system is at least $(1 - C/P)$ at any time. The problem with GPS is that the processor is reserved based on the worst case execution time requirement of the RT task. Thus, even though on average the media decoding task requires just under 10ms of execution time, any job is given a processor share of 24/40 = 0.6, which is more than twice the mean processor share requirement of 10/40 = 0.25.

3.2 Probability and Scheduling - Stochastic Processor Sharing

In this section, we start with discussion for a system with single RT task.

As pointed out above, variability in execution time requirement of RT tasks poses unique challenges. First, the scheduling algorithm should provide deadline/allocation guarantees to the RT tasks. Second, to provide guarantees, a non clairvoyant scheduler schedules each job of the RT task assuming that it would require its worst case execution time.

In this section we describe how the variability in execution time requirement can be efficiently handled. We first introduce the notion of expected processor share, $E[s(t)]$. This is the key notion in our analysis. Its importance lies in the fact that for variable requirement RT tasks, the value $s(t)$ is dependent upon whether the RT job has finished or not. If the RT job has finished then it does not require any processor share, but if it is active then it is allocated processor share given by its corresponding $g(.)$ function. So, while the value $s(t)$ at a time t can be thought of as a random variable which may be either 0 or $g(.)$ depending on whether the RT job has finished or not. If the probability distribution of the execution time requirement of the RT task is known, then the

probability that RT job is active after it has finished x units of execution time can be written as $\Pr[\chi > x]$, where χ is the random variable denoting the execution time requirement. Thus, $E[s(t)]$ can be expressed in terms of the $g(.)$ function and the probability distribution of the random variable. Next, we formally define $E[s(t)]$.

Definition $E[s(t)]$ represents the expected value of $s(t)$ for the RT task at time t. Formally,

$$E[s(t)] = g(t - a(t)) * \Pr[\chi > \int_0^{t-a(t)} g(x)\mathrm{d}x]$$

That is, $E[s(t)]$ at time t is the probability that the RT job is active at time t multiplied by the processor share given by its corresponding $g(.)$ function.

Let $\max E[s(.)]$ denote the maximum expected value of $s(t)$ at any time t.

In this paper, we propose the novel notion of choosing the function $g(.)$ (now we would refer to the $g(.)$ function as $g\_Proposed(.)$) for a RT task such that it satisfies the following key properties. For any t, where t is the time duration since arrival of RT job and x lies between 0 and P:

- $\int_0^P g\_Proposed(x)\mathrm{d}x \geq C$, where C is the WCET
- $g\_Proposed(.)$ is chosen such that $\max E[s(.)]$ is minimized

3.2.1 Calculating $g\_Proposed(.)$ for a Single RT Task System

In this section we show how $g\_Proposed(.)$ can be calculated which satisfies the above mentioned properties for a system containing single a RT task. At any time some non-RT task is assumed to be active.

Suppose we want to calculate a schedule where $E[s(t)] \leq K$ for any t. Since $E[s(t)]$ is a periodic function with period P, it is sufficient to enforce this relation in the interval $[0, P]$.

For t between 0 and P, $E[s(t)]$ can be written as $g\_Proposed(t)\Pr[\chi > \int_0^t g\_Proposed(x)\mathrm{d}x]$. Thus the constraints can be expressed as,

$$g\_Proposed(t)\Pr[\chi > \int_0^t g\_Proposed(x)\mathrm{d}x] \leq K$$

and,

$$\int_0^P g\_Proposed(t)\mathrm{d}t \geq C$$

This constraint enforces the fact that any job should finish atmost C execution time by its deadline.

In this section assume $0 \leq t \leq P$. Now, a schedule with $E[s(t)]$ at most K would do maximum work by any time t if $E[s(t)] = K$ for any t. This is because if $E[s(t)] \leq K$ for some time t, the the function $g\_Proposed(t)$ can be increased at that time t thereby leading to greater processor share to the RT task.

The first solution to function $g\_Proposed(.)$ can then be written recursively as

$$\begin{aligned} g\_Proposed(0) &= K/\Pr[\chi > 0] \\ g\_Proposed(t+\delta t) &= \frac{K}{\Pr[\chi > \int_0^t g\_Proposed(x)\mathrm{d}x]} \end{aligned}$$

But this may lead to $g\_Propose(t)$ outside the range $[0, 1]$ (specifically as the probability approaches 0, g_Proposed(.) approaches infinity). So we limit the value $g\_Proposed(.)$, which gives the solution to $g\_Proposed(.)$ as,

$$\begin{aligned} g\_Proposed(0) &= K/\Pr[\chi > 0] \\ g\_Proposed(t+\delta t) &= \min(1, \frac{K}{\Pr[\chi > \int_0^t g\_Proposed(x)\mathrm{d}x]}) \end{aligned}$$

Note that if $g\_Proposed(t)$ is reduced the 1 because $\frac{K}{\Pr[\chi > \int_0^t g\_Proposed(x)\mathrm{d}x]}$ is greater than 1, then $E[s(t)]$ is less than K.

So now the function $g\_Proposed(.)$ is defined in terms of a constant K which represents the maximum value of $E[s(t)]$ for this schedule at any time t. What remains is to find the minimum K for which a job of this RT tasks meets its deadline under worst case execution time requirement of C, i.e.

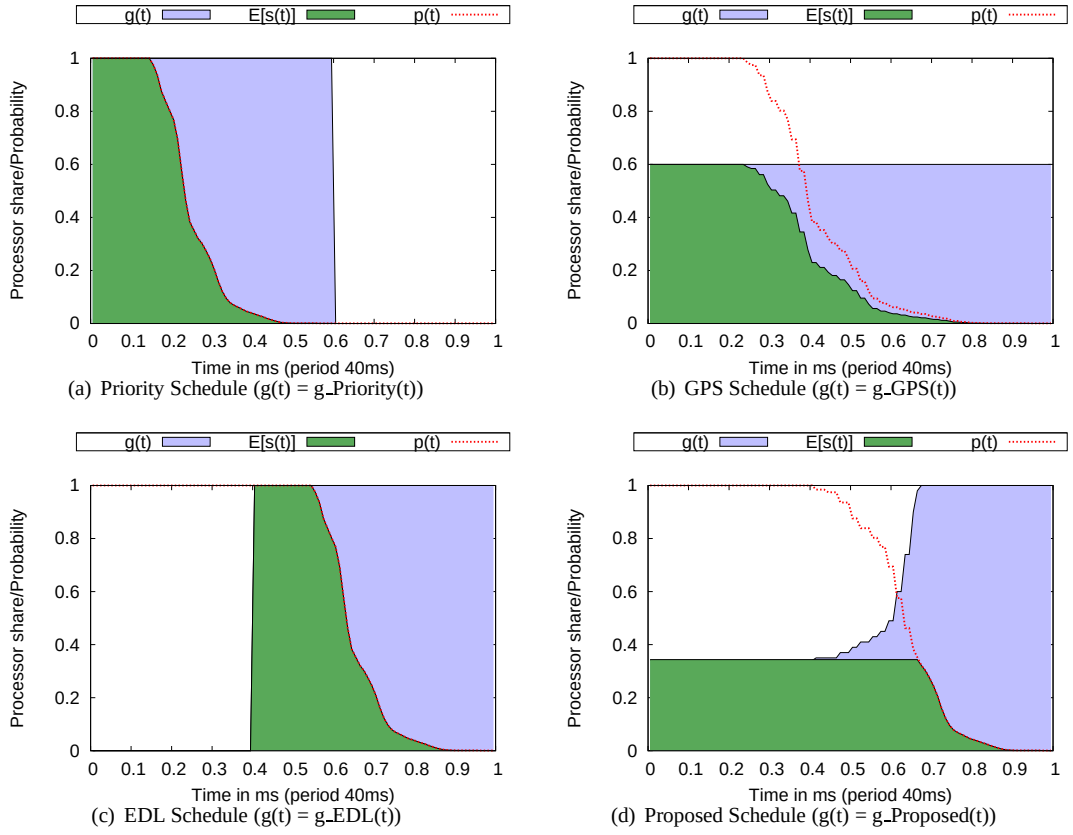
$$\text{minimum } K \; s.t. \; \int_0^P g\_Proposed(t)\mathrm{d}t \geq C$$

Now the value of K (the maximum expected processor share) lies between 0 and 1. As the value of K is increased from 0, the execution time allocation to the RT job increases. Therefore a reasonable approximation to K can be efficiently calculated using binary search on the value of K in the interval $[0, 1]$.

3.2.2 Schedule Illustration - Media Decoding Task Example

Here we present the application of proposed approach on the media decoding task example. The processor cycles used to decode Star Wars trailer using mpeg_play were calculated on a FreeBSD 4.8 machine with 800 MHz Intel Pentium III processor. The worst case execution time requirement was found to be 24ms and the mean execution time requirement was 10ms. To run the movie at 25fps, frames need to be decoded every 40ms. Thus the RT task has a period of 40ms, worst case utilization if 24/40=0.6 and mean utilization of 10/40=0.25.

The K calculated for this RT task was 0.34 which is near the mean utilization 0.25 and nearly half the worst case utilization of 0.6 (Refer Fig 3.2).



(a) Priority Schedule (g(t) = g_Priority(t))

(b) GPS Schedule (g(t) = g_GPS(t))

(c) EDL Schedule (g(t) = g_EDL(t))

(d) Proposed Schedule (g(t) = g_Proposed(t))

Figure 3.2: *Figure showing job schedules for a MPEG task with period 40ms, WCET 24 ms and mean requirement of 10ms. The filled curves $g(t)$ and $E[s(t)]$ represent the fraction of processor share given to the RT job as a function of the time duration since its arrival. While $g(t)$ curve shows the processor share function corresponding to the scheduling algorithm, $E[s(t)]$ curve shows the expected value of $s(t)$ for a job ($E[s(t)] = g(t)\Pr[\chi > \int_0^t g(x)\mathrm{d}x]$, for t in the range $[0, P]$, where χ is the random variable representing job execution times.*

3.2.3 Handling Multiple RT Tasks

The analysis so far considered a system with only a single RT task. In this section we extend the above methodology to a system with multiple RT tasks. But this extension is not trivial. The primary reason being that schedule of one RT task impacts the schedule of other RT tasks. That is, if a job of RT task i is allocated processor share given by function $s_i(t)$, then for a different RT task j, its corresponding processor share function $s_j(t)$ cannot be independent of the value of $s_i(t)$ since the total processor share available to all RT jobs is atmost 1 (full utilization).

For the case when all tasks have same period and arrival time, the analysis as done for a single task system can be used, the only difference being the random variable χ will now be $\sum_{i=0}^{n} \chi_i$ where χ_i is the random variable for the execution time of task i and there are n tasks in the system.

If the tasks have different periods then the problem gets tougher. Calculating $s_i(.)$ are dependent because they are constrained by the relation $\sum_{i=1}^{n} s_i(t) \leq 1$.

Though this is a serious problem, we propose a novel and efficient solution.

Consider a set $\mathcal{T}$ of RT tasks with n tasks. Let task $\tau_i \in \mathcal{T}$ has period P_i, , worst case execution time requirement of C_i and χ_i the random variable representing job execution time.

We form a virtual task $\tau_{virtual}$ with period as 1 time unit, worst case execution time requirement of $U = \sum_{i=1}^{n} C_i/P_i$, and the random variable representing the execution time requirement of a job as $\chi = \sum_{i=1}^{n} \chi_i/P_i$.

The schedule minimizing the maximum expected processor share is calculated for this virtual task. Let $g_{virtual}(.)$ be the resulting job function which is defined in the interval $[0, 1]$ and $\int_0^1 g_{virtual}(t)\mathrm{d}t = U$.

Now the tasks are scheduled as follows. The function $s_i(t)$ for task τ_i is defined as

$$s_i(t) = \begin{cases} g_{virtual}(\frac{t-a_i(t)}{P_i}) & if\ \tau_i\ active \\ 0 & otherwise \end{cases}$$

Note that the function $g_{virtual}(.)$ is such that for a task τ_i with period P_i, the cumulative allocation available to RT tasks while a job of task τ_i is active is $\int_0^{P_i} g_{virtual}(t/P_i)\mathrm{d}t = P_i * \int_0^1 g_{virtual}(x)\mathrm{d}x = P_i * U$ (note the integral was transformed by the substitution $x * P_i = t$).

At any moment the RT tasks are given a processor share of $s(t) = \max_{1 \leq i \leq n} s_i(t)$, and this processor share is allocated to the earliest deadline active job.

The intuition behind this approach is as follows. Suppose the tasks are given a constant processor share of

U when active ($U = \sum_{i=1}^{n} C_i/P_i$). Then all jobs meet their deadline. And in the worst case a job j may finish exactly on its deadline. In this scenario, the cumulative allocation to RT tasks from the time of arrival of j until its deadline is $U * P_i$ where P_i is the period of task corresponding task for job j. So, in this scenario if the cumulative allocation available to RT tasks from the arrival of job j until its deadline is $U * P_i$, then the job j does not miss its deadline.

Next we formally show that using the proposed scheduling approach all RT jobs meet their deadlines.

Lemma 3.2.1 *Consider a set $\mathcal{T}$ of RT tasks with n tasks. Let task $\tau_i \in \mathcal{T}$ has period P_i, worst case requirement C_i and χ_i be the random variable denoting the execution time requirement of a job. Let $g(.)$ be an an increasing function such that $\int_0^1 g(t)\mathrm{d}t = U$, where $U = \sum_{i=1}^{n} C_i/P_i$. The jobs are scheduled using preemptive EDF and the processor share at time t is given by $s(t) = \max_{1 \le i \le n} s_i(t)$, where*

$$s_i(t) = \begin{cases} g(\frac{t-a_i(t)}{P_i}) & if\ \tau_i\ active \\ 0 & otherwise \end{cases}$$

All jobs meet their deadline.

Proof For this proof, we would assume that all the RT tasks are released at time 0, and SRT task τ_i releases a job every P_i time units.

If the tasks were given a constant processor share of U then all jobs meet their deadline (preemptive EDF schedulability). Suppose all tasks are scheduled in EDF order and the processor share at time t is given by $g(\frac{t-a_i(t)}{P_i})$ i.e. $s(t) = g(\frac{t-a_i(t)}{P_i})$.

Now consider task τ_i. It can be easily seen that if $s(t) = g(\frac{t-a_i(t)}{P_i})$ then all jobs of τ_i meet their deadline. To see this consider at time kP_i,

$$\int_0^{kP_i} s(t)\mathrm{d}t = \int_0^{kP_i} g(\frac{t-a_i(t)}{P_i})\mathrm{d}t$$

This gives,

$$\int_0^{kP_i} s(t)\mathrm{d}t = \int_0^{P_i} g(\frac{t-0}{P_i})\mathrm{d}t + \cdots + \int_{(k-1)P_i}^{kP_i} g(\frac{t-(k-1)P_i}{P_i})\mathrm{d}t$$

$$\int_0^{kP_i} s(t)\mathrm{d}t = (\sum_{j=1}^{N} R_j/P_j)P_i + \cdots + (\sum_{j=1}^{N} R_j/P_j)P_i$$

$$\int_0^{kP_i} s(t)\mathrm{d}t = (\sum_{j=1}^{N} R_j/P_j)kP_i$$

Thus $\int_0^{kP_i} s(t)\mathrm{d}t$ is same as what GPS allocation for integer values of k. This implies all jobs of τ_i meet their

deadlines (remember jobs are scheduled in EDF order) while jobs of other tasks may miss their deadlines.

Similarly, using $s(t) = g(\frac{t-a_m(t)}{P_m})$ would lead to all jobs of τ_m meeting their deadlines. Now, if $s(t) = \max_{1 \leq i \leq N} g(\frac{t-a_i(t)}{P_i})$, then all RT jobs meet their deadlines.

Once a RT job is finished, it no longer requires processor share till next job of this task arrives. Thus, $s(t) = \max_{1 \leq i \leq N} s_i(t)$ would imply that all the RT jobs meet their deadlines. □

Lemma 3.2.2 *If $g_{virtual}(.)$ is used as the $g(.)$ function, then the maximum value of $E[s(t)]$ for any time t is K, where K is the maximum expected processor share of $\tau_{virtual}$.*

Proof Consider at some time t, job j of task τ_i has maximum $s_i(t)$.

Now job j is scheduled using preemptive EDF, so jobs with deadlines earlier than j are scheduled before it. From the arrival of job j until it finishes, the maximum execution time required by RT tasks is $U * P_i$, where U is the cumulative worst case utilization and P_i is period of task τ_i.

Now the cumulative utilization of RT tasks is represented as a random variable χ. The cumulative allocation to RT tasks from arrival of job j until its deadline can be approximated by this random variable χ. This is because, job j is scheduled after all earlier deadline jobs and in the worst case all the active jobs may have a deadline earlier than job j, in which case job j is scheduled last. In general, let t_a be the arrival time of j and t_f its finish time, then the cumulative utilization of RT tasks in this interval is at least $\frac{\int_{t_a}^{t_f} s_i(t)\mathrm{d}t}{P_i}$ (since during some intervals $s_i(.)$ may not be maximum). This utilization is approximately upper bounded by the cumulative utilization of all active tasks during the interval $[t_a, t_f]$ which we already represent as χ.

Coming back to the function $s_i(t)$, the expected processor share $E[s_i(t)]$ is given as,

$$E[s_i(t)] = g_{virtual}(\frac{t-a_i(t)}{P_i}) * \Pr[\chi > \frac{\int_0^{t-a_i(t)} g_{virtual}(x/P_i)\mathrm{d}x}{P_i}]$$

which gives,

$$E[s_i(t)] = g_{virtual}(\frac{t-a_i(t)}{P_i}) * \Pr[\chi > \int_0^{\frac{t-a_i(t)}{P_i}} g_{virtual}(y)\mathrm{d}y]$$

The RHS can be simply written as $g_{virtual}(z) * \Pr[\chi > \int_0^z g_{virtual}(y)\mathrm{d}y]$. And as explained before, for a single task system with task $\tau_{virtual}$ with unit period, χ as execution time, and U as worst case utilization requirement, the minimum value of K (maximum expected processor share) is calculated such that $g_{virtual}(z) * \Pr[\chi > \int_0^z g_{virtual}(y)\mathrm{d}y] \leq K$ and $\int_0^1 g_{virtual}(y)\mathrm{d}y \geq U$.

Hence $E[s_i(t)] \leq K$.

□

3.3 Performance Comparison

In this section we theoretically compare the performance of our algorithm to Priority, GPS and EDL in terms of the measures $A(t)$ and $s(t)$. $A(t)$ represents the cumulative allocation to non-RT tasks by time t and is given by $\int_0^t (1 - s(t))\mathrm{d}t$. An algorithm with greater $A(t)$ for any t provides better response time to large non-RT tasks, while an algorithm with lower maximum expected value of $s(t)$ for any t provides better instant service and hence improves responsiveness of shorter non-RT tasks.

The processor share functions for Priority, GPS, EDL and Proposed algorithm for a multiple RT task system are approximated by following $g(.)$ functions. Note that as before, we represent the $g(.)$ function for certain algorithm by prefixing $g\_$ before the algorithm name. So $g\_Priority(.)$ represents the $g(.)$ function for Priority algorithm.

Note that $0 \leq t \leq 1$, since $g(.)$ functions are assumed to be defined for unit period task. Assume as before we have n RT tasks, where the i^{th} task is represented as τ_i and its period, worst case requirement and execution time requirement are represented as P_i, C_i and χ_i respectively. For feasibility $\sum_{i=1}^{n} C_i/P_i \leq 1$. Let $U = \sum_{i=1}^{n} C_i/P_i$. Let $\chi = \sum_{i=1}^{n} \chi_i/P_i$ which is the random variable representing the combined utilization of all RT tasks.

- Priority, $g\_Priority(t) = 1$
- GPS, $g\_GPS(t) = U$
- EDL, $g\_EDL(t) = \begin{cases} 1 & if\ t \geq 1 - U \\ 0 & otherwise \end{cases}$
- Proposed, $g\_Proposed(t) = min(1, K/p(t))$, where $p(t)$ is the probability that the cumulative RT utilization is greater than $\int_0^t g\_Proposed(t)\mathrm{d}t$.

Note that for all these algorithms, $\int_0^1 g(t) \geq \sum_{i=1}^{n} C_i/P_i$ and the $g(.)$'s are increasing functions. So from Lemma 3.2.1, all these algorithms correctly schedule a given set of RT tasks. Thus all can schedule set of RT tasks.

For the GPS processor share function $g\_GPS(t) = U$ is an approximation since processor share of cumulative worst case utilizations of just the active RT tasks is sufficient to correctly schedule the RT tasks. But the

simpler formulation is assumed to simplify the proof, though it penalizes GPS in terms of the measure $A(t)$. Note that the maximum expected value of $s_i(t)$ remains unchanged.

For the EDL processor share function, a simplified approximation is again used. This approximation reduces $A(t)$ (the cumulative allocation to non-RT tasks in the system by time t) as compared to the $A(t)$ attainable using the true EDL scheme (as in Chetto and Chetto [CC89]), where the schedule calculation is done offline for the hyper-period. Despite this simpler EDL formulation, EDL still achieves better $A(t)$ than the the other three algorithms (so the simpler formulation does not skew the results). Again, the maximum expected value of $s_i(t)$ for any task remains unchanged.

Lemma 3.3.1 *For any given t between 0 and 1,*

$$\int_0^t g\_EDL(t)\mathrm{d}t \leq \int_0^t g\_Proposed(t)\mathrm{d}t$$
$$\leq \int_0^t g\_GPS(t)\mathrm{d}t \leq \int_0^t g\_Priority(t)\mathrm{d}t$$

Proof First, note that $\int_0^t g\_Proposed(t)\mathrm{d}t \leq \int_0^t g\_GPS(t)\mathrm{d}t$. This is because $g\_Proposed(t)$ is a non decreasing function and $g\_GPS(t)$ is a constant function. And $\int_0^1 g\_Proposed(t)\mathrm{d}t = \int_0^1 g\_GPS(t)\mathrm{d}t = U$. This implies that $\int_0^t g\_Proposed(t)\mathrm{d}t \leq \int_0^t g\_GPS(t)\mathrm{d}t$ for $t \leq 1$.

For $\int_0^t g\_EDL(t)\mathrm{d}t \leq \int_0^t g\_Proposed(t)\mathrm{d}t$, note that while g_EDL(t) is 0 for t less than (1-U), while $g\_Proposed(t)$ is non zero during the interval.

Thus the RT task execution is delayed for the maximum amount in EDL. However, as pointed out earlier, this leads to blocking of non-RT tasks when RT tasks are scheduled. Under the proposed algorithm, RT tasks are delayed lesser than in EDL but their schedule is determined by the execution time requirement probability distribution, thereby reducing blocking of non-RT tasks. □

Lemma 3.3.2

$$Proposed \max E[s(.)] \leq GPS \max E[s(.)]$$
$$\leq EDL/Priority \max E[s(.)]$$

Proof The maximum value of $E[s(.)]$ for Priority and EDL is $1 * \Pr[\chi > 0] = \Pr[\chi > 0]$ and for GPS, it is

$U * \Pr[\chi > 0]$, where U is the worst case cumulative utilization of RT tasks. This is attained when a RT job begins execution. As proved earlier in Lemma 3.2.2, Proposed $\max E[s(.)]$ is K.

What remains to show is that $K \leq U * \Pr[\chi > 0]$. To see this note that, $g\_Proposed(0) \leq g\_GPS(0)$. This is because while $g\_GPS(.)$ is constant and equal to U, $g\_Proposed(t)$ is a non decreasing function, so it can start with processor share less than U while still finishing U execution time in a unit sized interval. This gives, $g\_Proposed(0) * \Pr[\chi > 0] = K \leq g\_GPS(0) * \Pr[\chi > 0]$. □

Lemma 3.3.3 *The cumulative allocation to non-RT tasks follows the following relation -*

$$Priority\ A(t) \leq GPS\ A(t) \leq Proposed\ A(t) \leq EDL\ A(t)$$

Proof Under Priority, the RT tasks get full processor share whenever active and the non-RT tasks are blocked while RT tasks are active, giving the worst value of $A(t)$. Under EDL RT tasks are maximally delayed so EDL has the maximum $A(t)$ value for any t.

What remains to show is the order between GPS and proposed approach. For a single task system, the proposed approach clearly provides larger $A(t)$ for any t. For multiple RT task system, under GPS the RT tasks always get a constant processor share of U whenever active. Under the proposed approach, the RT task may get a smaller value of processor share during some intervals leading to delayed RT task execution thereby increasing $A(t)$. □

In summary the proposed scheme delays execution of RT tasks to increase allocation to non-RT tasks by any time t, but at the same time it maintains the maximum expected processor share of RT tasks at any time to be bounded by its minimum value. Thus, under the proposed scheme, larger non-RT tasks get better response times (due to better $A(t)$ than GPS and priority), and smaller non-RT tasks get better response time because of low maximum expected value of $s(t)$ at any time t.

3.4 Quantum-Based Scheduler

Here we propose a generic approach to schedule RT tasks if the $s(t)$ function is given.

In particular, we want the allocation to approximate $s(t)$ over reasonably long intervals, while over shorter intervals the allocation may be off by some value.

We propose a quantum-based algorithm that keeps the allocation to RT tasks within one quantum of the allocation given by the function $s(t)$. As the quantum size is decreased, the allocation accuracy increases. Making the quantum size too small is not efficient because at each quantum boundary, the tasks are switched which may lead to cache flushes and reading for the new task from memory, which is a slow process. Also, context-switch overhead is encountered on every task switch. Current GPOS have quantum size in the range of 10ms. Older systems had quantum size in the range of 100ms. So while the current quantum size is still larger than what we would like for our algorithm, but it is decreasing.

3.4.1 The Algorithm

Consider a system with RT tasks $\mathcal{T}$ and the function $s(t)$, that is the cumulative processor share of RT tasks is known.

The following approach is then used to map this schedule to a quantum based scheduler. First, the RT tasks are allocated execution time in discrete units each of size one quantum or q.

Each quantum allocation unit to RT tasks is characterized by its arrival time and deadline. The arrival time of a quantum is the deadline of previous quantum. The deadline of a quantum arriving at time t_a is calculated as follows. Find the value of Δ for which the cumulative allocation to RT tasks in the interval $[t_a, t_a + \Delta]$ is q time units.

Formally, find Δ such that the following equation holds.

$$\int_{t_a}^{t_a+\Delta} s(t)\mathrm{d}t \geq q$$

Then the deadline of this quantum t_d is given by $t_a + \Delta$.

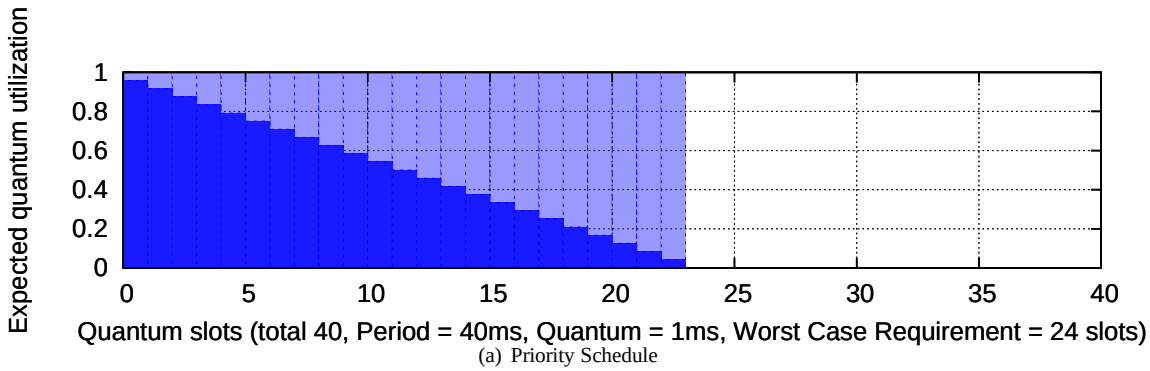
Lemma 3.4.1 *The allocation error that is the difference between allocation to RT tasks under the quantum based schedule and under a schedule which can allocate a share $s(t)$ of the processor to RT tasks at time t is at most q at any time t, where q is the quantum size.*

Proof As can be seen from the algorithm formalization, the allocation under both schedules matches at any quantum deadline. And for any time in between, the allocation difference can be at most q, the quantum size. □

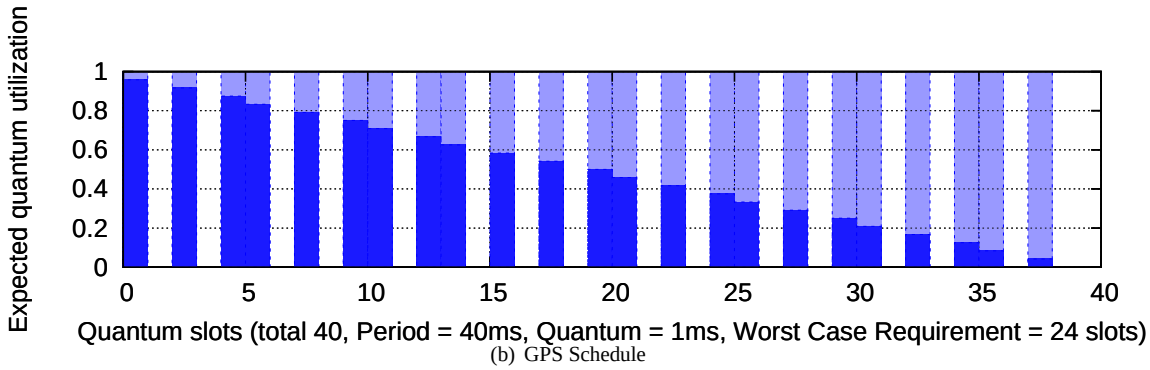
Lemma 3.4.2 *If the worst case requirement of all RT tasks is a multiple of q, the quantum size, then all deadlines are met.*

Proof Since the RT tasks are allocated in discrete quantum execution time units, and the allocation at quantum boundaries is equal to $\in_0^t s(t)\mathrm{d}t$ for any t such that t is a quantum deadline. □

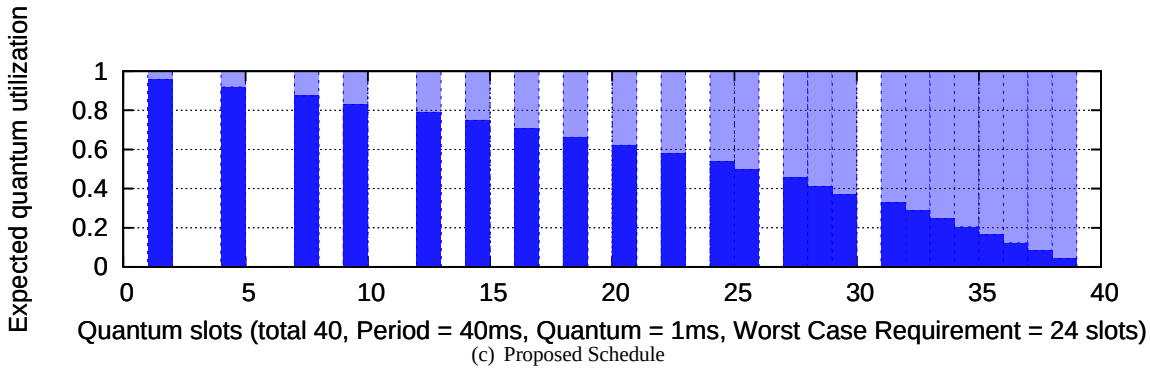
3.4.2 Simulation Results



(a) Priority Schedule



(b) GPS Schedule



(c) Proposed Schedule

Figure 3.3: *Schedule for a task with period 40ms, worst case execution time requirement of 24ms and mean execution time requirement of 12ms. The execution time is assumed to be uniformly distributed between 0 and 24ms. The quantum size is assumed to be 1ms.*

Fig 3.3 shows the quantum based schedule for a task whose requirement is uniformly distributed between 0 and 0.024. The period is assumed to be 0.040 seconds and the quantum size is $1ms$.

The first schedule is for the case when this task is given highest priority. In this case the task executes until completion when it arrives. The second schedule is for the case when the task is allocated at constant worst case rate, that is $\frac{0.024}{0.04} = 0.6$ for all quanta. Note that the mean requirement of task is 0.012. The last schedule represent the quantum based schedule for the proposed reservation scheme. Note that initially the quanta are more spaced and towards the end they are closer together. The dark shade represents the probability that the task would actually require that quantum and not finish before or in between it. Thus, though the proposed schedule has closer quanta towards the schedule end, the probability of requiring them is very small, given by the low height of the dark shaded region.

3.5 Summary

In this work we proposed the novel notion of varying the processor share of the RT tasks with their progress. What this achieves is that the RT job starts off requiring lesser processor share than its worst case utilization, and its processor share increases as the RT job progresses, and so does the probability that the RT job will finish. So while some RT jobs which require execution time requirement near the WCET may end up consuming greater processor share near their deadline, the probability that this scenario arises is less. And the function $s(t)$ is calculated using the probability distribution of execution time of RT task obtained through online profiling such that the maximum expected value of $s(t)$ for any time t is minimized.

In this work we assumed the GPS model of processor sharing. The RT jobs get a share $s(t)$, and the non-RT tasks get the remaining processor share $(1 - s(t))$. Most current processors execute tasks sequentially and use a quantum based scheduler to schedule multiple tasks concurrently. Our proposed scheduling algorithm can be adapted to a quantum-based scheduler, and the quantum size would determine the allocation accuracy. Smaller the quantum size better the allocation accuracy.

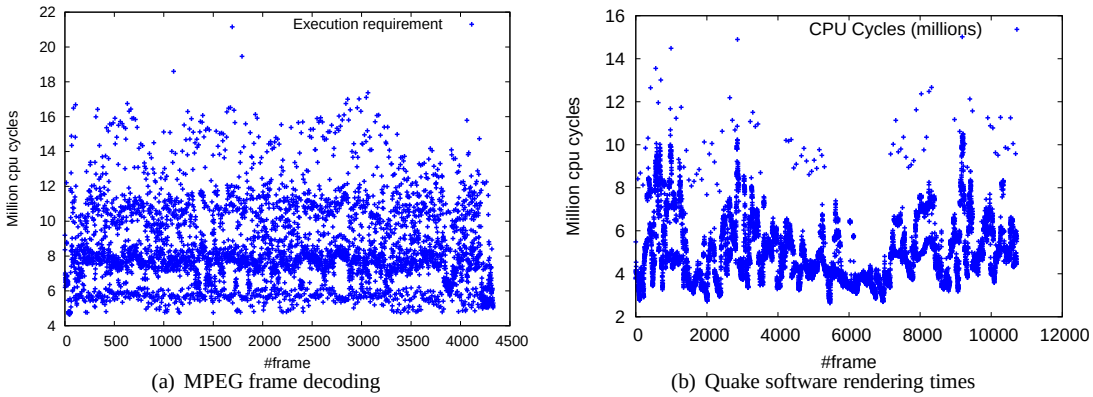
While current scheduling algorithms like Priority, EDL or GPS may perform arbitrarily in terms of response time to non-RT tasks, the proposed scheduling algorithm works well for both measures $A(t)$ and $(1 - s(t))$. Our work opens new doors in the area of scheduling variable requirement RT tasks and we believe that many more exciting applications are possible using this approach. From hereon we would refer to this algorithm as Stochastic Processor Sharing or SPS.

CHAPTER 4
Soft Real-time Scheduling

In the previous chapter we addressed the problem of scheduling variable requirement RT jobs while providing better timely allocation to the non-RT tasks in the system. RT task model is useful for critical applications like medical instruments, space missions etc., but frequently, the tasks encountered in a GPOS are not exactly time critical, even though they may have a notion of deadline. An example of such a task is media (audio/video) playback, and other examples are computer games and any sort of interactive application. Though, these tasks can be assigned deadline (based on frame rate for media playback and animation, and based on human sensitivity to response time for interactive jobs), but the performance is not affected by response times in close vicinity of the deadline. This means that SRT tasks need not be allocated based on their worst case requirement values (which is required for critical RT tasks), and they may be guaranteed a smaller allocation for each job (which we would call its reservation) and the burden falls upon the scheduler to provide good response times for the SRT jobs that require greater execution time than their reservation.

One of the common approaches used in scheduling SRT tasks is to bound their deadline miss ratios. In this approach, only a certain fraction of jobs are allowed to miss their deadline (say 1%), and no assumption is made about the response times of these 1% of jobs. Now, consider the execution time requirement distribution for MPEG decoding and Quake I software frame rendering (Fig 4.1). For the MPEG decoding task, giving 14 Million cycles to each job leads to around 2% deadline misses and still most frames may require less than 11 Million cycles. The other important observation here is that a deadline miss ratio of 2% means that approximately one frame in every fifty frames misses its deadline. At 25 fps, one frame every 2 seconds misses its deadline by an unspecified amount of time. In the worst case, each of these deadline misses may contribute to performance degradation.

This brings out the importance of taking into considering the overrun times of the SRT jobs. For example, even if the deadline miss ratio is 10%, if 99% of the jobs missing their deadlines finish within reasonable times, then the performance degradation may not be visible. And at the same time the processor share committed to the task is reduced drastically (by allocating just enough so that 90% jobs are guaranteed to finish before their



(a) MPEG frame decoding (b) Quake software rendering times

Figure 4.1: Execution time variation for an MPEG frame decoding task and Quake I software rendering task. The frame decoding times for a StarWars trailer using mpeg_play are reported. Note that the minimum execution time frame requires around 5 million cycles, some frames requireme more than 20 million cycles. Some frames require more than 20 million cycles. Even from frame to frame, the execution time variation is significant. For Quake I software rendering task, though frame to frame execution time variation is not as substantial as MPEG decoding task and the execution time between adjacent frames is similar, the overall execution time variation is large, with mean requirement of around 5M cycles and maximum of more than 15 M cycles.

deadline). But to achieve these gains, the 99% of the 10% jobs missing deadline should have small overrun time. And current scheduling algorithms do not address this problem. SPS does exactly that. The goal of SPS is to guarantee allocation to RT jobs while minimizing response times of the non-RT jobs. Thus irrespective of the value of reservation chosen, using SPS to guarantee that reservation would provide smaller overrun times to the non-RT jobs in the system.

In the following sections, we formally describe the task model for SRT tasks and show how SPS can be adopted to better handle SRT tasks.

4.1 SRT Tasks

We address the problem of scheduling overrun time sensitive variable requirement soft real-time (SRT) tasks. We break each job of the SRT task into two parts, real-time (RT) component and non-RT time sensitive (TS) component. The goal of scheduling is to provide timely allocation guarantees to the RT component while reducing response times for the TS components. The response time of a SRT job is the time when its RT and TS component both have finished execution. The deadline overrun time for a SRT job is defined as its response time minus its period.

Formally, a SRT task τ_i is represented by the tuple (P_i, χ_i, C_i, R_i).

- P_i is the period. Jobs of τ_i arrive every P_i time units.
- χ_i is the random variable representing the execution time requirement of a job of this task
- C_i is the worst case execution time requirement of this task (WCET)
- R_i is the reservation, that is any job of the SRT task should receive R_i execution time before its deadline.

For schedulability, $\sum_{i=1}^{N} R_i/P_i \leq 1$, where N is the number of tasks in the system. Note that $\sum_{i=1}^{N} C_i/P_i$ may be greater than 1, i.e. the system may be overloaded.

Any SRT job starts as a RT job. If the SRT job does not finish after receiving R_i execution time, then it is scheduled as a TS job for its remaining duration. The SRT job finishes when the RT as well as the TS component (if any) have finished. SRT jobs requiring not greater than R_i execution time are guaranteed to finish within P_i time units of their arrival. Jobs requiring greater than R_i execution time may miss their deadline, and the goal of scheduling is to reduce their deadline overrun time.

The question of how R_i is chosen is left open. R_i may be prespecified or adaptively determined using algorithms like feedback-control [APLW02] [LSST02]. The problem we are addressing is that the RT component may require execution time anywhere between 0 and C_i, and this variability can be leveraged to provide better response times to the TS components, thereby reducing SRT jobs' deadline overrun times.

We found that giving RT jobs higher priority over other jobs in the system is very inefficient, yet most current General Purpose Operating System (GPOS) rely on priority based schedulers. Reserving processor share for RT jobs (Generalized Processor Sharing (GPS)) or delaying RT jobs (Earliest Deadline as Late as possible (EDL)) provides smaller overrun times to SRT jobs in some scenarios but may perform badly in others (as explained in next Section). We propose Stochastic Processor Sharing (SPS) algorithm that uses the empirical probability distribution of execution time requirements of the SRT jobs to determine the processor share of the RT jobs as a function of time. In particular, the RT jobs start with a lesser processor share than they would in GPS, but may end up using greater processor share as they approach their deadline. The probability distribution of execution time requirement is used to determine how the processor share increases with progress, in particular, the higher processor share requirement phases are rare. That is, most jobs finish while in the smaller processor share phase. We show that our algorithm consistently provides small overrun times to SRT jobs, closely matching (or outperforming) the best performing algorithm (out of GPS, EDL or Priority) in any given scenario.

4.2 TS Tasks

Practical systems would consist of non-RT response time sensitive tasks, which we call Time Sensitive (TS) tasks, along with SRT tasks. While these tasks do not require any allocation guarantees, their performance depends upon their response times. In a system containing a mix of SRT tasks and TS tasks, the response times of TS tasks would be dependent upon how the SRT tasks are scheduled because the SRT tasks require guaranteed allocation by their deadline.

To evaluate the impact of SRT scheduling on response time of TS tasks, we consider a system containing mixed task set i.e. SRT and TS tasks. The SRT tasks have a RT component and a TS component, as defined in previous section. The RT components are scheduled in EDF order while getting processor share of at least $s(t)$ at time t. The TS (overrun) components if any, are scheduled in LAS order i.e. the TS job which has received the least service till a given time gets allocated before other TS jobs. This selected TS job is scheduled for at most one quantum time, after which again the TS job with least received allocation is selected.

When scheduling SRT and TS tasks together, there are two ways that SRT overrun jobs and TS jobs can be scheduled. One way is that the SRT overrun jobs are treated at par with the other TS jobs. The other scenario is when the SRT overrun jobs are given priority over the TS jobs. We assumed a system where the SRT overrun jobs are given priority over the other TS jobs in the system. This is because the SRT overrun jobs may actually end up finishing before their deadline in this scenario which reduces the number of deadline misses. Thus, lesser RT reservation would achieve the desired deadline miss ratio, as compared to the scenario when SRT overrun are treated at par with other TS tasks.

One important area that needs more work is to come up with a continuous priority range between SRT overrun and other TS tasks. This may be dependent upon the utilities of each of the tasks, and scheduling would be based on these utility functions, on the lines of those proposed by Jensen et. al. [JLT85]. This is an important problem because if the SRT overrun jobs require a lot of computation on average, then scheduling them at higher priority than TS jobs may not be efficient, because the SRT overrun jobs would block the TS jobs and themselves not benefit much.

4.3 What makes a Good Co-Scheduling Algorithm?

The notion of optimality in co-scheduling algorithm is difficult to define in a general sense. Task sets can be constructed such that any one of Priority, GPS, or EDL may perform better than the others.

For example, consider a single RT task with period 1 time unit, and utilization of 0.5. Now, assume that there are two TS tasks – one very large requirement TS task with very large period and the other a very small requirement TS task with period 1 time unit. Now, depending upon the arrival time of the small requirement TS task, either Priority or EDL or GPS may be optimal. For example, if the small requirement TS task arrives at time 0, then EDL is optimal. If the the small requirement TS task arrives at time 0.5 then Priority is optimal. If the small requirement TS task arrives at time 0-0.5, then GPS provides better response times than Priority, and if the small requirement task arrives at time 0.5-1, then GPS provides better response times than EDL.

One way to define optimality would be to define the worst case response time distribution for a very small requirement TS task with period equal to hyper-period of the task-set over all possible arrival times between 0 and hyper-period. The optimal scheduling algorithm then would be the one that minimizes the maximum expected processor share of RT tasks at any instant. And for any other scheduling algorithm, there would be intervals during hyper-period where the expected processor share of RT task is higher than that under the optimal algorithm. And choosing any of these intervals as arrival time for the very small requirement TS task with period equal to hyper-period would give worst response times under the other algorithm as compared to the optimal algorithm.

Now, for a single RT task system, with no idle allocation to RT jobs, the SPS schedule is the optimal schedule. Note that idle allocation i.e. allocation received by the RT jobs outside of their RT share, impacts the expected value of processor share. This is because the function $g(.)$ is calculated assuming the only allocation available to RT jobs is the one obtained through RT share $s(t)$. To see this note that, $E[s(t)]$ is defined as $s(t) * \Pr[\chi > \int_0^{t-a(t)} g(x)\mathrm{d}x]$. If the RT jobs get idle allocation then

$$E[s(t)] = s(t) * \Pr[\chi > (idle\ allocation + \int_0^{t-a(t)} g(x)\mathrm{d}x)]$$

For a system with multiple tasks, minimizing the maximum expected processor share of RT jobs may be costly. For example, consider a two task system. When one task completes, the distribution of execution time requirements is no longer the sum of the execution time requirement distributions of the two tasks, rather it is the execution time requirement distribution of the active task. So the $g(.)$ function minimizing the maximum expected processor share of RT jobs is different now. As the number of tasks increases, for each unique set of active tasks, the cumulative execution time requirement distribution may be different, and hence the optimal $g(.)$ function would be different in each of these cases. Maintaining separate $g(.)$ function for each possible

combination of active tasks may not be feasible. This, combined with RT jobs getting idle allocation makes the problem more complicated. Part of the problem here is adjusting to changing requirement distribution (due to idle allocation or job completions).

It is important to note here that algorithms other than SPS have intervals when the $E[s(t)]$ is smaller than that in SPS, so the TS jobs arriving during these intervals get better response times. But, reducing $E[s(t)]$ during some intervals means $E[s(t)]$ is higher in some other intervals. And having higher $E[s(t)]$ has a two-fold negative impact. First, during the interval when $E[s(t)]$ is high, the TS jobs get lesser processor share. And second, during these intervals, the TS jobs may get queued up, leading to greater delays and during the intervals when $E[s(t)]$ is small, these queued up TS jobs compete with the fresh TS jobs, thereby diluting the impact of intervals when $E[s(t)]$ is small.

4.4 TS Job Size and Impact on Response Time

At this point it is important to mention that the response time benefits are greater for tasks that operate at smaller time scales (small period or small requirement) and decrease as the time scale increases (large requirement tasks). To understand this consider the following example.

Consider a task system with one RT task. Suppose the period if this task is 1 time unit, mean requirement is 0.3 time units and worst case requirement is 0.6 time units. Let x_i represent the execution time requirement of job i of this task.

Now suppose this RT task and a TS job enter the system at time 0. Let the requirement of TS job be y time units. Now, under Priority scheduling the TS job finishes during the k^{th} job of RT task where k is the minimum integer for which,

$$\sum_{i=1}^{k} x_i + y \leq k$$

That is the cumulative execution time requirement of k RT jobs ($\sum_{i=1}^{k} x_{k-1}$) and the TS job y, should be at most the total execution time available, which is k.

Now consider for the job $k-1$ of the RT task (assume $k > 1$). From the discussion above,

$$\sum_{i=1}^{k-1} x_i + y > k - 1$$

Because k is the minimum integer for which the execution time requirement of k RT jobs plus the execution time

requirement for the TS job is not greater than k.

This is important because irrespective of the scheduling algorithm used to schedule the RT task (i.e. irrespective of choice of $s(t)$), the TS job finishes during the k^{th} RT job. The RT scheduling algorithm dictates when actually during the k^{th} RT job will the TS job finish. The TS job would finish the earliest under EDL, later in SPS, still later under GPS and latest under Priority scheduling.

Thus, the response time benefit is most visible for TS jobs that finish within the period of the RT job that is active on their arrival (which is the case if $y << 0.3$, where 0.3 is the mean execution time requirement of the RT job).

In general, small TS jobs see greater benefits depending upon the way RT tasks are scheduled. For our case, we will focus on task systems with a large number of tasks (both SRT and TS), so the individual job requirements are small, hence the benefits seen are large.

In the later sections we discuss the case when the system has relatively large requirement TS jobs, in which case, all scheduling algorithms perform equally.

4.5 The SPS Scheduler

The scheduler has the following key components -

- EDF ordered queue of RT jobs
- LAS ordered queue of SRT Overrun jobs
- LAS ordered queue of TS jobs
- SPS_Share function $s(t)$,

$$s(t) = \max_{1 \leq i \leq N_{srt}} s_i(t)$$

  This choice of $s(t)$ ensures that the allocation requirements for the SRT tasks are guaranteed always, as proved in previous chapter.

We assume a GPS capable processor. The scheduling is done in time steps of Δ (for simulations we assumed $\Delta = 1ms$). At the end of each time step, the RT share $s(t)$ ($O(N_{srt})$) is calculated. The RT jobs are allocated $s(t) * \Delta$ execution time in EDF order. The remaining allocation, which is $\Delta(1 - s(t))$ plus any allocation remaining from the $s(t) * \Delta$ (if all RT jobs finish without consuming all the available allocation), is allocated

first to the overrun jobs in LAS order and then to the TS jobs in LAS order. If the overrun and TS jobs do not consume the allocation available to them (all overrun and TS jobs have finished), then the remaining allocation is given to RT jobs in EDF order and this allocation is the *idle* allocation.

4.6 Measuring and Reporting Response-times - $\Phi(.)$ Function

Quantifying the scheduler performance is not an easy problem. because there are too many variable that can impact the performance. The response times depend on the following factors –

- Choice of SRT workload
- Choice of TS workload

As proved theoretically in the previous chapter, irrespective of what the RT task set is, SPS minimizes the maximum expected RT processor share at any instant. What this means that our problem is not to quantify scenarios where SPS works and where it does not work, this is because SPS provides a better RT schedule irrespective of the RT task set characteristics in terms of the measures $E[s(t)]$ and $A(t)$. The problem we address is what is the performance benefits that can be achieved using SPS. In some cases, the performance benefits would be less and more in other cases. And our goal is to conduct a broad enough range of experiments to give a good understanding so as to when SPS would provide significant performance benefits and when it would perform at par with other algorithms.

For the SRT task set, for simulations we assumed normally distributed execution time requirement SRT jobs with random mean utilizations and periods randomly distributed between 30 and 200 time units. The cumulative mean SRT utilization is specified as a parameter and a task set with large number of SRT tasks is generated (50 in our case). The reason of choosing large number of SRT tasks is because, as the number of SRT tasks increases, their cumulative distribution approached normal distribution and hence the simulations are of greater practical relevance to actual workloads.

While choice of SRT task set (and their requirement distribution) is one problem, the choice of TS task set is another similar problem. But, the choice of TS task set is limited by remembering that the performance benefits are most visible when the TS jobs are small requirement jobs and they finish within the duration in which currently active RT jobs are still active. This is because, irrespective of the scheduling algorithm used, the cumulative RT and TS allocation at RT job deadline is equal in case of single RT task system, and nearly the

same in case of multiple RT task system. The only difference in RT allocation is because SPS/EDL delay the currently active RT job. For TS tasks with large execution time whose execution lasts over several periods of RT jobs is not impacted by SPS and their response time distribution is just slightly improved by using SPS/EDL as compared to GPS/Priority. Thus, for our case we would focus on TS task set with small jobs. The TS tasks are generated exactly like the SRT tasks, and the only difference is that the TS tasks have zero reservation while the SRT tasks have a cumulative reservation of R.

Now that we have the SRT task set and the TS task set, what remains is to present the simulation results such that they can be easily visualized and understood. Now, what we are optimizing is for the TS response times while providing RT guarantees. So, it is natural that we report the SRT overrun times and the TS jobs response times to quantify the performance of the various algorithms (EDL, Priority, GPS, SPS). Now, the most natural measure is to report the mean response times, but this measure is biased in favor of jobs requiring large response times. Also, the impact of response time of a job can be thought of as being dependent upon its period. So we chose to report the response times scaled by their respective period. Note that in this model, TS tasks with smaller period are more sensitive to response time than TS tasks with greater period. But the scaling is essential to find common ground to compare all the results. Even the SRT overrun times are scaled by their respective period.

So we report the mean scaled SRT overrun time and mean scaled TS response time. But this does not give the complete picture. This is because for SRT overrun jobs, the jobs with small overrun time (less than say 0.3 times their period) may not actually incur any performance degradation. So, it would be nice to have a quantification which tells how many SRT jobs missed their deadline by more than say 0.3 times their period. We represent this measure as the $\Phi(x)$ measure. $\Phi(x)$ denotes the number of jobs with response time greater than x times their respective period. So for the discussion above SRT overrun $\Phi(0.3)$ would give the number (or percentage) of SRT jobs with overrun time greater than 0.3 times their respective period.

Now we have most things in place to get into presenting actual simulation results. But before that there is an important topic which needs attention and that is how to get χ_{RT}.

4.7 Online Profiling - Constructing χ_{RT}

As pointed out earlier, χ_{RT} is tightly coupled with the SPS scheduling algorithm. The cumulative execution time distribution which is represented as χ_{RT} forms the eyes of SPS algorithm in determining the appropriate

$g(.)$ function.

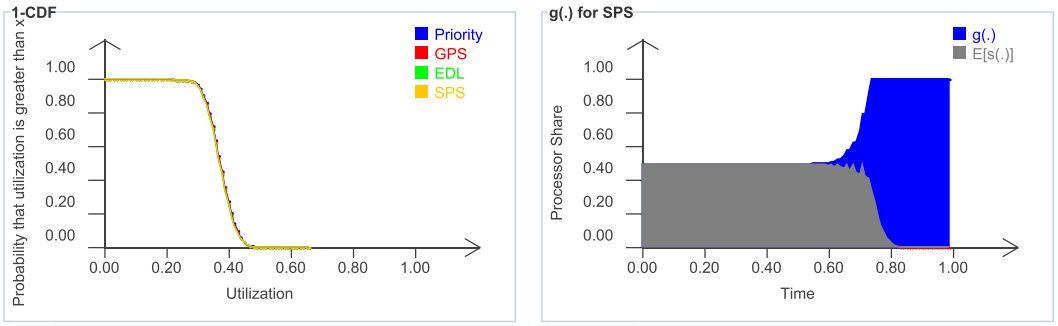
Now, the simplest interpretation of χ_{RT} would be

$$\chi_{RT} = \sum \frac{\chi_i}{P_i}$$

To construct this distribution using online profiling, we follow the following approach. At time 0, we assume that the previous (hypothetical) job of each RT task required its respective reservation R_i amount of execution time. Now whenever a RT job finishes, its utilization is updated and the sum of the utilizations of all the RT tasks in the system is taken as a value in histogram for distribution of χ_{RT}. As time progresses, the histogram becomes a better approximation of χ_{RT}.

To understand the impact of χ_{RT} on the actual performance, lets run through an actual example.

Suppose we have a task set [N_{srt}=50,U_{srt}=0.40,R=0.65,N_{ts}=50,U_{ts}=0.30]. Now assuming that the χ_{RT} is calculated as mentioned above (sum of distributions of individual RT tasks), the schedule looks as in Fig 4.2.



(a) (1-CDF) for χ_{RT}

(b) $g(.)$ for SPS

Figure 4.2: These figures show the χ_{RT} obtained using just summing the individual SRT task distributions. We call this the naive version of χ_{RT} the reasons for which will become clear in the next figure.

All looks well and good, the $g(.)$ function for SPS has a nice shape, starting with lesser processor share and increasing its processor share with progress. Now lets look at the TS response time distribution. Figure 4.3 shows the $\Phi(.)$ function for the four scheduling algorithms. As can be seen, EDL is significantly better than the other three scheduling algorithms. The reason for this is that the cumulative mean system utilization is just 0.30 + 0.30 = 0.60. Hence there is lot of idle time, so while the RT component jobs of the SRT tasks are waiting for execution under EDL, they get idle allocation and hence finish without requiring any RT allocation. While under SPS, the RT jobs start with certain processor share and their share keeps increasing with progress. Hence, even though SPS performs better than GPS, because the RT jobs start off with lesser processor share, it gets beaten

well by EDL.

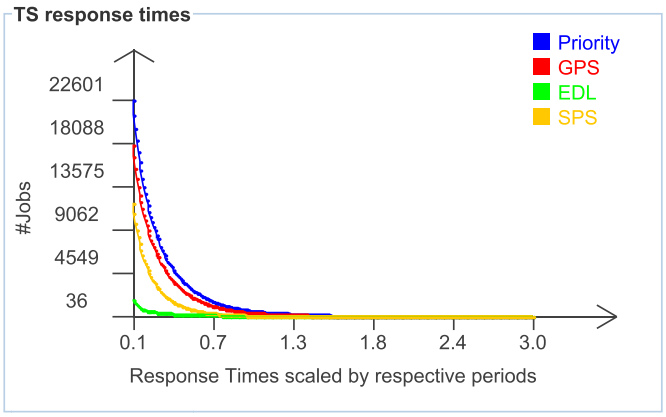


Figure 4.3: TS response time $\Phi(.)$. As can be seen from the figure, EDL outperforms the other algorithm by significant margin. So why does SPS not perform well ?

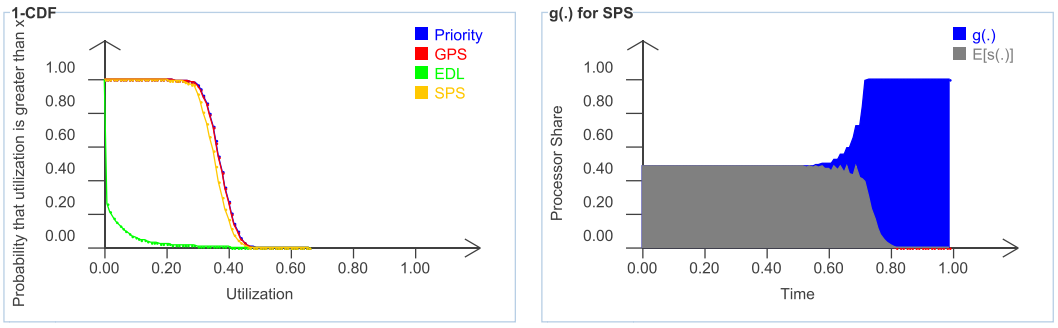
Is this it for SPS ? Well not actually. Note that, if there is enough idle time in the system, the the RT jobs may not require any RT allocation at all. That is there is no actual need for an RT scheduling algorithm if the RT jobs can finish before their deadline even if they are scheduled after TS jobs (though in EDF order). How can this fact be incorporated in SPS. Well, its not that difficult after all. Note that SPS scheduling algorithm would give a schedule like EDL (RT jobs wait for maximum possible time) if the requirement distribution (χ_{RT}) is such that there is very high probability that the RT job requires 0 execution time and insignificantly small probability that the RT job requires R execution time. Since SPS needs to guarantee the R execution time, hence, it would come up with a schedule that looks like EDL. The RT job starts with minimal processor share, and as in EDL, it gets full processor share later in the schedule to ensure that no RT job misses its deadline.

This raises the question so as to what needs to change in the profiling to account for the idle time in the system. The first avenue for improvement comes from the scheduler. Note that the way we defined the SPS scheduler, it first allocates $s(t) * q$ execution time to the RT jobs and then allocates $(1 - s(t)) * q$ time to the TS jobs and if there is some execution time still remaining then the remaining execution time is allocated to the RT jobs as idle allocation. Now, we change the χ_{RT} distribution construction as follows. When a RT job finishes, instead of using the sum of utilizations of all the RT jobs as a value of χ_{RT}, we use the sum of RT allocation divided by the respective periods for the finished RT jobs as the value to be used for constructing χ_{RT}.

What does it mean to leave out the idle allocation to RT jobs in construction the χ_{RT} schedule? Well, intuitively it means, that if the RT job actually required only the RT allocation worth of computation, and rest of

the computation was available to it as idle allocation so that does not need to be included in the RT requirement of the RT job.

So what does this buy us ?



(a) (1-CDF) for χ_{RT} (b) $g(.)$ for SPS

Figure 4.4: These figures show the χ_{RT} obtained after discounting the idle allocation to the RT jobs in constructing χ_{RT} histogram. Their is a slight change in the schedule from the naive approach discussed before.

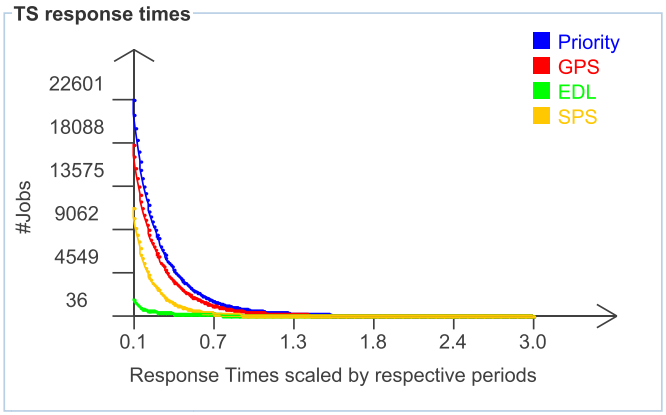


Figure 4.5: TS response time $\Phi(.)$. As can be seen from the figure, EDL still outperforms the other algorithm by significant margin. SPS improves, but slightly.

Fig 4.4 and Fig 4.5 show the impact of discounting idle allocation in construction of χ_{RT}. Though, there is slight improvement in SPS performance but it is too small to account for anything significant. The reason for this is that SPS is not able to capture the idle time fully. This is because, under SPS, the RT jobs start off with certain processor share and even after discounting the idle allocation, the RT jobs usually finish quickly. And once the RT jobs are finished, the idle time in the system goes unnoticed by SPS. On the other hand, EDL delays RT jobs maximally, hence they are active for the greatest duration and hence they have increased chance of finishing while using idle time.

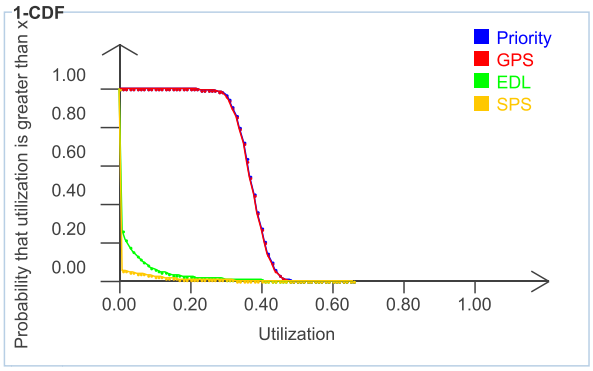
So how can we account for the idle time in the system when no jobs are active. Well, to do this we just maintain the total duration of time during which the processor is idle. This duration divided by the total elapsed time give the mean idle time in the system. And the χ_{RT} now approximately equals

$$\chi_{RT} \approx \sum \frac{\chi_i}{P_i} - (1 - U_{srt} - U_{ts})$$

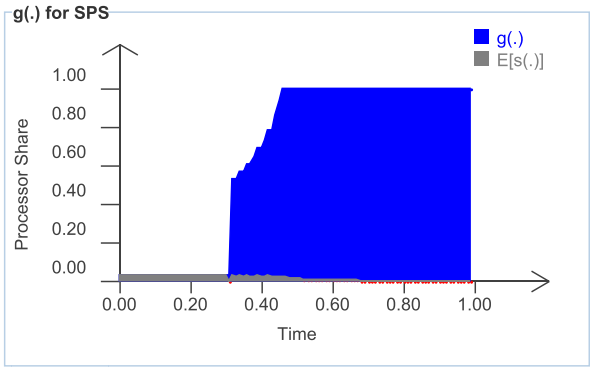
where U_{srt} is the mean SRT utilization and U_{ts} is the mean TS utilization and hence $(1 - U_{srt} - U_{ts})$ is the mean idle system utilization. What this means is that we retroactively assign the mean idle utilization to the RT jobs, assuming that they used it and hence we subtract that much utilization from their cumulative utilization to construct the histogram for χ_{RT}. Note that this technique of retroactively assigning computation to job has use in other scenarios, specifically for slack reclamation under static priority systems where it is referred to as history rewriting [BBB04].

Does this help ? Actually very much. Fig 4.6 and Fig 4.7 show the impact of discounting idle time in construction of χ_{RT}. Note that now SPS and EDL perform at par, outperforming GPS/Priority by considerable margin. From this point on, the χ_{RT} distribution will be calculated discounting the idle allocation and idle time in the system, which gives

$$\chi_{RT} \approx \sum \frac{\chi_i}{P_i} - (1 - U_{srt} - U_{ts})$$



(a) (1-CDF) for χ_{RT}



(b) $g(.)$ for SPS

Figure 4.6: These figures show the χ_{RT} obtained after discounting the idle time in constructing χ_{RT} histogram. Note that the $g(.)$ function nearly resembles EDL which is what was needed.

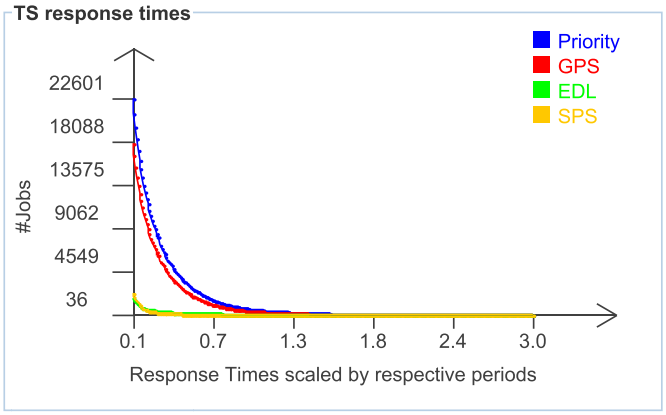
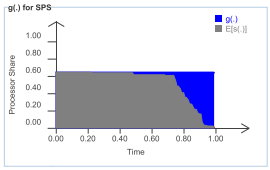


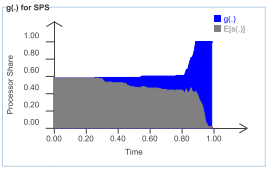
Figure 4.7: TS response time $\Phi(.)$. As can be seen from the figure, SPS now matches EDL, which is not surprising since the SPS $g(.)$ function closely matches that of EDL.

4.8 Learning χ_{RT}

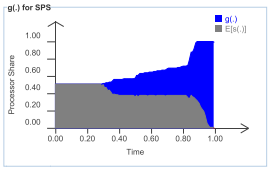
Fig 4.8 shows the impact of continuous online profiling on the shape of $g(.)$ function for SPS.



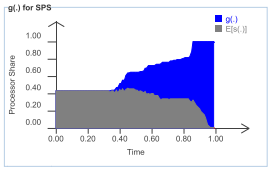
(a) $g(.)$ after 65 RT jobs, calculated once at t=0



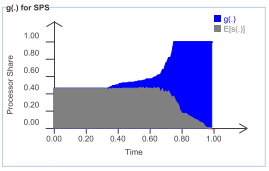
(b) $g(.)$ after 142 RT jobs, calculated at t=100



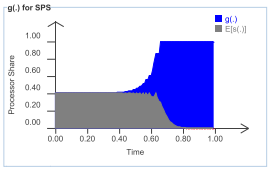
(c) $g(.)$ after 194 RT jobs, calculated at t=200



(d) $g(.)$ after 248 RT jobs, calculated at t=300



(e) $g(.)$ after 602 RT jobs, calculated at t=1000



(f) $g(.)$ after 57953 RT jobs, calculated at t=100000

Figure 4.8: These figures show the χ_{RT} obtained through online profiling as time progresses for task set (N_{srt}=50,U_{srt}=0.50,R=0.65,N_{ts}=50,U_{ts}=0.35). The $g(.)$ function is recalculated every 100 time units. As the histogram becomes richer and more accurate, the $g(.)$ function becomes nearly constant.

4.9 Putting All the Pieces Together - Design of a Practical Scheduler

Our claim is that SPS is a practical algorithm that can provide performance gains in actual systems. In this chapter, we address this problem in detail. We start with a discussion of how our scheduling model of specifying $g(.)$ function and using it to schedule RT jobs in EDF order can be mapped onto actual practical systems. In particular, the $g(.)$ function may vary continuously, so there needs to be a mapping mechanism to map the $g(.)$ function to sequential processors. Second, there are issues regarding how to find the period boundaries of a task. in particular, current applications do not inform the OS when a job finishes, instead they might just call sleep function. But in order to provide timeliness guarantees, it is required that the job arrival time and deadline be known. Third, for systems with very large number of tasks, like web servers, the periodic task model for each task may not be applicable. Because on these servers, the number of active tasks at any instant may be variable. So, it is required that we come up with a scheduling model for systems with a large number of tasks, where the number of tasks may be variable. Fourth, large scale p2p networks are useful architectures with wide application domain from content distribution to massively multi-player video games. Our scheduling model handles systems with cooperative tasks well.

4.9.1 Periods and Reservation

The only information that is required of the SRT jobs is their periods and reservation. Though minimal, these requirements are still difficult to meet in current applications. For example, most applications do not communicate their job arrival time and deadline to the OS. There are two ways that can be used to get around this problem. First, for applications like media decoding, animation their is an implicit notion of period if the frames displayed per second is known. For interactive applications, the response time range of 50-200 ms is usually considered good. In a similar fashion, the time sensitivities of various applications can be approximated.

This leaves us with two problems –

- The actual arrival times and deadlines may still be unavailable
- We still need to figure out the value of reservation

But this kind of problem has already been addressed for tasks like media playback (Abeni et. al. [AB98a] [APLW02]), where the Constant Bandwidth Server (CBS) is used in conjunction with feedback scheduling to provide predictable service to media playback tasks.

For our purpose, the only thing that needs to change is the way the allocation is guaranteed. So while the remaining system components remain the same – a feedback-control loop to determine the reservation required using deadline miss ratio or frame decoding time as the control variable – instead of using CBS, SPS is used to guarantee the reservation to the RT component of the SRT media playback jobs.

While this is one possible path that can be taken, there are other possibly better ways to address this problem. For example, instead of using just the deadline miss ratio or controlling the frame decoding time to be equal to the period as in [APLW02], the reservation may be determined using the entire response time distribution ($\Phi(.)$) function to check if the performance requirement of the application are satisfied. But this would require the application programmers to provide a metric for their applications acceptable performance response times. The feedback-control loop design methodology proposed by Lu et. al. [LSST02] can then be used to determine the actual reservation required by the application to satisfy its performance requirements.

These are important problems and should be addressed for the realization of a practical system using SPS scheduler. And this becomes one of the important components of our future work.

In the following sections we look at some possible application scenarios.

4.10 Possible Application Scenarios

To understand how SPS would fit into practical systems, we work through two examples – one of a server (like web-server) supporting large number of concurrent clients, and the second scenario is a network node supporting large number of flows with bandwidth guarantees.

4.10.1 Server System Supporting Large Number of Clients

We consider a simple model, there is a set of premium clients who pay greater money for the service and require assured service rates and then there is a set of normal clients who do not pay as much (or may not pay at all) and get the remaining execution time. Now, the goal of the scheduling algorithm is to keep the paying customers satisfied while supporting as many of the less paying customers as possible.

Suppose through some magic or oracle, we come up with 0.65 as the cumulative reservation provided to the premium customers as a whole. Now Let the mean cumulative utilization of the premium customers be 0.30. That is, due to variation in number of premium clients and the tasks they do, their cumulative utilization may vary but the mean is 0.30 and suppose that the actual cumulative utilization of the premium customers is normally

distributed with mean 0.30 and standard deviation of 0.057. This means that probability that the cumulative utilization of premium customers is greater than 0.30 + 3*0.057 = 0.47 is less than 0.15%. Hence no premium customers miss their deadline.

Assume that the premium customers are scheduled together as a single task, with a period of 150ms, so if premium customers arrive only on this period boundary then they are guaranteed a response time of at most 150ms. Now suppose there are 100 non premium customers that we would like to support (since the mean utilization of the system with premium customers is just 0.3). We model these customers are 100 TS tasks with periods uniformly distributed in the range 30-200 ms and, and a job arrives every period time units and the job requirements are normally distributed and their cumulative mean utilization is 0.40.

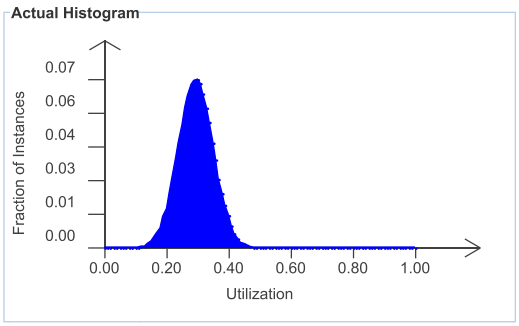
The Table 4.1 shows the summary of simulation results for such a task system.

Stats	Scheduler			
	Priority	GPS	EDL	Proposed
Simulation time	100002	100002	100002	100002
SRT jobs completed	667	667	667	667
SRT job overruns	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
Mean scaled overrun time	0.0000	0.0000	0.0000	0.0000
Overrun $\Phi(0.0)$	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
Overrun $\Phi(0.10)$	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
Overrun $\Phi(0.20)$	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
Overrun $\Phi(0.40)$	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
Overrun $\Phi(0.80)$	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
Overrun $\Phi(1.6)$	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
TS jobs completed	117472	117472	117472	117472
Mean scaled TS response time	0.2134	0.0854	0.0560	0.0363
TS response time $\Phi(0.10)$	43357 (36.91%)	20451 (17.41%)	15529 (13.22%)	9428 (8.03%)
TS response time $\Phi(0.20)$	35599 (30.30%)	12359 (10.52%)	8962 (7.63%)	3462 (2.95%)
TS response time $\Phi(0.40)$	22690 (19.32%)	6429 (5.47%)	3793 (3.23%)	960 (0.82%)
TS response time $\Phi(0.80)$	9157 (7.80%)	2181 (1.86%)	771 (0.66%)	185 (0.16%)
TS response time $\Phi(1.6)$	1389 (1.18%)	426 (0.36%)	35 (0.03%)	25 (0.02%)

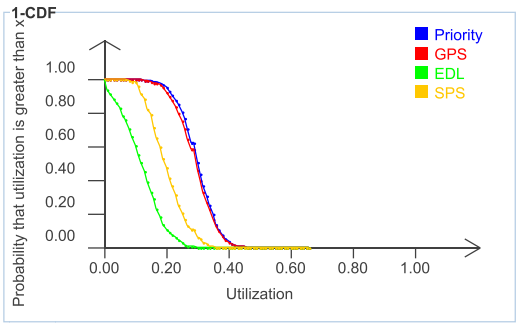
Table 4.1: Summary statistics for (N_{srt}=1,U_{srt}=0.30,R=0.65,N_{ts}=100,U_{ts}=0.40).

As expected, there are no deadline overruns using any scheduling algorithm. But note the significant impact on the TS response times. In terms of mean scaled TS response times, SPS reduces it by a factor of 6 as compared to Priority and a factor of 2 as compared to GPS. Also, under SPS, less than 1% of the non premium customers have response time greater than 0.4 times their respective job periods. On the other hand, nearly 20% under Priority, 5% under GPS and 4% under EDL have response times greater than 0.4 times their respective job periods. If this was a performance threshold then SPS satisfies 99% of the non premium customers and Priority satisfies only 80% of those. Note that all algorithms guarantee that the premium customers do not suffer any

performance loss.



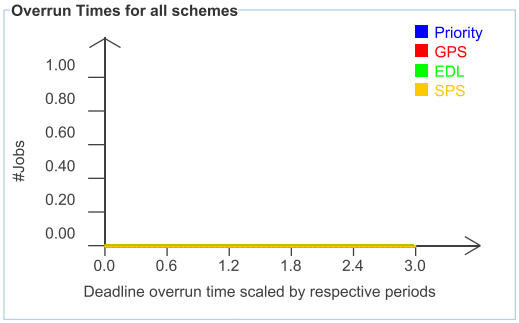
(a) Cumulative SRT utilization histogram



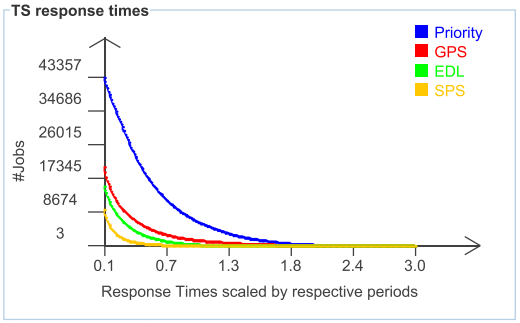
(b) Cumulative RT utilization (1-cdf)

Figure 4.9: (N_{srt}=1,U_{srt}=0.30,R=0.65,N_{ts}=100,U_{ts}=0.40) execution time requirement distribution.

Fig 4.9 shows the execution time requirement distribution. The histogram represents the actual requirement histogram of RT task, while the (1-CDF) curve represents the probability that a RT job may have RT utilization greater than the value on X-axis. Note that under EDL, the RT utilization is the least, which means that the RT task gets the largest amount of idle allocation as compared to other algorithms.

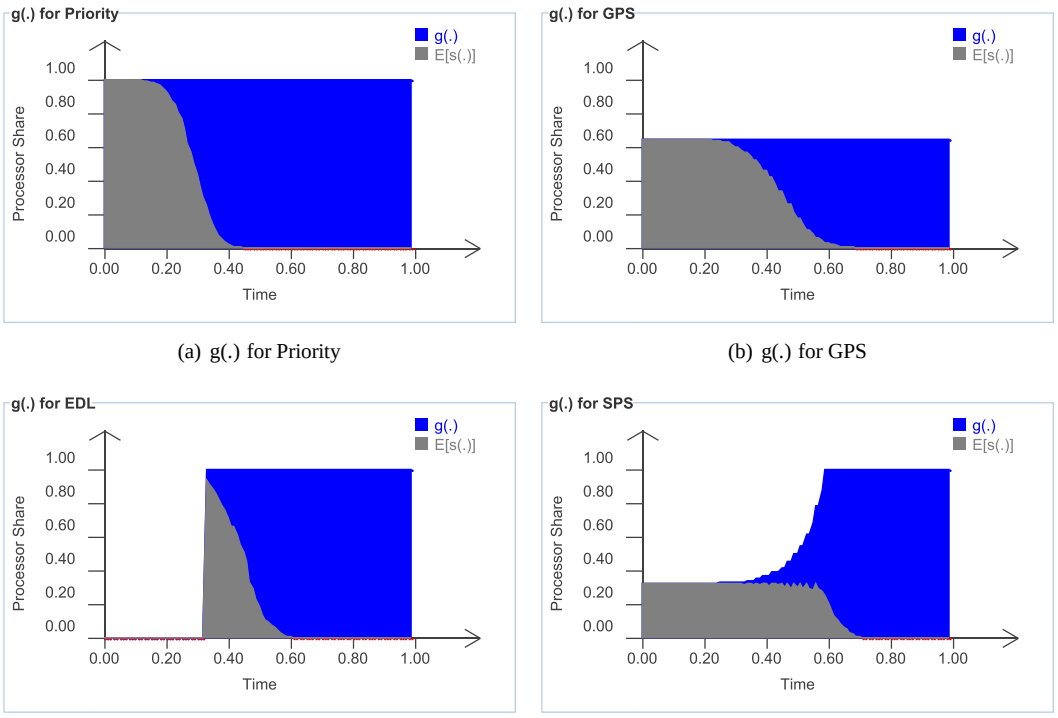


(a) SRT overrun $\Phi(.)$



(b) TS response times $\Phi(.)$

Figure 4.10: (N_{srt}=1,U_{srt}=0.30,R=0.65,N_{ts}=100,U_{ts}=0.40) $\Phi(.)$ values for SRT and TS tasks



(a) g(.) for Priority

(b) g(.) for GPS

(c) g(.) for EDL

(d) g(.) for SPS

Figure 4.11: (N_{srt}=1,U_{srt}=0.30,R=0.65,N_{ts}=100,U_{ts}=0.40) g(.) functions for the four schemes. The $g(.)$ functions for the four scheduling algorithms. Note that SPS has a expected value of $s(t)$ nearly 0.3, which is the mean utilization of the RT task. Hence, SPS does a good job in scheduling the RT task.

4.10.2 Supporting Bandwidth Reservations on a Network Node

Now consider another scenario. Suppose there are 100 clients who pay and get bandwidth reservation for themselves. Let the cumulative mean requirement of these clients be 0.65 fraction of available bandwidth. Suppose, the oracle comes up with the figure 0.80 as the fraction of bandwidth to be reserved for these clients. Now since there is 35% idle capacity, the network owner decides to support 20 more clients with cumulative mean bandwidth utilization of at most 0.15 who pay based on the quality of service they receive. And the acceptable performance for the paying 100 clients is that the probability that the packet spends more than the respective period time at the network node should not be more than 1%. The periods for the premium clients may be chosen based on the timeout values for their congestion control protocol (TCP). So the packets with deadline overrun may actually be considered lost by TCP leading to decreased bandwidth. Also, the non paying customers only pay for packets that are serviced within 0.4 times their respective period. That is for $\Phi(0.4)$ packets (or TS jobs) do not pay.

The way the presence of TS jobs impacts the schedule of RT jobs is that a fraction of the idle allocation is consumed by the TS jobs. This has a negative impact on GPS/EDL schedules where now there are longer periods with high expected value of processor share of RT tasks ($E[s(t)]$). SPS on the other hand adapts to the available idle time, coming up with a schedule for the RT jobs such that they encounter fewer deadline misses, while the TS jobs also get smaller response times. So this is a win-win situation.

Let us construct the task set for this scenario. The 100 paying clients are represented as 100 SRT tasks. Each SRT task has a mean utilization M_i and reservation utilization R_i. And let, $\sum M_i = U_{srt} = 0.65$ and $\sum R_i = R = 0.80$. The 20 other customers are represented as TS tasks, with cumulative mean bandwidth requirement of 0.15 of available capacity. And $\Phi(0.4)$ jobs do not pay for the service. Table 4.2 summarizes the results.

Note that EDL performs significantly better than the other algorithms in terms of the measures TS response time $\Phi(0.10)$ and $\Phi(0.20)$, SPS nearly catches up with EDL at $\Phi(0.40)$. But in terms of deadline overruns for the paying customers, EDL has nearly 0.2% deadline misses, which is twice more than SPS (0.1%). As compared to Priority and GPS though, EDL and SPS reduce the number of non paying packets by nearly 3 times.

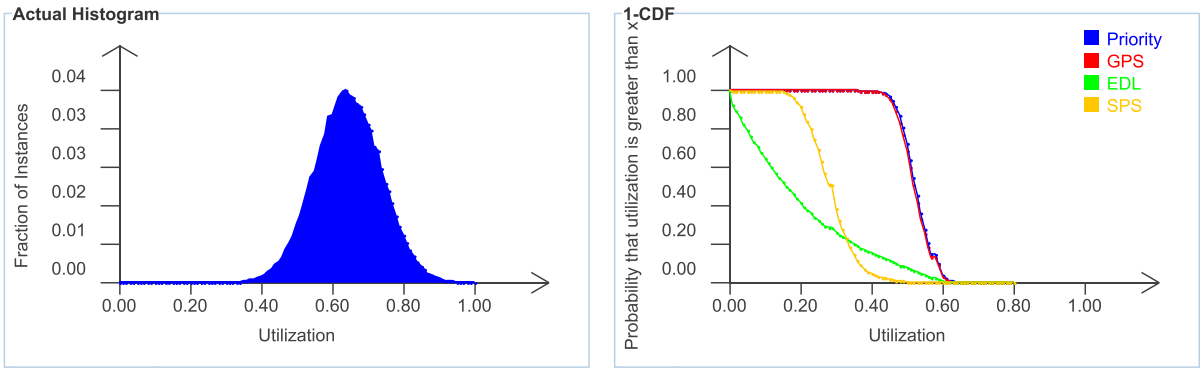
The important thing to note here is there may be a possibility to improve TS response time $\Phi(0.1)$ and $\Phi(0.2)$ for SPS in this scenario by more aggressively assigning idle time to the RT jobs, so that the SPS $g(.)$ function resembles EDL more. But that would come with a price, in terms of more deadline overruns for the SRT tasks.

Stats	Scheduler			
	Priority	GPS	EDL	Proposed
Simulation time	100002	100002	100002	100002
SRT jobs completed	117469	117470	117467	117469
SRT job overruns	329 (0.28%)	221 (0.19%)	223 (0.19%)	118 (0.10%)
Mean scaled overrun time	0.0042	0.0034	0.0032	0.0026
Overrun $\Phi(0.0)$	329 (0.28%)	221 (0.19%)	223 (0.19%)	118 (0.10%)
Overrun $\Phi(0.10)$	274 (0.23%)	192 (0.16%)	192 (0.16%)	114 (0.10%)
Overrun $\Phi(0.20)$	235 (0.20%)	169 (0.14%)	168 (0.14%)	109 (0.09%)
Overrun $\Phi(0.40)$	186 (0.16%)	138 (0.12%)	137 (0.12%)	100 (0.09%)
Overrun $\Phi(0.80)$	121 (0.10%)	100 (0.09%)	107 (0.09%)	88 (0.07%)
Overrun $\Phi(1.6)$	81 (0.07%)	73 (0.06%)	72 (0.06%)	61 (0.05%)
TS jobs completed	22342	22343	22346	22342
Mean scaled TS response time	0.4802	0.4623	0.1830	0.2583
TS response time $\Phi(0.10)$	15533 (69.52%)	14789 (66.19%)	4981 (22.29%)	9195 (41.16%)
TS response time $\Phi(0.20)$	11725 (52.48%)	11126 (49.80%)	3503 (15.68%)	5625 (25.18%)
TS response time $\Phi(0.40)$	7392 (33.09%)	7043 (31.52%)	1946 (8.71%)	2838 (12.70%)
TS response time $\Phi(0.80)$	3265 (14.61%)	3123 (13.98%)	800 (3.58%)	1115 (4.99%)
TS response time $\Phi(1.6)$	1063 (4.76%)	1021 (4.57%)	314 (1.41%)	421 (1.88%)

Table 4.2: Summary statistics for (N_{srt}=100,U_{srt}=0.65,R=0.80,N_{ts}=20,U_{ts}=0.15).

There is no clear way to handle this trade-off. And this is one avenue for future work.

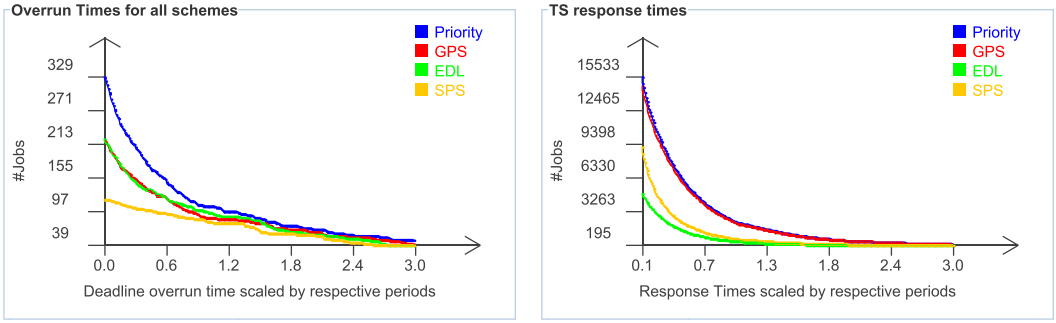
Note that EDL performs significantly better in terms of $\Phi(0.1)$ and $\Phi(0.2)$ as compared to SPS. But SPS catches up with EDL at around $\Phi(0.4)$. EDL pays the price for this performance in encountering more deadline overruns. Now SPS schedule can be made to resemble EDL schedule by better accounting of idle times. But is this required ? The reason why EDL performs better for in terms of $\Phi(0.1)$ and $\Phi(0.2)$ is because when the RT jobs are waiting under EDL, the TS jobs get serviced at a faster rate. While under SPS, the RT jobs take a certain share of processor. It would be interesting to know if the TS response times can be further improved under SPS without incurring additional SRT deadline overruns. And we would like to address this problem in future.



(a) Cumulative SRT utilization histogram

(b) Cumulative RT utilization (1-cdf)

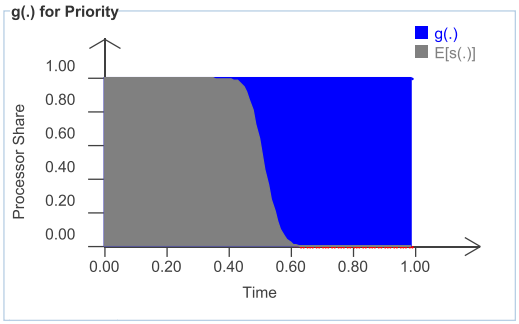
Figure 4.12: (N_{srt}=100,U_{srt}=0.65,R=0.80,N_{ts}=20,U_{ts}=0.15) execution time requirement distribution



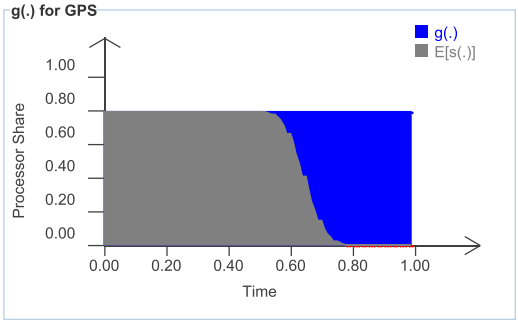
(a) SRT overrun $\Phi(.)$

(b) TS response times $\Phi(.)$

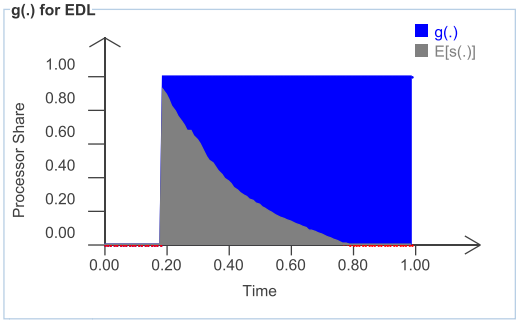
Figure 4.13: (N_{srt}=100,U_{srt}=0.65,R=0.80,N_{ts}=20,U_{ts}=0.15) $\Phi(.)$ values for SRT and TS tasks



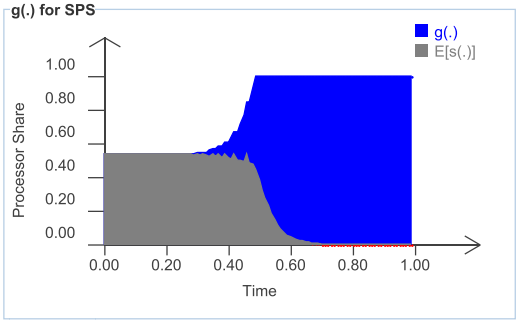
(a) g(.) for Priority



(b) g(.) for GPS



(c) g(.) for EDL



(d) g(.) for SPS

Figure 4.14: (N_{srt}=100,U_{srt}=0.65,R=0.80,N_{ts}=20,U_{ts}=0.15) g(.) functions for the four schemes

4.11 Summary

We propose a scheduling framework for variable requirement SRT tasks, where a SRT job has two execution phases. In the first phase it is scheduled as a RT job with the requirement to finish certain execution (R_i or reservation) before its deadline. If the RT job is still active after R_i RT allocation, then the SRT job is scheduled as a TS job for the remainder of its execution time.

We examined four scheduling algorithms to schedule the SRT tasks - Priority which gives priority to RT jobs over TS jobs,GPS which reserves constant processor share of $\sum R_i/P_i$ for the RT jobs, modified EDL which delays RT jobs while still meeting their deadlines and the proposed SPS algorithm.

CHAPTER 5
Experimental Setup

In the previous sections we presented the theoretical analysis, which establishes the usefulness of SPS, but given the unconventional nature of SPS and probabilities playing an important part in scheduling, it is difficult to form a good intuitive understanding of how, why and by how much does SPS improve scheduling performance as compared to current algorithms. To build this understanding, which is not only useful to build familiarity with SPS but also to get insights into the factors determining job response times while guaranteeing allocation to RT jobs, we present a wide range of experiments, along with detailed discussion on why the experiments were conducted and what were the performance benefits.

In this chapter, the simulation results are presented for a wide range of scenarios. We have a Java testbed which is automated to run and document the results for the experiments. The experimental report contains all relevant information about the experiment, and should be easy to follow once familiarity is built with the reporting mechanism.

In the following sections, we first explain the experimental setup and the SPS scheduling algorithm. This is followed by a in-depth discussion of the experiment reporting format by running through an example. This section should build familiarity with the figures and tables explain in detail the factors impacting the performance of SPS through relevant set of experiments. Finally, we present detailed experiment reports for all the experiments discussed in this section.

5.1 Experiment Parameters

The task set was generated based on the following parameters.

- N_{srt} – number of SRT tasks
- U_{srt} – mean cumulative utilization of SRT tasks
- R – cumulative reservation utilization of SRT tasks

- N_{ts} – number of TS tasks
- U_{ts} – mean cumulative utilization of TS tasks
- P_{min} – minimum period of any task (30 ms)
- P_{max} – maximum period of any task (200 ms)

The tuple (N_{srt}=50,U_{srt}=0.50,R=0.65,N_{ts}=50,U_{ts}=0.35) describes an experiment, where N_{st} is the number of SRT tasks, U_{srt} is average cumulative SRT utilization, R is the cumulative N_{ts} is the number of TS tasks and U_{ts} is the cumulative TS utilization.

The task set is generated using N_{srt}, U_{srt}, R, N_{ts} and U_{ts}. Each SRT and TS task is assumed to have normally distributed execution time requirement (bell shaped distribution). The reason for this choice of workload is that normal distribution has the properties where most values are near mean and there are fewer values away from the mean. This kind of variability has he most scope for improvement since allocating for values greater than mean may be necessary to provide allocation guarantees, but in most cases, the values are near the mean.

The other reason for considering normal distribution is that since for SPS, we require the cumulative requirement of RT tasks rather than their individual requirements, and from Central Limit Theorem, the sum of a large number of random variables follows normal distribution since the constituent random variables cancel out each others variability. So, for large number of independent tasks, their cumulative execution time requirement distribution is most likely to be normal.

5.1.1 SRT Tasks Generation

The mean utilizations of SRT tasks were chosen to be N_{srt} random numbers (normal distribution) with their sum equals U_{srt}. Denote the mean utilization for task τ_i as M_i. Then $\sum M_i = U_{srt}$. The periods were chosen as uniformly distributed integers in the range of $[30, 200]$.

The individual reservations for SRT tasks were chosen as follows. Suppose SRT task τ_i has mean utilization of M_i. Then its reservation R_i is given by $M_i * (R/U_{srt})$.

Note that,

$$\sum R_i = \sum M_i * (R/U_{srt}) = R\frac{\sum M_i}{U_{srt}} = R$$

5.1.2 TS Tasks Generation

The mean utilizations of TS tasks were chosen to be N_{ts} random numbers (normal distribution) with their sum equals U_{ts}. The periods were chosen as uniformly distributed integers in the range of $[30, 200]$.

5.2 Simulation Platform

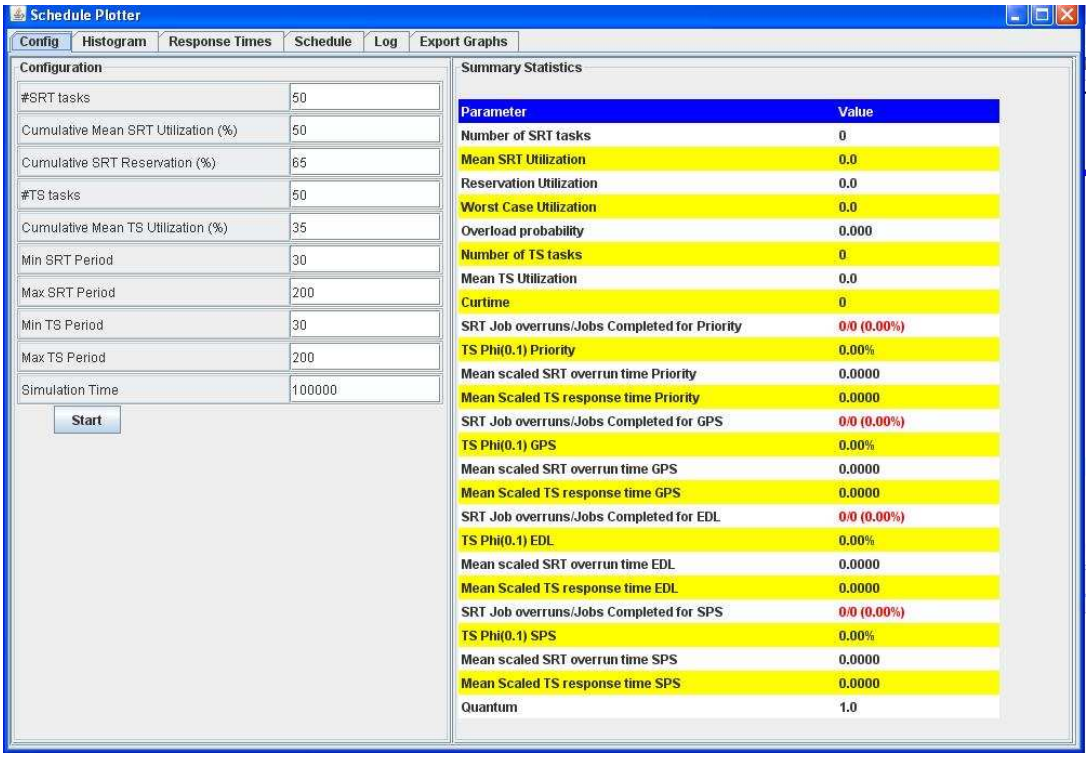


Figure 5.1: Java GUI input form

We implemented a Java based GUI to simulate task sets with Priority, GPS, EDL and SPS schedulers. The interface looks like this Fig 5.1. The input parameters are -

- N_{srt} – Number of SRT tasks
- U_{srt} – Cumulative mean utilization of SRT tasks
- R – Cumulative reservation utilization of SRT tasks
- N_{ts} – Number of TS tasks

- U_{ts} – Cumulative mean utilization of the TS tasks
- $\min P_{srt}$ – Minimum SRT period
- $\max P_{srt}$ – Maximum SRT period
- $\min P_{ts}$ – Minimum TS period
- $\max P_{ts}$ – Maximum TS period
- Simulation time – This is the duration in time units for which the simulation is run.

As can be seen in Fig 5.1, the right panel shows summary statistics for the experiment. The last entry in summary statistics is the allocation quantum size. We fixed this to be 1. This value represents the allocation granularity for emulating GPS. The quantum size of 1 means that at the end of each quantum, the processor time allocated to each task is what an ideal GPS schedule would have allocated.

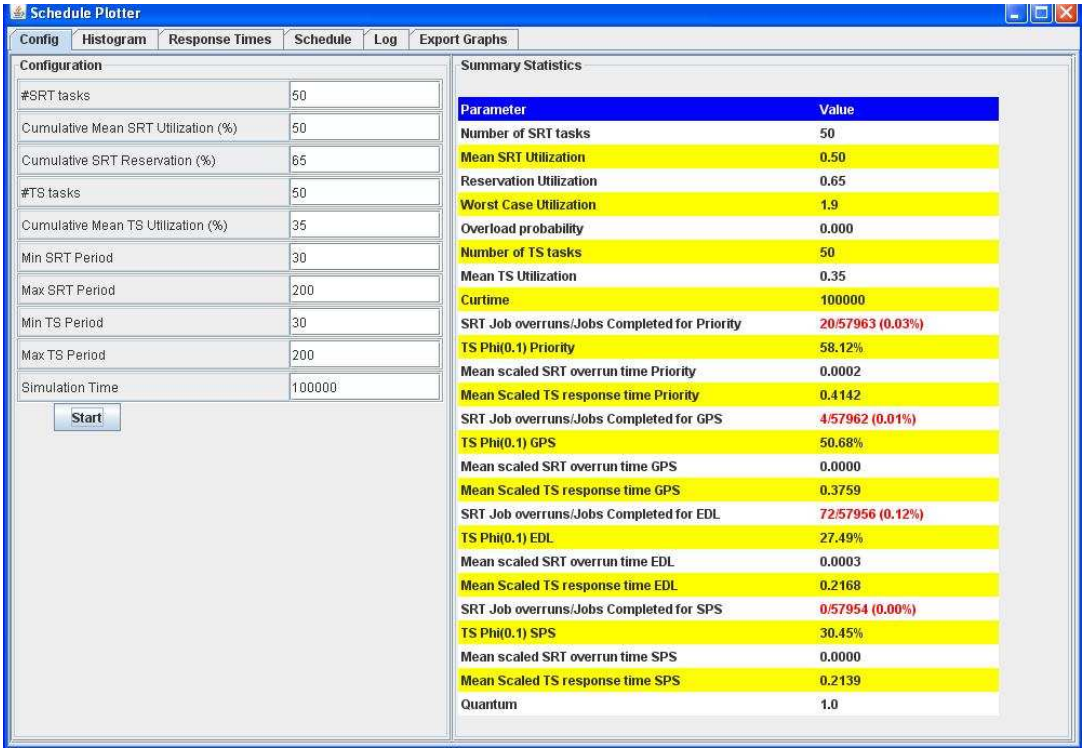


Figure 5.2: Java GUI summary statistics at the end of simulation.

Fig 5.2 shows the summary statistics after the experiment is over. Along with these statistics, the other tabs show the distribution of RT requirements, the $\Phi(.)$ functions for RT overruns and TS response times and the $g(.)$ functions for all the four scheduling algorithms. These graphs are dynamically updated as the simulation progresses. There are also multiple tabs which show dynamic performance graphs as the simulation progresses.

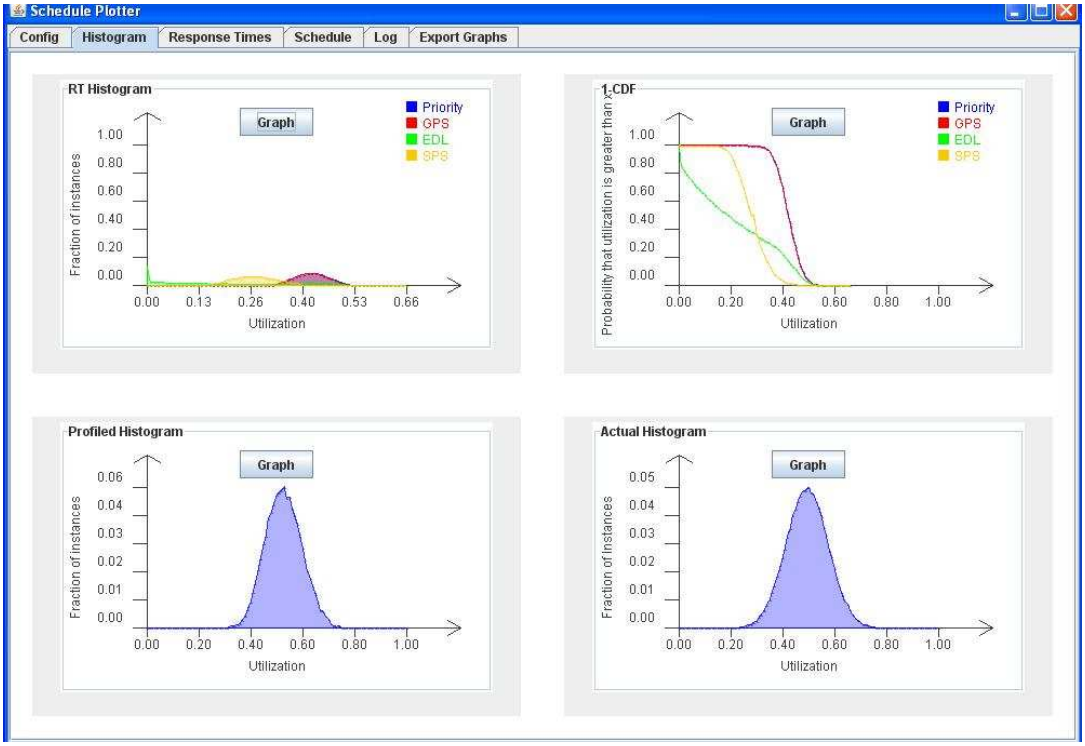


Figure 5.3: Java GUI RT requirement distribution

Fig 5.3 shows the RT requirement distribution for the four scheduling algorithms, as well as the profiled and actual cumulative RT requirement histogram. The RT requirement distribution is the amount probability of the amount of cumulative RT allocation that is required for the RT jobs. Note that the RT jobs can get idle allocation, and this idle allocation does not count as RT allocation. In particular, note that the RT requirement histogram for SPS and EDL shows that there is higher probability that the cumulative RT requirement of the RT jobs is lower as compared to that for GPS/Priority. This is because under EDL/SPS, the RT jobs get more idle allocation (because they are active longer and hence have greater chances of claiming idle time), hence there RT requirements are lesser under EDL/SPS. The lower pair of graphs is just for verification purposes to visually verify that the profiled SRT requirements match the actual (theoretical) cumulative requirement distribution.

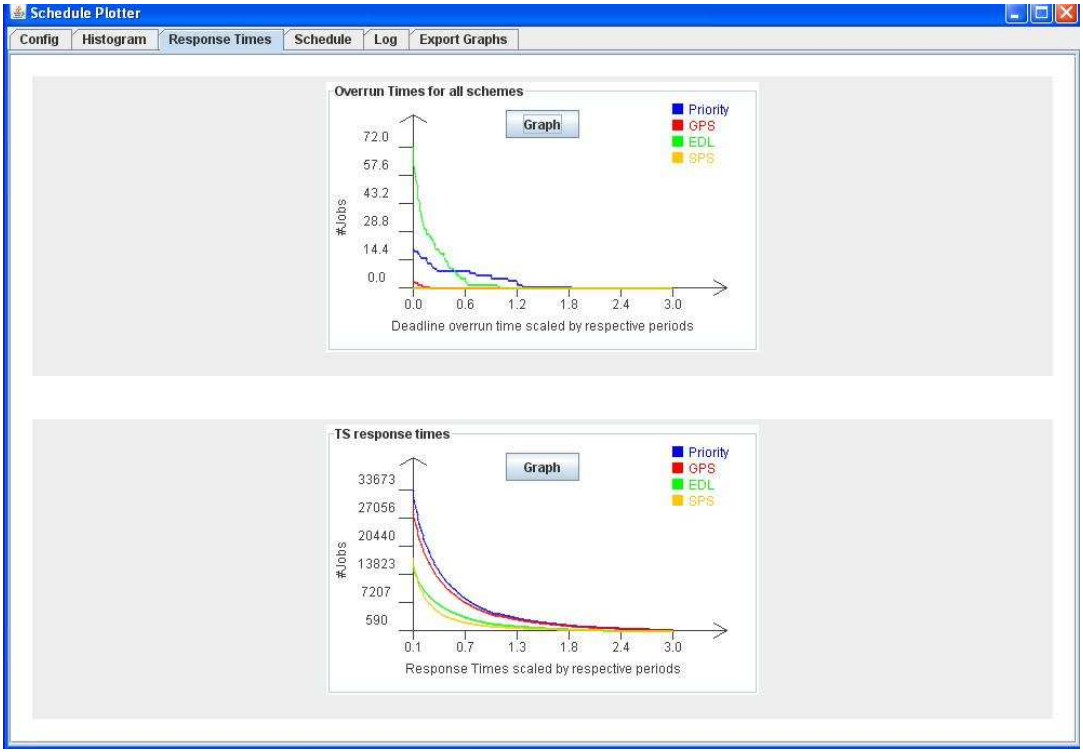


Figure 5.4: Java GUI $\Phi(.)$ functions

Fig 5.4 shows the SRT overrun time $\Phi(.)$ functions and the TS overrun time $\Phi(.)$ functions. Note that for given x, higher the value of $\Phi(x)$ means there are greater number of jobs with their scaled response time or overrun time greater than x. Thus, lower values of $\Phi(.)$ are better. Also, we maintain and show the $\Phi(.)$ function over the range $[0, 3]$. This is not just for information, rather the $\Phi(.)$ measure would be useful in checking whether the performance requirements of the applications are met or not.

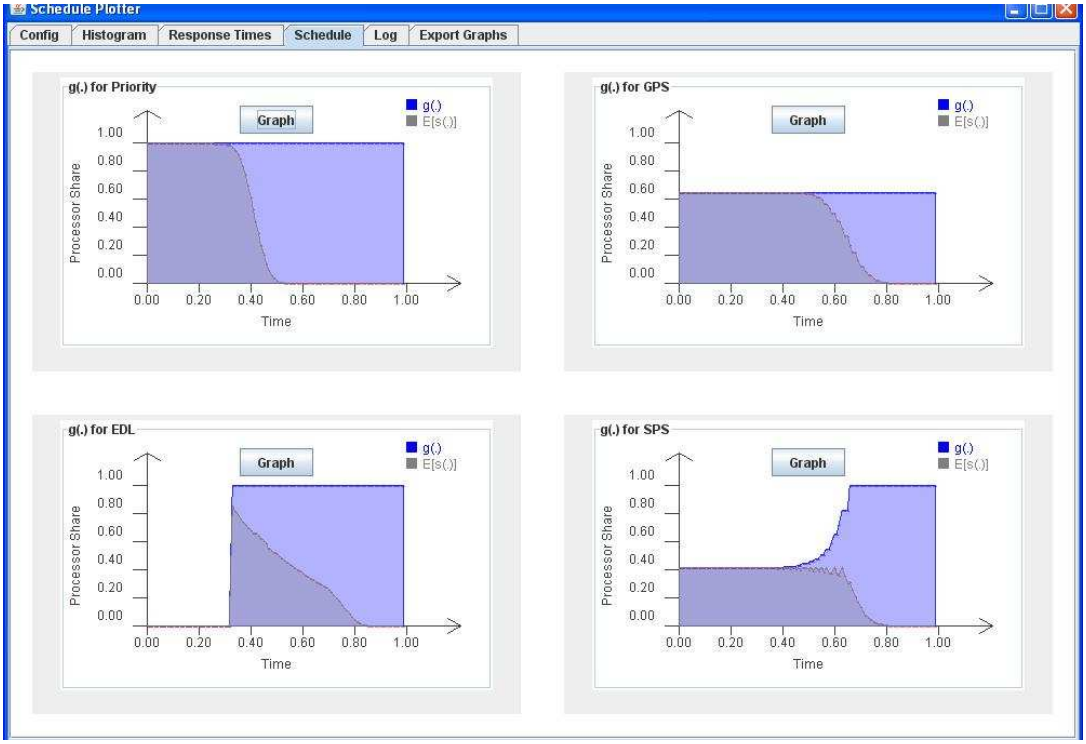


Figure 5.5: Java GUI $g(.)$ functions

Fig 5.5 shows the $g(.)$ functions for the four scheduling algorithms, as well as the expected value of $s(t)$. This is the key graph showing the difference between the four algorithms. Understanding this graph is the key to understanding all the results. The lighter shaded region is the $g(.)$ function for a unit period job with execution requirement distribution same as the cumulative RT utilization of all the RT jobs. The darker region shows the expected cumulative processor share of the RT jobs. The shape of SPS $g(.)$ function lies in between that of GPS and EDL, matching EDL when the processor is under-loaded and GPS when the processor is heavily loaded. For intermediate values, the SPS $g(.)$ function is hybrid between the EDL $g(.)$ function and the GPS $g(.)$ function, with maximum expected processor share of RT jobs lesser than both.

Our Java platform not only reports these dynamic graphs but also generates a latex report for the experiment. In the following section we look at the generated report for typical experiment.

5.3 Typical Experiment

A typical experiment is reported as follows.

The tuple (N_{srt}=50,U_{srt}=0.50,R=0.65,N_{ts}=50,U_{ts}=0.35) represents the parameters for the experiment as described in previous section.

Stats	Scheduler			
	Priority	GPS	EDL	Proposed
Simulation time	100002	100002	100002	100002
SRT jobs completed	57963	57962	57951	57953
SRT job overruns	20 (0.03%)	4 (0.01%)	67 (0.12%)	0 (0.00%)
Mean scaled overrun time	0.0002	0.0000	0.0002	0.0000
Overrun $\Phi(0.0)$	20 (0.03%)	4 (0.01%)	67 (0.12%)	0 (0.00%)
Overrun $\Phi(0.10)$	16 (0.03%)	2 (0.00%)	34 (0.06%)	0 (0.00%)
Overrun $\Phi(0.20)$	13 (0.02%)	0 (0.00%)	24 (0.04%)	0 (0.00%)
Overrun $\Phi(0.40)$	9 (0.02%)	0 (0.00%)	11 (0.02%)	0 (0.00%)
Overrun $\Phi(0.80)$	7 (0.01%)	0 (0.00%)	1 (0.00%)	0 (0.00%)
Overrun $\Phi(1.6)$	1 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
TS jobs completed	57939	57943	57952	57955
Mean scaled TS response time	0.4180	0.3775	0.2243	0.2143
TS response time $\Phi(0.10)$	33417 (57.68%)	28904 (49.88%)	15911 (27.46%)	17272 (29.80%)
TS response time $\Phi(0.20)$	23635 (40.79%)	20202 (34.87%)	10951 (18.90%)	9738 (16.80%)
TS response time $\Phi(0.40)$	14255 (24.60%)	12345 (21.31%)	6624 (11.43%)	4787 (8.26%)
TS response time $\Phi(0.80)$	6667 (11.51%)	5863 (10.12%)	3143 (5.42%)	2214 (3.82%)
TS response time $\Phi(1.6)$	2612 (4.51%)	2359 (4.07%)	1323 (2.28%)	1118 (1.93%)

Table 5.1: Summary statistics for (N_{srt}=50,U_{srt}=0.50,R=0.65,N_{ts}=50,U_{ts}=0.35)

Table 5.1 presents the summary statistics for the experiment.

- Simulation time - The time for which the simulation was run.
- SRT jobs completed - The number of SRT jobs completed under each of the scheduling algorithms. This may differ between the scheduling algorithms, because while the RT components of all the SRT tasks finish by their deadline, the overrun TS components have different response times under different scheduling algorithms.
- SRT job overruns - The number of SRT jobs missing their deadline. Note that all the RT components meet their deadline. But if the job requirement is greater than its reservation then it may miss its deadline, and this is the number of SRT jobs missing their deadline under each of the four scheduling algorithms.
- Mean scaled overrun time - This is the mean of overrun times of all the jobs divided by their respective periods. Note that SRT jobs finishing on or before their deadline have overrun time of 0.

- Overrun $\Phi(x)$ - This is the number (and percentage) of SRT jobs with scaled overrun time greater than x. Note that SRT jobs with scaled overrun time between 0-0.4 may not have significant impact on performance.

- TS jobs completed - The number of TS jobs completed under all the four algorithms. It may differ between the four algorithms.

- Mean scaled TS response time - The mean of scaled response times of all the TS jobs. Smaller value is better.

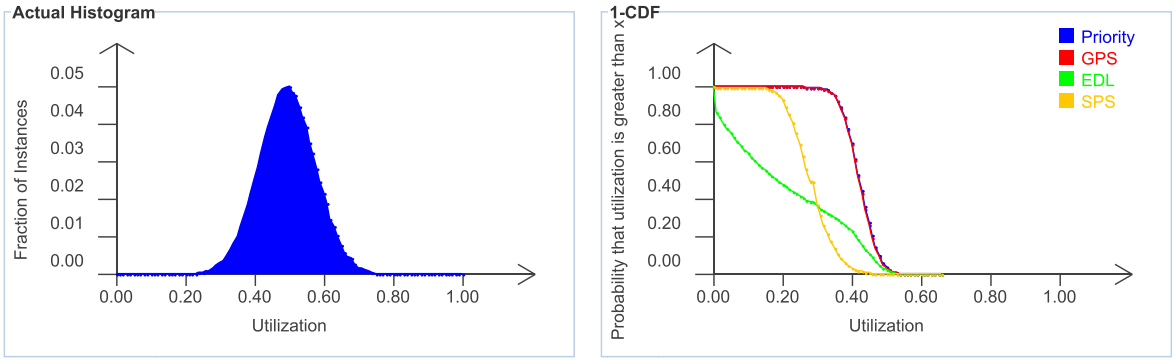
- TS Response time $\Phi(x)$ - This is the number (and percentage) of all TS jobs with their scaled response time greater than x.

This table provides the summary of key results, but it does not provide any explanation on the performance. The next set of figures provide that explanation. This table serves to highlight the results, and provides a single place where all the performance aspects of the experiment are available.

Note that smaller response time values are better, and smaller values of $\Phi(.)$ functions are better. It should also be noted that the response times are measured from time 0, and at time 0, the SPS schedule is like the GPS schedule. In the absence of any information, the SPS schedule resembles that of GPS, because the SPS schedule is calculated assuming constant execution time requirement of R_i for the SRT task. As time progresses, the probability distribution χ_{rt} is constructed and the shape of SPS changes according to it. So in cases where GPS does not perform well, SPS is at a disadvantage because it starts of as GPS. These experiments do not ignore any response time values. So in steady state, the SPS performance would be better than what is reported here because in steady state, SPS would have a more or less constant $s(t)$ function.

The reason why scaled response times are compared is because there is no straightforward way to directly compare response times of jobs with varying job sizes. So we made the implicit assumption that the response time sensitiveness of tasks is proportional to their period. That is, smaller period tasks are more sensitive to their response time as compared to tasks with greater period. This is a reasonable assumption if the tasks are assigned periods based upon their time sensitiveness. Also, there is little meaning to period of TS tasks, but for our simulations we assumed that TS jobs arrive as periodic jobs, just like SRT jobs with the only difference that the TS tasks have 0 reservation and hence no RT component. In actual systems, the TS task time sensitiveness may not be directly proportional to their period and also the TS jobs may not arrive periodically. But by choosing

periods randomly for the TS tasks we insure that the TS tasks arrive randomly and hence create a random arrival time scenario, while arriving at fixed rates.



(a) Cumulative SRT utilization histogram

(b) Cumulative RT utilization (1-cdf)

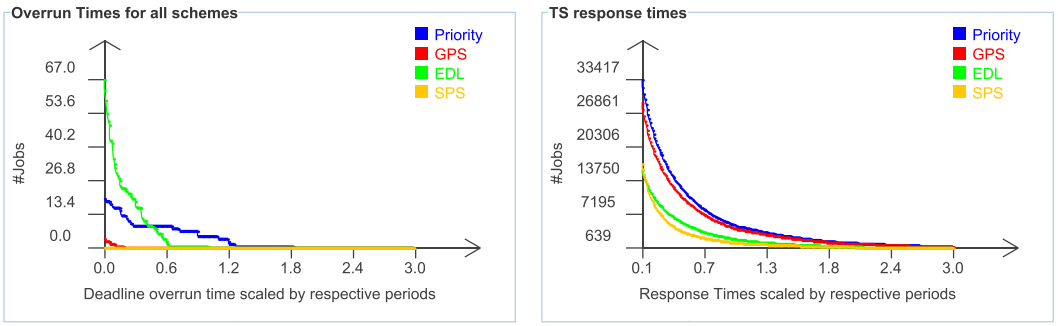
Figure 5.6: (N_{srt}=50,U_{srt}=0.50,R=0.65,N_{ts}=50,U_{ts}=0.35) execution time requirement distribution

The first set of figures (Fig 5.6) shows the cumulative utilization distribution histogram for the SRT tasks. Note that since we generate SRT workload using normal distribution hence the cumulative histogram is also normal. Also note that while the cumulative worst case utilization requirement may be considerably greater than the mean utilization requirement the probability that the actual cumulative execution requirement is greater than the mean decreases as the number of jobs in the system is increased. The second graph shows the tail distribution of cumulative RT utilization of the RT components of the SRT tasks. Remember that the allocation to the RT component of the SRT job is given by the processor time received by this job due to its RT share. If the RT job receives any idle allocation then that does not count towards its RT allocation.

Also for given value of utilization on the X-axis, the height of the curve represents the probability that the cumulative RT utilization of the RT components of the SRT jobs will be greater than that value of x. Note that for EDL, the probability distribution is such that the cumulative RT component finishes has greater probability to use lesser utilization. This is because, under EDL, the RT jobs get maximum possible idle allocation which does not count towards RT allocation. Thus, under EDL the RT jobs require minimum RT allocation. And hence the shape of the distribution.

The reason why SPS resembles EDL is because for SPS we are using the cumulative RT requirement distribution given by EDL. The actual cumulative RT requirement for SPS would be a curve which will lie between the EDL and GPS curves. That is under SPS, the RT jobs get greater idle allocation as compared to GPS, but lesser as compared to EDL. Idle allocation is the the allocation available to RT job outside its processor share if

there are no active TS jobs present.



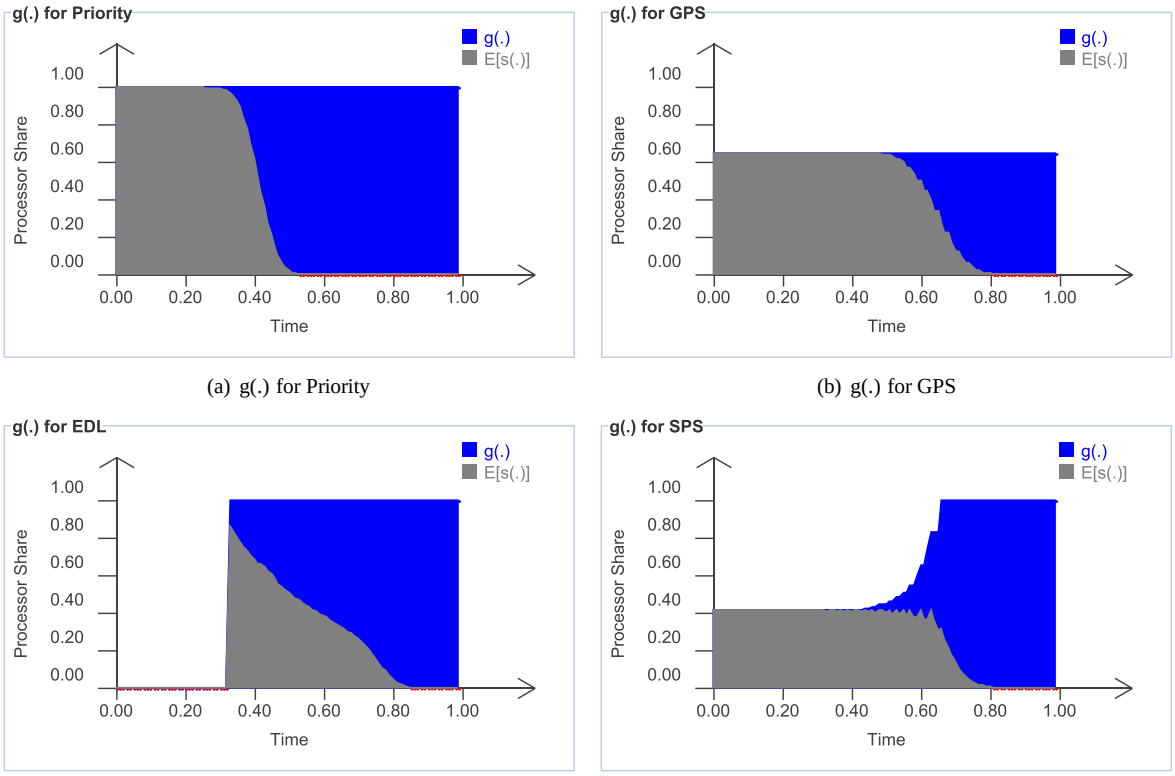
(a) SRT overrun $\Phi(.)$ (b) TS response times $\Phi(.)$

Figure 5.7: (N_{srt}=50,U_{srt}=0.50,R=0.65,N_{ts}=50,U_{ts}=0.35) $\Phi(.)$ values for SRT and TS tasks

Fig 5.7 graphically shows the $\Phi(.)$ function for SRT overrun times as well as TS response times. The reason these two are considered separately is because under the scheduling algorithm we used for simulation the SRT overrun TS jobs are given priority over normal TS jobs. This is based on the assumption that the SRT overrun jobs would usually be very small, and finishing them early is more important than TS response times. In comparison to the approach of treating the SRT overrun jobs at par with other TS jobs, this approach provides much smaller SRT overrun times. Also this formalization helps to bring about the problem with EDL scheduling where the SRT deadline misses are increased if the mean processor utilization is high. If all the TS jobs are treated equally, then this impact is diluted.

The other thing to note is that SPS provides response time benefits over x as high as 1, which is important, because the jobs with scaled response time greater than 0.4 would have greater performance impact as compared to jobs with their scaled response time in the range 0 to 0.4. This is because, tasks may not be very sensitive to small increase in response times. Although, smaller response times may be better for some tasks, and as we would show through the experiments, SPS performs the best in terms of $\Phi(0 - 0.4)$ measures, so that is no problem.

Fig 5.8 shows the $g(.)$ and $E[s(.)]$ functions for the four scheduling algorithms. This is the key figure representing the differences between the four scheduling algorithms. The first thing to note is that under SPS the expected processor share of the RT jobs is nearly constant, and lesser as compared to any of the other three algorithms. While under SPS, the processor share of RT jobs may vary with progress, under the other three scheduling algorithms it is a constant value or 0. The SPS $s(t)$ function also shows that the higher processor share for RT jobs is reached only rarely, as seen by the expected value of processor share for the interval when



(a) g(.) for Priority

(b) g(.) for GPS

(c) g(.) for EDL

(d) g(.) for SPS

Figure 5.8: (N_{srt}=50,U_{srt}=0.50,R=0.65,N_{ts}=50,U_{ts}=0.35) g(.) functions for the four schemes

the $g(.)$ function value is high. This is the novelty of our approach. Until now, the scheduling algorithms focused on giving the RT jobs a constant processor share, but this approach though appropriate for constant requirement RT jobs, does not perform well with variable requirement RT jobs. And varying processor share of RT jobs is an effective tool in handling requirement variability.

5.4 Experiments and Observations

Experiments were conducted to study SPS performance in following setups (Table 5.2).

Name	Description
Impact of SRT utilization on SRT overrun times	Measure the impact of SRT overrun time with respect to cumulative mean utilization of SRT tasks. It is assumed that there are no TS tasks in the system.
Impact of SRT utilization on TS response times	Measure the impact of cumulative mean SRT utilization on TS response times.
Impact of SRT requirement variability on TS response times	Measure the impact of SRT reservation (for given SRT mean utilization) on TS response times
Impact of TS utilization in TS response Time	Measure the impact of TS workload on TS response times (for given SRT mean and reservation utilization)
Impact of size of RT jobs on TS response Times	Measure the impact of size of SRT jobs on TS response times.
Impact of size of TS jobs on TS response Times	Measure the impact of size of TS jobs on TS response times.

Table 5.2: Experiment sets

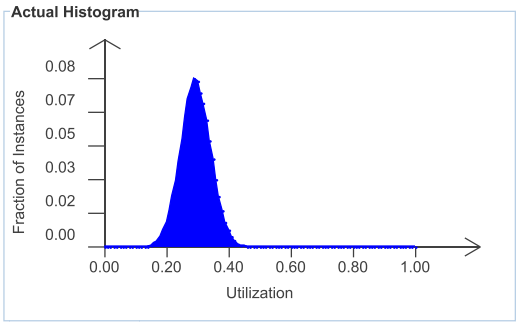
5.4.1 Impact of SRT Utilization on SRT Overruns

- For cumulative mean SRT utilization of 0.30, 0.40 and 0.50, the probability that the system utilization is greater than 1 is nominal (Fig 5.9). So while the cumulative reservation is 0.35, 0.45 and 0.55 respectively, the SRT overruns finish before deadline, since the remaining processor share (after allocating the reservation) is considerable.

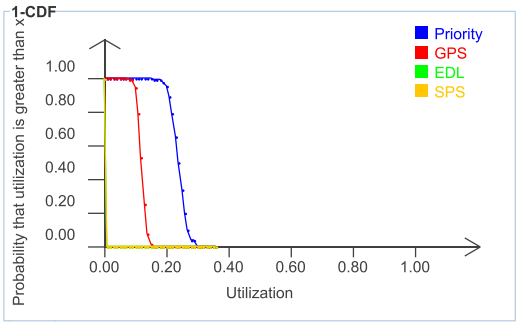
  Also not that EDL and SPS schedules are very similar, this is because RT jobs finish using just idle allocation under both the schemes. So the probability that RT jobs require any RT allocation is nominal under EDL or SPS, as seen in the tail distribution (1-cdf) of cumulative RT requirements (Fig 5.9)

- For cumulative mean SRT utilization between 0.6 and 0.8, the performance benefits of EDL and SPS become evident over Priority and GPS (Fig 5.10). This is because under EDL and SPS, the RT job execution is delayed (Fig 5.10), giving better response times to the overrun jobs.

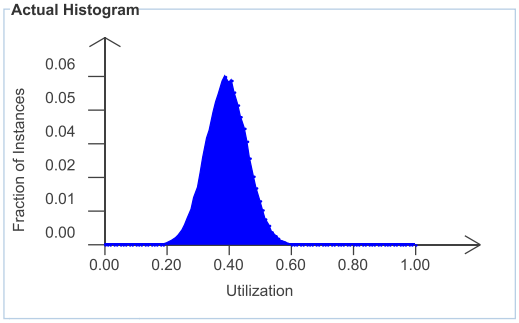
- For cumulative mean SRT utilization 0.9 or more, nearly all the schemes perform equally. This is because the processor is busy most of the time with the RT jobs (which gives similar $g(.)$ functions under all the four schemes). Also note that the cumulative RT requirement distribution is nearly same under all the



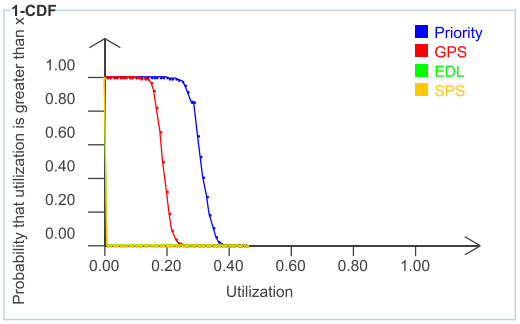
(a) (N_{srt}=50,U_{srt}=0.30,R=0.35,N_{ts}=0,U_{ts}=0.0)



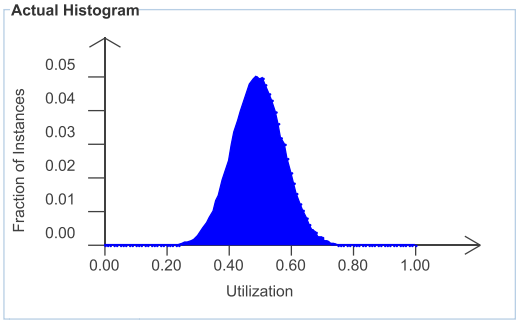
(b) (N_{srt}=50,U_{srt}=0.30,R=0.35,N_{ts}=0,U_{ts}=0.0)



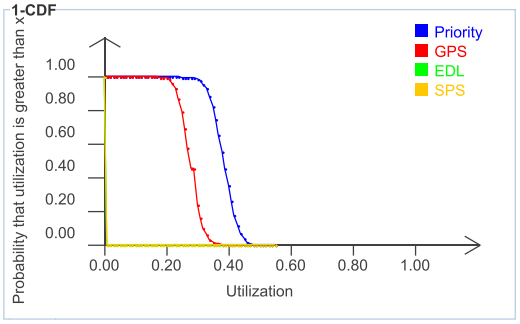
(c) (N_{srt}=50,U_{srt}=0.40,R=0.45,N_{ts}=0,U_{ts}=0.0)



(d) (N_{srt}=50,U_{srt}=0.40,R=0.45,N_{ts}=0,U_{ts}=0.0)



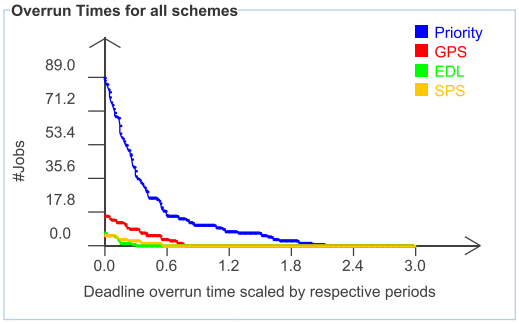
(e) (N_{srt}=50,U_{srt}=0.50,R=0.55,N_{ts}=0,U_{ts}=0.0)



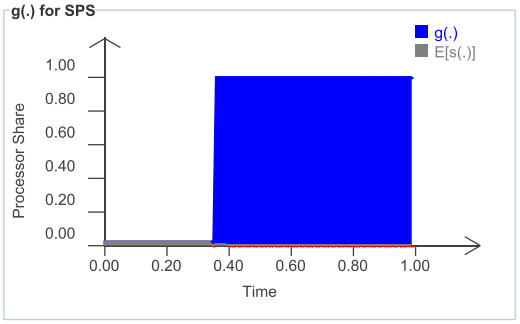
(f) (N_{srt}=50,U_{srt}=0.50,R=0.55,N_{ts}=0,U_{ts}=0.0)

Figure 5.9: This figure shows the cumulative SRT utilization histogram and RT requirement distribution for low to medium loaded system. Note that under SPS/EDL, the probability that the RT jobs require any RT allocation is nominal.

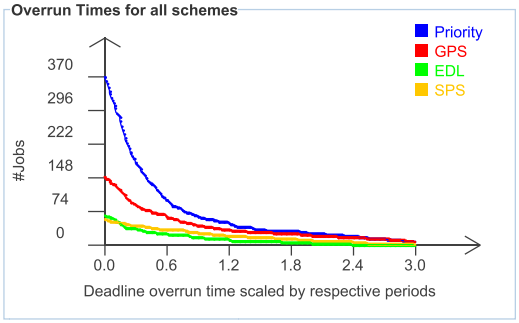
schemes (Fig 5.11) , and the processor is busy with RT jobs for majority of time, leaving little time for the overruns under any scheme.



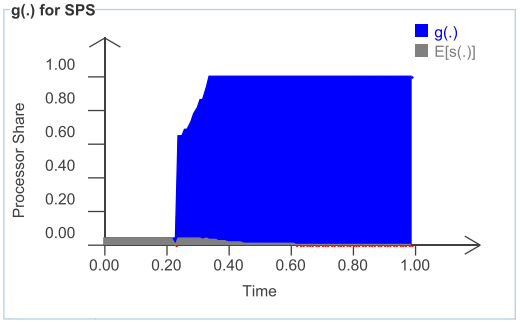
(a) (N_{srt}=50,U_{srt}=0.60,R=0.65,N_{ts}=0,U_{ts}=0.0)



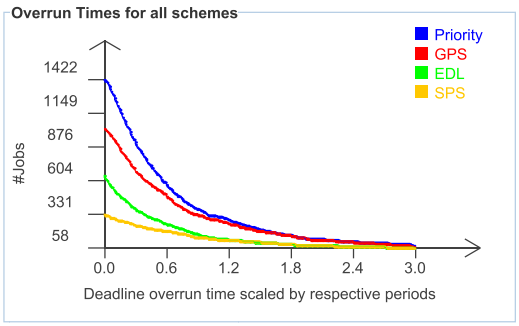
(b) (N_{srt}=50,U_{srt}=0.60,R=0.65,N_{ts}=0,U_{ts}=0.0)



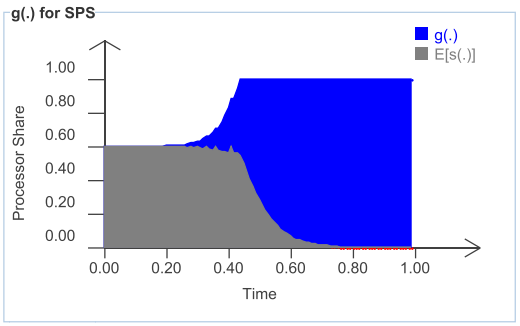
(c) (N_{srt}=50,U_{srt}=0.70,R=0.75,N_{ts}=0,U_{ts}=0.0)



(d) (N_{srt}=50,U_{srt}=0.70,R=0.75,N_{ts}=0,U_{ts}=0.0)

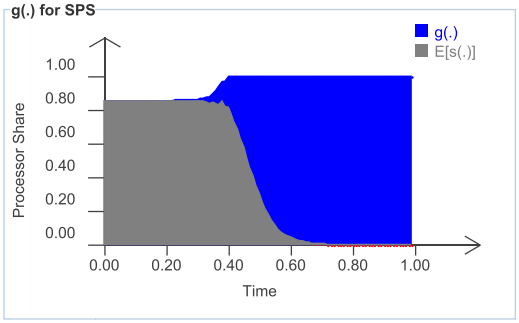


(e) (N_{srt}=50,U_{srt}=0.80,R=0.85,N_{ts}=0,U_{ts}=0.0)

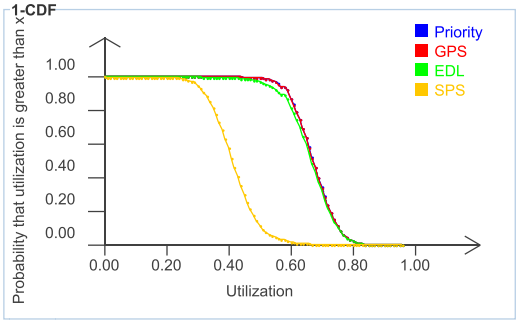


(f) (N_{srt}=50,U_{srt}=0.80,R=0.85,N_{ts}=0,U_{ts}=0.0)

Figure 5.10: Overrun times and $g(.)$ for SPS



(a) (N_{srt}=50,U_{srt}=0.90,R=0.95,N_{ts}=0,U_{ts}=0.0)



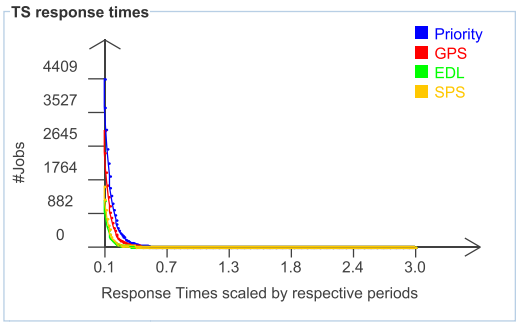
(b) (N_{srt}=50,U_{srt}=0.90,R=0.95,N_{ts}=0,U_{ts}=0.0)

Figure 5.11: SPS $g(.)$ and RT requirement distribution

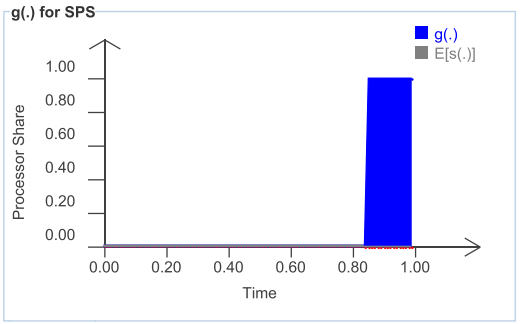
5.4.2 Impact of SRT Utilization on TS Response Times

While in the previous section we looked at a system with no TS tasks, in this section we look at a system which has a mix of SRT and TS tasks. SRT overrun tasks are scheduled with the TS tasks in LAS order.

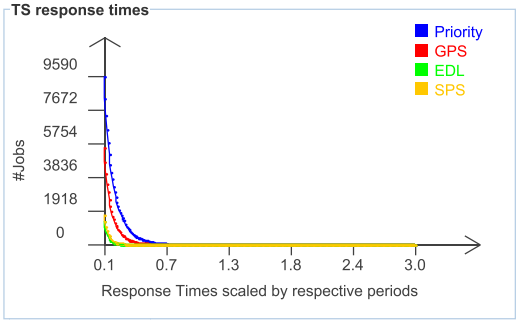
The aim of this set of experiments is to understand the impact of SRT utilization on TS response times. To do this we fix the number $N_{srt} = 50$, $N_{ts} = 50$ and $U_{ts} = 0.35$, and vary U_{srt} (while choosing R=$U_{srt} + 0.05$). This will enable us to understand how much is the effect of SRT utilization on the response times of TS tasks.



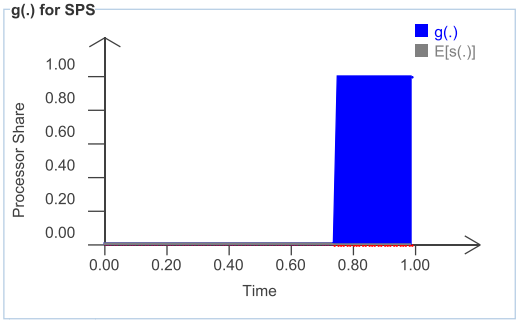
(a) (N_{srt}=50,U_{srt}=0.10,R=0.15,N_{ts}=50,U_{ts}=0.30)



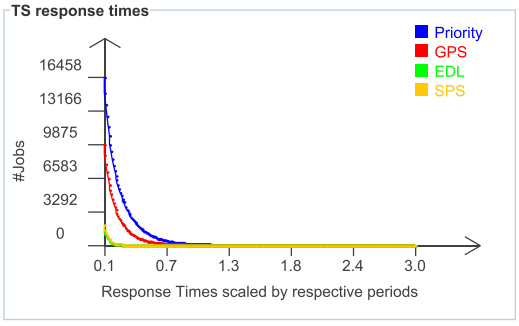
(b) (N_{srt}=50,U_{srt}=0.10,R=0.15,N_{ts}=50,U_{ts}=0.35)



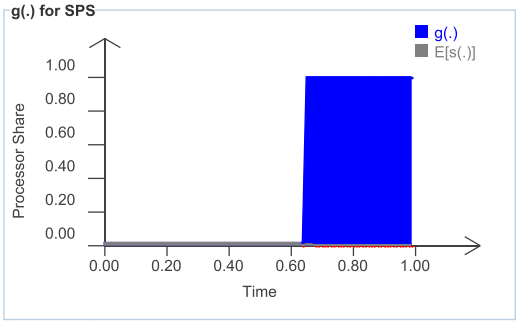
(c) (N_{srt}=50,U_{srt}=0.20,R=0.25,N_{ts}=50,U_{ts}=0.35)



(d) (N_{srt}=50,U_{srt}=0.20,R=0.25,N_{ts}=50,U_{ts}=0.35)



(e) (N_{srt}=50,U_{srt}=0.30,R=0.35,N_{ts}=50,U_{ts}=0.35)

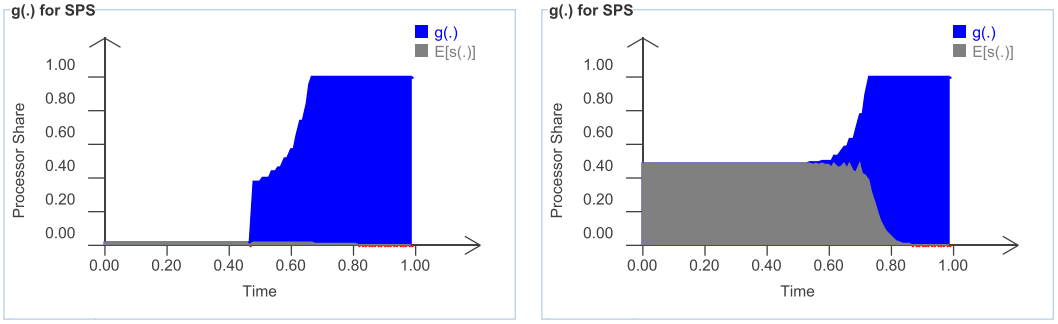


(f) (N_{srt}=50,U_{srt}=0.30,R=0.35,N_{ts}=50,U_{ts}=0.35)

Figure 5.12: Low utilization system, TS response time $\Phi(.)$ and SPS $g(.)$

Following are the key observations –

- Even for very low (0.1 - 0.3) cumulative SRT task utilization, the impact on TS response time is significant. and independent of scheduling algorithm used. This is because no matter how small cumulative mean utilization of SRT tasks be, the execution of RT job components of SRT tasks interferes with the execution of TS jobs, more so under Priority and GPS as compared to SPS and EDL. This can be seen from the $g(.)$ functions (Fig 5.12). Under SPS, the $g(.)$ function resembles the EDL $g(.)$ function i.e. the RT allocation is delayed maximally, thereby giving preference to TS tasks which is evident from the TS response time $\Phi(.)$ functions. Note that with high probability, the RT allocation is not actually required and the RT jobs finish using just idle allocation.



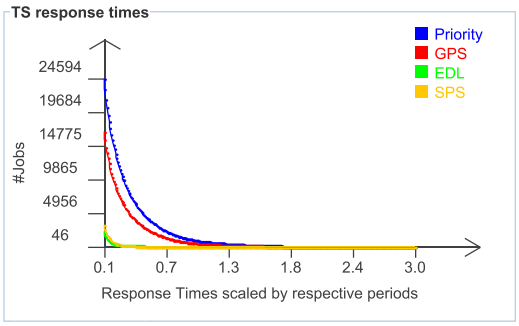
(a) (N_{srt}=50,U_{srt}=0.40,R=0.45,N_{ts}=50,U_{ts}=0.35) (b) (N_{srt}=50,U_{srt}=0.60,R=0.65,N_{ts}=50,U_{ts}=0.35)

Figure 5.13: Medium utilization system, $g(.)$ function

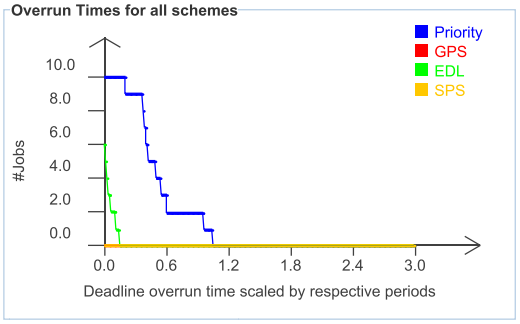
- For cumulative mean SRT utilization 0.4, 0.5 and 0.6, the cumulative mean overall utilization of the system (SRT and TS) is 0.75, 0.85 and 0.95 respectively. Due to the high mean overall utilization, the RT jobs get lesser idle allocation under all schemes. The effect of this can be seen on the $g(.)$ function for proposed algorithm (Fig 5.13). Note that their is a greater probability that the RT job requires its allocation, as compared to the previous low utilization scenarios where the RT jobs finished using just the idle allocation.

  Also, some SRT jobs miss their deadline under EDL and SPS. This is because under these two algorithms, the RT execution is delayed, hence the RT jobs finish later as compared to under Priority and GPS. Thus, the SRT overrun component is available later as compared to Priority and GPS, and it has to compete with the (nearly always present) TS jobs for processor. Thus, there is a higher probability of missing SRT deadline under EDL. SPS provides TS response times comparable to EDL scheme while keeping SRT deadline misses substantially smaller. This is because, the $g(.)$ function under SPS is a cross between GPS

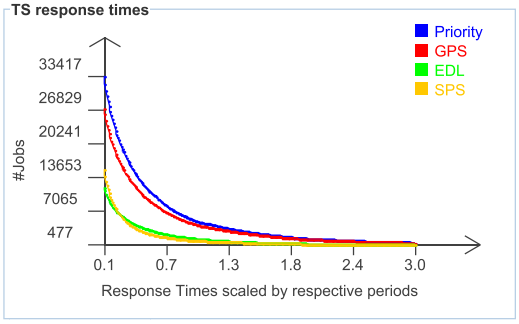
and EDL, so while the RT allocation is delayed, it is not delayed by too much. As the overall system utilization increases the SPS $g(.)$ function morphs from being like that of EDL to being like that of GPS.



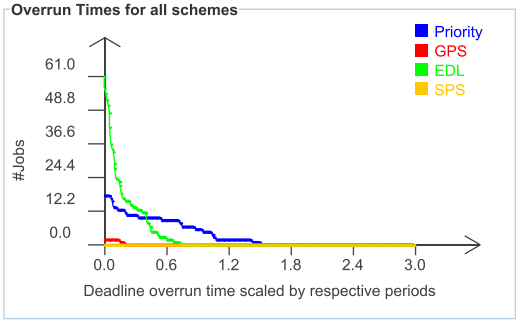
(a) (N_{srt}=50,U_{srt}=0.40,R=0.45,N_{ts}=50,U_{ts}=0.35)



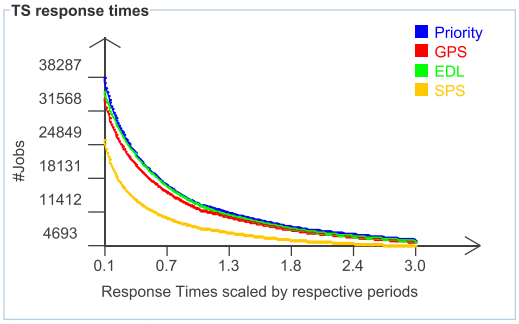
(b) (N_{srt}=50,U_{srt}=0.40,R=0.45,N_{ts}=50,U_{ts}=0.35)



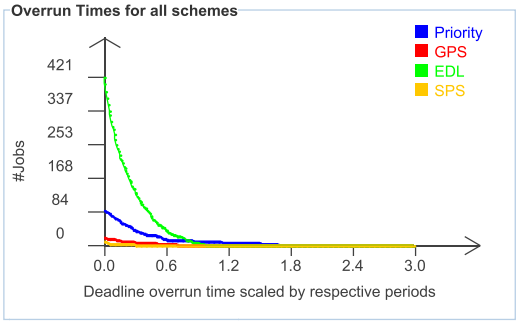
(c) (N_{srt}=50,U_{srt}=0.50,R=0.55,N_{ts}=50,U_{ts}=0.35)



(d) (N_{srt}=50,U_{srt}=0.50,R=0.55,N_{ts}=50,U_{ts}=0.35)



(e) (N_{srt}=50,U_{srt}=0.60,R=0.65,N_{ts}=50,U_{ts}=0.35)

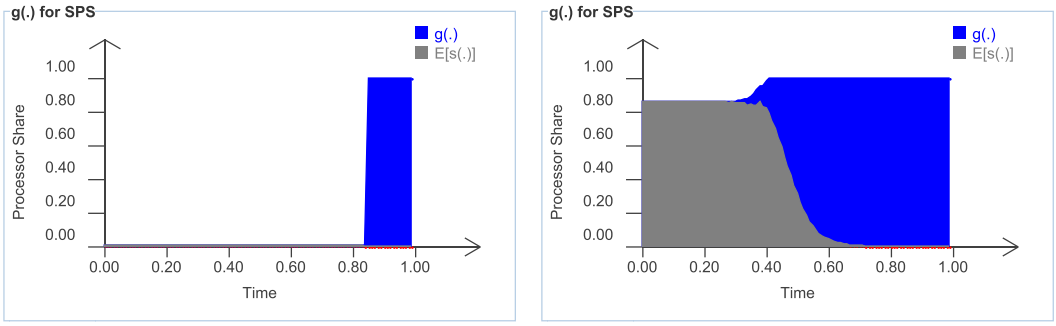


(f) (N_{srt}=50,U_{srt}=0.60,R=0.65,N_{ts}=50,U_{ts}=0.35)

Figure 5.14: Medium utilization system, TS response time and SRT overrun time $\Phi(.)$

- For high cumulative mean SRT utilization (¿0.6), the cumulative mean system utilization is (¿1). Hence there is little idle time in the system. This impacts the $g(.)$ function, and SPS $g(.)$ increasingly resembles GPS $g(.)$. This trend can be seen in the Fig 5.15.

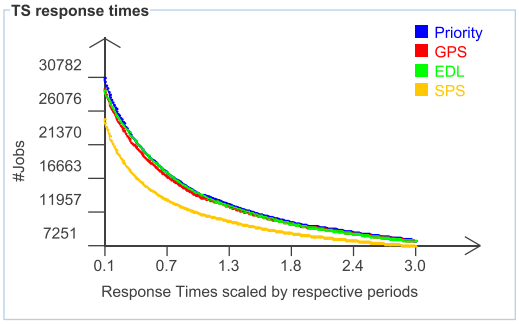
  Also, note in the TS response time $\Phi(.)$ function that a bulk of TS jobs have response time greater than their



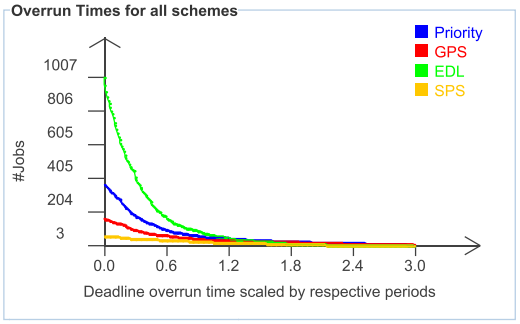
(a) (N_{srt}=50,U_{srt}=0.10,R=0.15,N_{ts}=50,U_{ts}=0.35) (b) (N_{srt}=50,U_{srt}=0.90,R=0.95,N_{ts}=50,U_{ts}=0.35)

Figure 5.15: SPS $g(.)$ trend from EDL like for low overall system utilization to GPS like for high overall system utilization

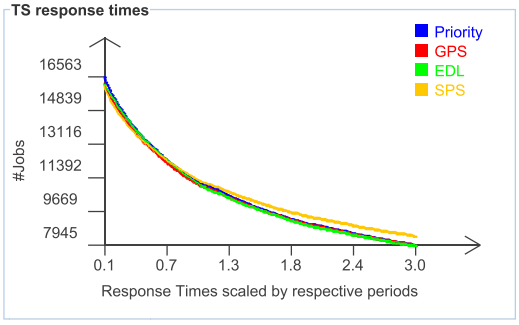
period. In fact, many TS jobs are perpetually queued because they are starved (new smaller requirement TS jobs keep coming in). Thus it is important to keep the cumulative mean system utilization to be less than 1 (Fig 5.16).



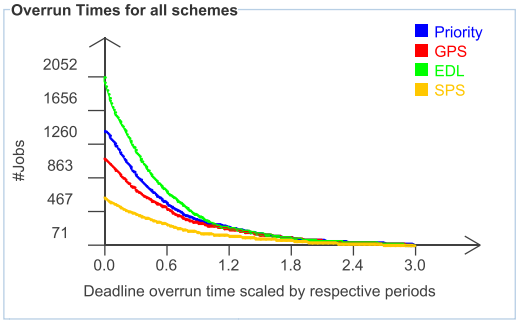
(a) (N_{srt}=50,U_{srt}=0.70,R=0.75,N_{ts}=50,U_{ts}=0.35)



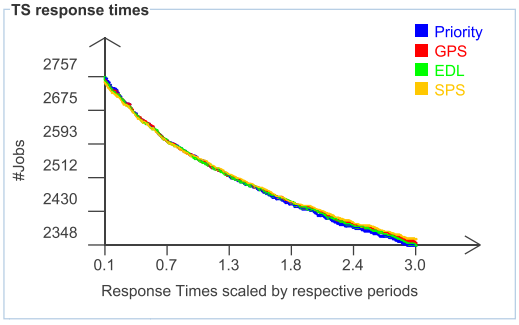
(b) (N_{srt}=50,U_{srt}=0.70,R=0.75,N_{ts}=50,U_{ts}=0.35)



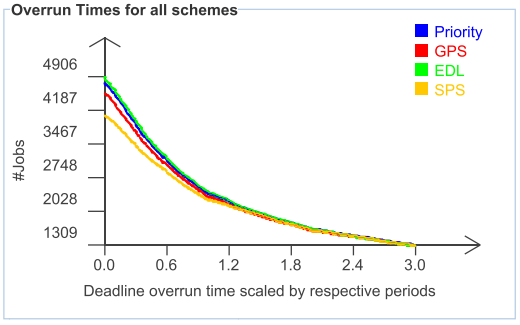
(c) (N_{srt}=50,U_{srt}=0.80,R=0.85,N_{ts}=50,U_{ts}=0.35)



(d) (N_{srt}=50,U_{srt}=0.80,R=0.85,N_{ts}=50,U_{ts}=0.35)



(e) (N_{srt}=50,U_{srt}=0.90,R=0.95,N_{ts}=50,U_{ts}=0.35)



(f) (N_{srt}=50,U_{srt}=0.90,R=0.95,N_{ts}=50,U_{ts}=0.35)

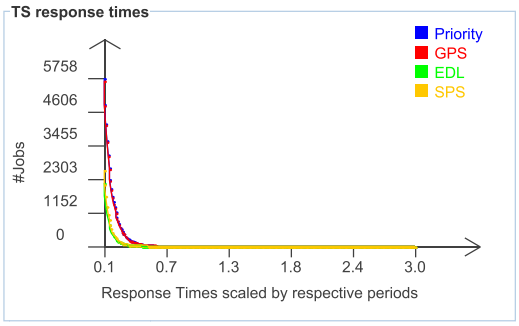
Figure 5.16: High utilization system, TS response time and SRT overrun time $\Phi(.)$. TS jobs may be starved.

5.4.3 Impact of SRT Requirement Variability

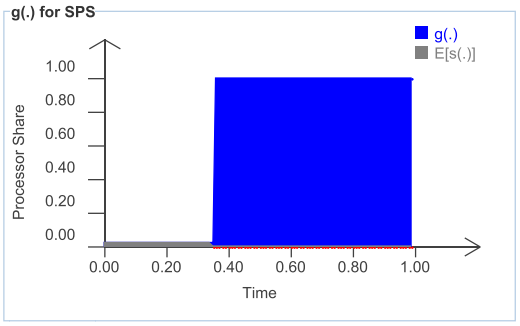
The difference between SRT reservation and mean utilization directly impacts the allocation to TS jobs. The choice of reservation may depend upon a variety of factors like perceived performance, response time distribution etc. In this set of experiments we do not figure out how to choose the value of reservation, rather given a set of RT tasks with given cumulative mean utilization, how does choice of reservation impact the response time of TS tasks. Task sets with reservation considerably greater than their mean SRT utilization are task sets with high variability in their execution time requirement.

To do this we run experiments for a system with 50 SRT tasks with with cumulative SRT reservation fixed at 0.65, and cumulative TS utilization of 0.40 and 40 TS tasks. We vary the cumulative mean utilization of SRT tasks from 0.10 to 0.60 in steps of 0.1, and observe its impact on the response time distribution of TS tasks.

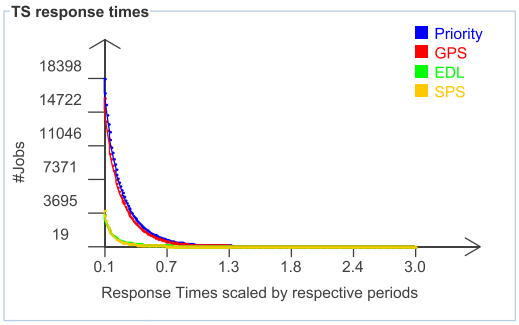
From the Fig 5.17 it can be seen that as difference between cumulative mean SRT utilization and cumulative SRT reservation utilization decreases, so does the advantage of using SPS or EDL. This is understandable, because if U_{srt} is nearly equal to R, then the cumulative RT requirement of RT jobs is pretty much constant and equal to the mean requirement. Note that if there is idle time in the system, then EDL and SPS will outperform Priority or GPS, since RT jobs would use this idle time and thereby reduce their interference with TS jobs, as can be seen from previous set of experiments detailing the impact of U_{srt} on SRT overrun and TS response times. But if there is no idle time in the system, then all the schemes provide near similar performance.



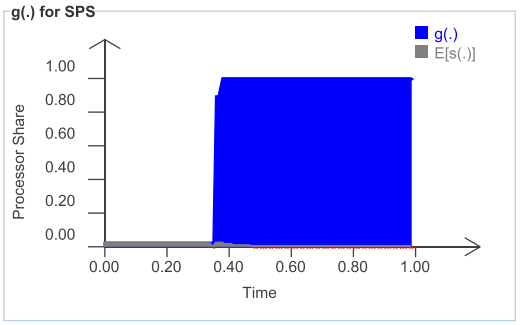
(a) (N_{srt}=50,U_{srt}=0.10,R=0.65,N_{ts}=50,U_{ts}=0.40)



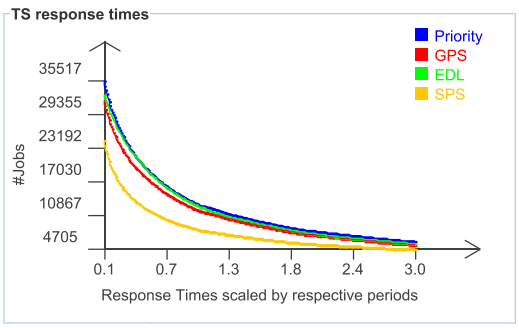
(b) (N_{srt}=50,U_{srt}=0.10,R=0.65,N_{ts}=50,U_{ts}=0.40)



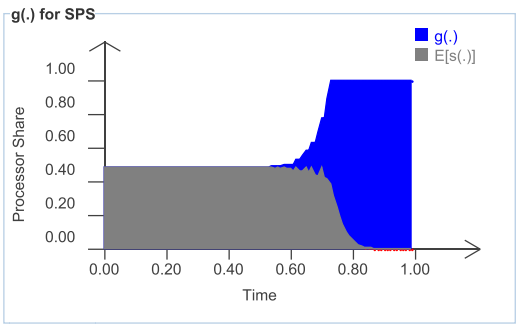
(c) (N_{srt}=50,U_{srt}=0.30,R=0.65,N_{ts}=50,U_{ts}=0.40)



(d) (N_{srt}=50,U_{srt}=0.30,R=0.65,N_{ts}=50,U_{ts}=0.40)



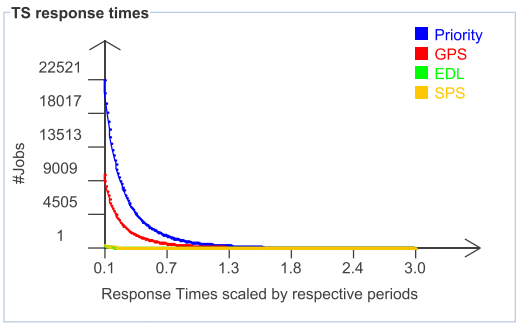
(e) (N_{srt}=50,U_{srt}=0.60,R=0.65,N_{ts}=50,U_{ts}=0.40)



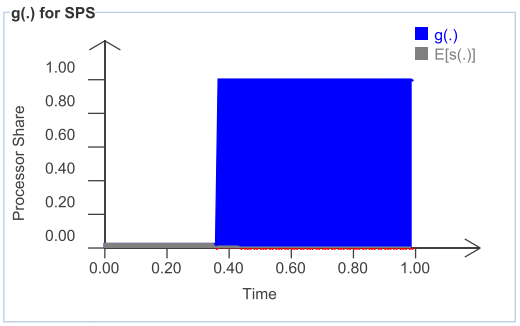
(f) (N_{srt}=50,U_{srt}=0.60,R=0.65,N_{ts}=50,U_{ts}=0.40)

Figure 5.17: Impact of difference between U_{srt} and R on TS response times

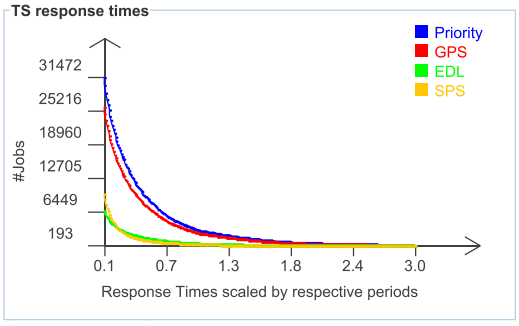
5.4.4 Impact of Mean TS Utilization



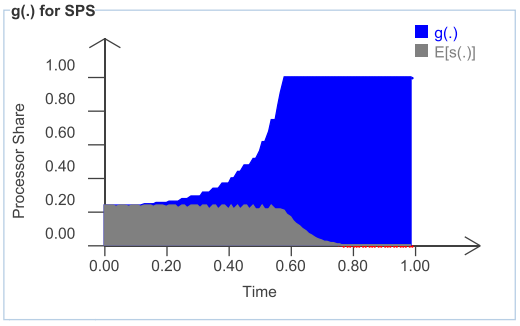
(a) (N_{srt}=50,U_{srt}=0.5,R=0.64,N_{ts}=50,U_{ts}=0.10)



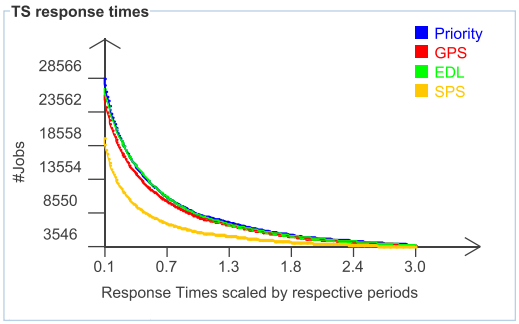
(b) (N_{srt}=50,U_{srt}=0.50,R=0.64,N_{ts}=50,U_{ts}=0.10)



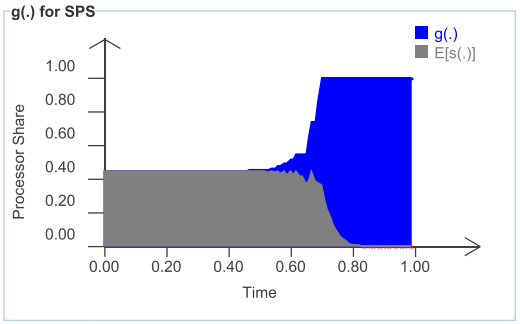
(c) (N_{srt}=50,U_{srt}=0.50,R=0.64,N_{ts}=50,U_{ts}=0.30)



(d) (N_{srt}=50,U_{srt}=0.50,R=0.64,N_{ts}=50,U_{ts}=0.30)



(e) (N_{srt}=50,U_{srt}=0.50,R=0.64,N_{ts}=50,U_{ts}=0.60)



(f) (N_{srt}=50,U_{srt}=0.50,R=0.64,N_{ts}=50,U_{ts}=0.60)

Figure 5.18: Impact of TS Workload

While in the previous set of experiments, we looked at the impact of SRT utilization on TS response times and SRT overrun times, in this section we look at the impact of TS utilization on TS response times (Fig 5.18). To do this we fix the number of SRT tasks to be $N_{srt} = 50$, their mean utilization $U_{srt} = 0.5$, their reservation $R = 0.65$, and the number of TS tasks $N_{ts} = 50$ and vary the mean TS utilization U_{ts}.

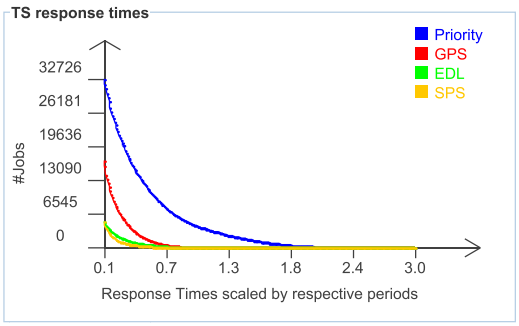
- For small cumulative TS utilization (0.1-0.3), EDL and SPS perform significantly better than GPS or

Priority in terms of TS response times, with $\Phi(0.1)$ for SPS and EDL being more than 10 times smaller than those for GPS/Priority. The reason for such huge performance benefit is twofold. First, due to small TS utilization, the idle allocation to RT jobs is high (hence RT jobs under EDL and SPS can take advantage of this fact by starting RT jobs with very low processor share (small $E[s(t)]$). Second, for 50 TS tasks with cumulative utilization of just (0.1-0.3), the individual job requirements are very small. Thus, the TS jobs usually finish when they are scheduled. Under EDL/SPS the expected value of RT processor share is very small for any time, and hence the small TS jobs get nearly instant service without any interference from the RT jobs. Thus the key factor impacting the response times of TS tasks is the measure $E[s(t)]$. As can be seen from $g(.)$ function for SPS (which resembles that of EDL), $E[s(t)]$ is an insignificant value at any instant. Hence the drastic performance benefit over GPS or Priority schemes.

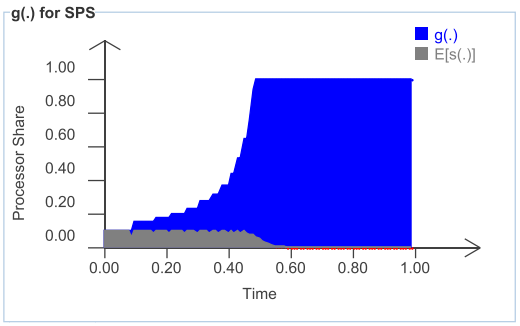
- As the cumulative TS utilization increases(0.3-0.5), there are two things happening. First, the idle allocation to RT jobs is decreasing leading to more and more RT jobs requiring their RT allocation (increasing $E[s(t)]$). Second, the average requirement of TS jobs is higher and hence the TS jobs may not finish whenever they are scheduled. Thus, there is greater probability that a TS job may encounter instances of higher value of $E[s(t)]$ leading to larger response times.

- For TS utilization of 0.6, the system is overloaded on average (mean utilization is $0.5 + 0.6 = 1.1$, leading to accumulation of high requirement TS jobs (which are starved by the small requirement TS jobs). Hence nearly all schemes perform equally, since there is little scope for improvement.

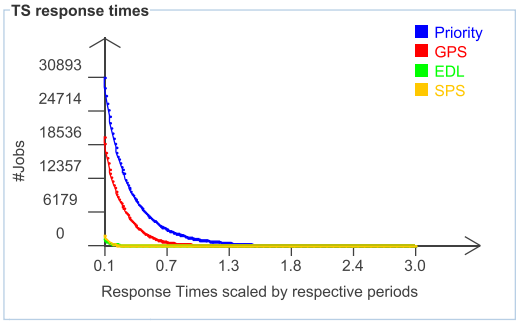
5.4.5 Impact of Size of SRT Jobs



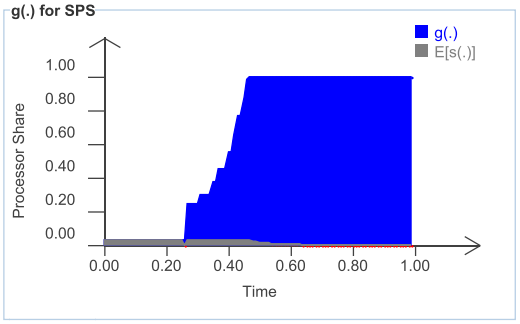
(a) (N_{srt}=5,U_{srt}=0.50,R=0.65,N_{ts}=50,U_{ts}=0.35)



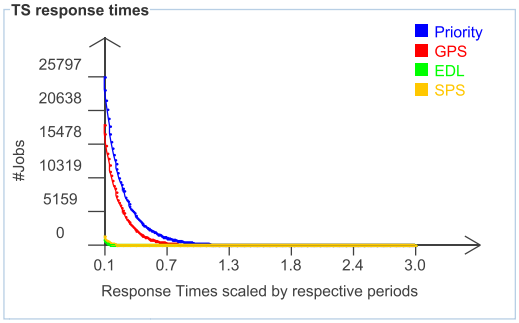
(b) (N_{srt}=5,U_{srt}=0.50,R=0.65,N_{ts}=50,U_{ts}=0.35)



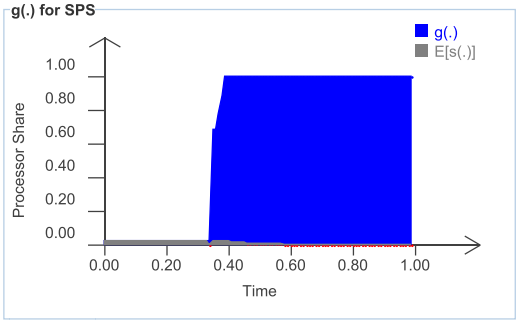
(c) (N_{srt}=30,U_{srt}=0.50,R=0.65,N_{ts}=50,U_{ts}=0.35)



(d) (N_{srt}=30,U_{srt}=0.50,R=0.65,N_{ts}=50,U_{ts}=0.35)



(e) (N_{srt}=100,U_{srt}=0.50,R=0.65,N_{ts}=50,U_{ts}=0.35)



(f) (N_{srt}=100,U_{srt}=0.50,R=0.65,N_{ts}=50,U_{ts}=0.35)

Figure 5.19: Impact of Number of SRT Tasks on TS response times

In this section we look at the impact of size of SRT jobs on the TS response times.

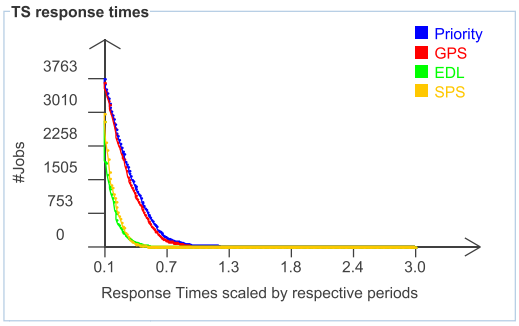
To do this we vary the number of SRT tasks N_{srt}, while keeping mean SRT utilization to be $U_{srt} = 0.5$, SRT reservation to be $R = 0.65$, the number of TS tasks to be $N_{ts} = 50$ and mean TS utilization to be $U_{ts} = 0.35$.

If number of SRT tasks is small (5-10), the duration of time during which a RT job is active is longer as compared to the case when there are many SRT tasks in the system and the SRT jobs have lesser execution time requirement. The way longer RT jobs impact the performance of TS jobs is that during the interval when RT jobs are active TS jobs get processor share of $(1 - s(t))$, and if this processor share is small then the TS jobs may get queued up and even small requirement TS jobs may have to wait. The longer the duration of such interval where RT jobs are active, the greater the queuing of TS jobs and correspondingly higher delays for them. On the other hand, if RT jobs are small, then there may be intervals of low value of $s(t)$ when a RT job finishes, giving greater processor share to the TS jobs under SPS/EDL. And this leads to improvements in TS response times.

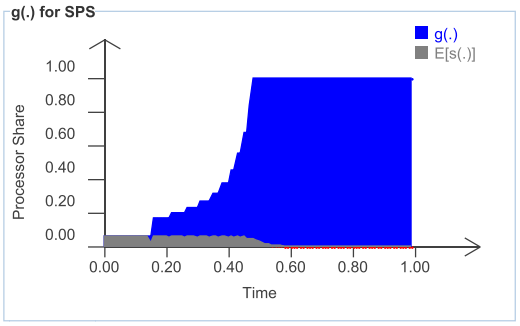
Also, as the requirements of individual SRT jobs become much smaller, then EDL performance becomes better than SPS. This is because, under EDL the RT jobs wait maximally and even when they are scheduled they last for small duration thereby minimally impacting the TS jobs, on the other hand under SPS, the RT jobs are not delayed as much as in EDL. For EDL to perform better than SPS, there should be a large number of SRT tasks and also the processor should have enough idle time so that bulk of RT jobs can finish using just the idle time. If there is no idle processor time then EDL performance suffers and EDL gives greater SRT deadline overruns than all the other three scheduling algorithms.

On the other hand, SPS adjusts and adapts according to the available idle time, changing from EDL like to GPS like as idle processor time decreases. But this adaptation may lead to EDL performing better than SPS by small margins under certain scenarios specifically when there is processor idle time and the SRT job sizes are small.

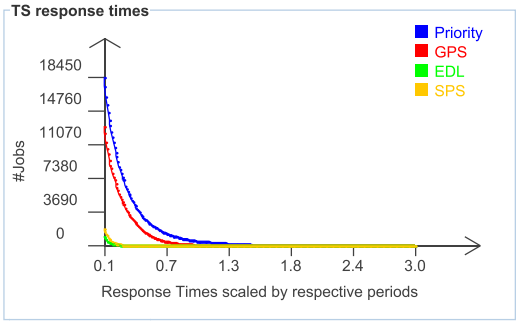
Note that SPS can be made more aggressive in delaying the RT jobs (more EDL like) by accounting for the number of tasks in the system, but this will not be a general solution, and the performance benefits are also marginal and not very significant. Also EDL is unstable in this scenario, in the sense that if a large requirement TS job arrives leading to transient system overload, then EDL may incur greater SRT deadline overruns and also increased TS response times. On the other hand SPS is much more robust and stable to transient overloads.



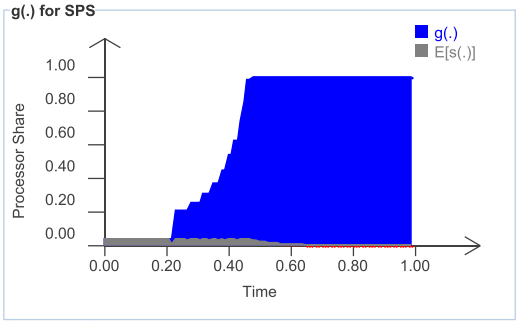
(a) (N_{srt}=50,U_{srt}=0.50,R=0.65,N_{ts}=5,U_{ts}=0.35)



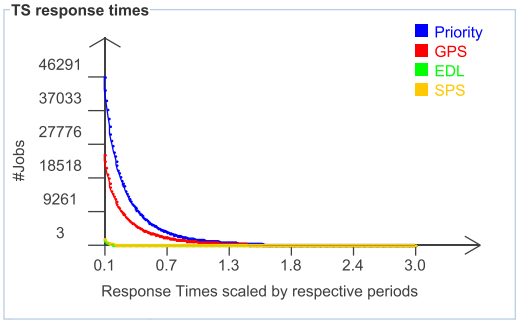
(b) (N_{srt}=50,U_{srt}=0.50,R=0.65,N_{ts}=5,U_{ts}=0.35)



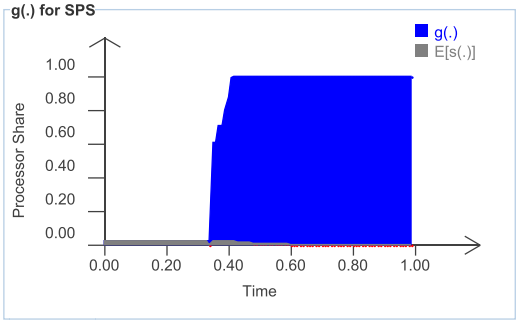
(c) (N_{srt}=50,U_{srt}=0.50,R=0.65,N_{ts}=30,U_{ts}=0.35)



(d) (N_{srt}=50,U_{srt}=0.50,R=0.65,N_{ts}=30,U_{ts}=0.35)



(e) (N_{srt}=50,U_{srt}=0.50,R=0.65,N_{ts}=100,U_{ts}=0.35)



(f) (N_{srt}=50,U_{srt}=0.50,R=0.65,N_{ts}=100,U_{ts}=0.35)

Figure 5.20: Impact of size of TS jobs on TS response times

5.4.6 Impact of Size of TS jobs

While in the previous section we looked at the impact of the number of SRT tasks on the system performance, in this section we fix rest of the parameters and just vary the number of TS tasks to describe the impact of size of TS jobs on the system performance.

If the number of TS tasks is small, then the execution requirement of individual TS job is high. Large requirement TS jobs have greater response times, and this diminishes the response time benefits seen, because

SPS is most beneficial for small requirement TS jobs. The reasons for this have been explained before, but we would briefly explain them here again. The RT scheduling algorithm impact the performance of TS jobs only in terms of currently active RT jobs. That is, until the previous deadlines of all the active RT jobs, all the RT scheduling algorithm have equal cumulative allocation to the TS tasks. So the allocation difference comes on time scale which is of the order of mean period of SRT tasks. Also, if a TS job finishes quickly then its performance is impacted by instantaneous $E[s(t)]$, which is minimized under SPS. A TS job that is active for a longer duration sees reduction in the available processor share as the active RT jobs progress and hence the difference in response time between SPS and GPS/Priority decreases. Same line of reasoning also explains EDL performance, which is just like SPS.

As the number of TS tasks is increased, the individual TS jobs become small. With smaller TS jobs, the TS jobs finish once scheduled and hence their response time is a function of $(1 - E[s(t)])$, and as can be seen $E[s(t)]$ is nearly 0 for SPS/EDL.

In the next chapter, conclusions and future work is presented.

CHAPTER 6
Conclusion

Managing resources is an important problem faced by operating systems. Processor is one of the key resources that is shared amongst multiple tasks, and properly managing it can bring significant performance benefits. The tasks may have different timeliness requirements, for example some tasks may have deadlines and others may have response time constraints. Through this work we address the problem of co-scheduling tasks with deadlines and tasks whose response times need to be minimized. The problem of providing hard guarantees like deadlines and minimizing response times have either been considered in isolation (for example, the real-time scheduling is considered independent of the non-RT tasks in the system [AB98a] [RH95] [APLW02]) or in restricted scenarios, for example EDL provides optimal response times to the non-RT tasks if the jobs of the non-RT tasks are scheduled in FIFO order.

In this work, we focused on task sets with RT tasks and response time sensitive TS tasks. The goal of the scheduling algorithm is to provide deadline guarantees to the RT tasks while reducing response times of the TS tasks. The performance of the scheduling algorithm is measured in terms of scaled response times of the SRT job overruns and TS jobs. The goal is to reduce the number of TS jobs (or SRT job overruns) with scaled response time (or SRT deadline overrun time) greater than any given fraction x. This value is represented as $\Phi(x)$. For example, if the number of SRT overrun jobs with scaled deadline overrun time greater than say 0.4 leads to performance degradation then the scheduling algorithm with least SRT overrun $\Phi(0.4)$ would be the best. For TS jobs, a scheduling algorithm that keeps the response time of TS jobs smaller is better. In terms of the TS response time $\Phi(.)$ function, the scheduling algorithm with smaller $\Phi(x)$ for any value of x would be better.

6.1 Co-scheduling Algorithm Performance

We present a summary of the factors that most heavily impact the performance of a co-scheduling algorithm.

- Minimizing expected RT processor share - This is one of the key factors impacting the performance of co-scheduling algorithms. If there is enough idle time such that all the RT jobs finish no later than their

Measure	Scheduling Algorithm			
	Priority	GPS	EDL	SPS
(N_{srt}=50,U_{srt}=0.30,R=0.65,N_{ts}=50,U_{ts}=0.40)				
SRT job overruns	1 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
Mean scaled overrun time	0.0000	0.0000	0.0000	0.0000
Mean scaled TS response time	0.1113	0.1002	0.0357	0.0377
TS response time $\Phi(0.10)$	18398 (31.74%)	16280 (28.09%)	3617 (6.24%)	4036 (6.96%)
TS response time $\Phi(0.20)$	9101 (15.70%)	7954 (13.72%)	1155 (1.99%)	1077 (1.86%)

Table 6.1: Performance numbers for a very lightly loaded processor.

deadlines, then the RT jobs do not actually require any preference instead the RT jobs can be scheduled at the lowest priority, and in this case the RT jobs do not interfere at all with the TS jobs.

Also, the RT jobs may have variable execution time, and so many RT jobs may require less than their reservation execution time. The RT share can be calculated such that the expected processor share of RT jobs is minimized.

So the RT execution time variability is the combined effect of the execution time variability and the idle allocation that the RT jobs get. The RT execution time for a job is the actual execution time requirement minus the idle allocation to this job. Both these factors together impact the co-scheduling performance.

Task set (N_{srt}=50,U_{srt}=0.30,R=0.65,N_{ts}=50,U_{ts}=0.40) shown in Table 6.1 represents a scenario where the mean SRT utilization is equal to the mean idle utilization of the processor. To see this note that the mean SRT utilization plus the mean TS utilization is $0.30 + 0.40 = 0.70$, so on average 0.30 fraction of time, the processor is idle. EDL/SPS perform significantly better than GPS/Priority in this scenario, which follows from the fact that the expected RT processor share under EDL/SPS is nearly 0. That is, with high probability the RT jobs finish using just the idle processor time without interfering with the jobs of TS tasks.

Task set (N_{srt}=50,U_{srt}=0.40,R=0.45,N_{ts}=50,U_{ts}=0.35) and (N_{srt}=50,U_{srt}=0.50,R=0.64,N_{ts}=50,U_{ts}=0.20) shown in Table 6.2 have substantial idle processor time, but lesser than the previous task set. SPS/EDL take advantage of the idle time by delaying the RT jobs, hence increasing the chances of RT jobs getting idle allocation. GPS/Priority do not take advantage of this fact and its impact can be seen on the response times of TS jobs.

Task set (N_{srt}=50,U_{srt}=0.50,R=0.65,N_{ts}=50,U_{ts}=0.35) shown in Table 6.3 has little idle time, and as before SPS/EDL take advantage of this idle time whereas Priority/GPS do not. The difference in the TS response time distribution is not as significant as in the previous cases, but still the mean scaled TS

Measure	Scheduling Algorithm			
	Priority	GPS	EDL	SPS
(N_{srt}=50,U_{srt}=0.40,R=0.45,N_{ts}=50,U_{ts}=0.35)				
SRT job overruns	10 (0.02%)	0 (0.00%)	6 (0.01%)	0 (0.00%)
Mean scaled overrun time	0.0001	0.0000	0.0000	0.0000
Mean scaled TS response time	0.1714	0.1220	0.0337	0.0382
TS response time $\Phi(0.10)$	24594 (42.44%)	17008 (29.35%)	2331 (4.02%)	3331 (5.75%)
TS response time $\Phi(0.20)$	14415 (24.87%)	9386 (16.20%)	545 (0.94%)	695 (1.20%)
(N_{srt}=50,U_{srt}=0.50,R=0.64,N_{ts}=50,U_{ts}=0.20)				
SRT job overruns	20 (0.03%)	4 (0.01%)	17 (0.03%)	0 (0.00%)
Mean scaled overrun time	0.0002	0.0000	0.0001	0.0000
Mean scaled TS response time	0.1998	0.1461	0.0279	0.0280
TS response time $\Phi(0.10)$	27234 (46.99%)	19418 (33.50%)	1265 (2.18%)	1500 (2.59%)
TS response time $\Phi(0.20)$	16417 (28.33%)	11393 (19.66%)	589 (1.02%)	563 (0.97%)

Table 6.2: Performance numbers for a lightly loaded processor.

Measure	Scheduling Algorithm			
	Priority	GPS	EDL	SPS
(N_{srt}=50,U_{srt}=0.50,R=0.65,N_{ts}=50,U_{ts}=0.35)				
SRT job overruns	20 (0.03%)	4 (0.01%)	67 (0.12%)	0 (0.00%)
Mean scaled overrun time	0.0002	0.0000	0.0002	0.0000
Mean scaled TS response time	0.4180	0.3775	0.2243	0.2143
TS response time $\Phi(0.10)$	33417 (57.68%)	28904 (49.88%)	15911 (27.46%)	17272 (29.80%)
TS response time $\Phi(0.20)$	23635 (40.79%)	20202 (34.87%)	10951 (18.90%)	9738 (16.80%)

Table 6.3: Performance numbers for a moderately loaded processor.

response time for Priority and GPS is 0.4180 and 0.3775 respectively and for EDL and SPS, the mean scaled TS response time is 0.2243 and 0.2143 respectively.

For task set (N_{srt}=50,U_{srt}=0.80,R=0.85,N_{ts}=0,U_{ts}=0.0) shown in Table 6.4, the cumulative SRT utilization is high. The number of SRT job overruns is high under EDL in this scenario as compared to SPS, though EDL/SPS perform better as compared to GPS/Priority in terms of mean scaled overrun times. For task set (N_{srt}=50,U_{srt}=0.80,R=0.85,N_{ts}=50,U_{ts}=0.35), the system is overloaded. EDL performance suffers in this case as compared to SPS in terms of SRT overruns. The expected processor RT share is high under EDL for overloaded systems because delaying RT jobs does not bring much performance benefit if the RT execution time variability is small.

- The performance benefits are more pronounced for smaller TS jobs rather than bigger ones. The difference between the co-scheduling algorithms is only at RT jobs' periods time scales. So, if a TS job has large execution time then the TS job may have a response time extending over a few periods of RT jobs. Therefore the response time benefit is only seen due to the allocation difference amongst the various algorithms for

Measure	Scheduling Algorithm			
	Priority	GPS	EDL	SPS
(N_{srt}=50,U_{srt}=0.80,R=0.85,N_{ts}=0,U_{ts}=0.0)				
SRT job overruns	1422 (2.45%)	1032 (1.78%)	651 (1.12%)	332 (0.57%)
Mean scaled overrun time	0.0228	0.0198	0.0119	0.0103
Overrun $\Phi(0.20)$	1087 (1.88%)	807 (1.39%)	447 (0.77%)	275 (0.47%)
Overrun $\Phi(0.80)$	426 (0.74%)	366 (0.63%)	201 (0.35%)	168 (0.29%)
(N_{srt}=50,U_{srt}=0.80,R=0.85,N_{ts}=50,U_{ts}=0.35)				
SRT job overruns	1422 (2.45%)	1092 (1.88%)	2052 (3.54%)	638 (1.10%)
Mean scaled overrun time	0.0228	0.0207	0.0262	0.0145
Overrun $\Phi(0.20)$	1087 (1.88%)	851 (1.47%)	1430 (2.47%)	495 (0.85%)
Overrun $\Phi(0.80)$	426 (0.74%)	383 (0.66%)	515 (0.89%)	253 (0.44%)
Mean scaled TS response time	34.6884	33.1181	31.6116	37.9978
TS response time $\Phi(0.10)$	16563 (88.93%)	16249 (86.93%)	16226 (86.32%)	16126 (83.23%)
TS response time $\Phi(0.20)$	15448 (82.94%)	15173 (81.18%)	15373 (81.78%)	15026 (77.55%)

Table 6.4: Performance numbers for a loaded and overloaded case.

the RT job during whose execution the TS job finishes execution. The following Table 6.5 shows the variation in performance benefits with the size of TS jobs. In this set of experiments, the mean TS utilization is kept constant at 0.35, and the number of TS tasks is increased. Greater the number of TS tasks, smaller the TS job sizes. As can be seen, the mean scaled TS response time decreases and the difference between the performance of SPS/EDL and Priority/GPS increases with decreasing TS job sizes.

The results in Table 6.5 are explained using our theoretical framework as follows. If the TS jobs' executions span over several jobs of the same RT tasks, then the measure $A(.)$ better determines the response time difference under the four scheduling algorithms. This is because, when the TS jobs finishes, most of the active RT jobs saw this TS job as already active when they arrived and hence $A(.)$ would represent the allocation to the TS job as a function of time. EDL maximizes $A(.)$ so the best response times are seen under EDL, followed by SPS, GPS and Priority. As the average TS jobs sizes decrease with increasing number of TS tasks making up the same mean cumulative utilization, the response time benefits with EDL/SPS increase. This is because, the smaller TS jobs have greater probability of finishing before the active RT jobs. So, the response time is determined more by $\int(1 - s(t))\mathrm{d}t$ rather than just $A(.)$ because the allocation received during the time when the RT job is active differs between the four algorithms. SPS provides the most unbiased allocation, that is on average the TS job get similar allocation irrespective of their arrival times. The performance of other algorithms depends upon the idle processor time. If there the processor is idle for a large fraction of time then EDL performs better, and GPS performs better if the processor idle time is small. SPS on the other hand adapts itself to the available idle time, changing from

Measure	Scheduling Algorithm			
	Priority	GPS	EDL	SPS
(N_{srt}=50,U_{srt}=0.50,R=0.65,N_{ts}=5,U_{ts}=0.35)				
Mean scaled TS response time	0.3008	0.2783	0.1302	0.1438
TS response time $\Phi(0.10)$	3763 (83.11%)	3694 (81.58%)	2823 (62.33%)	2986 (65.93%)
TS response time $\Phi(0.20)$	2769 (61.15%)	2687 (59.34%)	815 (18.00%)	1122 (24.77%)
TS response time $\Phi(0.40)$	1311 (28.95%)	1122 (24.78%)	105 (2.32%)	76 (1.68%)
TS response time $\Phi(0.80)$	121 (2.67%)	81 (1.79%)	4 (0.09%)	3 (0.07%)
(N_{srt}=50,U_{srt}=0.50,R=0.65,N_{ts}=10,U_{ts}=0.35)				
Mean scaled TS response time	0.2073	0.1808	0.0550	0.0631
TS response time $\Phi(0.10)$	6523 (63.07%)	5947 (57.50%)	1289 (12.46%)	1702 (16.45%)
TS response time $\Phi(0.20)$	4011 (38.78%)	3356 (32.45%)	210 (2.03%)	327 (3.16%)
TS response time $\Phi(0.40)$	1575 (15.23%)	1239 (11.98%)	25 (0.24%)	29 (0.28%)
TS response time $\Phi(0.80)$	100 (0.97%)	64 (0.62%)	6 (0.06%)	8 (0.08%)
(N_{srt}=50,U_{srt}=0.50,R=0.65,N_{ts}=100,U_{ts}=0.35)				
Mean scaled TS response time	0.1464	0.0907	0.0151	0.0176
TS response time $\Phi(0.10)$	46291 (39.41%)	25209 (21.46%)	1577 (1.34%)	1959 (1.67%)
TS response time $\Phi(0.20)$	24575 (20.92%)	12946 (11.02%)	257 (0.22%)	328 (0.28%)
TS response time $\Phi(0.40)$	9967 (8.48%)	5413 (4.61%)	94 (0.08%)	109 (0.09%)
TS response time $\Phi(0.80)$	2669 (2.27%)	1655 (1.41%)	50 (0.04%)	67 (0.06%)

Table 6.5: Performance numbers for varying TS jobs sizes. The number of TS tasks is varied while keeping the cumulative mean TS utilization constant. As the number of TS tasks increases, the individual tasks have lesser mean execution time requirement.

being EDL like when there is large idle processor time to GPS like when there is little or no idle time. In this particular scenario, there is some idle time, and EDL is able to take advantage of this fact giving the best performance with SPS closely following it. Priority/GPS are not suited for this scenario and hence their performance suffers.

6.2 Contributions

The key contributions are enumerated below –

- Current GPOS scheduling algorithms use multi-level feedback queues along with task priorities to schedule the workloads. Through this work we have shown that as the GPOS move from best-effort to assured service systems with guarantees, the scheduler would be required to provide more than just best-effort service. We have exposed the inadequacies of current scheduling algorithms in dealing with workloads with varying timeliness requirements. In particular, giving priority to RT jobs over non-RT jobs, which is the most widely available scheduling option, is grossly sub-optimal. In fact, it is the worst possible way to co-schedule RT and non-RT tasks and tremendous gains are possible by migrating to better scheduling algorithms.

The prime reason why current GPOS still rely on giving Priorities to RT jobs is because the systems are usually underutilized and hence the problems with Priority scheduling are not exposed. As the attention is shifting towards power efficiency and heat reduction, it is desirable to have higher system utilization. This implies that there will be shift from underutilized systems to nearly fully utilized systems. And it is in this scenario that task scheduling will become the single most important factor determining system performance. And this is the scenario we address through this work.

- We propose Stochastic Processor Sharing (SPS) as a practical, efficient and smart scheduling algorithm, that adapts itself based on the actual RT requirements distribution (obtained through online profiling), such that the RT guarantees are maintained while the response times of the non-RT jobs are reduced. In fact, under SPS the maximum expected utilization of the RT tasks at any time instant is minimized. This gives an unbiased schedule and non-RT jobs get nearly same expected service rate irrespective of other factors like their arrival times. In a way this is an optimal schedule, because any slight variation to this schedule would give intervals during which the expected processor share of the RT jobs is greater than that ever reached under SPS, and other intervals during which it is less. It is during the times of higher expected processor share of RT jobs that the TS jobs would suffer as compared to SPS. It finishes if it is scheduled. Under SPS, there is nearly equal probability of this job getting scheduled at any instant, while under any scheduling algorithm other than SPS, there are intervals in which the probability is higher and others in which it is lower, and this is not conducive for good response times to the non-RT jobs.

- We evaluated SPS performance both analytically and empirically, and showed that it performs significantly better than the current algorithms under a wide range of common scenarios under reasonable assumptions. The extensive empirical analysis not only shows the performance gains achievable but it also provides good understanding on the working of SPS algorithm, and how RT scheduling algorithms should behave when they are co-scheduled with response time sensitive computation.

- The RT allocation guarantees are upheld irrespective of the probability distribution (RT allocation guarantee is provided on the basis of R and does not depend upon χ). Thus, any reasonable approximation to the RT execution time requirement probability distribution provides smaller response times to the TS jobs. In fact, in the beginning (at time 0) it is assumed that the execution time requirement is of RT jobs is constant and equal to R. As time progresses, the execution time probability distribution is created empirically.

- SPS is an intelligent algorithm, and it evolves and adapts as it gathers more information about the work-

load. It is independent of the application, and requires minimal information about the task set, basically only the period and reservation information about each RT task. Current scheduling algorithms on the other hand are fixed based on the task set parameters or require design of appropriate application dependent feedback-control loop or other heuristics to handle workloads with variable execution time requirements. All these advantages make SPS a good choice for practical systems.

6.3 Limitations and Future Work

- The primary limitation of SPS is that it is based on GPS processor sharing model. The accuracy of emulation of GPS on a sequential processor is determined by the quantum size. Smaller quantum size gives better allocation accuracy but increases the time spent in context switches ([Reg02]). Current GPOSes have time quantum in the range of 10 ms, while the average period of RT tasks (interactive, media playback , computer games) is in the range of (20ms - 200ms). Ideally, we would like the quantum size to be at least less than the RT task period. Quantum size of 1ms would give good performance, but it may lead to significant context switch overhead. This problems will not be encountered by systems with a lot of tasks (100 or more). This is because in such a scenario most jobs would be very small requirement and would finish once scheduled and in less than 1ms (given 100 tasks with 50ms period and mean utilization of 0.8, the mean utilization is 0.008 and the mean job requirement is 0.4ms).

- While SPS requires only the period P_i and reservation R_i information for the SRT task, the choice of R_i is left open. R_i can be chosen using a feedback-control loop as described in [LSST02] [APLW02]. But a better way would be to use a feedback-control loop based on the $\Phi(.)$ function, that is, the entire response time distribution rather than focusing on a threshold value. But this would require information about acceptable response time distributions for applications, which may vary considerably from application to application. This kind of problem has been addressed before by Jensen et. al. [JLT85] in 1985, and later. The primary difference from the approach followed by these with ours is that, Jensen et. al. looked at a very general form of the problem where the utility functions may be arbitrary and the scheduling problem is then to maximize the utility. In our case we focus on the problem of guaranteeing allocation to certain job while providing timely allocation to the non-RT jobs in the system. Note that the constraints SPS handles are more focused and simpler than maximizing utility. In fact, SPS focuses exclusively on how to schedule the variable requirement RT jobs, and the scheduling of the non-RT jobs can be done in arbitrary fashion

(we used Least Attained Service First (LAS) scheduling which minimizes expected response times) based on utility functions. The problem of utility arises in calculating the value of reservation R_i for a RT task, which would be the primary focus of our future work. And it should be noted that given any value of R_i, SPS provides better response times to the non-RT jobs as compared to any other algorithm.

- An optimal co-scheduling algorithm would minimize the expected processor share of RT tasks at any time. Though SPS minimizes the maximum expected processor share of RT tasks, during some intervals the processor share of RT tasks may be greater than what an optimal algorithm can achieve. Currently, this gap may be wide but it will shrink once slack reclamation is incorporated in SPS. For example, consider a two RT task system with one very large requirement TS task. At any time, atmost two RT jobs are active, one belonging to each of the two RT tasks. During the intervals when only one RT job is active, the $g(.)$ function which determines the processor share of the RT tasks overallocates because the $g(.)$ function was defined such that the allocation in a unit interval would be equal to the cumulative reservation of all the RT tasks present in the system. But, if only one RT task is active, clearly using this $g(.)$ function is overkill. To find a schedule that minimizes the expected processor share of RT tasks would require the exact probability distribution of execution time requirement known at any instant. Now, this distribution would depend upon the active RT tasks. Maintaining probability distribution for all possible combinations of active RT tasks would be highly inefficient because for a task set with n tasks, it would require 2^n different execution time distributions. Another avenue for improvement is when the RT tasks get idle allocation. If a RT job receives greater allocation than required by the $g(.)$ function by time t, then it can appropriately reduce its processor share for its remaining time while still meeting the allocation guarantees. A RT job may receive greater allocation than that given by its $g(.)$ function because it may get allocation when the processor is idle.

  Though incorporating these optimizations in SPS is not trivial, the complexity of the optimal solution does not rule out the existence of efficient approximations that can match the performance of the optimal solution without the associated computation complexity. We are currently working on a slack reclamation scheme for SPS that would take into account the active RT tasks as well as idle allocation to compute the processor share for RT tasks.

  One more avenue for optimization is treating very large requirement TS jobs separately. The performance impact of co-scheduling algorithm would be minimal on TS jobs whose execution spans several jobs of

the active RT tasks and these TS jobs can be scheduled at the absolute lowest priority, without incurring performance loss. But handling large TS jobs like this would increase the probability of the RT jobs getting idle allocation, thereby reducing the real-time execution time requirement for the RT jobs.

- We considered LAS (Least Attained Service time first) discipline to schedule TS jobs. This algorithm gives optimal mean response time for the TS tasks if the execution time requirements of the TS jobs are not known in advance. In practical systems, other scheduling policies for TS jobs may be more appropriate. For example, some TS tasks may be given priority over others, or more elaborate measures like TUF [JLT85] may be used to schedule the TS jobs. Even for these scheduling algorithms, SPS would provide performance benefits because it performs well on both $A(t)$ and $E[s(t)]$ measure, but quantifying them would be essential for wide scale acceptance of SPS.

- This work addresses task set with independent tasks and issues like task inter-dependency and parallelism are not addressed. Addressing these issues would be important for wide applicability of SPS. Simple share transfer technique may work well, but this is just a conjecture and it needs to be verified. This would be one avenue which requires further research in future.

6.4 Workload Consolidation and Power Savings

Power is becoming a significant resource in server installations. Big companies like Google are seeing a massive increase in their energy bills [BDH03] [Bar05]. For any server that needs to support a large number of clients, optimizing the system performance for power is becoming essential.

At the other hand of computing spectrum, for handheld devices operating on limited battery life, minimal power consumption is essential to provide longer service times.

Datacenters are moving towards workload consolidation. This consolidation would require the operating system to support allocation guarantees like processing time guarantees to the tasks. Also, frequently the tasks would require variable execution time and this variability can be effectively accommodated using a scheduling algorithm like SPS. These workloads do not fall into hard real-time category, rather the allocation guarantee requirements are soft. If the response times fall within some predefined reasonable range then the performance can be considered acceptable. Our modeling allows to tackle these kinds of problems where the performance is dependent upon the response time distribution rather than fixed deadlines.

There is a whole research area on varying processor speed to save energy and reduce heat generation, in particular the work done by Lorch et. al. [LS04] and Gruian [Gru01]. The theoretically optimal way to schedule variable requirement RT tasks on a DVS capable processor is to continuously vary its speed with job progress. Such a schedule minimizes the expected energy consumed by the processor. This approach applies to task system with single RT task [LS04] or to each task independently for multiple task systems [YN03]. Second, in case of deadline overrun, the job missing its deadline is serviced at full processor speed [YN03]. Instead of varying processor speed, we propose varying processor share of the tasks to maintain their allocation guarantees. Our approach better suits the problem requirements and warrants further study in this direction.

6.5 Conclusions

From application to research, to entertainment and gaming, computers have established themselves as an integral part of the day to day lives of many. With the rich set of functionality that computers are providing today, their performance is also becoming important. For example, for web servers the response times are important, for hand-held devices power is important, for embedded devices meeting deadline is essential etc.

While there is a large body of works addressing the issues of RT scheduling and those addressing response time minimization problems, a fusion of the these two works is required to provide an effective solution to the scheduling problems faced by current operating systems. RT scheduling focuses on the worst case values, providing deadline guarantees assuming the worst case scenario. Response time minimization approaches on the other hand focus on the mean values and neglect the worst case values which may be too rare to be taken into account.

Domains which are purely RT and those which are purely non-RT can be found, but there is a large area in between, where the task sets are composed of a mix of RT (hard and soft) and non-RT (and probably response time sensitive) tasks. The co-scheduling of RT and non-RT tasks thus becomes an important problem.

The main difficulty faced by practical systems is the requirement variability in RT jobs and non-RT jobs. Allocating for worst case may be essential in some scenarios but it may be overkill in others. Ideal scenario would be when the processor is nearly fully utilized on average while delivering acceptable performance (meeting deadlines and keeping TS response times small). In this work we propose a novel scheduling algorithm that achieves precisely this goal. SPS automatically reaches a balance between GPS (constant processor share to RT jobs) and EDL (delaying RT jobs), to achieve better performance than both GPS and EDL. The fact that SPS is

intelligent enough to reach this point, makes it stand apart from all the current scheduling algorithms.

One of the prime applications of SPS would be in the area of energy savings for task sets composed of RT and non-RT tasks. As we have shown theoretically and empirically, SPS provides guaranteed allocation to RT tasks while significantly improving response times of non-RT tasks as compared to current algorithms. For energy efficiency, the processor needs to be run at minimum speed to satisfy the performance requirements of the given task set. Given processor running at certain speed and servicing a given task set, by switching to SPS scheduling algorithm, the overall system performance improves significantly while keeping the power consumption constant. Most current GPOSes support just Priority scheduling, and as we have shown through experiments, priority scheduling is very inefficient performing significantly badly as compared to SPS, and substantial benefits can be reaped by switching to SPS algorithm.

BIBLIOGRAPHY

[AAA06] Eitan Altman, Konstantin Avrachenkov, and Urtzi Ayesta. A survey on discriminatory processor sharing. *Queueing Syst.*, 53(1-2):53–63, 2006.

[AABN04] Konstantin Avrachenkov, Urtzi Ayesta, Patrick Brown, and Eeva Nyberg. Differentiation between short and long tcp flows: Predictability of the response time. In *INFOCOM*, 2004.

[AANO04] Samuli Aalto, Urtzi Ayesta, and Eeva Nyberg-Oksanen. Two-level processor-sharing scheduling disciplines: mean delay analysis. In *SIGMETRICS*, pages 97–105, 2004.

[AB98a] Luca Abeni and Giorgio C. Buttazzo. Integrating multimedia applications in hard real-time systems. In *IEEE Real-Time Systems Symposium*, pages 4–13, 1998.

[AB98b] Alia Atlas and Azer Bestavros. Statistical rate monotonic scheduling. In *IEEE Real-Time Systems Symposium*, pages 123–, 1998.

[AMMMA01] Hakan Aydin, Rami G. Melhem, Daniel Mossé, and Pedro Mejía-Alvarez. Optimal reward-based scheduling for periodic real-time tasks. *IEEE Trans. Computers*, 50(2):111–130, 2001.

[AMMMA04] Hakan Aydin, Rami G. Melhem, Daniel Mossé, and Pedro Mejía-Alvarez. Power-aware scheduling for periodic real-time tasks. *IEEE Trans. Computers*, 53(5):584–600, 2004.

[APLW02] Luca Abeni, Luigi Palopoli, Giuseppe Lipari, and Jonathan Walpole. Analysis of a reservation-based feedback scheduler. In *IEEE Real-Time Systems Symposium*, pages 71–80, 2002.

[Bar05] Luiz André Barroso. The price of performance. *ACM Queue*, 3(7):48–53, 2005.

[BBB04] Guillem Bernat, Ian Broster, and Alan Burns. Rewriting history to exploit gain time. In *RTSS*, pages 328–335, 2004.

[BCM98] Michael A. Bender, Soumen Chakrabarti, and S. Muthukrishnan. Flow and stretch metrics for scheduling continuous job streams. In *SODA '98: Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, pages 270–279, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics.

[BDH03] Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003.

[BMP98] Andy C. Bavier, Allen Brady Montz, and Larry L. Peterson. Predicting mpeg execution times. In *SIGMETRICS*, pages 131–140, 1998.

[BS99] Giorgio C. Buttazzo and Fabrizio Sensini. Optimal deadline assignment for scheduling soft aperiodic tasks in hard real-time environments. *IEEE Trans. Computers*, 48(10):1035–1052, 1999.

[BZ96] Jon C. R. Bennett and Hui Zhang. Wf^2q: Worst-case fair weighted fair queueing. In *INFOCOM*, pages 120–128, 1996.

[CC89] Houssine Chetto and Maryline Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Trans. Software Eng.*, 15(10):1261–1269, 1989.

[Che98] Ludmila Cherkasova. Scheduling strategy to improve response time for web applications. In *HPCN Europe 1998: Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking*, pages 305–314, London, UK, 1998. Springer-Verlag.

[CKR96] Ludmila Cherkasova, Vadim E. Kotov, and Tomas Rokicki. The impact of message scheduling on a packet switching interconnect fabric. In *HICSS (1)*, 1996.

[CKZ01] Chandra Chekuri, Sanjeev Khanna, and An Zhu. Algorithms for minimizing weighted flow time. In *STOC '01: Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 84–93, New York, NY, USA, 2001. ACM Press.

[DGK+02] José Luis Díaz, Daniel F. García, Kanghee Kim, Chang-Gun Lee, Lucia Lo Bello, José María López, Sang Lyul Min, and Orazio Mirabella. Stochastic analysis of periodic real-time systems. In *IEEE Real-Time Systems Symposium*, pages 289–, 2002.

[DKS89] Alan J. Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM*, pages 1–12, 1989.

[DW95] Robert Davis and Andy J. Wellings. Dual priority scheduling. In *IEEE Real-Time Systems Symposium*, pages 100–109, 1995.

[FH03] Eric J. Friedman and Shane G. Henderson. Fairness and efficiency in web server protocols. In *SIGMETRICS '03: Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 229–237, New York, NY, USA, 2003. ACM Press.

[GCW95] Kinshuk Govil, Edwin Chan, and Hal Wasserman. Comparing algorithm for dynamic speed-setting of a low-power cpu. In *MobiCom '95: Proceedings of the 1st annual international conference on Mobile computing and networking*, pages 13–25. ACM Press, 1995.

[GLIN00] Dirk Grunwald, Philip Levis, Charles B. Morrey III, and Michael Neufeld. Policies for dynamic clock scheduling. *OSDI 2000*, 2000.

[Gru01] Flavius Gruian. Hard real-time scheduling for low-energy using stochastic data and dvs processors. In *ISLPED '01: Proceedings of the 2001 international symposium on Low power electronics and design*, pages 46–51, New York, NY, USA, 2001. ACM Press.

[GVC97] Pawan Goyal, Harrick M. Vin, and Haichen Cheng. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Trans. Netw.*, 5(5):690–704, 1997.

[Hor74] W A Horn. Some simple scheduling algorithms. *Naval Research Log. Qurt.*, 21, 1974.

[JLT85] E. Douglas Jensen, C. Douglas Locke, and Hideyuki Tokuda. A time-driven scheduling model for real-time operating systems. In *IEEE Real-Time Systems Symposium*, pages 112–122, 1985.

[LB00] Giuseppe Lipari and Sanjoy Baruah. Greedy reclamation of unused bandwidth in constant-bandwidth servers. *ecrts*, 00:193, 2000.

[Leh97] John P. Lehoczky. Real-time queueing network theory. In *IEEE Real-Time Systems Symposium*, pages 58–67, 1997.

[LL02] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. pages 179–194, 2002.

[LLS+91] Jane W.-S. Liu, Kwei-Jay Lin, Wei Kuan Shih, Albert Chuang shi Yu, Jen-Yao Chung, and Wei Zhao. Algorithms for scheduling imprecise computations. *IEEE Computer*, 24(5):58–68, 1991.

[LS04] Jacob R. Lorch and Alan Jay Smith. Pace: A new approach to dynamic voltage scaling. *IEEE Trans. Computers*, 53(7):856–869, 2004.

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

[LSST02] Chenyang Lu, John A. Stankovic, Sang Hyuk Son, and Gang Tao. Feedback control real-time scheduling: Framework, modeling, and algorithms. *Real-Time Systems*, 23(1-2):85–126, 2002.

[MT03] Malena Mesarina and Yoshio Turner. Reduced energy decoding of mpeg streams. *Multimedia Syst.*, 9(2):202–213, 2003.

[PG93] Abhay K. Parekh and Robert G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The multiple node case. In *INFOCOM*, pages 521–530, 1993.

[PS01] Padmanabhan Pillai and Kang G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 89–102. ACM Press, 2001.

[RCGF97] Ismael Ripoll, Alfons Crespo, and Ana García-Fornes. An optimal algorithm for scheduling soft aperiodic tasks in dynamic-priority preemptive systems. *IEEE Trans. Software Eng.*, 23(6):388–400, 1997.

[Reg02] John Regehr. Inferring scheduling behavior with hourglass. In *USENIX Annual Technical Conference, FREENIX Track*, pages 143–156, 2002. http://www.usenix.org/publications/library/proceedings/usenix02/tech/freenix/regehr.html.

[RH95] Parameswaran Ramanathan and Moncef Hamdaoui. A dynamic priority assignment technique for streams with (m, k)-firm deadlines. *IEEE Trans. Comput.*, 44(12):1443–1451, 1995.

[SAWJ+96] Ion Stoica, Hussein M. Abdel-Wahab, Kevin Jeffay, Sanjoy K. Baruah, Johannes Gehrke, and C. Greg Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *IEEE Real-Time Systems Symposium*, pages 288–299, 1996. http://computer.org/proceedings/rtss/7689/76890288abs.htm.

[SBA+01] Tajana Simunic, Luca Benini, Andrea Acquaviva, Peter Glynn, and Giovanni De Micheli. Dynamic voltage scaling and power management for portable systems. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 524–529. ACM Press, 2001.

[SBS95] Marco Spuri, Giorgio C. Buttazzo, and Fabrizio Sensini. Robust aperiodic scheduling under dynamic priority systems. In *IEEE Real-Time Systems Symposium*, pages 210–221, 1995.

[SJ07] Abhishek Singh and Kevin Jeffay. Co-scheduling variable execution time requirement real-time tasks and non real-time tasks. In *ECRTS*, 2007.

[TDS+95] Too-Seng Tia, Zhong Deng, Mallikarjun Shankar, M. Storch, Jun Sun, L.-C. Wu, and Jane W.-S. Liu. Probabilistic performance guarantee for real-time tasks with varying computation times. In *IEEE Real Time Technology and Applications Symposium*, pages 164–173, 1995.

[USR02] Bhuvan Urgaonkar, Prashant J. Shenoy, and Timothy Roscoe. Resource overbooking and application profiling in shared hosting platforms. In *OSDI*, 2002. http://www.usenix.org/events/osdi02/tech/urgaonkar.html.

[WWDS94] Mark Weiser, Brent Welch, Alan J. Demers, and Scott Shenker. Scheduling for reduced CPU energy. In *Operating Systems Design and Implementation*, pages 13–23, 1994.

[XXMM04] Ruibin Xu, Chenhai Xi, Rami Melhem, and Daniel Moss. Practical pace for embedded systems. In *EMSOFT '04: Proceedings of the fourth ACM international conference on Embedded software*, pages 54–63. ACM Press, 2004.

[YN03] Wanghong Yuan and Klara Nahrstedt. Energy-efficient soft real-time cpu scheduling for mobile multimedia systems. In *SOSP*, pages 149–163, 2003.

[YN04] Wanghong Yuan and Klara Nahrstedt. Practical voltage scaling for mobile multimedia devices. In *ACM Multimedia*, pages 924–931, 2004.