IMPLEMENTATION AND EVALUATION OF PROPORTIONAL SHARE SCHEDULER ON LINUX KERNEL 2.6

A thesis presented to

the faculty of

the Russ College of Engineering and Technology of Ohio University

In partial fulfillment

of the requirements for the degree

Master of Science

Pradeep Kumar Srinivasan

March 2008
ABSTRACT

SRINIVASAN, PRADEEP KUMAR, M.S., March 2008, Computer Science

Implementation and Evaluation of Proportional Share Scheduler on Linux Kernel 2.6 (121pp.)

Director of Thesis: Frank Drews

There is a steady proliferation of Time Sensitive applications, adding more diversity to the regular workload on General Purpose Operating Systems (GPOS). Priority based Scheduling in GPOS are not designed to handle a mix of regular workload and Time Sensitive applications. An Alternative approach to handle this mixed workload is to use Proportional Share Scheduling (PSS). In PSS model, each task receives a share of the resource in proportion to the weight assigned to it. Most of the PSS schemes proposed in the literature assume the existence of an interface, that can translate application requirements in to weights. And some leave the onus of specifying weights to the application on the User/Application Developer. In this thesis, we describe our implementation of Earliest Eligible Virtual Deadline First (EEVDF) based PSS on Linux Kernel. We have designed and developed an Abstract Scheduling Interface (ASI) to our Proportional Share Scheduler. ASI provides an interface to translate resource requirements of tasks in the form of shares to weights for PSS Tasks.

Approved: ________________________________

Frank Drews

Assistant Professor of Electrical Engineering and Computer Science
To Suseela, Srinivasan, Ramaswamy

& in fond memory of Gomathy Ramaswamy
Acknowledgments

I am deeply indebted to Dr. Frank Drews for trusting my abilities, understanding my interests by offering me an opportunity to work on this interesting area of Resource Management. I am also thankful to him for his immense patience in bearing with me, constant motivation, support and encouragement throughout my graduate curriculum at Ohio University.

I would like to thank Dr. Shawn Ostermann and Dr. Robert Judd for being part of my thesis committee. I would always cherish the projects that I worked during my class work under their guidance.

I am grateful to Dr. Vardges Melkonian for being college representative as part of my thesis committee.

I would like thank Bryan Jordan and Paul Deering for providing me an opportunity to work with them, which secured my graduate student life financially.

I would also like to thank Mike Dunn who helped me at various stages of this thesis in the form of development environment, interesting ideas and suggestions.

Finally, I am thankful to the almighty God for blessing me with wonderful family and friends, who bestowed their full support especially in really critical times.
Table of Contents

Abstract .................................................. 3
Dedication .................................................. 4
Acknowledgments .......................................... 5
List of Tables ............................................. 10
List of Figures ............................................. 11
1 Introduction ............................................. 12
  1.1 Generalized Classification of Workload ................ 12
  1.2 Scheduling - An Overview ............................ 14
  1.3 The Problem ......................................... 15
  1.4 Proportional Share Scheduling ....................... 17
  1.5 Our Contributions ................................... 18
2 Our Motivation .......................................... 20
  2.1 Dissecting a MPEG-1 Video Player .................... 20
      2.1.1 An Overview of MPEG-1 Video Compression ....... 21
      2.1.2 Inside mpeg_play - MPEG-1 Player ............... 23
      2.1.3 Exposing GPOS Video Playback with Load ........ 24
  2.2 Motivating Examples ................................ 27
      2.2.1 Resource Reservation in Virtualized Environment .28
      2.2.2 Differentiated Service Levels in Web Hosting .... 29
3 Scheduler Improvements Proposed in Literature ......... 31
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Priority Based Scheduling</td>
<td>31</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Unsuitability of Priority Based Scheduling in GPOS</td>
<td>34</td>
</tr>
<tr>
<td>3.2</td>
<td>Scheduling Based on Reservation</td>
<td>36</td>
</tr>
<tr>
<td>3.2.1</td>
<td>CBS - An Example for Server Based Allocation Model</td>
<td>37</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Costs and Benefits of Using Reservation Based Scheduling</td>
<td>38</td>
</tr>
<tr>
<td>3.3</td>
<td>Proportional Share Scheduling</td>
<td>39</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Ideal Fluid Flow Paradigm</td>
<td>39</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Emulation of Fluid Flow Model</td>
<td>40</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Terminology</td>
<td>41</td>
</tr>
<tr>
<td>3.3.4</td>
<td>The Concept of Virtual Time</td>
<td>42</td>
</tr>
<tr>
<td>3.3.5</td>
<td>Earliest Eligible Virtual Deadline First - EEVDF</td>
<td>43</td>
</tr>
<tr>
<td>3.3.6</td>
<td>Other Variations of PSS</td>
<td>48</td>
</tr>
<tr>
<td>4</td>
<td>ASI - Enhancements to EEVDF Based PSS</td>
<td>51</td>
</tr>
<tr>
<td>4.1</td>
<td>Design Goals</td>
<td>51</td>
</tr>
<tr>
<td>4.2</td>
<td>Why EEVDF Based PSS Solution?</td>
<td>52</td>
</tr>
<tr>
<td>4.3</td>
<td>Issues with EEVDF</td>
<td>53</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Allocation Uncertainties</td>
<td>54</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Weight Assignment</td>
<td>55</td>
</tr>
<tr>
<td>4.4</td>
<td>Abstract Scheduling Interface</td>
<td>56</td>
</tr>
<tr>
<td>4.4.1</td>
<td>An Overview</td>
<td>56</td>
</tr>
<tr>
<td>4.4.2</td>
<td>How Does it Work?</td>
<td>57</td>
</tr>
<tr>
<td>5</td>
<td>Our Prototype Implementation</td>
<td>63</td>
</tr>
</tbody>
</table>
5.1 Dissecting Linux 2.6 Scheduler .............................................. 63
  5.1.1 Process - States Inside Kernel ........................................ 64
  5.1.2 Linux Scheduling Policy .............................................. 66
  5.1.3 Calculating Priorities ................................................ 66
  5.1.4 Choosing the Highest Priority Process ............................. 68
  5.1.5 Kernel’s Notion of Time ............................................. 69
  5.1.6 An Overview of Scheduler Related Functions ....................... 69

5.2 Our Implementation of EEVDF in Linux 2.6.18 ......................... 71
  5.2.1 Implementation Strategies ........................................... 72
  5.2.2 Data Structures ..................................................... 73
  5.2.3 How Does Our EEVDF Implementation Work? ....................... 74

5.3 Implementation of Abstract Scheduling Interface .................... 82
  5.3.1 Summary - ASI Design .............................................. 82
  5.3.2 Implementation Strategy .......................................... 83
  5.3.3 Data Structures ................................................... 84
  5.3.4 Interfaces ......................................................... 84
  5.3.5 How Does it Work? ................................................ 86

6 Experiments and Results .................................................... 89
  6.1 Our Approach to Analyze Scheduling behavior ....................... 89
    6.1.1 Custom Workload Generator ..................................... 91
    6.1.2 Issues with Custom Workload Generator ......................... 93
    6.1.3 Hourglass ....................................................... 93
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2</td>
<td>Experiments with PSS</td>
<td>94</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Demonstration of Fairness in PSS</td>
<td>94</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Measurements with Hourglass</td>
<td>95</td>
</tr>
<tr>
<td>6.3</td>
<td>Experiments With ASI</td>
<td>97</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Mix - Constant Shares and Proportional Use of Spare Resource</td>
<td>98</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Proportional Use of Spare Resource</td>
<td>101</td>
</tr>
<tr>
<td>6.3.3</td>
<td>Constant Shares Only</td>
<td>103</td>
</tr>
<tr>
<td>7</td>
<td>Conclusions and Future Work</td>
<td>106</td>
</tr>
<tr>
<td>7.1</td>
<td>Limitations and Improvements to Our Prototype Implementation</td>
<td>107</td>
</tr>
<tr>
<td>7.2</td>
<td>Future Work</td>
<td>108</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>110</td>
</tr>
<tr>
<td>A</td>
<td>EEVDF Implementation</td>
<td>119</td>
</tr>
<tr>
<td>A.1</td>
<td>Data Structures Used</td>
<td>119</td>
</tr>
<tr>
<td>A.1.1</td>
<td>Request Data Structure</td>
<td>119</td>
</tr>
<tr>
<td>A.1.2</td>
<td>Request List</td>
<td>119</td>
</tr>
<tr>
<td>A.1.3</td>
<td>PSS Queue</td>
<td>120</td>
</tr>
<tr>
<td>A.2</td>
<td>Fixed Point Macros</td>
<td>120</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Details of the Video Stream Used in the Experiments</td>
<td>24</td>
</tr>
<tr>
<td>3.1</td>
<td>$V_e$ and $V_d$ Computations for Sample Tasks T1 and T2</td>
<td>47</td>
</tr>
<tr>
<td>6.1</td>
<td>System Description</td>
<td>89</td>
</tr>
<tr>
<td>6.2</td>
<td>Measurements on Execution Traces Using Hourglass for 3 PSS Tasks</td>
<td>97</td>
</tr>
<tr>
<td>6.3</td>
<td>Observed Shares of ASI Tasks - Mixed Scenario</td>
<td>100</td>
</tr>
<tr>
<td>6.4</td>
<td>Iterations Performed During Time Intervals - Mixed Scenario</td>
<td>101</td>
</tr>
<tr>
<td>6.5</td>
<td>Observed Shares of 3 ASI Tasks - Proportional Allocation of Spare Resources</td>
<td>103</td>
</tr>
<tr>
<td>6.6</td>
<td>Iterations Performed During Time Intervals - All Tasks Requesting Benefits</td>
<td>104</td>
</tr>
</tbody>
</table>
List of Figures

2.1  A Plot of Processing Times of I,P and B Frames in Sample Video Stream . 24
2.2  Frame Rate of Video Player on a Lightly Loaded System .................. 25
2.3  Frame rate of Video Player in the Presence of Synthetic Load ............ 27
2.4  Frame Rate of Video Player as a Real Time Task with Synthetic Load ... 28
3.1  An Example EEVDF Schedule with 2 Tasks T1 and T2 ....................... 46
4.1  A High Level Overview of Abstract Scheduling Interface for PSS .......... 57
4.2  A Figure Capturing ASI IDLE State Transitions ............................ 61
5.1  A Figure Capturing Process State Transitions ............................... 65
5.2  A Figure Showing Runqueue layout ........................................ 67
5.3  A Call Graph of Functions Invoked During PSS Task Create ............. 76
6.1  A Schematic Describing How Our Workload Generator works. ............. 92
6.2  A Plot of 3 PSS Tasks with Weights 4, 3 and 1 ............................ 95
6.3  Execution Trace of 3 Regular Linux Tasks with Priorities: Low, Normal, High 96
6.4  Execution Trace of 3 PSS Tasks with Weights 4, 3, and 1 ................ 97
6.5  Mix - ASI Tasks with Constant Shares and Proportional Allocation of Resource 99
6.6  ASI Tasks - Proportional Allocation of Spare Resource ................... 102
6.7  ASI Tasks with Constant Shares Only ...................................... 105
Chapter 1

Introduction

The rate of progress in the area of Computing is simply mind blowing. In the 90’s, the workload on a desktop computer system typically consisted of a mix of Console based command line terminals, Text editors, Browsers capable of rendering static web pages, Document management using word processors, Spread sheet editors etc [1]. We had been making slow but steady transition from static content, typical of 90’s workload to media rich Interactive content. A Testimony to this shift is the current crop of Media rich applications that we use in day to-day life including: Video players in various flavors like stand alone playing video encoded in various formats (mpeg1, mpeg2, etc), Flash based players integrated in to browsers etc, Voice over Internet Protocol (VoIP) applications etc. The Critical point is, that the workload on today’s computing platform is much more diverse as it has been in the past.

1.1 Generalized Classification of Workload

Traditionally, a process in execution is viewed as a cycle of CPU burst and Input/Output (I/O) waits [2]. Based on the duration of CPU bursts or I/O waits, the processes can be classified CPU bound processes or I/O bound processes. For example, a process that is spending more time doing I/O would typically have very short CPU bursts and it is classified as I/O bound process. At the highest level, the
workload on the General Purpose Operating Systems (GPOS) can be classified into 3 categories [3, 4]: Interactive, Batch and Time Sensitive Applications.

- **Interactive Applications** - This class of applications is driven by user interactions. An example of Interactive application is a simple Text Editor waiting for the user input. Tracing the events: User presses key $\rightarrow$ interrupts keyboard interrupt controller $\rightarrow$ which interrupts the CPU $\rightarrow$ keyboard interrupt handler is called $\rightarrow$ keyboard device driver is invoked and it copies the character pressed $\rightarrow$ character sent to the Pseudo-terminal $\rightarrow$ wake up Text Editor. Since the editor is waiting for events, it is put to sleep. It is woken up, once it has data for processing. This class of applications usually requires short CPU bursts. Hence they are I/O bound processes.

- **Batch Applications** - Batch applications are typically CPU bound processes spending most of their allotted CPU Slice executing code. The user may not expect a definitive response time on these applications. For example, compiling and building an operating system kernel, generating reports by relating huge volume of data are batch applications.

- **Time Sensitive (TS) Applications** This is an emerging class of applications, which have temporal characteristics. In these applications, not only correctness of the result matters, but also providing logically correct results with in the time constraints matters. A failure to meet the time constraints result in loss
of quality. They are also referred to as “Soft Real Time” applications, in which occasional misses in time constraints are tolerated [5, 6]. A typical example for this class of applications is video playback. A video playback application has to decode and display the media samples at a steady rate. A substantial delay between the processed output stream and incoming media stream results in loss of quality (also known as QoS -Quality of Service) in the form 'jitter' [7, 8]. These applications perform CPU intensive computations and hence they are CPU bound processes.

1.2 Scheduling - An Overview

Clearly, the diverse mix of applications described above have demanding requirements in terms of resources such as CPU processing time, memory, etc and present new challenges to managing Resources. Are General Purpose Operating Systems (GPOS) able to handle these requirements? This question can be answered by developing a high level understanding of operating system. The design of the state-of-the-art GPOS revolves around the concept of Multiprogramming [2]. Multiprogramming achieves two critical design goals of modern operating systems: maximizing CPU utilization by overlapping CPU usage with Input/Output; maintaining an illusion of steady progress of all processes to the user. This is made possible by the CPU Scheduler, which is a core component of operating systems. Given a set of runnable processes, the Scheduler decides which process should be executed at
any given point of time. This decision is referred to as 'Scheduling Policy' and it determines the overall feel of the system [9].

Predominantly, the GPOS Scheduler make this policy decision based on two important criteria: 'response time' 1, 'throughput' 2. GPOS Scheduler strives to maximize the throughput and lower the response time [10]. Most of the GPOS achieve this by employing Priority based Scheduling. There are several variants of Priority based scheduling used in GPOS [11, 12]. But the common theme behind them is to rank processes, based on their need for CPU processing time and assign priorities to them. Whenever the scheduler is invoked to make a decision, it chooses a process with the highest priority for execution.

1.3 The Problem

Most of the GPOS attempts to lower the response time for interactive applications by penalizing applications, which shows patterns of high CPU usage. That is, priority of a process and its CPU usage are inversely related [11]. For example, in an I/O bound processes, CPU usage is lesser which translates in to higher priority. This way, most of the GPOS explicitly prefer I/O bound processes over CPU bound processes. This approach is fine, as long as the workload is mainly comprising of Interactive or Batch applications. But in the presence Time Sensitive (TS) applications,  

---

1Response Time - the time between process getting activated to the time when results presented to the user [2].
2Throughput - number of processes active and making progress in a given time.
it may yield lower quality for time sensitive applications. Since time sensitive applications are CPU intensive, they may receive lesser priority. Further, when I/O bound processes wakes up from sleep, they can preempt time sensitive processes with their high priority. For example, consider the preemption of video playback application by interactive task. It may cause the video application to be unable to process the media sample stream on time. This can result in frames getting dropped and hence unacceptable quality. We had verified this scenario experimentally and a discussion is presented in chapter 2.

An obvious solution is to run time sensitive applications at the highest available priority in the system. This would guarantee that the TS applications get scheduled at the right instance and thus better QoS. But in this approach, TS applications could starve other applications. There is a tendency to oversubscribe for resources and thus it leads to under utilization of resources. There are other reasons presented in chapter 3 making a case for us to look for other solutions.

This thesis attempts to address following problems with scheduling in GPOS:

- Inability to provide sufficient support for time sensitive applications. For example, an application may not be able to honor its temporal constraints because of other possibly misbehaving tasks in the system.
Most of the GPOS do not handle mix of applications in an efficient way. They tend to favor a particular class of applications which may not be acceptable all the time.

When the CPU is heavily contended, the GPOS scheduler allows a single process to monopolize the system. This precludes the chance of graceful degradation in overload scenarios.

Priority based scheduling does not map or interpret the QoS levels of various applications very well [13]. There is no natural way to change QoS level of one application explicitly as a high priority task always runs at the expense of low priority task. This scheme is inflexible [14].

1.4 Proportional Share Scheduling

We believe that it is possible to address each of the problem listed in previous section by using Proportional Shared Scheduler (PSS). Rephrasing the description on PSS by authors of [15]: In PSS, all the tasks in the system are associated with a 'weight'. The 'weight' determines the share the that task is supposed to receive. Resource is allocated in quanta of fixed size to each task based on the share. Thus all the tasks make progress in a uniform rate [16, 4, 17, 18, 5]. Two important characteristics PSS [19, 20] are:

- Flexibility - support for dynamic operations in a seamless manner.
• Fairness - resource allocation is in proportion to weight of the task.

There are few issues with using Proportional Share approach, in the context of supporting mix of applications in GPOS. They need to be addressed:

• In Proportional Share approach, even in the presence of sufficient resources, dynamic operations such as tasks entering/leaving competition for the resource could cause fluctuations in the share received by the Tasks. We refer to these fluctuations in share as ‘Allocation Uncertainties’ and some tasks that require constant share may not tolerate these fluctuations.

• In PSS, "weight" of a task determines its share. If this is the case, how would an application convey its weight to the scheduler?

• Some of the PSS models assume that there exists an higher level interface that can translate application requirements in to weights [15]. To our knowledge, the problem of assigning weights in the PSS model has received little attention.

1.5 Our Contributions

Earliest Eligible Virtual Deadline First (EEVDF) based Proportional Share Scheduling was first proposed by Stoica et.al [4]. To the best of our knowledge, there is no implementation of EEVDF based PSS available for current state of the art GPOS. Many modern GPOS supports Kernel Preemption, in which multiple threads
of control are interleaved in kernel. An Implementation on such a system would be non-trivial. Our primary contribution is an Implementation of EEVDF based PSS in Linux Kernel 2.6.

Our second major contribution towards this thesis is the design and implementation of Abstract Scheduler Interface (ASI) for EEVDF based PSS CPU Scheduler. ASI is designed to provide an interface to PSS, so that the user intending to proportionally share the resource may be freed from dealing with weights to tasks. In EEVDF, there is no provision to handle an application requesting a constant share of the resource. ASI is also designed to address the issue of constant share resource requirement by applications. ASI achieves constant share requested by applications by re-computing weights assigned to tasks and schedules them as PSS Tasks.
Chapter 2

Our Motivation

In this chapter, we consider the example of video playback application and try to understand the time sensitive nature of this application. In the process, we will also be able to see the impact of other interactive applications on time sensitive applications in GPOS. Armed with this understanding, we can look at some motivating examples, which would give us an insight into where the improvements proposed in this thesis can be applied.

2.1 Dissecting a MPEG-1 Video Player

The Video playback applications like MPEG-1 video player, may resemble soft Real Time applications, in the aspect the of being periodic or requiring fixed quantum of processing time at regular intervals. But they are not strictly periodic. Their demand for processing time may not begin at the beginning of the new interval. In the following sub section, we will develop a high level understanding of MPEG-1 video compression, which will provide the answer to why video playback applications are not strictly periodic. A more detailed discussion on this topic is beyond the scope of this thesis and it is available in [21, 22].
2.1.1 An Overview of MPEG-1 Video Compression

MPEG-1 is a set of standards developed by Motion Picture Experts Group in the year 1988, at the initiation of CCITT (International Telephone and Telegraph Consultative Committee) also known by current name ITU-T. MPEG-1 aimed for video compression at 1.5 Mbps and audio compression of digital audio at 64, 128 and 192 Kbps per channel [21], so that video could be delivered over the communication and computer networks of that time. It was developed with the intention of stopping propagation of Proprietary encoding technologies that would not inter-operate with multitude of devices.

Following details are summarized from explanations in [23, 22, 21]. Video can be viewed as a set of still images/frames. Each of these frames contains lots of information, some of which are redundant across the frames, some of them contain information that are not perceptible to human eye. MPEG-1 exploits these intricacies in encoding the video, by using the following techniques:

- Transform Coding
  - exploits human eye’s sensitivity to very high frequencies.
  - By applying mathematical transforms, represents the information in an image using fewer values and thus better compression
  - MPEG-1 performs transform coding by Discrete Cosine Transform (DCT).
Motion Compensation

- exploits redundant information across the frames. For example, let us consider consecutive frame I1, I2 and I3. Frame I1 can be looked up as a key frame containing most of the information and subsequent frames differ from I1 in subtle way. By encoding just the differences between previous frame and the current frame a better compression is achieved.

Entropy Coding

- information encoded in the Transform Coding and Motion Compensation techniques are further encoded using Huffman coding to achieve even better compression.

MPEG-1 processes each image/frame by treating it to be made of multiple macro blocks. Each macro block is 16X16 pixels and made of 4 blocks. Each block is 8X8 pixel. Each block is subjected to DCT to perform Transform Coding. To perform Motion Compensation, MPEG-1 differentiates the frames to be of 3 types:

- I-Frame:
  - Intra Frame/I-Frame are independent self contained frames. They serve as reference frame to do motion compensation for other type of frames
  - Motion compensation is not performed on these frames, as they are self contained.
• P-Frame:
  – Predicted Frame/P-Frame are constructed from preceding frames

• B-Frame:
  – Bi-Directional Predicted Frame/B-Frame are constructed from preceding and succeeding frames.

2.1.2  Inside mpeg_play - MPEG-1 Player

An MPEG Player has to undo all the steps followed in the encoding process. In order to understand the requirements and constraints of video playback application, we chose a video playback application called ”mpeg_play” [23, 22]. This is a publicly available MPEG decoder developed by Berkeley Multimedia Research Center and it is ported on wide variety of platforms. It can decode only MPEG-1 videos and uses X-Windowing system to display decoded video.

We observed the processing time required for Intra, Predicted and Bi-directional Predicted (I,P, B) frames for a Sample Video Stream. As expected, the processing time for each frame were not the same. Processing I-Frame takes longer time as compared to B an P Frames. The number of I, B and P frames are dependent on the video stream. This observation re-confirms our understanding, that video playback applications resemble Soft Real Time, but they are not strictly periodic Soft Real Time application. A plot of processing time for various types of frames in a sample
video stream is shown in the Figure 2.1. The details of the sample video stream used in the experiments are presented in Table 2.1.

Table 2.1: Details of the Video Stream Used in the Experiments

<table>
<thead>
<tr>
<th>Description</th>
<th>Detail</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dimensions</td>
<td>Width: 640 Height: 360</td>
</tr>
<tr>
<td></td>
<td>2328</td>
</tr>
<tr>
<td>Total number of Frames</td>
<td>77.68 seconds</td>
</tr>
<tr>
<td>Length of Stream</td>
<td>29.97 frames/second</td>
</tr>
<tr>
<td>Required Display Speed for MPEG viewer</td>
<td></td>
</tr>
</tbody>
</table>

2.1.3 Exposing GPOS Video Playback with Load

The frame rate for video playback application is an important QoS (Quality of Service) metric. The human eye can perceive a drastic change in frame rate,
especially when the video is played at less than 30 frames/second. To maintain a constant frame rate, the video playback application has to decode I, P and B Frames at a steady rate irrespective of the amount of information encoded in them. This can be guaranteed on a system, on which video player is being scheduled at the exact intervals, as needed by video decoder. Repeated failure to get the application scheduled at the exact interval would result in key frames not getting decoded. This would result in drop in frame rate and hence an unacceptable QoS.

With GPOS preferring interactive tasks explicitly by according higher priorities, CPU intensive applications like video player may receive lesser priority. Hence, when the system has too many interactive tasks competing for CPU slices, video playback applications would not get scheduled at precise intervals resulting in frame drops. We observed the above scenario experimentally.

Figure 2.2: Frame Rate of Video Player on a Lightly Loaded System
Video Player with Less Load

We instrumented mpeg_play, to record frame rate at regular intervals as the video is displayed on X-windows. The frame rate recorded as part of the experiment is cumulative frame rate. It indicates the total number of frames decoded, since the start of the experiment. First, we observed frame rate for a sample video stream, on a system running essential system processes and daemons. The frame rate holds steady at 29.9 frames/second as the video player is scheduled at the right time always. The result of this experiment is shown in the figure 2.2.

Video Player with Synthetic Applications

We observed the frame rate again, running the same video stream on a system running synthetic applications. The synthetic applications are designed to emulate regular tasks (Interactive and CPU bound). In this experiment, the synthetic stress applications are started off after the video player has executed for 3 seconds. The video player is about to stabilize to a steady frame rate as shown in the figure 2.3. But the synthetic stress applications preempts the video applications at regular intervals. This results in video player dropping frames. As it is evident from the figure 2.3, between time 3 to 25 seconds, the synthetic stress applications compete for CPU slices and the frame rate drops during this period. When synthetic applications quit after completing its execution at 24 seconds, the frame rate picks up to stable value.
Video Player as a Real Time Task with Load

We did set up another experiment to observe frame rate, with the video player running as a Real Time Task. The synthetic stress applications are executed as regular linux tasks (they do not belong to Real Time scheduling class). With this setup, as expected the video player has a perfect monopoly over the system Resources. The synthetic stress applications did not have any impact on the Frame Rate of the Video Player, which is confirmed from Figure 2.4. The synthetic applications makes progress at a steady rate whenever, the video application running as Real Time Task is put to sleep waiting for events.

2.2 Motivating Examples

We present other motivating Examples in this Section:
2.2.1 Resource Reservation in Virtualized Environment

There used to be one to one mapping between the operating system and
the hardware on which the operating system runs. With Virtualization, it no more
holds true, as it is possible to share the underlying hardware with multiple oper-
ating systems. Virtualization is a technology in which a thin software layer (called
Hypervisor/Virtual Machine Monitor) manages underlying hardware and it creates
components called Virtual Machines [24]. Virtual Machines can be viewed as independ-
dent containers in which guest operating systems run on virtual hardware emulated
by Hypervisor.

Virtualization is an area of tremendous growth potential. Its benefits include server
and storage consolidation [25, 26], better resource utilization of hardware infrastruc-
ture, and debugging/evaluating performance and robustness of operating systems etc.

With the benefits of virtualization as mentioned above, we envision the scheduler improvements suggested in this Thesis to be used in the Hypervisor Scheduler. A Hypervisor scheduler could partition the resource requirements to each guest Operating Systems as configured by the administrator/user.

2.2.2 Differentiated Service Levels in Web Hosting

With the ever-rising popularity of Internet, it is very common for small businesses and individuals to host their content on Internet and they use services of web Hosting companies. The Hosting companies may offer differentiated service levels in the system, in processing the incoming requests, based on the content and maximize their revenue. For example, in a web server if the incoming requests are processed in the order in which they are received, there is a chance of the most popular content monopolizing the system. This could potentially cause the requests for contents of other customer’s be delayed. The hosting company could charge the customer, a premium in hosting the on demand contents. The hosting company by proportionately charging customer with popular content can offer diverse service levels.

The idea of providing differentiated service levels in web hosting has been addressed in [27]. But their approach is based on priority based requests scheduling. This can be achieved in a more natural way by using Proportional Share approach discussed in this Thesis. The web server in the hosting system can be modified to
have a proportional share based Request Scheduler. Each incoming Request could be associated with a weight, based on the type of Content being accessed. In this approach, the requests for content of non-premium customers would not be starved at the expense of premium content.
CHAPTER 3

Scheduler Improvements Proposed in Literature

In this chapter, we look at the various scheduling improvements proposed in the literature especially to handle mix of continuous media applications and conventional desktop applications in GPOS. We do not attempt to survey all the Scheduling algorithms proposed in the literature. We consider a high level overview of Scheduling paradigms and consider a few representative examples to understand the benefits of each paradigm. In the context of providing better support to Time Sensitive applications, the Scheduling paradigms can be classified in a broad way in to 3 categories [28]:

- Priority based Scheduling
- Reservation based Scheduling
- Proportional Share Scheduling.

3.1 Priority Based Scheduling

Priority based Scheduling is a very generic Scheduling method in which the scheduler chooses the Highest priority task for execution, whenever it is activated
to make a scheduling decision. It is used in wide variety of Systems ranging from regular General Purpose Operating Systems, Mission Critical Space Shuttles and Rovers (NASA Mars PathFinder robot \[29\]), to Industrial and Safety control systems. There are various flavors of Priority based Scheduling in use: Fixed Priority, Dynamic Priority, etc. In Fixed Priority Scheduling the priority of the task is fixed and remains the same throughout the life of task \[30\]. This strategy is commonly used in Real Time Systems Scheduling, where the tasks have well defined time constraints. Failure to honor time constraints could result severe consequences such as System breakdowns. In Computer literature these tasks are referred as Hard Real Time tasks. Scheduling in a predictable and deterministic way is more important in such Systems.

Classical Real Time Scheduling was first mathematically analyzed by Liu and Layland \[31\]. Two such popular Hard Real Time Scheduling algorithms are Rate Monotonic (RMA) and Earliest Deadline First (EDF) algorithm. Afore mentioned algorithms are preemptive and priority based. These algorithms consider a periodic Task model, in which prior knowledge of following information about each task is assumed: Worst Case Execution time, \(C_i\) and periodicity of the task \(T_i\). RMA is a static priority algorithm in which all the tasks that will be contending for resource in the system is known in advance. RMA works by assigning the highest priority to a task with shortest period \(T_i\). So, for a set of tasks say from 1 \(...n\), the Total CPU utilization would be computed as \(f\) allows:
\[ \sum_{i=1}^{n} \frac{C_i}{T_i} = U \]  

(3.1)

It is a simple and well understood algorithm, but the Scheduler utilization bound that can be achieved with RMA is:

\[ U_b = n \times (2^{\frac{1}{n}} - 1) \]  

(3.2)

An extensive theoretical proof of Utilization bound is available in [31]. The above equation means that with RMA, it is possible to schedule all the periodic tasks if the Total CPU utilization is less than the Utilization bound:

\[ U \leq U_b \]

If \( U > U_b \), it may not be possible to schedule all periodic tasks using RMA [32, 31]. Adding support for Rate Monotonic Algorithm on a fixed priority based Scheduling is feasible and very common [32].

EDF is a Dynamic priority algorithm, which tries to improve upon the Utilization bound for RMA as in (3.2). It is dynamic in the sense that, task priorities are computed on the fly based on their \((C_i, D_i)\), where \(D_i\) is the deadline by which the task should complete. As the name implies, a task with Earliest Deadline will be given highest priority. No Scheduling algorithm can guarantee a schedule if the total CPU Utilization exceeds 100%. With EDF, the scrollable utilization bound is (theoretical
proof in [31]):

\[ U_b \leq 1 \quad (3.3) \]

EDF offers improved resource utilization as long as Total Utilization is below 100%.
But once the utilization of the resource shoots above 100 %, it would start missing deadlines invariably for all the tasks in the system irrespective of their priority.
Because of non-graceful handling of overload scenarios, a systems developer implementing EDF needs to care about ensuring the task’s utilization is maintained under 100%.

### 3.1.1 Unsuitability of Priority Based Scheduling in GPOS

Most of the Commercial Hard Real Time Systems like VxWorks, pSOS, QNX has implemented a Fixed Priority Scheduling. If the fixed priority scheduling is used in Hard Real Time Systems, where honoring time constraints is highly critical one might wonder why can’t it be used in GPOS ? For reasons mentioned below, we find Priority based Scheduling may not be able to address the drawbacks that we listed in Section 1.3:

- **Temporal Protection** - Considering the mix of Time Sensitive applications and Conventional interactive applications in GPOS, that may enter or leave the system dynamically, it is not possible to provide temporal protection to Time Sensitive tasks using Priority based Scheduling. For example, a Time Sensitive
task could be preempted by a other tasks with higher priority, which could potentially cause the Time Sensitive task to fail the time constraint.

- **Task Model** - In order to guarantee timeliness, this model assumes that the scheduler does have prior knowledge about task’s periodicity, execution time and deadlines. But in a GPOS with mix of applications that we are considering, it is not always possible to have these information for all the tasks in the system. For example, let us consider a text editor being used to edit a file. The speed at which the user is typing characters controls when the editor would receive characters for processing.

- **Overload Scenario** - In situations where the resource is heavily contended, there is no chance of graceful degradation as a higher priority task would continue to monopolize the system leaving very little room for other tasks to progress. Both algorithms behave unpredictably [33, 32]

- **Priority Inversion** - This is a scenario which most of the priority based Scheduling algorithms has to deal with. In order to protect shared resources, Operating Systems use Synchronization primitives (mutex, semaphores, etc). When there are shared resources between say high and low priority task, and low priority task has acquired lock to work in that critical section, high priority task could be locked out for a long time. The situation becomes even worse in the case of unbounded priority inversion where high priority task would be
forced to wait indefinitely. This type of scenarios has to be dealt with Priority Inheritance or Priority Ceiling protocols [34] which complicates the scheduling model.

- **Over-subscription of Resources** - Tasks are characterized based on Worst Case Execution Times (WCET). Hence there is a tendency to oversubscribe for resources. This could lead to inefficient use of Resources. This is also referred to as Priority inflation [35, 36].

- **Escalation of user Privilege** - In GPOS with support for running Real Time tasks, a user may need Special User privileges (such as Super User access in Unix flavors) to run Real Time tasks. As observed by authors of [37], it is important to answer the question of to whom should we give Special User privileges? If we give Special privilege to all users, then the value of the privilege is diluted. If each User of a shared System, is launching applications as Real Time Tasks with conflicting Resource Requirements, the System may not be shared in a beneficial way between any user.

### 3.2 Scheduling Based on Reservation

Like the Classical Real Time Scheduling model, the Reservation based Scheduling paradigm also offers strong Temporal Protection to tasks. The origins of Reservation based Scheduling can be traced to the problem of scheduling aperiodic tasks in
the Real Time Systems [38, 13]. As we saw in the previous section, the Task Model
of most Real Time Scheduling algorithms requires the tasks be periodic. To handle
Aperiodic Tasks, the authors of [38] proposed a Sporadic Server which is a Real Time
Periodic task. When this Sporadic Server is scheduled to run, it looks for any pending
Aperiodic request and services it. Reservation based Scheduling can be seen as an
extension of Sporadic Server to GPOS, also referred in the literature as Server based
Allocation [13, 37, 28].

3.2.1 CBS - An Example for Server Based Allocation Model

To better understand Server Based Allocation model, we consider Constant
Bandwidth Server (CBS) [37, 39]. The Core of CBS approach is to allocate a fraction
of CPU to a task and to make sure that it doesn’t over shoot its utilization by
policing task’s usage. The CBS algorithm can be explained as follows: A Server $S_i$
has a fraction of CPU bandwidth dictated by Server’s defined Capacity $Q_i$ and Period
of Server $T_i$. The Server’s bandwidth can be expressed as

$$S_i = \frac{Q_i}{T_i}$$

The Server internally maintains a deadline $d_i^{s}$ and capacity/budget $q_i$. A Task $t_i$ that
enters the system, if served by $S_i$, is allocated a dynamic deadline by the Server and
placed in a queue. The the tasks are scheduled by Earliest Deadline First (EDF)
algorithm. When the task $t_i$ executes, the the Server’s capacity $q_i$ reduces. When the
Server’s bandwidth becomes zero, its defined capacity $Q_i$ is restored and its deadline
is pushed ahead by a Period. The Task’s deadlines in the EDF queue are recalculated to reflect the change. This way the task’s will not over shoot its utilization.

### 3.2.2 Costs and Benefits of Using Reservation Based Scheduling

With a high level overview of Reservation based Scheduling, we look the benefits of using this scheme:

- It offers a strong temporal protection for tasks that demand a fixed quantum of processing time

The Costs involved in using Reservation based Scheduling are listed below:

- The Task Model of this scheme is not so simple. The Task of how to choose the number of Servers and partition the CPU among them is left to the System Designer (One solution to this issue would be group the application in to various Application Classes and have Server to cater to each Application Class). It is not very clear on how a System Designer should go about in implementing it.

- This model will serve very well if the Tasks Characteristics doesn’t fluctuate much. But as we saw in Chapter 2 for video applications, which have wide fluctuations in their processing requirements, a static bandwidth allocation may be unsuitable [40]. To handle these kind of fluctuations, adaptive schemes may need to analyzed [40].
3.3 Proportional Share Scheduling

The Proportional Share Scheduling (PSS) paradigm was initially conceived as a Network Flow Control Technique in which a switch forwards packets based on the Bandwidth Reservation made by the Data Stream [41]. Proportional Share Scheduling (PSS) is an interesting model as it treats both Time Sensitive and conventional desktop applications identically, yet provide provides temporal protection. The Central idea behind PSS is to partition the resources in such a way that all the applications in the system makes uniform progress as determined by their shares. The PSS task model associates each task with a ‘weight’ that determine the ‘share’ that particular task is supposed to receive. PSS is an approximation of Ideal Fluid Flow paradigm [16, 15, 4, 18].

3.3.1 Ideal Fluid Flow Paradigm

In an Ideal Fluid Flow model, irrespective of the length of interval we look at, a task with a fixed share, would always receive it, if there is no change in tasks in the system. If there N flows $l_1, l_2, \ldots, l_N$ and each of them are associated with weights $w_1, w_2, \ldots, w_N$, then total capacity can be given by the following relation:

$$C = \sum_{i=1}^{N} w_i$$ (3.4)
Share of flow \( l_i \) at time 't' can be expressed as:

\[
SH_i(t) = \frac{w_i}{\sum_{j=1}^{N} w_j} \quad \forall l_1, l_2 \ldots l_N
\] (3.5)

In the time interval say \( t_1 \) to \( t_2 \), if the share of the flow \( l_i \) remains constant then the Service time for that flow can be estimated as:

\[
S_i(t_1, t_2) = w_i \times (t_2 - t_1)
\] (3.6)

If the share is varying in the time interval \((t_1, t_2)\), then the Service time for the flow can estimated as follows:

\[
S_i(t_1, t_2) = \int_{t_1}^{t_2} SH_i(t) dt \quad \forall l_1, l_2 \ldots l_N
\] (3.7)

The equation 3.7 represents Ideal Fluid Flow paradigm, as the flow \( l_i \) is receiving a share that is exactly in proportion to its weight.

### 3.3.2 Emulation of Fluid Flow Model

Most of the uni-processor systems allow only one thread of execution to be using the CPU at a time. As discussed in Chapter 1, these systems emulate multitasking by switching CPU among tasks in fixed size ‘quanta’. To implement Ideal Fluid flow paradigm in uni-processor system, so that we can share the resources in proportion to weights of various tasks, the quanta needs to be very small \((q \rightarrow 0)\) [4, 15]. This is practically impossible because to allocate resources in infinitely small quanta, the scheduler has to pick a task all the time. The time attributed to scheduler
picking up a task for execution is an overhead in the Operating System. If the system is spending all its time in picking a task, the utility of the system degrades.

For this reason, the resources are allocated in discrete sized quanta 'q'. This will result in a difference between the Service time received by the flow $l_i$ in the ideal case and the actual Service time received by the flow in emulated case. This difference is referred to as 'lag' [15]. For a flow $l_i$ which becomes active at time $t_i^0$, the lag can can be estimated as:

$$
lag_i(t) = S_i(t_i^0, t) - s_i(t_i^0, t)
$$

(3.8)

A Positive lag would indicate that the task is getting more share than it requested and similarly a Negative lag would indicate that the task has received less share than it has requested. To emulate the Ideal Fluid flow model closely, we will have to keep the 'lag' as small as possible.

3.3.3 Terminology

Some of the terms that we will be frequently using are explained below. We adopt the same terminology as in [15, 4]

- request - is a unit of processing time requested by the client

- client - any task competing for the resource

- weight - determines the share a task is supposed to receive
• active client - a task that has a pending request

• passive client - a task whose request is satisfied.

3.3.4 The Concept of Virtual Time

Virtual clock algorithm was developed to provide rate reservation network data flows. [42, 41]. It emulates the behavior of Time Division Multiplexing [42], in which multiple bit-streams share the physical communication link by taking turns to use it. It gives an illusion to the stream as if that they have the link for the entire duration of time they reserved. Similarly a task that requests a fixed share of resource is given an illusion that it is running on dedicated Virtual resource for the duration of its request [43]. Virtual clock based algorithms are work conserving, (i.e) as long as there are tasks in the system, at-least one of them would be selected for execution.

Most of the Virtual Clock based PSS algorithms track the progress made by the tasks by time-stamping them in virtual time domain and tracking their virtual time. And thus Virtual clock based algorithms abstracts the Ideal fluid flow model [43]. Virtual Time flows at the rate inversely proportional to the set of active clients in the system. If $A(t)$ is set of active clients at time $t$, then

$$V(t) = \int_0^t \frac{1}{\sum_{j \in A(t)} w_j} dt$$

(3.9)
The above equation also suggests that virtual time increases at a faster rate when the resource is less contended. When there are many tasks contending the resource, virtual time is said to slow down \cite{15}. It is also true that virtual time cannot be lower than the virtual time-stamp on the task with lowest share.

For a set of active clients $l_1, l_2, \ldots l_N$ with weights $w_1, w_2, \ldots w_N$, the service time for client $l_i$ from 3.7 can be expressed as follows:

$$S_i(t_1, t_2) = \int_{t_1}^{t_2} \frac{w_i}{\sum_{j \in A(t)}}$$  \hfill (3.10)

We can express the relation 3.10 in terms of virtual time 3.9 as follows:

$$S_i(t_1, t_2) = w_i \times (V(t_2) - V(t_1))$$ \hfill (3.11)

3.3.5 Earliest Eligible Virtual Deadline First - EEVDF

With the background information on Proportional Shared Scheduling, we can look at how EEVDF works, its capabilities and why it is of interests to us in this section. The Description in this section is summarized based on original work by authors of \cite{4, 15}. EEVDF is virtual clock based Proportional Shared scheduling algorithm. The Task Model of EEVDF algorithm is such that every client is associated with weight, which determines the share the client is supposed to receive. A task that requires the resource issues a request specifying the duration of processing time request size. If it demands a resource without a request, the scheduler can issue a request on
behalf of the task with default request size. The algorithm is flexible in allowing tasks to specify request size. The Tasks can generate fewer requests but each demanding longer durations, or it can generate many requests demanding shorter duration of time [4] The Resource is allocated by the scheduler in fixed size quanta. The Task can use the allocated quantum either in full or it may complete before the end of quantum.

Description of Virtual Time-stamps

EEVDF is an event driven algorithm in which events such as arrival of a new task, task exiting after completion influence the Scheduling decisions. The task’s entry or exit changes the summation of weights of tasks in the system, which in-turn impacts Virtual Time as show in Equation (3.9). On entering the competition for the resource, it is tagged with Virtual Eligible time \( V_e \) and Virtual Deadline \( V_d \). Both these time-stamps are on Virtual time scale.

If \( t_0^i \) is the time the task becomes active and \( t \) is the time when it issues a request, we can evaluate \( V_e \) as follows:

Substituting \((t_0^i, t)\) for \((t_1, t_2)\) in 3.11, we have

\[
s_i(t_0^i, t) = w_i \times (V(t) - V(t_0^i))
\]

\[
V_e = V(t_0^i) + \frac{s_i(t_0^i, t)}{w_i}
\]

(3.12)
Similarly, if \( r \) is the processing time requested, then virtual deadline can be evaluated as follows:

\[
V_d = V_e + \frac{r}{w_i}
\] (3.13)

The recurrence relation for calculating \( V_e \) and \( V_d \) for request \( k \) of client \( i \) can be computed using the following equations:

\[
V_e^0 = V(t_i^0)
\] (3.14)

\[
V_d^k = V_e^k + \frac{r^k}{w_i}
\] (3.15)

\[
V_e^{k+1} = V_d^k
\] (3.16)

**How Does EEVDF Work?**

With the understanding of Virtual Eligible time and Virtual Deadline, the way EEVDF works is simple and straightforward. In the system, the fields *Current Virtual Time* and summation of weights of all the tasks *Total Weight* are globally tracked. As the name implies, at the beginning of every quantum, the Scheduler filters from among a group of tasks, all the task that have *Virtual Eligible* time less than the current Virtual Time. These are the set of eligible tasks. From among the eligible tasks the task with earliest Virtual Deadline is chosen for execution for that quantum.

It can be better understood by considering the following simple example. There are two tasks T1 and T2 as shown the in the figure 3.1. Both of them issues
request for 1 unit of processing time at the same time, until they are done with computation. Task T2 exits after executing 2 quantum. We can observe that when both the tasks are active in the system, 1 Virtual Time unit = 3 Real Time Units. Once the Task T2 exits, the Virtual Time scale increases at a faster rate. The Virtual Eligible time and Virtual Deadlines are computed using 3.14 and are show in table 3.1.

Impact of Dynamic Operations on Virtual Time

In a General Purpose Operating System, tasks may change states frequently [For Example: from being in RUNNING state (currently executing) to WAIT (waiting
Table 3.1: $V_e$ and $V_d$ Computations for Sample Tasks T1 and T2

<table>
<thead>
<tr>
<th>Time</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>$V^0_e = 0$</td>
<td>$V^1_e = 0.3$</td>
<td>$V^2_e = 0.7$</td>
<td>$V^3_e = 0.7$</td>
<td>$V^3_e = 1.3$</td>
<td>$V^3_e = 1.3$</td>
<td>$V^3_e = 1.3$</td>
</tr>
<tr>
<td></td>
<td>$V^0_d = 0.5$</td>
<td>$V^1_d = 0.8$</td>
<td>$V^2_d = 1.2$</td>
<td>$V^3_d = 1.2$</td>
<td>$V^3_d = 1.8$</td>
<td>$V^3_d = 1.8$</td>
<td>$V^4_d = 1.8$</td>
</tr>
<tr>
<td>T2</td>
<td>$V^0_e = 0$</td>
<td>$V^0_e = 0$</td>
<td>$V^0_e = 0$</td>
<td>$V^1_e = 1$</td>
<td>$V^1_e = 1$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$V^0_d = 1$</td>
<td>$V^0_d = 1$</td>
<td>$V^0_d = 1$</td>
<td>$V^1_d = 2$</td>
<td>$V^1_d = 2$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Schedule</td>
<td>T1</td>
<td>T1</td>
<td>T2</td>
<td>T1</td>
<td>T1</td>
<td>T2</td>
<td>T1</td>
</tr>
</tbody>
</table>

for data from slower devices) etc.]. In an Ideal Fluid flow model, these dynamic operations can happen seamlessly as the resource is allocated in small quanta. But when the resources are allocated in discrete sized quanta as in EEVDF, we will have to consider the lag or Service Allocation Error as discussed in Section 3.3.2. The lag of each task has a direct impact on the Virtual Time. This can be explained by recalling the fact that, if task $i$ has acquired a negative lag, it means that it has received more service than it was entitled. It also means that other tasks in the system would have received less service. Stoica et.al [4, 15] rightly observe that ”gain for one client translates in to loss for other clients”. They suggest this loss/gain be distributed among other tasks in proportion to their weights.

The real beauty of their suggestion is in the way the distribution of loss or gain is carried out. It is done by recalculating the Virtual Time, which eliminates the need for estimating the lag for each client in the system. An extensive proof of Fairness in EEVDF is available in [4, 15]. When a client $i$ leaves the system at time $t$, Virtual Time would be recalculated as follows:
\[ V(t) = V(t) + \frac{\text{lag}_i(t)}{\sum_{j\in A(t)} w_j} \]  

Similarly when a client joins the system, Virtual Time will be recalculated as follows:

\[ V(t) = V(t) - \frac{\text{lag}_i(t)}{\sum_{j\in A(t)} w_j} \]  

### 3.3.6 Other Variations of PSS

There are several variations of Proportional Share Scheduling (PSS) proposed in the literature. Some of the PSS flavors are briefly described in this section:

- **Weighted Fair Queuing (WFQ)** - WFQ is a packet service discipline in packet-switched Networks \([41, 44]\). A Link at the Switch has multiple FIFO queues each corresponding to multiple Data Streams. One way to service packets in FIFO queues would be to process packets at the head of the queues simultaneously. Generalized Processor Sharing (GPS) \([45]\) is another way, in which each data stream is associated with service rate. Packets in the FIFO queues are serviced based on the rate associated with the stream. Since GPS describes the ideal fluid flow model, WFQ was proposed an approximation of Generalized Processor Sharing \([46, 45, 44]\). It works by ordering packets based on when they would be served in ideal system or Virtual Finish Time \([4, 44]\). In WFQ, the \text{lag} is bounded for systems where all clients are active all the time.
But in a dynamic system with clients leaving/entering continuously, fairness is not maintained [47].

- **Stride Scheduling** - This is the First PSS variation which applied Flow Control Techniques to allocate processor time among multiple processes [41]. In Stride Scheduling, each client is associated with:
  
  - 'tickets', which is similar to 'weight' in other PSS algorithms. It determines the amount of share the client is supposed to receive.
  
  - 'stride', which is similar to 'virtual time'

  - 'pass', similar to 'virtual finish time'.

This Algorithm works by selecting a client with lowest 'pass' in each quantum and advances the 'pass' for the client by a 'stride' [41]. It is similar to EEVDF in many aspects except the bounds for 'lag'. As the authors of [4] observe, even with hierarchical implementation of Stride Scheduling as suggested in [41], the lag bound is in the order of $O(n_c)$ where $n_c$ is the number of active clients in the system.

- **Start Time Fair Queuing (SFQ)** - It is a Virtual Time based PSS algorithm but has a unique approach where in the applications are grouped together in various classes and their allocations are handled by respective Schedulers. Each Scheduler in turn receive a fraction of Total CPU Bandwidth [48]. It has been
observed that SFQ provides an optimal upper bound for \( lag \), but fails to provide optimal bound for absolute \( lag \) \([47]\).

- **Scheduling for Multimedia and Real Time (SMART)** - SMART \([18, 7]\)
  is another unique PSS algorithm which not only aims to achieve fairness in Processor allocation for Conventional applications, but also it tries to meet as many deadlines of Real Time applications. Each Client is associated with a *share* and *priority*. As in other PSS algorithms, the *share* determines the virtual finish time. This Virtual Finish Time together with *priority* of the task constitutes *value tuple*. A Task with Higher Priority have have Higher Value Tuple. At the same priority level, a task with lower virtual finish time would have higher value tuple. Tasks are sorted based on their Value tuple as they are created. The Algorithm works by choosing a task with highest value tuple, if it is a conventional task. If it is a Real time task, a set of real time tasks value tuple greater than highest ranked conventional task is formed. A Schedule feasibility test is performed to verify if allowing this Real time task will cause failure to meet the guarantee to other tasks. A Task is chosen using Earliest Deadline First algorithm from this set for execution. \([18, 43]\).
Chapter 4

ASI - Enhancements to EEVDF Based PSS

The unsuitability of priority based CPU Scheduling in GPOS in the presence of mix of applications was described in chapter 1. This was also substantiated by a real application, failing to provide allocation guarantees in chapter 2. We described various solutions proposed in the literature in chapter 3. In this chapter, we revisit the Earliest Eligible Virtual Deadline First (EEVDF) based Proportional Share Scheduling (PSS), in the context of CPU scheduling in General Purpose Operating Systems (GPOS). We identify some issues with EEVDF based PSS approach in CPU Scheduling with mix of time sensitive and conventional applications. We also discuss the design of Abstract Scheduling Interface (ASI), which addresses the issues identified with EEVDF.

4.1 Design Goals

Based on analysis in the previous chapters, We listed out the requirements of a CPU Scheduler that would enable us address the problems related to scheduling in GPOS in Section 1.3:
• CPU scheduler should be able to handle mix of time sensitive and conventional
desktop applications without any preferential treatment for each class of applica-
tions from the scheduler. It means that the tasks would need to be treated
identically.

• It should also be able to provide allocation guarantees to time sensitive applica-
tions. If the system is not overloaded, it would guarantee resources to time
sensitive applications and at the same time best possible throughput to conven-
tional applications.

• If the system is overloaded, it would allow graceful degradation of QoS for
applications. This means it wouldn’t allow applications to monopolize, while
rest of them starve.

4.2 Why EEVDF Based PSS Solution?

From here on, Earliest Eligible Virtual Deadline First (EEVDF) based PSS is
referred as the representative example for PSS. Among the various solutions discussed
in the previous chapter on scheduling improvements, Proportional Share Scheduling
approach offers a lot of promising features that will help realize design goals mentioned
in 4.1:

• In PSS all tasks are treated identically [15]. Unlike GPOS, there is no preferen-
tial treatment of Tasks. That is, tasks are not favored based on whether they
are interactive, time Sensitive or batch applications. Each task in the system receives a share of the resource in proportion to their weights. [15, 16, 4, 17, 18, 5]. Thus they exemplify the notion of fairness.

- In PSS, resource is allocated in fixed size quantum. Implementing PSS in a GPOS can take advantage of existing infrastructure such as Periodic Timer Interrupt [49].

- In general, proportional share based resource allocation algorithms provide strong allocation guarantees. In PSS, over any given time interval, the allocation error or the lag is bounded [15]. Further the allocation error does not exceed one time quantum in EEVDF based PSS.

4.3 Issues with EEVDF

There are two issues with using EEVDF based Proportional Share Scheduling approach, that needs to be addressed, to get EEVDF based PSS to work in realizing our design goals described in Section 4.1. They are described in this section:

- Allocation uncertainties - There are applications, which may require constant Share to deliver an acceptable level of QoS. EEVDF based PSS approach cannot assure a constant minimum Share to such applications by itself. There is an inherent uncertainty in the allocation as explained in the Section 4.3.1.
• **Weight Assignment** - There is a need to formulate a strategy on how an application’s resource requirements can be translated into weights, which is discussed in Section 4.3.2.

### 4.3.1 Allocation Uncertainties

To understand the reason behind Allocation Uncertainties in EEVDF based PSS approach, we can recall the description of EEVDF algorithm in Section 3.3.5. In EEVDF, each Client gets a share of the resource in proportion to its weight. The share of a client is indeed dependent on the weights of other clients contending for the resource [50]. For example, the share \( f \) for a client with weight \( w \), in the presence of other clients is given by:

\[
f = \frac{w}{W},
\]

where, \( W \) is summation of weights of all clients in the system.

Thus, when the resource is contended by many clients, the share received by the client shrinks and when the contention for the resource is less, the client receives a bigger share. In a dynamic system, clients may enter or leave competition for the resource at any given point of time. These dynamic operations would result in fluctuations in share of the resource received by the client. These fluctuations are referred to as *Allocation Uncertainties*.

Time sensitive applications may not be able to tolerate these allocation uncertainties. They have some minimum resource requirements, which may need to
be guaranteed to deliver acceptable performance. As observed by authors of [19] in audio-playback application, if the application is assured a minimum share of CPU, for the period it plays an audio stream, they achieve acceptable quality. Wide fluctuations in the share received by this application may cause it fail to meet the temporal constraint and may yield low quality audio.

Some applications may request a "Constant" share to deliver a specified QoS. Further, if excess resources are available after satisfying "Constant" share requested by all tasks in the system, some applications may benefit by making use of spare resources.

There are applications, that may require a fixed share and may not benefit from making use of spare resources. An example of this class of application would be "Virtual Machine OS" running on "Virtual Machine Server". The user of such a system may have determined by profiling, that any allocation more than requested fixed share may not be useful to it. For example, "Virtual Machine OS" may waste the excess Resource allocated to it by running IDLE Task.

### 4.3.2 Weight Assignment

The issue of weight assignment is very critical and fundamental to EEVDF based PSS approach. It is complicated by the fact, that applications have conflicting resource requirements. On the one hand there are time sensitive applications that re-
quire guaranteed minimum shares and other hand there are conventional applications (Interactive, Batch) applications that require fair resource allocation.

Stoica et.al [19] attempted to address both the issues by considering ‘weights’ and ‘shares’ as duals of each other. In their model, each task is characterized by two components a ‘weight’ and a ‘share’. Conventional applications are associated with fixed weights and Time sensitive applications are associated with fixed shares. To maintain fixed shares for Time sensitive applications, in the event of dynamic operations such as applications getting activated/deactivated, they proposed recalculating weights of applications that needs a constant share. While this approach works, it does not offer provision for application to request a fixed share and still make use of spare resources available in the System.

4.4 Abstract Scheduling Interface

4.4.1 An Overview

Abstract Scheduling Interface (ASI) is designed to address the issues: allocation uncertainties, weight assignment discussed in the previous section. ASI provides an encapsulation for Proportional Share Scheduler. That is, without modifying underlying PSS functionality, it provides enhancements to address the afore mentioned issues. An architectural overview is provided in the figure 4.1.

ASI monitors the resource usage by all the clients, which use ASI Application Programming Interface (API). ASI recomputes weight for all clients, in the event of
dynamic operations and counter allocation uncertainties. It also provides a strategy to assign weights to the tasks.

4.4.2 How Does it Work?

In this section, we explain the user interface and weight estimation strategy used by ASI. This would also explain how ASI eliminates allocation uncertainties.
User Interface

ASI provides a user interface to capture such a diverse application properties.

Each user application/client in ASI is characterized by a 'tuple' consisting of two parameters:

- **Constant Share** - specifies the minimum amount of resource required to deliver a 'acceptable' QoS.

- **Benefit** - an integral value, which specifies how the excess resources can be allocated to the task. Or in other words, this parameter decides, if the task could benefit from making use of excess resources available, after satisfying the constant shares requested by all tasks in the system.

An application that may not benefit from making use of spare resource, may specify 'Benefit' value as zero. This would ensure, that such an application receives a "Constant Share" of resource and no more. 'Benefit' is an abstract term and the implementation is free to interpret it to be a parameter that would make better sense to the user. For example, an ASI implementation can treat 'Benefit' to be 'Static Weight' associated with the task. Such an implementation can allocate the spare resources proportionally, after satisfying "Constant Share" requested by other tasks. Our interpretation of 'Benefit' follows above example.
**Weight Estimation**

ASI monitors the resource usage of all the tasks using ASI API. Based on the resource requirements for each client and current resource usage, ASI computes a ‘Dynamic Share’. For a client $C_i$, with the resource requirement 'tuple' specified as $(S_i, B_i)$

$$D_i = S_i + IDLE_{capacity} \times \frac{B_i}{\sum_{j \in A(t)} B_j}$$

where,

$D_i$ = Dynamic share for client $C_i$,

$S_i$ = Constant share requested by the client $C_i$,

$B_i$ = Benefit associated with client $C_i$,

$A(t)$ = Set of active clients that are contending for the resource.

$IDLE_{capacity}$ = Amount of spare resources that can be allocated after satisfying Constant Share requested by all Tasks.

The "Share" (Dynamic and Constant) values in the above computation are percentile values. ASI translates these "Shares" in to "Weight". Weights computed for ASI tasks fall in a fixed range 1 to $W_m$, where $W_m$ is maximum possible weight that can be assigned to a task. The Range of weights used in this computation is implementation specific. Weight estimation in under-loaded scenario is straight
forward and it is described by the following Equation.

\[ w_i = D_i \times W_m \]  

(4.2)

where,

- \( w_i \) = Computed weight for client \( C_i \),
- \( D_i \) = Dynamic share for client \( C_i \) (percentile value),
- \( W_m \) = Maximum possible weight as chosen by the implementation

Weight assignment for client is carried out in the event of dynamic operations such as a client entering/leaving the competition for the resource in the System. Once the 'weight' is estimated based on "Dynamic Share", ASI makes use of the PSS API to create a PSS task with newly computed 'weight'.

A Sample ASI Computation

Let us consider a task \( T_1 \). The 'tuple' values of \( T_1 \) is specified as (10, 2). In this Example, \( T_1 \) is requesting a "Constant Share" of 10%. The non-zero 'Benefit' field indicates that this task can benefit from extra resource. If this is the only task in the system, the "Dynamic Share" as computed by ASI would be 100% based on 4.1.

Let us consider another task \( T_2 \) is active. Let the 'tuple' value for \( T_2 \) be specified as (20, 3). ASI recomputes the "Dynamic Share" that will be received by these tasks to 38% and 62% respectively. The computed "Dynamic Share" is translated to "Weight" using 4.2.
Consuming Resource With ASI IDLE

There is a tricky Scenario, in which all the tasks in the system may request "Constant Share" with their 'Benefit' parameter set to zero. In such a scenario, there may be spare resources left unallocated to any task in the System. This is handled by 'ASI IDLE'.

'ASI IDLE' is a pseudo-PSS task, similar to 'Idle' task in a system. An idle task spins and consumes CPU slices, when there is no other task to execute in a system. In a similar way, ASI IDLE consumes the remainder of the resource, that is available after being allocated to satisfy the "Constant Share" requirement of all
tasks in the system. 'ASI IDLE’ task can be in various states: INIT, INSERTED, IDLE_RUNNING. The state transitions involved in ASI IDLE task are captured in the figure 4.2.

The Behavior of 'ASI IDLE’ can be Implementation specific. For Example, if the PSS Implementation using ASI, is co-existing with another Scheduler, then it is possible to allocate the spare Resources to the Task chosen by the Secondary Scheduler. In Our Implementation of ASI, PSS is the primary Scheduler. When all the Tasks in the System are requesting a Constant Share without benefits, PSS Scheduler chooses to introduce 'ASI IDLE’ in the System. Whenever, 'ASI IDLE’ is chosen for execution, Our Implementation chooses the Task selected by regular Linux Scheduler for execution. This is explained in Section 5.3.5.

Handling Overload

An overload happens, when the resource is heavily contended. In overload conditions, it may no longer be possible to guarantee the "Constant Share" as requested by Tasks. ASI handles overload by using the 'Benefit’ parameter in the 'tuple’ as 'weight’ for the task and schedules them as PSS tasks. Thus, ASI is able to achieve the graceful degradation in overload scenarios.
CHAPTER 5

Our Prototype Implementation

We describe our Prototype implementation of EEVDF based Proportional Share Scheduler and Abstract Scheduling Interface (ASI) in this Chapter. We chose Linux as the developmental Platform as it is one of the most popular Open Source General Purpose Operating Systems.

5.1 Dissecting Linux 2.6 Scheduler

To understand our implementation, it is critical to gain some Perspective on various building blocks or Subsystems of the Linux Kernel Scheduler. These are basics relevant to understanding any Operating System. Our intentions are to provide the reader with enough information to understand our modifications. We provide a high level overview of relevant concepts in this following sections, and more detailed explanations are available in [9, 3, 2] In this section we dig in to Linux Kernel 2.6.18, which was the most up-to date version at the time of Development of this prototype. Kernel 2.6.18 is Symmetric Multiprocessing capable (i.e. It has markers in the code that will be compiled only on Symmetric Multiprocessing Systems). In our current prototype implementation, we concentrate exclusively on Uni-processor implementation.
5.1.1 Process - States Inside Kernel

The Fundamental Schedulable entity in the Operating System is a Process. Each Program that a user invokes is a bunch of instructions. It is referred as process, when it is being executed by the Processor. In a Uni-processor Multitasking System, Each Process maintains a set of exclusive resources. They include Program Counter, Stack, Processor Registers, an Address space etc. The Data structure which contains this information about a process is referred to as Process Descriptor or task_struct in Linux. Each Process that is created, is associated with a task_struct and they are maintained in a Circular Doubly Linked List [9]. A Process can be in any of the following states: TASK_RUNNING, TASK_INTERRUPTIBLE, TASK_UNINTERRUPTIBLE, TASK_ZOMBIE, TASK_STOPPED [9]. The State transitions can be visualized as shown in the following Figure 5.1

In traditional Unix, a process is created by fork system call, in which a new process descriptor is created for Child process. Except for few values that are not inherited by the Child from Parent process ¹, an identical copy of Parent’s process descriptor is made ². This followed by exec system call, which would load the program binary the Child is supposed to execute. At this stage, the Child process would be in TASK_RUNNING state. In Linux, the parent process creates a child using vfork or clone system calls. This implementation of fork helps Linux in achieving "Copy

---

¹such as Process ID, Parent Process ID, Pending Signals etc. [9].
²Parent and Child process executes same piece of code until a new image is loaded in to the child’s address space.
on Write”. The Salient feature of "Copy on Write" is that the costly duplication of Parent’s image in to the Child is deferred by letting Parent and Child share a single address space, until either of the processes first to tries to write in to its Address space. The Child Process is in TASK_RUNNING state and it waits in the runqueue. The Scheduler picks one of the task from the runqueue for execution dictated by the Scheduling Policy. When the Process is running, it may have to wait for certain in events, in which it will be put to one of the sleep states TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE.
5.1.2 Linux Scheduling Policy

The Scheduling Policy is the basis of the Kernel Scheduler as it essentially decides which Process should execute next. Linux supports four Scheduling Policies: two of them are POSIX compliant Real Time Scheduling (SCHED_FIFO \(^3\), SCHED_RR \(^4\)), third is to handle regular Non-Real Time processes (SCHED_OTHER) and fourth is to handle batch mode processes (SCHED_BATCH). Linux is Preemptive Priority based Multitasking System. There are 140 Priority levels implemented in two ranges. Real Time Priorities range from 0 to 99. The Priority levels 100 to 139 are used for Non-Real Time tasks in the system. Multiple Tasks with same priorities are scheduled in Round Robin order. With the Priority levels described as above, the Scheduling policy hierarchy looks like:

\[
SCHED\_FIFO > SCHED\_RR > SCHED\_OTHER > SCHED\_BATCH
\]

5.1.3 Calculating Priorities

The Priority of process is determined of 2 components: a \textit{niceness} or static priority \(^5\) (ranging from -20 to +19), and a \textit{Dynamic Priority}. The \textit{Dynamic Priority} is computed using \textit{niceness} value and \textit{Interactivity measure}. The \textit{Interactivity Measure} is estimated based on the heuristic of whether the process is Input/Output bound process or CPU bound process. A Process that spends more

\(^3\)Real Time First in First Out algorithm in which tasks execute from Start to Finish without preemption.

\(^4\)Real Time Round Robin algorithm in which tasks execute for fixed time quantum and yields to other tasks.

\(^5\)-20 corresponds to highest priority and +19 corresponding to lowest priority.
time in TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE state waiting for events is an I/O bound process. An example for an I/O bound process is a process that is trying to write data in to the Hard Disk Drive. The Disk Driver may put this process to sleep (process will be in TASK_INTERRUPTIBLE State) until data is written to the disk. This Disk Driver may wake up the process once the data is written. Just like other Unix flavors, Linux tries to reduce the latency to I/O bound process and maximize the throughput by providing *bonus* to I/O bound process. The nice levels -20 to 19 are mapped on to Priority range 100 to 139.

Figure 5.2: A Figure Showing Runqueue layout
5.1.4 Choosing the Highest Priority Process

Linux implementation of Multilevel Feedback Queue is very efficient and robust because of the Core Scheduling Data Structure \textit{runqueue}. It is defined as a Per-Processor Data Structure \footnote{In SMP machine, there is one run queue per processor.}. It is highly critical that Scheduler doesn’t spend too much time in finding the task the with Highest priority in the \textit{runqueue} as this is sheer overhead. Linux Versions till 2.5 didn’t scale well, as the Algorithmic Complexity in picking the task with Highest Priority task was \(O(n)\)\footnote{There are in total 140 bit fields as part of bitmap.}. With the current implementation, the Algorithmic Complexity of choosing highest priority task has improved to \(O(1)\). The reason for the improvement is because of intuitive design of \textit{Priority Arrays}, which forms a critical element of \textit{runqueue} data structure. The Priority Array is an array of Linked Lists. Each Priority Array also has a \textit{bitmap}. The \textit{bitmap} has a bit field corresponding to each Priority \footnote{There are in total 140 bit fields as part of bitmap.}. A simple visualization of \textit{runqueue} and \textit{Priority Array} is shown in the Figure 5.2. Whenever a task is created through \textit{fork} it is appended to the linked list at the computed priority level in \textit{active} Priority Array, and the the corresponding bit field in the bitmap is set. So finding the task with Highest Priority is as simple as finding the First bit set in the \textit{bitmap} and hence this is a constant time operation. Each task in the \textit{runqueue} is allotted a Time Slice which is scaled based on priority. Once the Task has finished using its allotted Time Slice it is moved in to the \textit{expired} priority array. When there are no
more tasks left in active priority array, the scheduler swaps the references to active and expired priority array.

5.1.5 Kernel’s Notion of Time

Kernel’s notion of time is provided by System Timer. The System Timer [9] is a Programmable hardware Interrupt generator, which is pre-programmed at the boot time with static value called as Timer Rate. The Timer Rate specifies the frequency at which the interrupt occurs. For Example, if the Timer Rate is specified as 1000HZ, then Timer Interrupt occurs every 1ms.

5.1.6 An Overview of Scheduler Related Functions

We summarize the purpose of core Scheduler Related Functions and an in depth analysis (available in [3]) is beyond the scope of this Thesis.

- The Function scheduler_tick() is responsible for updating time slices for each task. It is called every timer tick and it executes as part of handling Timer Interrupt. When a Task uses all its allotted time slices, this function sets a flag (TIF_NEED_RESCHED) to ensure it is rescheduled.

- The Core Scheduling Logic is implemented in schedule() and it is responsible for picking the highest priority task from the runqueue for Context Switch 8. It can be invoked in 2 ways:

---

8Context Switch is a term used to refer to the step of Saving the Current state of process so it can be resumed from exact point where it was preempted and loading a new process.
- The function `scheduler_tick()` sets the flag TIF_NEED_RESCHED which invokes `schedule()` immediately after completion of `scheduler_tick`.

- When the process has to wait for events, it may be put to sleep. Typically the section that is managing the event (For example, Disk Driver) will change the state of the task to TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE and invoke `schedule()`

  Inside `schedule()`, if any process is in one of the sleep states and found in `runqueue`, it is removed from the `runqueue` by the function `deactivate_task()`, so that it is not scheduled while it is waiting for data.

- When the process is woken up from sleep state by occurrence of the event, the code managing the event would invoke `try_to_wakeup()`. This routine would flip the state of task to TASK_RUNNING. It also invokes `activate_task()`, which is responsible for adding the task to the `runqueue`.

  All the Scheduling related functions runs in Privileged Mode \(^9\). The Applications that a regular user invokes runs in User Mode and they can \(^10\) can request services from kernel, which in turn executes Kernel functions through System Call Interface. The Linux Kernel implements several Scheduler related System calls that can be used to change/query Task’s priorities(`nice()`), `sched_setparam()`,

\(^9\)Privileged Mode - A Mode in which Core Kernel and Kernel data structures are maintained in protected address space, so that user space programs do not have direct access. This is usually done for the sake of Stability of the system so that no other User Space Program can corrupt the System Executive.

\(^10\)User Mode - In User mode, the applications cannot access or invoke Kernel functions or Kernel related Data Structures directly.
sched_getparam()), Scheduling Policy (sched_setscheduler(), sched_getscheduler()), etc.

5.2 Our Implementation of EEVDF in Linux 2.6.18

We discussed the Theory on Earliest Eligible Virtual Deadline First (EEVDF) based Proportional Share Scheduling (PSS) algorithm in Chapter 3, Section 3.3.5. Summarizing our discussion tersely, in EEVDF based Proportional Sharing, each task is associated with a weight that determines the share it is supposed to receive. Events like Task Entry, Task Exit, Task Blocking (sleeping/waiting for events), Task Activation (waking up from sleep/getting triggered on occurrence of events) affect the Virtual Time. When a Task enters the system, it issues a request to the scheduler. The Scheduler updates the Virtual time-stamps maintained for the requests and adds this request in Sorted Structure. At the beginning of the quanta, the scheduler selects a group of tasks from among all the tasks whose virtual eligible time is less than the current Virtual Time to form Eligible pool. Then it selects a task which has the Earliest Virtual Deadline from the tasks in the Eligible pool. We had implemented EEVDF based PSS for Linux Kernel 2.6. ¹¹

¹¹Stoica et.al [4], who proposed EEVDF had implemented a prototype version on Free-BSD Unix. To the best of our knowledge, there is no publicly available implementation of EEVDF for any platform.
5.2.1 Implementation Strategies

When we were looking into implementation strategies that could be adopted, we had the following choices:

- Export all Scheduling events like Task Creation, Task Deletion, Blocking and unblocking to a Loadable Kernel Module. This approach would be a clean implementation making minimal changes to actual Kernel. This approach has been taken by several prototype implementations [37, 40].

- Modify the actual Kernel Scheduler Source Code.

Though the Strategy 1 would result in a easily maintainable implementation, We followed the Strategy 2. In the regular Linux Kernel, most of the Scheduler Data Structures and Functions that manipulate them are not exposed out of the Scheduler Source file. This was done for the sake of preventing any accidental corruption of Scheduler related data and the consequences of which would be very bad. Further, once the implementation is stable, it would be easier to seek feedback from Kernel Development Community with this approach.

Addition of a New Scheduling Policy

In our implementation, we introduced a new Scheduling policy SCHED_PSS in the Linux kernel. Our intentions were to make minimal modifications to the Kernel and retain the functionality of other existing Scheduling policies in the kernel as much
as possible. With this modification, the Scheduling Policy Hierarchy would be:

\[ SCHED\_FIFO > SCHED\_RR > SCHED\_PSS > SCHED\_OTHER \]

This Hierarchy means that if there is any Task in the System with Real Time Scheduling policies (SCHED\_FIFO or SCHED\_RR) the scheduler will choose them for execution. If there are no tasks with Real Time Scheduling policies, PSS tasks will be scheduled based on EEVDF algorithm. Regular tasks are scheduled only when there is no Task with Real Time Scheduling policy/PSS \(^{12}\).

5.2.2 Data Structures

In this section, we list the Data Structures that were introduced or modified for the implementation of EEVDF.

- **EEVDF Request** - This is the fundamental Data Structure that describes and holds EEVDF Scheduling information. The Scheduling information held in this Data Structure struct pssrequest includes: weight, Virtual Eligible Time, Virtual Deadline etc. This Structure as defined in the implementation is shown in the Appendix A, Section A.1.1.

- **Request List** - PSS Requests are maintained in a Circular Doubly Linked List \(^{13}\). We made use of the List implementation in the Kernel. This is shown in Appendix A, Section A.1.2.

\(^{12}\)From here on, we will use the term PSS task to refer to a task scheduled based on EEVDF.

\(^{13}\)Before implementing EEVDF based PSS Scheduler in the kernel space, we developed a User Space application that emulated a regular Scheduler, but used EEVDF for scheduling threads. In this implementation, we used more efficient modified Binary Search Tree as suggested by authors of [4] to hold EEVDF requests.
• **Modified Process Control Block** - The Process Control Block of each task (struct task_struc) has a pointer to the EEVDF request. When a PSS Task is created, memory for EEVDF Request is dynamically allocated and initialized. For non-PSS tasks, this field is initialized to NULL in sched_init() 

• **PSS Queue** - It is similar to the runqueue data structure in the scheduler. All the EEVDF tasks are maintained in a Queue, implemented using List data structure as shown in Appendix A, Section A.1.3. 

5.2.3 How Does Our EEVDF Implementation Work?

In this section we will skim through the Code modifications made in the Scheduler that will provide enough insight into how our implementation works. With Scheduling hierarchy implemented as described in the section 5.2.1, when the kernel is booted up it functions just like any regular Linux 2.6 Kernel. The Virtual Time and Total weight of all PSS tasks are tracked globally and they are initialized during Scheduler Initialization at the time of system startup. 

EEVDF based Scheduling is triggered by occurrence of any of the 4 events mentioned below: PSS Task creation, PSS Task blocking, PSS Task Activation and PSS Task Exit. These events start a chain of actions in the Scheduler related functions scheduler_tick() and schedule(). In this Section, we will follow each event and
the series of actions that happen behind each event, which will explain our implementation:

**PSS Task Creation**

In our implementation, a PSS task is created, by forking of from a regular Task and invoking the system call wrapper function `sched_setscheduler()`, with the scheduling policy set to SCHED_PSS. The Priority field of `sched_param`, which is an argument to this system call is used to pass Weight for this PSS task. This System Call wrapper issues a software interrupt, which causes the switch in to the kernel mode and it starts executing `sys_sched_setscheduler()`. With the Scheduling policy set to be SCHED_PSS, an EEVDF request for this task is created and initialized by `create_request()`. After Request initialization, basic sanity checks like Verifying Correct Scheduling policy, the priority/weights supplied by user, are performed by `sched_setscheduler()` defined inside kernel. This Task is removed from the runqueue so that the Scheduling policy change can be effected by `_sched_setscheduler()`. PSS Tasks are added to the PSS Queue by `add_tocompetition()` and Virtual Time and Total Weight are updated. Virtual Time is computed as described by equation 3.9. Since floating point operations cannot be performed inside the kernel space we use fixed point representation of numbers. Fixed macro usage is explained in Appendix A.2. The Virtual Eligible time and Virtual Deadline are computed for this task using equations 3.14, 3.15 and 3.16 and these
values are updated in the request issued on behalf of this task (`insert_request()`).

The Implementation of `sched_setscheduler()` ends with this operation and it returns to the user space task that initiated the call. A Call graph of various function calls in the event of PSS Task Creation is shown in Figure 5.3.
PSS Task Blocking

Just like a regular task, a PSS Task may wait for occurrence of certain events. For Example, a Device Driver that receives request for data from PSS task, may put the task to sleep, by changing the state of the task to one of the blocked states (TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE. The Driver cannot manipulate runqueue, as by design only Scheduler has exclusive access to it. Hence it invokes schedule(). In the schedule() function, if a task in one of the blocked states is identified, deactivate_task() is invoked. It is the responsibility of deactivate_task(), to remove the blocked task from the runqueue and update the bitmap fields appropriately.

A PSS that is blocked waiting for events is treated as if it had left the competition for the resource. Hence for PSS tasks, leave_competition is invoked from deactivate_task(). The Total weight is decremented and Virtual Time is recomputed using equation 3.17. The Request corresponding to this task is deleted from the Request List by delete_request(). Though the Request is deleted from the list, the memory allocated for the PSS Request Data Structure, that was initialized at the time of creation of PSS Request is not released. This memory is released only when the PSS Task exits the system permanently.
PSS Task Activation

A PSS Task may be woken up by occurrence of the events on which they were waiting. The Driver code as in the PSS Blocking example will notify the occurrence of the event on which the process was waiting by invoking `try_to_wakeup()`. In a normal process, this routine tries to flip the state of the task to TASK_RUNNING and insert the task in to `runqueue` by invoking `activate_task()`. In the case of PSS task being woken up through `activate_task()`, it is equivalent inserting a new task in to the competition for the resource. Hence the routine `add_to_competition()` is invoked. Just as we observed in PSS Task creation, this routine does several things including recomputing Total Weight, Virtual Time, Virtual Eligible time and Virtual Deadline of the task. It also invokes the routine `insert_request()`, to re-insert the request Request List.

PSS Task Exit

Any Program which finishes all its computation, terminates by executing `exit` family of System calls. Either the user explicitly invokes `exit()` (a wrapper function to System Call `_exit()`, provided by the C library), or as the program terminates it may be invoked implicitly. By terminating this way, they release the resources allocated specifically for the task. Most of the work of `_exit()` system call is handled by the routine `do_exit()` [3]. The `do_exit()` routine begins by setting the state of the task to PF_EXITING (a preliminary state to becoming
TASK_ZOMBIE). It also un-links data structures in the Process Control Block of exiting task such as paged memory, files, namespaces etc through appropriate routines. (exit_mm(), _exit_files(), _exit_fs(), exit_namespace() etc). Finally the state of the process is changed to PF_DEAD and schedule() is invoked. Inside schedule(), when a current task is found in PF_DEAD state, the state is again changed to EXIT_DEAD so that the Process Control Block is released to the system completely. For the PSS task which is not going to contend for the resource any more, the routine leave_competition is invoked which in turn updates the Total Weight, Virtual Time and deletes this request from the Request List as observed in PSS Task Blocking event. This Request data structure for this PSS Task is freed and the Task entry is also deleted from the PSS queue. After this the schedule() goes ahead with its procedure to pick a task for execution.

**Action inside scheduler_tick()**

In this section, we revisit the scheduler function scheduler_tick() and schedule() to understand how they contribute to EEVDF implementation. Let us consider the case of a PSS Task, chosen for execution. We may also recall from the Kernel Notion of Time section 5.1.5, that the System Timer is programmed to generate a Timer Interrupt at Timer rate. The Scheduler function scheduler_tick() is executed as part of handling Timer Interrupt. Since EEVDF makes allocation of in fixed size quantum, the function scheduler_tick() tracks the number of ticks the
PSS task is executing by decrementing the 'pssslice' field in the Process Descriptor of the PSS task. If this field is non-zero, then PSS task hasn’t completed execution of full quantum. When this field becomes zero, the PSS Task has finished executing one full quantum. The field 'pssslice' is replenished to a default value\textsuperscript{14}. Virtual Time is updated by the routine \texttt{update\_virtual\_time()} as described in equation 3.9.

We view the task in execution as a continuous stream of PSS requests. So at the end of quantum, the request issued by PSS task is considered to have ended and a New Request is issued on behalf of the task. Since the current request has ended, we compute lag it has incurred (if any), delete this request from Request List. Virtual Eligible time and Virtual Deadline are recomputed for the new request and it is inserted in to Request List. This PSS Task is marked for reschedule.

**Action inside schedule()**

We may recall that the Core Scheduling routine \texttt{schedule()} plays a crucial role in taking actions for all occurrence of the PSS events. Whenever \texttt{schedule()} is invoked and currently executing task is a PSS task, it checks the state of task and take actions based on it:

- If the task is in PF\_DEAD state, then actions as discussed in Section 5.2.3 PSS Task exit are taken.

\textsuperscript{14}The Default value is the Fixed Quantum size of EEVDF. In our implementation, we had set it to 100ms.
• If the task is in one of the blocked states, the corresponding request is removed from Request List and actions as discussed in Section 5.2.3 are taken.

In both the cases mentioned above, it is responsibility of the scheduler to pick the most eligible task for execution. If there is a Real Time Task in the system, then it is the most eligible task in the system. If there are no Real Time tasks and there are few PSS Tasks and Regular tasks, then PSS tasks are scheduled based on EEVDF. The Regular Linux tasks are scheduled based Dynamic Priority, as in un-modified Linux Kernel when there are no Real Time or PSS tasks in the system.

This has been implemented by a regular invocation to `sched_find_first_bit()`, to get the bitmap of the priority level which has a valid Task ready for execution in the Active Priority Array. If the highest priority level falls in the Real Time Priority range (0-99), then the corresponding Real Time task is chosen for execution. Otherwise, `fetch_eligiblepss` is invoked. This routine skims through the list of request and identifies all the requests that have Virtual Eligible time less than or equal to the current Virtual Time. From among them, the request whose Virtual Deadline is the smallest is chosen and owner of the corresponding request is chosen for execution. If there are no Real Time/PSS tasks, the scheduler chooses the normal Linux tasks for execution.
5.3 Implementation of Abstract Scheduling Interface

The Design and Purpose of Abstract Scheduling Interface (ASI) was described in Chapter 4. In this Section, we describe our implementation of ASI for EEVDF based PSS in the Linux Kernel. Throughout this Section, we refer to ASI enhanced PSS Task as "ASI-PSS Task".

5.3.1 Summary - ASI Design

In a Proportional Share Scheduler (PSS), 'Weight' assigned to the task is very crucial, as it decides the proportion of resource the task would receive in a given time quantum. ASI is designed to provide a high level interface to Proportional Share Scheduler, to assign weights to the tasks. In ASI, each task specifies its Resource Requirements in the form of a 'tuple'. The 'tuple' consists of 'Constant Share' and 'Benefit', which specify fixed share needed by the Task to deliver a QoS level and value indicating if it could benefit from spare Resources available in the system. ASI computes weights for tasks, that uses ASI Application Programming Interface (API) and schedules them as PSS tasks with the 'estimated Weight'. The 'Weight' for a task is estimated by mapping the value specified in the 'tuple', into a value in fixed range of weights.

ASI also provides a solution to the issue of 'Allocation Uncertainties' in PSS. In PSS, dynamic events like task entering/leaving competition for the resource may cause fluctuations in the share received by tasks. These fluctuations are referred
to as 'Allocation Uncertainties' and they may adversely impact the performance of Time Sensitive applications. ASI counters the issue of 'Allocation Uncertainties' by recomputing weights of Tasks in the event of dynamic Operations such as Task entering/leaving contention for the Resource.

A more detailed explanation on handling under-load scenarios, overload scenarios are available in Section 4.4.

5.3.2 Implementation Strategy

ASI is implemented as a Dynamically Loadable Kernel Module (also referred as Kernel Module). The Kernel Modules are executables that can be inserted/removed from the running kernel [9], which is made possible by the Modular design of Linux Kernel. The Alternative approach is to implement ASI by making extensive modifications to the PSS Scheduler in the Linux Kernel. Unlike the implementation, which warrants making modifications to PSS Scheduler, the implementation as a Kernel Module provides a convenient feature to enable or disable ASI on the fly.

Note on ASI IDLE Implementation

ASI IDLE is a pseudo ASI task. That is in ASI, each ASI task has its own Process Control Block (PCB). But ASI IDLE does not have a PCB. Hence in our implementation of ASI, when ASI IDLE becomes the most eligible request to run, we schedule the 'regular Linux Task' (as picked by the regular Linux Scheduler) to run.
5.3.3 Data Structures

ASI Specific Data Structures inside Kernel Module

- **ASI Entry** - For every ASI task that is created using ASI API, an ASI entry is maintained by the module. It contains the values of Constant Share and Benefit parameter in the 'tuple', computed weight.

- **ASI List** - All the ASI entries are maintained in a List Data Structure.

ASI Specific Data Structures inside Kernel

- **ASI IDLE** - ASI IDLE is pseudo ASI task. It is statically defined inside the kernel.

5.3.4 Interfaces

Provided by ASI Module to User

The ASI Module provides the System Call `sys_pss_changeclass`, for the User Space application to interact with the ASI Module. This System Call takes following parameters from the User: Process ID, Constant Share and Benefit value. It creates an ASI Entry, to hold these parameters and ASI Entry is maintained in ASI List.

Provided by ASI Module to Kernel Scheduler

The ASI Module tracks Total Resource currently in use by all ASI Tasks. The Occurrence of Dynamic events like, a task getting created or exiting the system
directly impact the state of Competition for the Resource. These Events are first received and processed by the Kernel Scheduler. The ASI Module needs to be aware of these critical events, so that it can update the Resource usage and take corrective measures, if needed. The Corrective action by ASI typically is rescaling weights of task as described in Section 4.3.2. ASI Module receives notifications about these events from the Scheduler, through a set of Call back functions implemented by ASI Module. The Call back functions are invoked by EEVDF based PSS Scheduler and its purpose are briefly described here.

- **is_loaded** - to verify if the ASI Module is loaded;

- **update_blocked** - This Routine is invoked, when a ASI Task blocks. Since this task treated as if it has left competition for the resource, Total Available resource is updated.

- **update_exit** - This Routine is to convey to ASI Module that an ASI task has exited the System. ASI Module removes the ASI Entry corresponding to this task and updates the Total Available Resource.

- **update_activate** - This Routine is invoked, when a ASI Task is woken up from sleep.

- **insert_idle** - If the Total Resource currently in use is less 100%, the PSS Scheduler notifies the ASI module to insert the ASI IDLE task with an appropriate weight.
- **get_availpercent** - ASI Module tracks the amount of available Resource. This Routine is used by the Scheduler, to get the current Total Available Resource, to decide if there is a need for corrective actions by ASI.

The EEVDF based PSS Scheduler is unaware that ASI is managing the Resource usage by ASI tasks. It concerns itself mainly about finding the most eligible PSS Request and allocate the given quantum to it.

### 5.3.5 How Does it Work?

The ASI Module is enabled or brought online \(^\text{15}\), only after EEVDF based PSS Kernel is up. With no PSS or ASI Enhanced PSS tasks (ASI-PSS) tasks in the system up to this point, all tasks are scheduled as in a regular Linux Scheduler. All ASI Enhanced PSS Tasks (ASI-PSS) are treated identical to PSS Tasks, inside EEVDF based PSS Scheduler. In this Section, we will trace the Kernel Control path for ASI-PSS Task in the following events: Creation, Blocking, Activation and Exit.

#### Check For ASI IDLE

The Routine `check.asiidle` is statically defined inside the Kernel Scheduler as part of our ASI Enhancement to EEVDF based PSS Scheduler. This is called every time a Scheduling decision is to be made: such as, at the time of beginning of PSS Quantum, when a ASI-PSS Task is created, blocked, activated or on exit from the System. Internally, this routine uses the Call back functions defined by ASI Module.

\(^{15}\) Just like any other Kernel Module, ASI Kernel Module is enabled by using `modprobe` command.
The Purpose of this routine is to insert or remove ASI IDLE from contending for resource.

Internally, this Routine checks if the ASI Module is loaded using `is_loaded`. As described in Section 4.3.2, if the current Resource utilization is less than 100% and all the Tasks in the System are requesting Constant Share with 'Benefit' parameter in the 'tuple' specified as zero, this routine computes a weight for ASI IDLE by invoking `insert_idle`. If the ASI IDLE is not already inserted, then it is added to the competition for Resource. Otherwise, the weight of ASI IDLE is readjusted to its newly computed weight. If the Resource utilization is more than 100%, then ASI IDLE is forced out from contending for Resource.

**Tracing ASI-PSS Task Creation**

The User space Application may invoke the System Call `sys_pss_changeclass` to elevate itself to a ASI-PSS Task. An 'ASI Entry' is allocated to hold the 'tuple' information passed through this System Call. This 'tuple' is used to estimate a 'weight' for this task. The System Call 'sys_sched_setscheduler' is invoked, which causes a PSS Request to be introduced into the PSS List corresponding to this Task. From here on, the events traced are identical to a newly created PSS Task, as explained in Section 5.2.3.
Tracing ASI-PSS Task Exit

When a ASI-PSS Task exits, the state of the Task becomes "EXIT_DEAD". The Kernel Scheduler is invoked which performs Task cleanup. At this point, an up-call to ASI Module is made to update the Resource usage. The ASI Entry corresponding to this Task is deleted. The Weights of other Tasks are re-calculated if there are Tasks that can benefit from spare resources in the System. The PSS Scheduler proceeds to select the most eligible task for execution.

Tracing Dynamic Events

The Events that follow when a ASI-PSS Task blocks, is identical to a PSS Task blocking, up to the point where deactivate_task is invoked. A Call to the routine update_blocked, updates the Total Available Resource in the ASI Model. A Check is performed to recalculate or insert ASI IDLE Task through check_asiidle.

Similarly, when a ASI-PSS Task is activated before the Scheduler chooses the most eligible task to run, the newly activated task's weight is recomputed. This ASI IDLE Task is re-inserted/removed through check_asiidle.
Chapter 6

Experiments and Results

The experimental evaluation of our implementation of EEVDF based PSS and Abstract Scheduling Interface (ASI) to EEVDF based PSS is presented in this chapter. The experiments were conducted first, without ASI on EEVDF based PSS and then with ASI on EEVDF based PSS. All experiments were performed on a uniprocessor System running our prototype implementation of ASI, EEVDF based PSS on Linux kernel version 2.6.18. The system description is presented in the table 6.1.

Table 6.1: System Description

<table>
<thead>
<tr>
<th>Description</th>
<th>Detail</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Type</td>
<td>Pentium M</td>
</tr>
<tr>
<td>CPU Speed</td>
<td>1.5 GHZ</td>
</tr>
<tr>
<td>System Memory</td>
<td>512 MB @ 333 MHz</td>
</tr>
<tr>
<td>Kernel Version</td>
<td>2.6.18 (Patched for ASI/EEVDF)</td>
</tr>
<tr>
<td>Interrupt Frequency</td>
<td>100HZ</td>
</tr>
<tr>
<td>Timer Resolution</td>
<td>10ms</td>
</tr>
</tbody>
</table>

6.1 Our Approach to Analyze Scheduling behavior

The task of analyzing scheduling behavior is not trivial. This is due to the fact that, though the progress made by real world applications can be quantified by time, it is an 'implicit value'[36]. It is often not possible to extract timing information
from a real world application to track the progress made by it. There are two ways to analyze the scheduling behavior [36]:

- **Logging events by Modifying Kernel** - In this approach, the timing information pertaining to a particular application or system wide metrics are logged from within kernel. This is an intrusive approach, which involves modifying the existing kernel for inserting ‘markers’, that logs the occurrence of events that we are interested (such as amount of time a task is running, sleeping etc). This approach is useful in determining bottle-Necks in performance. A popular tool taking this approach is Linux Trace Toolkit (LTT and LTTng) [52, 53].

- **Logging events by Modifying Applications** - In this approach, the events of interests are logged by modifying real world applications.

Another alternative to modifying real world applications, but taking a similar approach is the use of synthetic applications [36, 5]. Synthetic applications are real programs, which emulates real world applications. The advantage of using synthetic application is that, a wide ranging usage scenarios can be emulated, without any modification to real applications. Since this method doesn’t require modifications to the kernel, stability of the system is not compromised.

We have taken an approach, which involves the use of set of tools specifically designed to analyze the scheduling behavior. In the subsequent sections, we will describe the design of the tools as well as the results gathered using them.
6.1.1 Custom Workload Generator

The design of our custom workload generator falls in to the second category, as described in the previous section. It is a synthetic application, designed to create tasks, that will be scheduled by our implementation of EEVDF based Proportional Share Scheduler. In this section, we will describe how this workload generator works.

We may recall from the scheduling hierarchy, described in section 5.1.2, that tasks belonging to Real Time Scheduling class (SCHED_RR or SCHED_FIFO) are preferred for execution over any other task in the system. The workload generator makes use of this implementation decision to its advantage. Our workload generator begins by elevating itself to a task belonging to real time class, so that no other task is chosen for execution, while running tests. It parses the command line arguments supplied to customize the workload and perform workload initialization. That is, the real time parent process forks off as many child PSS tasks as requested in the command line. As shown in the figure 6.1, up to this point, real time parent is the most eligible process to run in spite of presence of other PSS tasks. Once workload initialization is complete, real time parent waits on other PSS tasks to complete execution by sleeping.

The newly created PSS tasks are loaded with an appropriate copy of workload image, it is supposed to run. The workload image is programmed to perform logical computations (such as XOR operations) in a loop. The application’s progress is
tracked by periodically recording the time stamps and iteration count of the logical operations in the workload image. This is implemented by delivering a SIGALRM signal to the PSS task executing the workload image. The workload image allocates a chunk of memory buffer to record time stamps at the start. It also programs SIGALRM signal, to be delivered to the calling process periodically. The signal handler is designed to log the time-stamp and iteration count in the memory buffers. When the logical operations in the loop completes, the data in the memory is written in to a file and workload image terminates.
6.1.2 Issues with Custom Workload Generator

The workload generator uses routines provided by standard C Library such as \texttt{gettimeofday} for measuring the time-stamps. Further the iteration counts and time-stamps are logged from inside of a signal handler. When a \textit{Signal} is delivered to the process, it may be pending on the process or it may be delivered immediately. This asynchronous nature of the signal could also impact the measurements made by the Workload Generator. But nevertheless, the workload generator is useful to get an overall feel of Proportional Share Scheduler.

6.1.3 Hourglass

\textit{Hourglass} is a synthetic workload generator. It is more feature rich as compared to the custom workload generator described in the previous section 6.1.1. It is a multi-threaded user space real time application. The main thread is a high priority real time task, which launches as many threads as requested by the user. These threads can execute a range of workloads including: CPU bound, CPU bound yielding, Periodic, IO Bound etc. The type of workload to be executed by the thread is specified through command line. The execution traces are recorded in memory for each thread. To avoid delays caused by page faults, the thread is locked inside the memory. The time measurements are made using time-stamp counter provided by the Processor. Hence, they are very precise. For example, on Pentium processor the
assembly instruction ‘rdtsc’ is used to get the number of processor cycles, which in turn is used to compute the duration of in CPU slices allotted to each task.

6.2 Experiments with PSS

6.2.1 Demonstration of Fairness in PSS

The goal of this experiment is two fold:

- to demonstrate fairness achieved by using PSS.
- to demonstrate that fixed weights to PSS Tasks, translate in to variable shares.

It is performed with 3 PSS Tasks, without the ASI module online. The tasks are created using workload generator described in section 6.1.1. The weights for these tasks 4, 3 and 1 are assigned manually. The ratios 4 : 3 : 1 correspond to tasks requesting 50%, 37.5% and 12.5% of resources to deliver predefined level of QoS. Each task executes the workload as described in the section 6.1.1 and corresponding time-stamps are recorded. The result of this experiment is plotted in the figure 6.2. The plot show the typical characteristic feature of Proportional Share Algorithm. The tasks make progress in proportion to their weights. Up to 24 seconds, the summation weights of the tasks in the system is a constant. Hence the virtual time increment remains constant. When the Task 1 with larger weight exits, the virtual time increment increases at faster rate.
This experiment also shows the impact of dynamic operations such as, a task entering the system or a task leaving the system on other PSS tasks. Thus for a PSS task to maintain a constant share, its weight has to be re-calculated every time a dynamic operation occurs.

### 6.2.2 Measurements with Hourglass

In this section, we describe the experiment that was performed with Hourglass [36]. We use the capability of Hourglass to make very precise time measurements to record execution traces.
A sample execution trace is recorded for 3 regular Linux tasks, executing CPU bound workload. The trace is shown in the figure 6.3. The priorities of the Task/Thread 0, 1 and 2 are: Low, Normal, High. In Linux, the tasks are allotted slices based on priority. As it is evident from the figure 6.3, the highest priority task has longer slices as compared to other tasks. This experiment demonstrates the lack of fairness in priority based scheduling used in Linux.

Figure 6.3: Execution Trace of 3 Regular Linux Tasks with Priorities: Low, Normal, High

A similar sample execution trace for three PSS Tasks, executing CPU bound load is recorded and it is shown in figure 6.4. The weights of Threads 0, 1 and 2 are: 4, 3, 1. As can observed from Figure 6.4, each Thread receives a quantum of for 100ms. This is the fixed quantum size in our implementation of PSS. Each task is making progress exactly in proportion to its weight. This is confirmed from measured performance of each thread using Hourglass as shown in table 6.2.
The expected share for thread \( i \) is computed as a percentile value as shown below:

\[
\text{ExpectedShare}_i = \frac{\text{Weight}_i}{\text{TotalWeight of all threads}}
\]

Figure 6.4: Execution Trace of 3 PSS Tasks with Weights 4, 3, and 1

Table 6.2: Measurements on Execution Traces Using Hourglass for 3 PSS Tasks

<table>
<thead>
<tr>
<th>Thread</th>
<th>Weight</th>
<th>Running Time</th>
<th>Expected Percentile</th>
<th>Actual Percentile</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>5.099515</td>
<td>50</td>
<td>50.943168</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>3.698908</td>
<td>37.5</td>
<td>36.951734</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1.199666</td>
<td>12.5</td>
<td>11.984429</td>
</tr>
</tbody>
</table>

Total Running Time of Threads = 9.998088 seconds

6.3 Experiments With ASI

In this section, we describe the set of experiments, that were performed with ASI Module online. The resource requirements for the tasks are specified in the form of a 'tuple' consisting of "Constant Share" and "Benefit" value as described in
Section 4.4. We used a modified version of workload generator to take the resource requirements in the form of ‘tuple’.

6.3.1 Mix - Constant Shares and Proportional Use of Spare Resource

The goal of this experiment is to demonstrate, how a mix of tasks requesting ”Constant shares with non-zero Benefit” value, and tasks requesting ”Constant share alone” are handled using ASI. As we described in the design of ASI 4.4, tasks are allotted the requested ”Constant Shares”. If there are spare resources available, after satisfying the ”Constant Share” requirement of all tasks in the system, the excess resources are partitioned and allotted in proportion to their ”Benefit” value. The share of task requesting ”Constant Share” alone with ”Benefit” parameter set to zero, is not affected due to dynamics in shares of other tasks. Each task is started off to execute a workload for varying durations.

This experiment is started with the launch of ”Task 1” requesting ”Constant Share” of 10% and ”Benefit” value of 2. Since this is the only task in the system, its computed dynamic share is 100%. After 5 seconds, ”Task 2” enters the system requesting a share of 20% and ”Benefit” value of 3. At this point, each task requests a share of 10% and 20% respectively. Due to their non-zero benefit value, the remainder of the resource 70% is allocated proportionately to each of the active tasks. ASI computes a dynamic share for each of the tasks, which would be 38% and 62% re-
Figure 6.5: Mix - ASI Tasks with Constant Shares and Proportional Allocation of Resource

![Plot of 3 ASI Tasks against time each requesting Constant share with/without Benefits](image)

pectively. The dynamic share is computed using equation 4.1 and a sample dynamic Share computation for Task 1 is shown below:

\[ D_1 = S_1 + IDLE_{\text{capacity}} \times \frac{w_1}{\sum w_j} \]

\[ D_1 = 10 + 70 \times \frac{2}{5} = 38 \]

The results are plotted in figure 6.5. The figure 6.5 can be interpreted with the aid of table 6.3. The share of Task 1 drops from 100% to 41% with the arrival of Task 2. At the end of time interval A-B as shown in figure 6.5, "Task 3" enters the system requesting a constant share 30% and "Benefit" value of 0. The expected
dynamic shares of the clients at this point are 26%, 44% and 30%. The shares received by each client very closely follows the expected result. When task 3 exits, the dynamic shares of task 1 and 2 are 38% and 62% respectively. Thus ASI ensures that the share of a task never falls below the constant share requested by tasks. It is also evident that, a task requesting constant share of resource with "Benefit" value zero, receives the exact share and no more. The progress made by a non-ASI or regular Linux task is also shown in the figure 6.5. It is evident that regular linux tasks receives its share only after ASI tasks requesting "Constant Shares" with “Benefits” have completed their execution.

Table 6.3: Observed Shares of ASI Tasks - Mixed Scenario

<table>
<thead>
<tr>
<th>Time Interval</th>
<th>Description</th>
<th>Task 1</th>
<th>Task 2</th>
<th>Task 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - A</td>
<td>Iterations Executed</td>
<td>8441</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Share Expected</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Share Received</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A - B</td>
<td>Iterations Executed</td>
<td>3960</td>
<td>5605</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Share Expected</td>
<td>38</td>
<td>62</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Share Received</td>
<td>41.3</td>
<td>58.6</td>
<td>0</td>
</tr>
<tr>
<td>B - C</td>
<td>Iterations Executed</td>
<td>17600</td>
<td>29867</td>
<td>19965</td>
</tr>
<tr>
<td></td>
<td>Share Expected</td>
<td>26</td>
<td>44</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>Share Received</td>
<td>26.1</td>
<td>44.2</td>
<td>29.6</td>
</tr>
<tr>
<td>C - D</td>
<td>Iterations Executed</td>
<td>5667</td>
<td>9262</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Share Expected</td>
<td>38</td>
<td>62</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Share Received</td>
<td>37.9</td>
<td>62</td>
<td>0</td>
</tr>
<tr>
<td>D - E</td>
<td>Iterations Executed</td>
<td>13805</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Share Expected</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Share Received</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
The execution pattern in each interval was observed and is summarized in the table 6.4. The results indicate that total number of iterations is consistent throughout the execution of ASI tasks.

<table>
<thead>
<tr>
<th>Time Interval</th>
<th>Duration (seconds)</th>
<th>Total Iter-ations</th>
<th>Execution rate (Iterations per second)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - A</td>
<td>4.89</td>
<td>8441</td>
<td>1726.17</td>
</tr>
<tr>
<td>A - B</td>
<td>5.6</td>
<td>3960</td>
<td>1708.03</td>
</tr>
<tr>
<td>B - C</td>
<td>49.59</td>
<td>17600</td>
<td>1724.86</td>
</tr>
<tr>
<td>C - D</td>
<td>8.72</td>
<td>5667</td>
<td>1712.04</td>
</tr>
<tr>
<td>D - E</td>
<td>8.2</td>
<td>13805</td>
<td>1683.17</td>
</tr>
</tbody>
</table>

6.3.2 Proportional Use of Spare Resource

The previous experiment considered a mix of tasks requesting ”Constant Share alone” and tasks requesting ”Constant Share with non-zero Benefits”. In this experiment, we consider all tasks requesting constant share with non-zero ”Benefit” value.

At the start, ”Task 1” requesting a constant share of 10% with ”Benefit” value two enters the System. This task is allocated a share of 100%. After 5 seconds, ”Task 2” enters the system requesting a constant share of 20%. It also specifies a ”Benefit” value of 3. ASI computes a dynamic share for Task 2, as it can benefit from using available resources. Hence, in the time interval A-B as show in figure
Figure 6.6: ASI Tasks - Proportional Allocation of Spare Resource

The shares received by Task 1 and 2 are 38% and 62% respectively. In the time interval B-C in figure 6.6, "Task 3" enters the System with a constant share request of 30% and "Benefit" value 1. ASI recomputes the dynamic share of all Tasks to arrive at share values: 23.33%, 40% and 36.6%. The shares of the tasks are analyzed in the time intervals (A, B, C, D and E in 6.6) and presented in table 6.5. As it is evident from the computations, ASI allocates the spare resources to ASI tasks in a proportional way. The execution pattern in each of the interval is summarized in table 6.6.
Table 6.5: Observed Shares of 3 ASI Tasks - Proportional Allocation of Spare Resources

<table>
<thead>
<tr>
<th>Time Interval</th>
<th>Description</th>
<th>Task 1</th>
<th>Task 2</th>
<th>Task 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Iterations Executed</td>
<td>8440</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Share Expected</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Share Received</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 - A</td>
<td>Iterations Executed</td>
<td>3964</td>
<td>5635</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Share Expected</td>
<td>38</td>
<td>62</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Share Received</td>
<td>41.2</td>
<td>58.7</td>
<td>0</td>
</tr>
<tr>
<td>A - B</td>
<td>Iterations Executed</td>
<td>13788</td>
<td>24128</td>
<td>21305</td>
</tr>
<tr>
<td></td>
<td>Share Expected</td>
<td>23.33</td>
<td>40</td>
<td>36.6</td>
</tr>
<tr>
<td></td>
<td>Share Received</td>
<td>23.2</td>
<td>40.7</td>
<td>35.9</td>
</tr>
<tr>
<td>B - C</td>
<td>Iterations Executed</td>
<td>13345</td>
<td>0</td>
<td>13598</td>
</tr>
<tr>
<td></td>
<td>Share Expected</td>
<td>50</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Share Received</td>
<td>49.5</td>
<td>0</td>
<td>50.4</td>
</tr>
<tr>
<td>C - D</td>
<td>Iterations Executed</td>
<td>15437</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Share Expected</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Share Received</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The regular linux task is chosen for execution after the completion of all ASI tasks requesting "Benefits". If a user intends not to starve regular linux tasks, the ASI IDLE can be accorded a fixed share. This would allow ASI IDLE to be eligible for execution and this in-turn allows the task chosen by Linux scheduler to execute.

6.3.3 Constant Shares Only

Certain applications may not benefit from allocation of more resources than requested "Constant Shares". As we may recall, an example of such an application is "Virtual Machine OS" running on "Virtual Machine Server". A user may have found that it may be sufficient to assign a fixed share to such an Application. This
Table 6.6: Iterations Performed During Time Intervals - All Tasks Requesting Benefits

<table>
<thead>
<tr>
<th>Time Interval</th>
<th>Duration (seconds)</th>
<th>Total Iterations</th>
<th>Execution rate (Iterations per second)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - A</td>
<td>4.91</td>
<td>8440</td>
<td>1718.94</td>
</tr>
<tr>
<td>A - B</td>
<td>5.61</td>
<td>9599</td>
<td>1711.05</td>
</tr>
<tr>
<td>B - C</td>
<td>34.58</td>
<td>59221</td>
<td>1712.58</td>
</tr>
<tr>
<td>C - D</td>
<td>15.83</td>
<td>26943</td>
<td>1702.02</td>
</tr>
<tr>
<td>D - E</td>
<td>9.09</td>
<td>15437</td>
<td>1698.23</td>
</tr>
</tbody>
</table>

experiment is set up to demonstrate, how tasks requesting "Constant Share" alone, with "Benefit" value in the 'tuple' set to zero are handled in ASI.

In this experiment, three tasks 1, 2 and 3 are launched simultaneously. The resource requirements for these tasks are specified as (10, 0), (20, 0) and (30, 0). Each task is requesting a "Constant Share" of 10%, 20% and 30% respectively. To capture the dynamics of task exiting the system, each task executes the workload for different durations. The result of this experiment is plotted in the figure 6.7.

For a task, which specifies the "Benefit" value in the 'tuple' as zero, ASI does not compute dynamic share. Hence, with enough resources available, each task is allotted the requested "Constant Share". It is evident from the figure 6.7, that exit of a task does not impact the share received by other tasks in the system. The spare resource resulting after the allocation to all tasks in the system, is allocated to ASI IDLE. In our implementation of ASI, when ASI IDLE is the most eligible task in the system, the task chosen by regular Linux scheduler is allowed to execute.
Figure 6.7: ASI Tasks with Constant Shares Only

A Plot of Operations performed by ASI Tasks against Time

- Task 1 Share = 10 Benefit = 0
- Task 2 Share = 20 Benefit = 0
- Task 3 Share = 30 Benefit = 0
Chapter 7

Conclusions and Future Work

This thesis demonstrates the implementation of Earliest Eligible Virtual Deadline First (EEVDF) based Proportional Share Scheduler (PSS), which was first proposed by Stoica et.al [4]. Our implementation on Linux kernel 2.6, is the first implementation on a fully preemptive operating system kernel. Another unique aspect of our implementation is that it co-exists with the Linux kernel. We have accomplished this by introducing a new scheduling policy $SCHED\_PSS$ in the Linux kernel. We have demonstrated through the experiments on our implementation of EEVDF, that when tasks are assigned weights, each task receive a share of the resource in proportion to their weights and there by each task makes progress in proportion to their weights.

There are applications (such as Video playback), which may require a fixed share of resource to deliver an acceptable Quality of Service (QoS). It is not possible to achieve constant share for such applications using proportional share scheduling such as EEVDF. This is due to the dynamics of tasks entering or leaving the system, at any time in a given interval. These dynamics causes fluctuations in the share received by tasks in the system.

We have addressed the problem of achieving constant share for applications using EEVDF, through the design and implementation of Abstract Scheduler Interface
ASI is an abstraction layer designed on top of EEVDF based PSS. It provides an interface to the user to specify resource requirements for applications. Based on the requested share, ASI computes an appropriate weight. This task is scheduled as a PSS task with the computed weight. In the event of dynamic operations, such as a new task entering the system or an existing task leaving the system, the weights are recomputed to maintain a constant share. ASI also provides provision for tasks to request a constant share and still make use of spare resources, if they are available. We have also experimentally demonstrated the ability of ASI to provide constant share to applications using EEVDF based PSS, with out inherent modifications to EEVDF based PSS algorithm.

7.1 Limitations and Improvements to Our Prototype Implementation

In this section, we list out some limitations in our prototype implementation.

- **Interface to launch Tasks** - The Interface provided by our current prototype implementation to launch ASI/PSS Tasks is by using “System Call” explicitly. Hence, to launch a real application as a PSS/ASI Task, one may need to fork a process, elevate it to ASI/PSS class using the ‘System Call’ and use the ‘exec’ call to load the application. It may be cumbersome to launch applications this way.
• **Improvements to PSS Request Data Structures** - All PSS requests in current implementation are held in a 'List' data structure. It is implemented in Linux Kernel as a set of macros. Though the 'List' implementation is fast and reliable, the asymptotic complexity involved in searching for the most eligible request with earliest virtual deadline is in the order of $O(n^2)$.

• **Symmetric Multiprocessing (SMP) Support** - At the time of development, we focused on Scheduling sub-system in uni-processor Systems. But with rising popularity of Multi-core processors in the Desktop Computing Systems, it would be good to support Symmetric Multiprocessing Systems.

### 7.2 Future Work

The limitations in the current implementation listed in previous section could be addressed as part of Future work.

• **Scalability** - The Theoretical asymptotic complexity involved in searching for the most eligible request in a List based implementation is in the order of $O(n^2)$. It would be interesting to perform scalability tests to get an estimate of scheduling overhead involved, when a large number of tasks are launched. Stoica et.al has suggested a modified Binary Search Tree to hold PSS Requests in their Research [4]. With a Binary Search Tree based implementation, the
complexity in searching the most Eligible Request can be improved to the order of $O(lg n)$. This modification could make the implementation more scalable.

- **ASI Management Interface** It would be good to have an interface to manage ASI/PSS. The user can select an application and specify the resource requirements for the application. The Interface can handle launching the application by making appropriate calls. This could avoid the user developing programs to get benefits from using ASI. In other words, this will aid user in getting maximum benefits out of ASI.

Though our implementation of EEVDF focused on processor scheduling, the resource management ideas can be applied in other scheduler domains such as the Virtual Memory management sub-system in a GPOS. Berger et.al [54] make an interesting observation, that most of the virtual memory managers in the GPOS are not aware of the scheduler policies. Moreover, they are oriented towards increasing system wide throughput. Hence the strategy used by current virtual memory managers could affect the scheduler policy decisions. It would be interesting to explore the possibility of using Proportional Share based Virtual Memory Management.
Bibliography


Appendix A

EEVDF Implementation

A.1 Data Structures Used

A.1.1 Request Data Structure

Each EEVDF Task has a Request element which is defined in this section.

/*
 * the request structure that will be used by a PSS
 * process is defined below;
 */
struct pssrequest {
    int pid;    /* pid of owner task */
    int weight; /* weight associated with Task */
    int vstart; /* virtual start time */
    int vend;   /* virtual end time */
    int used;
    int lag;
    int ve;     /* virtual eligible time */
    int vd;     /* virtual deadline */
    int minvd;  /* minimum virtual deadline */
    struct task_struct *owner; /* pointer to owner task */
    struct list_head entry;
    struct list_head ellist;
};

A.1.2 Request List

All the EEVDF Requests are maintained in a Circular Doubly Linked list which is defined in our implementation as described below:

/*
 * the request for each PSS task is held in a list data structure
A.1.3 PSS Queue

All the PSS tasks are held in a List based Queue data structure as described below.

/*
 * PSS:
 * the pss queue to hold all PSS/ASI tasks
 * this is a per-cpu data structure.
 */
struct pssqueue {
    unsigned long nr_running;
    struct list_head q;
};

A.2 Fixed Point Macros

Inside kernel it is not feasible to floating point calculations. But our PSS calculations requires us to do floating point math. So we had following options to do calculations:

- Integer Scaling

- Fixed Point Arithmetic
We opted to use Fixed Point Arithmetic as they are fast, robust (bit-wise operators are a lot faster than multiplication by huge numbers). The macro definitions are provided here for reference.

```c
#define EXPONENT 8

/* to convert the normal number in to fixed point format */
#define TOFIX(a) ((int)( (a)*(double)(1<<EXPONENT )))

/*
 * basic operations performed on 2 numbers in fixed point q format returning
 * answers in q-format
 */
#define ADD(a, b) ((a)+(b))
#define SUB(a, b) ((a)-(b))
#define MUL(a, b) (((a)*(b))>>(EXPONENT))
#define DIV(a, b) (((a)<<(EXPONENT))/(b))
```