QoS IN PARALLEL JOB SCHEDULING

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the
Graduate School of The Ohio State University

By

Mohammad Kamrul Islam, B.S., M.S.

* * * * *

The Ohio State University

2008

Dissertation Committee:

Prof. P. Sadayappan, Adviser
Prof. Dhabaleswar K. Panda
Prof. Atanas Rountev

Approved by

Adviser
Graduate Program in
Computer Science and
Engineering
ABSTRACT

Considerable research has focused on the problem of scheduling dynamically arriving independent parallel jobs on a given set of resources to improve the performance with respect to various system and user metrics. However, there has been little work on provision of Quality of Service (QoS) in space-shared parallel job scheduling, in the form of hard deadline guarantees and service differentiation. Both of these functionalities offered by system providers are very desirable to the user. On the other hand, revenue maximization along with the optimal management of resources is appealing to a service provider.

This dissertation addresses these seemingly orthogonal aspects of parallel job scheduling in stages. At first, a new scheme called QoPS is developed, to provide QoS in the form of response time guarantees. Essentially, QoPS implements an admission control mechanism for jobs, and provides deadline guarantees for all accepted jobs. Secondly, a pioneer model is proposed to enable proportional service differentiation (PSD) in job scheduling. A PSD framework would basically allow proportional allocation of resources across users based on relative priorities. In addition, new schemes are designed to offer PSD to satisfy the varied expectations of users without hurting traditional performance metrics.
In order to address the revenue issue, two different charging models are investigated, determined by the resource provider and user respectively. Since no QoS-enabled charging model is currently deployed at any supercomputer center, a new provider-determined charging model is proposed. In this context, the impact of user tolerance towards missed deadlines is studied, as well as various techniques to further improve the overall revenue. Alternatively, a user-centric and market-based revenue approach originally proposed for non-QoS scheduling is adapted for QoS-aware scheduling. Using this charging model, an extension to QoPS called DVQoPS is being developed, that considers the opportunity cost using a history-based predictive technique and thus maximizes the overall revenue while maintaining the deadline guarantees in an integrated way.
I dedicate this dissertation to my parents, Sirajul Islam and Feroza Begum, my uncle

Fazlur Rahman and my wife Ayesha Akhter
ACKNOWLEDGMENTS

First of all, this dissertation could not have been completed without the active involvement of my adviser Professor P. Sadayappan who not only guided me, but also challenged me in every stage of my graduate studies. Without his continuous patience and understanding of my personal, family and other career responsibilities, I would not be where I am at this point. In addition, he, along with the other advisers - Professor D. Panda and Professor A. Rountev, helped me throughout the dissertation process, never accepting less than my best efforts. I thank them all.

I would like to acknowledge and extend my special gratitude to Pavan Balaji, Gerald Sabin, Mohammed Rahman, Gaurav Khanna for helping me in all respects of my research works. They have reasonably challenged my suggestions enabling me a better researcher and thinker. Specially, Pavan’s comprehensive explanations and in-depth analysis of my research topics have immensely improved this dissertation. I am also indebted to all of my graduate colleagues to whom I have collaborated in various scopes. I would like to convey thanks and cordial appreciation to all those friends at OSU. In particular, I would like to remember the cooperation and affability of Sitaram, Swarup, Uday, Vijay, Raj, Ranjit, Muthu, Albert and many others.

I offer my heartfelt gratitude to my family members for their continuous support and selfless blessings. My special thanks to my wife Ayesha Akhter who has encouraged me in this long journey. She has effectively taken care of my kids and family chores while
allowing me to continue the study beyond my usual office hours. Without her sacrifice, I could not have completed this challenging endeavor. Above all, I would like to take this opportunity with great pleasure to thank the most important persons in my life: my father Sirajul Islam, mother Feroza Begum and uncle Fazlur Rahman. Their solemn prayer and sincere affection have guided me to reach this current position.

And finally, I am grateful to God who makes all things possible for me.
VITA

April 10, 1968 ......................... Born - Bangladesh

1994 ................................. B.S Computer Science & Engineering,
                               Bangladesh University of Engineering
                               and Technology, Dhaka, Bangladesh

1997 ................................. M.S. Computer Science & Engineering,
                               Wright State University, Dayton, OH

2008 ................................. M.S. Computer Science & Engineering,
                               Ohio State University, Columbus, OH

1998-Present ........................ Senior Software Engineer
               Vero Systems Inc., Columbus, OH

1997-1998 ............................ Software Engineer
               Systran Corporation, Dayton, OH

1994-1995 ............................ Lead Developer
               PACS Bangladesh Ltd., Bangladesh

PUBLICATIONS

Research Publications

P. Balaji, P. Sadayappan and M. Islam “Chapter on Techniques on Providing Hard
Quality of Service Guarantees in Job Scheduling”. *Book Title: Market-Oriented Grid

N. Desai, P. Balaji, P. Sadayappan and M. Islam “Are Non-Blocking Networks Really
Needed for High-End-Computing Workloads?”. *in IEEE International Conference on


FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in Parallel Job Scheduling Strategies: Prof. P. Sadayappan
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>v</td>
</tr>
<tr>
<td>Vita</td>
<td>vii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xiv</td>
</tr>
<tr>
<td>List of Algorithms</td>
<td>xvi</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xvii</td>
</tr>
<tr>
<td>Chapters:</td>
<td></td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Problem Statement</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Proposed Solutions</td>
<td>4</td>
</tr>
<tr>
<td>2. Background and Related Works</td>
<td>10</td>
</tr>
<tr>
<td>2.1 Fundamentals of Parallel Job Scheduling</td>
<td>10</td>
</tr>
<tr>
<td>2.1.1 Parallel Computing System</td>
<td>11</td>
</tr>
<tr>
<td>2.1.2 Parallel Job</td>
<td>12</td>
</tr>
<tr>
<td>2.1.3 Job Preemption</td>
<td>14</td>
</tr>
<tr>
<td>2.1.4 Backfilling</td>
<td>15</td>
</tr>
<tr>
<td>2.1.5 Workload</td>
<td>18</td>
</tr>
<tr>
<td>2.1.6 Metrics</td>
<td>20</td>
</tr>
<tr>
<td>2.2 Related Works</td>
<td>22</td>
</tr>
</tbody>
</table>
7.5 Value-aware QoPS (VQoPS) ........................................ 141
  7.5.1 Design of VQoPS ........................................ 144
  7.5.2 Impact of Workload Characteristics on VQoPS ............ 145
7.6 DVQoPS: A Self-learning Variant of Value-aware QoPS ......... 147
  7.6.1 Dynamic Value-aware QoPS (DVQoPS) ...................... 147
  7.6.2 Balancing Sensitivity and Stability in DVQoPS ............ 149
  7.6.3 DVQoPS Algorithm ........................................ 151
7.7 Experimental Results ........................................... 154
  7.7.1 Impact of Urgent Job Mix on Performance .................. 154
  7.7.2 Impact of Cost of Urgent Jobs and System Load .......... 157
  7.7.3 Impact of User Runtime Estimate Inaccuracy .............. 159
7.8 Conclusions ..................................................... 162

8. Proportional Service Differentiation in Job Scheduling ............ 164
  8.1 Introduction .................................................. 164
  8.2 Job Scheduling Background ................................... 168
    8.2.1 Overview of Job Scheduling ............................ 169
    8.2.2 Preemption in Job Scheduling ......................... 169
8.3 Proportional Service Differentiation Model ....................... 170
  8.3.1 Properties of a Parallel Job Scheduler .................... 171
  8.3.2 Job Prioritization ....................................... 171
  8.3.3 Some Consequences of the Model ......................... 172
8.4 PSD-aware Schemes ............................................. 173
  8.4.1 Simple PSD-aware Scheme ................................ 174
  8.4.2 Preemption-Based Schemes ............................... 175
8.5 Experimental Results ........................................... 179
  8.5.1 Evaluation Setup ....................................... 179
  8.5.2 Performance Analysis of Migratable PSD ................. 180
  8.5.3 Comparing different PSD Schemes with respect to Service
        Differentiation Capability .............................. 182
  8.5.4 Comparing different PSD Schemes with respect to Absolute
        Performance ............................................... 184
  8.5.5 PSD vs. ASD ........................................... 186
  8.5.6 Load Adaptation ....................................... 187
  8.5.7 Inaccuracy in User Estimate ............................ 192
8.6 Related Work .................................................. 195
8.7 Concluding Remarks ............................................ 196

9. Conclusions and Future Works ..................................... 197
  9.1 Research Contributions ..................................... 197
9.2 Continuing Work ...................................................... 200
  9.2.1 SLA-based Service Differentiation ......................... 200
  9.2.2 Quantifying the Opportunity Cost ......................... 202

Bibliography ............................................................. 203
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Important Notations</td>
<td>31</td>
</tr>
<tr>
<td>6.1 Percentage Improvement of \textit{NormalizedUrgency} over \textit{FirstReward} for sequential job traces.</td>
<td>118</td>
</tr>
<tr>
<td>6.2 Percentage Revenue Improvement of different schemes over \textit{FirstReward} for sequential jobs with 20% urgent job</td>
<td>119</td>
</tr>
<tr>
<td>6.3 Percentage Revenue Improvement of different schemes over \textit{FirstReward} for sequential jobs with 50% urgent job</td>
<td>119</td>
</tr>
<tr>
<td>6.4 Slowdown (First Price) for offered load 88%</td>
<td>121</td>
</tr>
<tr>
<td>6.5 Response Time in hour (First Price) for offered load 88%</td>
<td>121</td>
</tr>
<tr>
<td>6.6 Slowdown (Present Value) for offered load 88%</td>
<td>121</td>
</tr>
<tr>
<td>6.7 Response Time in hour (Present Value) for offered load 88%</td>
<td>122</td>
</tr>
<tr>
<td>6.8 Slowdown (Normalized Urgency) for offered load 88%</td>
<td>122</td>
</tr>
<tr>
<td>6.9 Response Time in hour (Normalized Urgency) for offered load 88%</td>
<td>123</td>
</tr>
<tr>
<td>6.10 Slowdown (Opportunity Cost) for offered load 88%</td>
<td>123</td>
</tr>
<tr>
<td>6.11 Response Time in hour (Opportunity Cost) for offered load 88%</td>
<td>124</td>
</tr>
<tr>
<td>6.12 Slowdown (First Reward) for offered load 88%</td>
<td>124</td>
</tr>
<tr>
<td>6.13 Response Time in hour (First Reward) for offered load 88%</td>
<td>125</td>
</tr>
</tbody>
</table>
7.1 Revenue improvement for varying OC-Factors . . . . . . . . . . . . . . . . 146

7.2 Effects of rollback windows size for exact (perfect) user estimates, relative urgent job cost = 10X, and 50% urgent jobs at a high offered load . . . . . . . 150
List of Algorithms

1. Admission control mechanism followed in AREL scheme ........................................... 44
2. Admission control mechanism followed in Modified Slack-Based scheme ................. 46
3. Admission control mechanism followed in Modified Real-Time scheme .................. 48
4. Admission control mechanism used in QoPS scheme for FIRSTFIT ....................... 53
5. During backfill event of a Simple PSD-aware scheme ............................................. 174
6. PSD-aware Local Preemption based scheme to be run at every preemption interval .................................................. 177
7. PSD-aware Migratable Preemption based scheme to be run at every preemption interval .................................................. 178
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Generic Queuing System</td>
<td>11</td>
</tr>
<tr>
<td>2.2</td>
<td>Different types of parallel job</td>
<td>13</td>
</tr>
<tr>
<td>3.1</td>
<td>A simple model of parallel job scheduling</td>
<td>29</td>
</tr>
<tr>
<td>3.2</td>
<td>Flowchart of simulation-based evaluation of new scheme</td>
<td>33</td>
</tr>
<tr>
<td>4.1</td>
<td>Unaccepted Processor Seconds for different loads where 20% jobs have user specified deadline with exact user runtime estimate and Stringency Factor 0.2 for CTC trace</td>
<td>63</td>
</tr>
<tr>
<td>4.2</td>
<td>Unaccepted Processor Seconds for different loads where 20% jobs have user specified deadline with exact user runtime estimate and Stringency Factor 0.5 for CTC trace</td>
<td>63</td>
</tr>
<tr>
<td>4.3</td>
<td>Unaccepted Processor Seconds for different loads where 80% jobs have user specified deadline with exact user runtime estimate and Stringency Factor 0.2 for CTC trace</td>
<td>64</td>
</tr>
<tr>
<td>4.4</td>
<td>Unaccepted Processor Seconds for different loads where all jobs have user specified deadline with exact user runtime estimate and Stringency Factor 0.2 for CTC trace</td>
<td>64</td>
</tr>
<tr>
<td>4.5</td>
<td>Unaccepted Processor Seconds for different loads where 20% jobs have user specified deadline with inexact user runtime estimate and Stringency Factor 0.2 for CTC trace</td>
<td>66</td>
</tr>
<tr>
<td>4.6</td>
<td>Unaccepted Processor Seconds for different loads where 20% jobs have user specified deadline with inexact user runtime estimate and Stringency Factor 0.2 for SDSC trace</td>
<td>66</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>4.7</td>
<td>Number of unadmitted jobs for different loads where 20% jobs have user specified deadline with inexact user runtime estimate and Stringency Factor 0.2 for CTC trace</td>
<td>68</td>
</tr>
<tr>
<td>4.8</td>
<td>Number of unadmitted jobs for different loads where 20% jobs have user specified deadline with inexact user runtime estimate and Stringency Factor 0.2 for SDSC trace</td>
<td>68</td>
</tr>
<tr>
<td>4.9</td>
<td>Average Response Time for different loads where 20% jobs have user specified deadline with inexact user runtime estimate and Stringency Factor 0.2 for CTC trace</td>
<td>69</td>
</tr>
<tr>
<td>4.10</td>
<td>Average Response Time for different loads where 80% jobs have user specified deadline with inexact user runtime estimate and Stringency Factor 0.2 for CTC trace</td>
<td>70</td>
</tr>
<tr>
<td>4.11</td>
<td>Average Response Time for different loads where 20% jobs have user specified deadline with inexact user runtime estimate and Stringency Factor 0.2 for SDSC trace</td>
<td>70</td>
</tr>
<tr>
<td>4.12</td>
<td>Average Slowdown for different loads where 20% jobs have user specified deadline with inexact user runtime estimate and Stringency Factor 0.2 for CTC trace</td>
<td>72</td>
</tr>
<tr>
<td>4.13</td>
<td>Average Slowdown for different loads where 80% jobs have user specified deadline with inexact user runtime estimate and Stringency Factor 0.2 for CTC trace</td>
<td>73</td>
</tr>
<tr>
<td>4.14</td>
<td>Average Slowdown for different loads where 20% jobs have user specified deadline with inexact user runtime estimate and Stringency Factor 0.2 for SDSC trace</td>
<td>73</td>
</tr>
<tr>
<td>4.15</td>
<td>Utilization for different loads where 20% jobs have user specified deadline with inexact user runtime estimate and Stringency Factor 0.2 for CTC trace</td>
<td>74</td>
</tr>
<tr>
<td>4.16</td>
<td>Utilization for different loads where 80% jobs have user specified deadline with inexact user runtime estimate and Stringency Factor 0.2 for CTC trace</td>
<td>74</td>
</tr>
</tbody>
</table>
4.17 Utilization for different loads where all jobs have user specified deadline with inexact user runtime estimate and Stringency Factor 0.2 for CTC trace .......................................................... 75

4.18 Utilization for different loads where 80% jobs have user specified deadline with inexact user runtime estimate and Stringency Factor 0.2 for SDSC trace .......................................................... 75

5.1 The Feedback based QoPS Scheduler ................................................. 86

5.2 Resource Charge for different loads for Stringency Factor = 0.2: where (a) all jobs have user specified deadline (b) 80% jobs have user specified deadline ......................................................... 91

5.3 QoS Charge for different loads for Stringency Factor = 0.2: where (a) all jobs have user specified deadline (b) 80% jobs have user specified deadline ......................................................... 92

5.4 Resource Charge for different Slack Factors with Stringency Factor = 0.2 and load 1.6: (a) all jobs have user specified deadline (b) 80% jobs have user specified deadline ......................................................... 94

5.5 QoS Charge for different Slack Factors with Stringency Factor = 0.2 and load 1.6: (a) all jobs have user specified deadline (b) 80% jobs have user specified deadline ......................................................... 95

5.6 Resource Charge with Kill-and-Restart and without Kill-and-Restart for Stringency Factor = 0.2, load = 1.6 and TF = 4.0 (a) all jobs have user specified deadline (b) 80% jobs have user specified deadline ......................................................... 96

5.7 Resource Charge with Kill-and-Restart and without Kill-and-Restart for Stringency Factor = 0.2, load = 1.6 and TF = 4.0 (a) all jobs have user specified deadline (b) 80% jobs have user specified deadline ......................................................... 97

6.1 Utility function used in Value-Based Job Scheduling ......................... 103

6.2 Revenue improvement relative to FirstPrice assuming exact runtime estimate at different offered loads .......................................................... 113

6.3 Revenue improvement relative to FirstPrice assuming exact runtime estimate at different offered loads with 80% urgent job .......................................................... 114
7.7 Revenue improvement relative to the QoPS assuming exact estimates for original and high offered load for SDSC trace ........................................ 157

7.8 Revenue improvement relative to the QoPS assuming exact estimates for original and high offered load of CTC trace ........................................ 158

7.9 Revenue improvement and Accepted load relative to QoPS assuming actual inexact estimates at the original offered load (89%) for SDSC trace. Cost of urgent jobs is 10X compared to normal jobs. In (b), 50% urgent jobs are used. ..................................................... 160

7.10 Revenue improvement and Accepted load relative to QoPS assuming actual inexact estimates at the original offered load (74%) for CTC trace. Cost of urgent jobs is 10X compared to normal jobs. In (b), 50% urgent jobs are used. ..................................................... 161

7.11 Revenue improvement relative to the QoPS assuming inexact estimates for actual and high offered load for SDSC trace ........................................ 162

7.12 Revenue improvement relative to the QoPS assuming inexact estimates for actual and high offered load for CTC trace ........................................ 163

8.1 Provided relative service (response time) to different priorities of jobs for Migratable scheme for different file transfer rate while offered load varies. ................................................................. 181

8.2 Response time for various offered loads in different schemes w.r.t. different file transfer rates ................................................................. 182

8.3 Provided relative service (response time) to different priorities of jobs for different schemes while offered load varies. Ideal ratios are $\frac{\text{low}}{\text{high}} = 3$ and $\frac{\text{low}}{\text{medium}} = 2$ ................................................................. 183

8.4 Response time and slowdown of jobs for different schemes for load=97% 185

8.5 Utilization achieved in different schemes for various offered loads ...... 185

8.6 Category-wise slowdown and response time for various schemes (Offered Load = 97%) ................................................................. 186
8.7 Average response time and service ratio for different grouping of jobs comparing PSD, ASD based schemes (Local and Migratable) at offered load(97%)  .......................................................... 187

8.8 Provided relative service (response time) to different priorities of jobs for different schemes with exact user runtime estimate, same offered load for each priority class and burstiness factor = 0.5 where ideally expected ratios are $\frac{\text{low}}{\text{high}} = 3$ and $\frac{\text{low}}{\text{medium}} = 2$  .................................................. 188

8.9 Response time and slowdown of jobs for different schemes for load=97% with exact runtime estimate, same offered load for each priority class and burstiness factor = 0.5  ................................................. 189

8.10 Utilization achieved in different schemes for various offered loads with exact runtime estimate, same offered load for each priority class and burstiness factor = 0.5  .................................................. 189

8.11 Provided relative service (response time) to different priorities of jobs for different schemes with exact user runtime estimate, offered load ratio for priority class high:medium:low = 4:2:1 and burstiness factor = 0.0 where ideally expected ratios are $\frac{\text{low}}{\text{high}} = 3$ and $\frac{\text{low}}{\text{medium}} = 2$  .................................. 190

8.12 Response time and slowdown of jobs for different schemes for load=97% with exact runtime estimate, offered load ratio for priority class high:medium:low = 4:2:1 and burstiness factor = 0.0  ................................................. 191

8.13 Utilization achieved in different schemes for various offered loads with exact runtime estimate, offered load ratio for priority class high:medium:low = 4:2:1 and burstiness factor = 0.0  ................................................. 191

8.14 Average response time and service ratio for different grouping of jobs comparing PSD, ASD based schemes (Local and Migratable) at offered load(97%) with exact runtime estimate, offered load ratio for priority class high:medium:low = 4:2:1 and burstiness factor = 0.0  .................................. 192

8.15 Provided relative service (response time) to different priorities of jobs for different schemes with inexact user runtime estimate, same offered load for each priority class and burstiness factor = 0.0 where ideally expected ratios are $\frac{\text{low}}{\text{high}} = 3$ and $\frac{\text{low}}{\text{medium}} = 2$  .................................. 193
8.16  Response time and slowdown of jobs for different schemes for load=97% with inexact user runtime estimate, same offered load for each priority class and burstiness factor = 0.0 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 193

8.17  Utilization achieved in different schemes for various offered loads with inexact runtime estimate, same offered load for each priority class and burstiness factor = 0.0. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 194
CHAPTER 1

INTRODUCTION

In recent years, the increase in projects with massive computing demands has enabled breakthroughs in areas such as atmospheric science, genomics, computational astronomy, molecular dynamics. To address this growing computational demand, various parallel machines with massive computing power have been built with thousands of processors. Effective scheduling of such huge resources is a big challenge in high-performance computing and a significant volume of research has focused on parallel job scheduling considering various objectives.

1.1 Problem Statement

Typically, providers such as supercomputer centers offer computing resources to various users to execute parallel applications. In addition, these centers provide an interactive application interface called scheduler that users often take advantage of when they submit jobs. In general, a job scheduler determines when and where to execute a job while given a set of computing resources and a stream of parallel jobs. In a standard working model, when a parallel job arrives in the system, the scheduler tries to allocate the required number of processors for the job’s runtime and starts the job immediately, if processors are available. If the requested processors are currently unavailable, the job
is queued and scheduled to start at a later time. The most common metrics evaluated include system metrics such as utilization, throughput [35, 20] and user metrics such as turnaround time, wait time, [44, 62, 22, 37, 60, 43]. The typical charging model is based on the amount of total resources used \((resources \times runtime)\) by any job.

While the scheduler provides the basic service of mapping jobs to the required number of processors, there are some highly desirable features such as fairness and QoS (Quality of Service) that the scheduler can support. Recently, Sabin [68, 66, 67, 65] extensively investigated the fairness aspect of job scheduling. On the other hand, this dissertation exclusively concentrates on the QoS aspect of parallel job scheduling that signifies the measurement of service quality offered to the user. Although QoS has received much attention as a field of extensive research in the networking community, little research [77, 6] has been done on QoS in parallel job scheduling. There has been limited work [73, 6] toward offering service differentiation, another form of QoS, for different classes of jobs. In addition, there has been some recent research [71, 76, 77] done to provide the deadline guarantees in a time-shared system [32].

However, no work has addressed the provision of Quality of Service (QoS) guarantees in parallel job scheduling for a space-shared system [32]. Using the current job schedulers, for instance, a user cannot specify the deadline by which a job should be completed, although the user would pay more for this guarantee. Such hard QoS capabilities are useful in several cases, such as a situation in which a scientist submits a job before leaving work in the evening and requests an 8:00 am deadline for the job’s completion the following day.

In addition, QoS support in the form of service differentiation is another quality of job scheduling desired by the end-users. For example, sometimes two jobs with identical
attributes are requested for scheduling in a system at the same time. The only difference between these two jobs is the urgency sought: one job is very urgent and the other one is not. It is desirable that the urgent job gets better service in the form of acceptance and early completion. There are two basic ways of providing service differentiation: absolute service differentiation (ASD) and proportional service differentiation (PSD). In ASD, the jobs from the higher priority would always execute first resulting potential starvation for lower priority jobs. In PSD, the higher priority job would get preferential treatment over the lower priority job as long as the predefined service ratio is maintained. Although PSD concepts received substantial attention [50, 49, 52, 51, 25] in the networking community, to the best of our knowledge, there is no research performed regarding the provision of PSD in job scheduling.

While hard deadline guarantees and service differentiation are appealing QoS attributes from the user’s perspective, overall revenue maximization with the optimal management of resources is the key motivational factor from some providers’ viewpoints. Revenue computation primarily requires a suitable QoS-aware charging or cost model. But most of the existing charging models that depend only on the resource usage in non-QoS scheduling cannot serve the purpose of the QoS-aware system. Once the appropriate charging model is identified, determining some effective techniques to further improve the system revenue is highly expected. In summary, we study the overall issues of providing QoS for job scheduling in terms of two related aspects, which can be decoupled as follows:

• **QoS-aware job scheduling:** A QoS-aware scheduler needs to provide the hard deadline guarantees for supercomputer center users without overly penalizing other common performance metrics. Although deadline-based scheduling has
been a topic of much research in the real-time community, it has not been addressed in the context of parallel job scheduling. Likewise, service differentiation gets extensive attention in the networking arena, but limited attention in job scheduling.

- **QoS-aware revenue management**: The benefit of a QoS-aware scheme can be evaluated in terms of hard currency, the satisfaction offered to the customer, or both. Therefore, a good scheduler without associated charging models and built-in revenue maximization techniques might be less appealing to the service provider. There is no current research that deals with the revenue aspect of QoS-aware scheduling.

### 1.2 Proposed Solutions

We first study the issues associated with providing the deadline guarantees in a QoS-aware scheduler, as there is no such existing scheme. Initially, we consider the adaptations of different existing algorithms like Advanced Reservation, Slack-based algorithm [73] and the Real-time algorithm [64] to provide the deadline guarantees. But, since all the schemes are originally designed to accomplish diverse objectives, these modified schemes are not perfectly suited for providing QoS guarantees. Therefore, we put forward a new scheme termed as *QoPS (QoS for Parallel Job Scheduling)* that concentrates solely on parallel job scheduling to exploit the inherent characteristics of the problem. QoPS basically implements an admission mechanism for incoming jobs, attempting various schedule rearrangements of previously admitted jobs to make a decision as to whether or not the requested deadline is achievable without violating the deadlines provided for the other admitted jobs.
Secondly, we study the issues of providing the proportional service differentiation (PSD) service to different priorities of jobs through scheduler. A PSD-aware scheduler essentially ensures when a higher priority job is relatively over-served, the lower priority job will get the preference in scheduling over the higher priority job. We address the obvious challenges of providing PSD service in job scheduling by first specifying a pioneer PSD-aware model. Then, we design three new schemes to study PSD, ensuring minimal impact on other important metrics.

In addition to hard QoS service, we also address the revenue aspect of such QoS-aware scheduling by evaluating two charging models independently derived from the perspectives of provider and user. First, we propose a provider-centric charging model founded on the notion that the quicker the response time sought, the larger the charge should be. The charge is generally a function of many factors, including but not limited to the resources used and the load on the system. We examine two separate charging components associated with the resource usage and QoS guarantees. The resource usage relies mainly on the amount of resources used and the duration of the service. The QoS part of the charge constituents essentially depends on the flexibility of the requested deadline.

To further improve revenue, we study the user characteristics in which users have different degrees of flexibility with respect to their requested deadlines. We also enhance the basic QoPS scheme to improve the negotiation capability for the initially unaccepted jobs. Accordingly, the enhanced mechanism labeled as $FQoPS$ ($\text{Feedback Based QoPS}$) presents a new deadline to the user when the job fails to meet its schedule within the requested deadline. We further extend $FQoPS$ to examine revenue maximization by introducing two new enhancements. The first extension is accomplished by introducing
slack in the offered deadline to the end user. We then incorporate the Kill-and-Restart mechanism, in which a running job can be killed to enable a new job to be accepted, into the Slack-based FQoPS scheduling that some supercomputer centers support. Afterward, the terminated job is restarted from the beginning. We evaluate whether such a mechanism can be utilized to improve the overall profit of the supercomputer centers in the QoS-based scheduling context.

Secondly, as a user-centric approach, we adopt a market-based charging model originally proposed by Chun et. al. [21] and Irwin et. al. [39] for sequential jobs directing the non-QoS scheduler. In this model, users can specify the maximum price that they are willing to pay and a linearly decaying cost function over time. At first, we investigate the revenue maximization technique for the non-QoS scheduler by proposing a novel heuristic backed by theoretical reasoning. More significantly, we prove the optimality of our approach in a simplified scenario involving a uni-processor system and an offline batch of jobs. Then, we propose sufficient conditions which when true, guarantee optimality, for an online stream of jobs on a uni-processor system. Finally, we apply our proposed scheduling scheme in a generic multiprocessor system with parallel jobs.

After that, we broaden our study to QoS-aware system in user-centric revenue model through the enhancement of QoPS scheduler. While the basic QoPS admits the job after checking the deadline constraints of all the accepted jobs, it is totally unaware of the jobs’ revenue aspects. But, as mentioned earlier, the service provider is eager to maximize the overall system revenue. So, we study the issues related to revenue maximization and ultimately propose new schemes. In addition to deadline constraints, the new scheme should consider the possible revenue gains during job admission. However, it is particularly difficult to determine how much revenue gain is sufficient for a job to
be accepted that would ultimately maximize the revenue, because admitting a job could sometimes cause the rejection of more profitable jobs arriving later. With regard to this, we introduce the notion of opportunity cost related to the job acceptance.

Accordingly, a job is admitted if the revenue gain expected from the new job’s acceptance exceeds the corresponding opportunity cost. Ideally, the opportunity cost of accepting a job can be evaluated as the difference between the maximum revenue earned from the two independent schedules of the whole workload: the schedule where the new job is definitely accepted and the schedule where the new job is certainly rejected. But, in an online system where jobs arrive dynamically, the information about the future jobs is nearly impossible to know in advance. Therefore the exact estimation of opportunity cost is impossible.

To address these issues regarding opportunity cost, we extensively analyze the impact of opportunity cost as related to different system factors, and eventually propose a history-based predictive technique to estimate the opportunity cost. Essentially, we propose an extension to QoPS, termed VQoPS (Value-aware QoPS), in which the value of the job is considered during job acceptance. With VQoPS, we also study the overall revenue earned with respect to various statically assumed opportunity costs. However, as we will see in a later section, no single statically assumed opportunity cost performs superiorly in all possible circumstances. Therefore, to deal with this problem, we introduce Dynamic Value-aware QoPS (DVQoPS), which is a self-learning variant of VQoPS, to analyze the past jobs and predict the opportunity costs for future jobs.

We present a detailed analysis of different proposed schemes with simulation based on different real workloads of jobs gathered from Feitelson’s workload archive [29]. At first, our results substantiate QoPS as the best quality admission control mechanism
compared with different adapted schemes. Furthermore, QoPS achieves better or comparable performance with respect to all other widely used metrics. The FQoPS-based simulation result exhibits the impact on both QoS charge and resource charge for different assumed user behaviors. It also shows how the introduction of artificial slack and the adoption of Kill-and-Restart affect the system revenue. Our results collected from non-QoS schemes in user-centric model demonstrate that the proposed heuristic based scheme provides significantly higher revenue as compared to existing schemes. The data obtained from VQoPS simulation elucidates the impact of opportunity cost on different dynamic system behaviors in QoS-aware scheduling. In addition, the simulation result for the DVQoPS scheme shows that it can provide several factors of higher revenue compared with QoPS. Furthermore, we present the detailed analysis of our proposed PSD-aware schemes. We can summarize the main contributions of the current dissertations as follows:

- **Job scheduling scheme with QoS guarantees**: We propose a pioneer scheme to provide the QoS guarantees in the form of response time for the consumer without overly penalizing other common metrics.

- **PSD-aware schemes**: We address the obvious challenges of providing proportional service differentiation (PSD) service in job scheduling by first specifying a pioneer PSD-aware model. Then we design three new schemes to study PSD, ensuring minimal impact on other important metrics.

- **Propose a new provider-centric charging model**: We suggest a novel and effective charging model, based on the urgency sought in the response time of a job. In this model, the provider decides the price by splitting it into two distinct components: resource charge and QoS charge.
• **Revenue maximization in provider-centric model:** Using the provider-centric pricing model, we explore the influence of user behavior. We also study the effect in the revenue constituents by proposing two approaches: incorporating the artificial slack in user deadlines and utilizing the Kill-and-Restart mechanism.

• **Revenue maximization in user-centric model:** We propose a new scheduling heuristic to improve the system revenue in a user-centric model with non-QoS aware scheduler. More significantly, we prove the optimality of our approach in a simplified scenario involving a uni-processor system and an offline batch of jobs. Then, we propose sufficient conditions which when true, guarantee optimality, for an online stream of jobs on a uni-processor system. Finally, we apply our proposed scheduling scheme in a generic multiprocessor system with parallel jobs.

• **Revenue management considering opportunity cost:** We perform extensive analysis regarding the impact of opportunity cost on revenue maximization using the user-centric charging model. Moreover, we present a history-based predictive technique to further increase the overall system revenue.
CHAPTER 2

BACKGROUND AND RELATED WORKS

2.1 Fundamentals of Parallel Job Scheduling

Scheduling, in general, is a process of assigning or mapping tasks to a set of resources for a period of time according to some organization-specific policy [30]. It is heavily used in many real world fields such as production and computing processes. In parallel computing system, single multiprocessor parallel job can execute on multiple processors concurrently while data communication occurs among the processors as needed. In this context, the scheduling process assigns the parallel jobs into the required number of processors and, eventually, determines the start time of job execution. In other words, parallel job scheduling arranges the contending jobs in a two-dimensional division of processors, both in time and in space. An efficient job scheduler experiences the challenges of satisfying users and system providers at the same time. In this section, we present the high-level elementary concepts regarding some prevalent issues of parallel job scheduling.
2.1.1 Parallel Computing System

A parallel computing system is a computer composed of multiple processors to execute the same task in parallel. Since, large-scale applications demand huge amounts of computing resources to fulfill their functionalities in an effective way, a parallel system can help to achieve this goal by providing a framework to complete an application in a shorter time, and make use of larger aggregate cache, physical and secondary memory [32]. In essence, a parallel computing system is a queuing system in which the job arrives, may wait for service, executes and leaves the system. The following classification [34] is proposed for this type of queuing system which is also shown in Figure 2.1:

**Offline model:** In this model, all the jobs and their resource requirements are available in the system at the beginning. No job arrives after the system starts processing the jobs. In the offline model, the scheduler can arrange the job more effectively as there is no dynamic behavior of the system. In addition, the performance of the offline model can be predicted by using different analytical methods [34]. However, this type of system is less frequent in the real world.

**Online model:** In this model, jobs arrive over a period of time. In other words, jobs can arrive at any time when the system is active [34]. The scheduler requires arranging...
the jobs in real time without any knowledge of future arrivals. The online model system can be subdivided [34] into open and closed system.

*Closed Model:* Although, jobs can arrive dynamically to a closed system, there is a fixed set of jobs that can arrive to the system. Moreover, there is an upper bound on the maximum number of jobs that can arrive to a closed system. In this model, arrival of a new job is linked with the departure of the previous job. While predicting the behavior of a closed system using analytical methods is very difficult, it is possible with some restricted assumptions [34].

*Open Model:* In an open model system, jobs can arrive at any time to the system and there is no bound on the maximum number of jobs. In this endless stream of job arrival, there is no association between the arrival of a new job and the departure of some other job. In other words, the arrivals and attributes of every job are independent and dynamic. This model represents a more realistic picture of the current parallel computing system. Due to the dynamic nature of this type of system, it is very complex and problematic to analytically model this system [34].

### 2.1.2 Parallel Job

A parallel job is a single application that is composed of many units of work running in multiple processors concurrently [32]. Generally, each unit of work runs on different processors independently but synchronizes and exchanges the required data as necessary. The following classification of parallel job is proposed [30, 35, 33]:

**Rigid Job:** The job that requires a fixed number of dedicated processors throughout its execution is called a rigid job. The required number of processors is specified by some external entity (i.e. the user) other than the job scheduler. If the scheduler accepts
this job, it has to confirm the availability of that predefined number of processors during the job’s runtime. In processor-time space, a rigid job is represented as a fixed rectangle (Figure 2.2(a)) [30, 35].

**Moldable Job:** The moldable job allows the number of processors to be specified at any time before the start of the job’s execution. However, once the job is started, the required number of processors can’t be changed. The scheduler usually determines the number of processors depending on the system load at that instance and some job-specific constraints. In contrast to rigid jobs, a moldable job can have different rectangular shapes (shown in Figure 2.2(b)) determined just before the start of the job’s execution [30].

**Evolving Job:** The job that changes the requirement of processors during its execution is known as an evolving job. This type of job passes through different phases during execution with requirements of different numbers of processors at each phase. At the start of each phase, the evolving job acquires the necessary number of processors for that phase. At the end of that phase, it relinquishes all the processors. The user, who submits the evolving job, specifies the required number of processors for different phases. The scheduler has less flexibility, as it cannot change these requirements during scheduling. Figure 2.2(c) shows one such representation in processor-time space [30].
Malleable job: The job that permits the change in the required number of processors during its execution is called a malleable job. The scheduler has more flexibility with this type of job. The scheduler can start a malleable job immediately with the processors available. As soon as a job finishes and releases some processors, the scheduler assigns those processors to running jobs. Likewise, if there is not an available processor to start a new job when it arrives in the system, the scheduler can preempt some running jobs and assign the processor to the new job to start. While the user of the evolving job specifies the processor requirement for different phases of execution, the scheduler, in contrast, determines the processor requirements for the malleable job. The scheduler has greater flexibility in scheduling the malleable job. Like the evolving job, the malleable job has the same presentation in Figure 2.2(c), though conceptually they are distinct [30].

2.1.3 Job Preemption

Job preemption allows the job scheduler to suspend a job (low priority) to allow another job (high priority) to use the resources utilized by the first job. In this case, the suspended job is restarted as soon as the required resources become available. There are different types of job preemption [35] available upon which the job-scheduling algorithm depends very much.

No preemption: In this scheme, the job is never suspended once started. The job holds all the resources including the processors throughout its execution period [35]. The system that supports non-preemptive scheduling places no extra overhead on the underlying system. Most of the current schedulers do not support preemption in job scheduling. This scheme has less flexibility but puts less burden on the scheduler.
**Local Preemption:** In this method, each thread of a job can be preempted at any time but each thread can be restarted only in the same processor. This scheme provides some flexibility to the scheduler and doesn’t need any data movements across the processors [35]. The underlying system that supports the preemption needs to handle all the details of suspending and restarting the job in a seamless manner. This is the easiest preemption to support.

**Migratable Preemption:** According to Migratable preemption, each thread of a job can be preempted at any time and can be resumed in any processor later on [35]. This preemption requires a lot of data movements across the processors, adding extra overhead.

**Gang Preemption:** In this model, all the threads of a job can be suspended together at any time. Later on, when resources become available, the threads are resumed simultaneously. Gang preemption can work with or without migration [35].

### 2.1.4 Backfilling

Parallel job scheduling is generally a space-sharing problem in a two-dimensional chart, with time along the horizontal axis and the number of processors along the vertical axis. In this representation, a rectangle symbolizes a job in which the required number of processors and the user-estimated runtime of the job determine the height and width, respectively, of the rectangle. This representation facilitates the visualization of the different concepts of job scheduling, including the widely used notion of backfilling. In this section, we describe the importance of backfilling and its different commonly used forms.
In general, most of the schedulers adopt the First Come First Served (FCFS) policy as a simple but fair strategy for scheduling jobs [69, 28]. According to the policy, jobs are scheduled in the order of their arrival. When a new job arrives, the scheduler starts the job immediately if the requested number of processors is available and no other jobs are waiting. Otherwise, the new job has to wait in a reservation queue till the requested number of processors becomes free and no other earlier jobs are in the reservation queue. To maintain the FCFS policy, if any single job is waiting in a reservation queue, all the subsequently arriving jobs have to wait until the first waiting job gets its required resources. Although FCFS policy provides fairness in scheduling, this policy may lead to lower system utilization [28, 46]. Backfilling [53, 47] is proposed to improve the system utilization without violating the FCFS-based fairness. Backfilling allows the small jobs to move ahead and run on processors that would otherwise remain idle [53]. Backfilling first identifies the holes (unoccupied space) in the 2D chart of the schedule. At the same time, it selects small jobs that fit the holes and then moves the jobs forward into the holes in an effective manner. Backfilling can be categorized [28] using two criteria: depth of reservations and queuing policy. Depending on the number of reservations, the following classification is proposed [28]:

**Aggressive or EASY:** EASY (Extensible Argonne Scheduling System) backfill backfill is the original of all backfill technique variants [53, 28]. In EASY, only the job at the head of the queue has a reservation, and the jobs arriving later do not get any reservation. A smaller job is permitted to move ahead as long as the reservation of the first job is not violated.

**Conservative:** In this type of backfill, all the jobs in the system get their respective reservations in the schedule [60, 28]. A smaller job is allowed to move forward in
the schedule as long as it does not delay any previously reserved jobs. Conservative backfill generally needs more computations from the scheduler because it requires the maintenance of the reservations for all the jobs.

**No guarantee:** In no guarantee backfill, there is no reservation provided to any job. A smaller job is allowed to move ahead without any constraints from earlier jobs. This technique is rarely used in production because it is very much prone to starvation.

During backfilling, jobs in the reservation queue are sorted using different queuing policies [28]. Backfill chooses the job to fill up a hole according to the sorted reservation queue. Depending on that queuing policy, backfilling is classified as follows [78]:

**First come First Serve (FCFS):** In this policy, the job that will be backfilled is selected by ordering the jobs according to their arrival time [53]. Initially, the scheduler tries the first job in the list to find any hole in the existing schedule that fits the job. Then, the subsequent jobs in the ordered list also attempt to get a respective reservation in the updated schedule. Although it is a very basic and simple policy that provides fairness to the jobs, it performs worse in other system and user metrics.

**Shortest Job First (SJF):** According to this strategy, jobs in the queue are ordered according to their estimated runtime [19, 28]. The smaller jobs get preference in backfilling compared to the longer jobs. This policy improves some performance metrics, like slowdown, that are sensitive to the short jobs. However, this policy can lead to starvation for the long jobs.

**Best Fit:** In this approach, jobs are sorted in decreasing order according to their requested number of processors. The job that fits the maximum number of empty processors is given the higher priority in backfilling. Although the wider jobs get the benefit of early completion, the narrow jobs may experience the starvation.
**Worst Fit:** In this policy, jobs are arranged according to the increasing order of their requested number of processors. The narrower jobs get priority over the wider jobs, which eventually can lead to starvation for the wider jobs. It is the complete opposite of the Best Fit policy.

**Largest expansion Factor (LXF):** According to this policy, the jobs are ordered based on the ratio of the expected response time to the estimated runtime of the job.

### 2.1.5 Workload

When a new job schedule scheme is proposed, the extensive evaluation of the proposed design is essential prior to any real-world implementation. In parallel job scheduling, this assessment profoundly relies on either the analytic technique or the simulation approach. Formal analysis can provide a concise description of the system behavior by presenting a mathematical or analytical model. But, since there are so many dynamic and seemingly confusing attributes in parallel system, the actual mathematical representation of such system is a very challenging task. Instead, the simulation approach is an effective method of evaluation because it can exhibit the accurate and real-world characteristics of the actual system. Most parallel job scheduling schemes adopt the simulation as a medium for evaluating the respective schemes. In essence, the quality of the simulation results depends on two factors: the selection of representative workloads [36, 18, 15, 34, 17, 54] and appropriate metrics[18, 34]. In this section, we discuss the various aspects of workloads and in the next section 2.1.6, we describe several widely used metrics.
Generally, the workload is a collection of parallel jobs presented as a plain ASCII file. Each line in the workload corresponds to a job. Each job has two major components [34]: job arrivals and job structure, which defines the job’s attributes (e.g. the job’s runtime, number of required processors etc.). Workload is either collected from some real supercomputer centers or generated synthetically. Accordingly, the workload can be classified in either of the two following categories:

**Workload log:** almost all parallel supercomputer centers maintain accounting logs for all the job-specific information including job arrival, its various internal attributes and features [18]. Since these workloads from production systems represent an accurate portrayal of the system, running the simulation using this input gives a correct assessment of a proposed scheduler. Based on these real jobs, we also can generate different workload traces that are suitable for specific needs. For example, if there is a possibility that the targeted new system will experience a high load compared to the existing system, we can expand the load by using various techniques but keeping the basic trace characteristics unchanged. There are a lot of such workloads compiled in [29, 18], including SDSC paragon [75], CTC SP2 [38], NASA-Ames iPSC/860 [27] and LANL CM-5 [31].

**Workload model:** Generation of the workload models [18] is profoundly based on the rigorous statistical analysis of the real workload logs. Using the analytical findings, a comparable synthetic workload can be generated with more control of modifying the features at will [26]. While the workload model offers liberties to the modeler to manipulate the expected features, it is prone to the inclusion (exclusion) of the characteristics that are (not) present in the actual system. There are few utilities [23, 55] that are used to generate various workload models depending on specific needs.
2.1.6 Metrics

The usefulness of the results obtained from simulation or analysis depends on the metrics that are used. The metrics serve as the objective function of the scheduler, whose goal is to optimize its value. Since the measurements using different metrics may lead to conflicting conclusions, it is important to select the pertinent metric(s) judiciously in accordance with the system requirements [32]. In this section, we briefly discuss the widely used metrics in job scheduling considering a parallel system of N jobs. To facilitate the explanation of the different metrics, we use the following notations to describe the attributes of a parallel job \( J_i \): arrival time of job \( J_i \) \((T^a_i)\), start time of job \( J_i \) \((T^s_i)\), user-estimated execution time of job \( J_i \) in non-preemptive system \((T^e_i)\), completion (finish) time of job \( J_i \) \((T^f_i)\), and the number of processors used by job \( J_i \) \((N_i)\).

**Response Time:** In parallel job scheduling, the response time of a job is defined as the time elapsed between the job’s arrival and the job’s completion. Response time is a suitable metric for online open system. For better performance, the response time of a job should be less. However, the performance using individual metric (e.g. response time) often does not signify the effectiveness of a scheme correctly. In this case, the aggregate metrics such as average response time can be considered a suitable metric to evaluate. Average response time specifies, on average, how quickly the system completed a job.

Mathematically, Response Time (RT) and Average Response Time (ART) can be defined as follows:

\[
\text{RT of job } J_i = (T^f_i - T^a_i)
\]

\[
\text{ART} = \frac{\sum_{i=1}^{N}(T^f_i - T^a_i)}{N}
\]
**Slowdown:** Job slowdown measures how much slower the system appears to the user compared to a dedicated machine. Slowdown of a job is evaluated as the ratio of the response time to the runtime of a job. Slowdown for job $J_i$ can be expressed as:

$$SD_i = \frac{\text{Response Time}_i}{\text{Runtime}_i} = \frac{(T_f^i - T_a^i)}{T_e^i}$$

As shown in the above equation, very short jobs play an important role in the slowdown metric and thus can misrepresent the result. Therefore, to reduce the statistical impact of very short jobs, the new metric Bounded Slowdown (BSD) is introduced; it considers 10 seconds as a lower bound of execution time. Besides, Average Bounded Slowdown (ABSD) is a better metric to evaluate the system wide performance.

$$BSD = \frac{\text{MAX}(T_f^i - T_a^i, 10)}{\text{MAX}(T_e^i, 10)}$$

$$ABSD = \frac{1}{N} \sum_{i=1}^{N} \frac{\text{MAX}(T_f^i - T_a^i, 10)}{\text{MAX}(T_e^i, 10)}$$

**Utilization:** Utilization is the ratio of resources used by the jobs to the resources offered. In job scheduling, resources are generally evaluated by multiplying the number of processors by the runtime of a job. A lower value of utilization means the system is idle most of the time. A system provider usually expects a higher value of utilization. The maximum possible value of utilization can be 1. Let $M$ be the total number of processors, and $N$ is the number of jobs in the system, where job $N$ is the last job to complete the execution. Also, assume that the first job arrives at time $t = 0$. In this situation, utilization ($U$) of a system can be defined as

$$U = \frac{\text{Used Processor-seconds}}{\text{Offered Processor-seconds}} = \frac{\sum_{i=1}^{N} (N_i T_e^i)}{MT_m^f}$$

**Makespan:** Makespan is defined as the time taken to complete the execution of the whole workload. Usually, makespan is a suitable metric in offline scheduling [32]. In some system, the objective of a scheduler is to minimize the total makespan. Assuming
that the first job started as time $t=0$, makespan can be determined by finding the completion time of the last job. It can be defined as

$$\text{Makespan} = \text{Latest Completion Time} - \text{Earliest Start Time} = \max_{1 \leq i \leq n}(T_i^f)$$

### 2.2 Related Works

Several job schedulers such as Portable Batch Systems (PBS) [8], Moab [5], Load Sharing Facility (LSF) [4], Sun Grid Engine (SGE) [13], etc. have been deployed at shared-resource supercomputer centers to schedule parallel jobs. These schedulers primarily focus on performance by improving utilization, throughput, average turnaround time [35, 20], etc. Even though providing QoS service is a very desirable feature from the resource user’s viewpoint, it has received very little attention from both the industry [6, 10, 3] and the research community [73, 71] of parallel system. In this section, we review the QoS-related works performed in job scheduling or some other related fields that can be loosely divided into three broad categories: studies that provide deadline guarantees, works that offer service differentiation and studies that address revenue aspects associated with two previous services. For convenience, we discuss the respective literatures of these categories in separate sub-sections.

#### 2.2.1 Deadline-based schemes

There has been a substantial amount of research [64, 56, 59, 16] accomplished in the real-time community involving deadline guarantees. In a real-time system, where a timely completion of the task is crucial, the task is given the hard-guarantees in the upper bound of task completion. One of the significant studies [64] proposed by Ramamritham, et. al, is an approach to scheduling uni-processor tasks with hard real-time deadlines on multi-processor systems, and is evaluated in a static scheduling scenario.
The algorithm tries to meet the specified deadline of tasks by using various heuristic functions. The tasks are characterized by worst-case computation times, deadlines and resource requirements. The scheme starts with an empty partial schedule, and extends in each step of the search the current partial schedule with one of the tasks yet to be scheduled. The heuristic functions used in the algorithm actively direct the search for a feasible schedule, i.e., they help to choose the task that extends the current partial schedule efficiently. Earliest Deadline First (EDF) and Least Laxity First (LSF) are examples of two such heuristic functions. While, like Ramamritham [64], our work also deals with deadline constraints of jobs, we mainly consider the dynamically arriving multiprocessor jobs in parallel computing system.

Recently, Buyya et. al., [71, 77, 76] has carried out a considerable amount of research to support deadline-based scheduling. The studies consider the time-shared [32] system model where multiple jobs can be scheduled and time sliced on a single node. Buyya et. al., [71, 77, 76], primarily, check the admissibility of a new job by considering whether there are sufficient shares of CPU cycles available in any node at any instance. The work presented in this dissertation, however, addresses the space-shared [32] system that is applicable to many supercomputing centers. In the space-shared system, each processor is exclusively dedicated to a specific job for a fixed duration.

2.2.2 Service Differentiation studies

The service differentiation concept has been widely investigated in the computer networking community [80, 51, 52]. In parallel job scheduling, a few studies have been done with very limited analysis. Only a few supercomputer centers provide differentiated service among multiple classes of jobs utilizing a very simple concept. According
to this simple approach, the scheduler provides best effort relative prioritization for different classes of jobs. Such priority is assigned statically to the jobs, in which jobs from a group of users might be given a higher priority compared to others. The NERSC computing center [6] is an example of an environment that uses such a notion in their scheduler. Specifically, the NERSC center offers three different queues in which each queue has its respective static priorities and charges. Depending on the urgency sought, users can submit their jobs in any of the three queues: normal queue with usual charge, high priority queue which double the charge, and low priority queue with half the usual charge. Typically, jobs in the high priority queue gets priority over the normal queue, until some threshold on the number of serviced jobs is exceeded. Although the NERSC center implements a very basic requirement with a simple design, the service differentiation needs a rather thorough investigation in diversified scenarios with an impact analysis on various basic metrics.

Feitelson, et. al, [73] propose the slack-based (SB) algorithm essentially to improve system utilization and user response time. The main idea of the algorithm is to allow a scheduler calculated slack or laxity for each job. During backfilling 2.1.4, the scheduler assures that no job is delayed more than its assigned slack. The proposed scheduler also considers priority during dynamic slack assignments of any job. There are three different priorities determined by three different entities. the system administrator statically assigns the administrator priority to a group of users depending on the importance of the group. Each user can statically specify the user priority to a job depending on the importance of the specific job. Additionally, the scheduler dynamically calculates the scheduler priority depending on the queue waiting time of the job. For example, if any job is waiting longer than the expected average wait time, the scheduler increases the
value of scheduler priority, making the job critical. SB [73] algorithm examines different ways of combining these three priorities into an integrated priority and that ultimately helps to determine the slack of the job. Since the main intention of the proposed scheme is to improve the utilization and response time, the scheme does not reasonably provide any direct service differentiation to the user.

2.2.3 Revenue Management works

Typically, the service provider calculates the revenue by using either the provider-selected charging model or the user-specific charging model. Most supercomputer centers [10, 1, 3] adopt the provider-centric approach, in which a user is charged in proportion to the resource usages. In most of the cases, the basic resource is the processor; the charge is proportional to the product of the required number of processors and the run time of a job. Some supercomputer centers [10, 3] that provide multiple queues for different levels of services determine the charges depending on the resource used and quality of services sought. In this model, the charge is generally proportional to 

\[(\text{processor} \times \text{runtime} \times \text{queuecharge})\]. While these models work well in a basic system, the system that offers deadline guarantees cannot use these models.

Alternatively, there has been some research [21, 39, 63] conducted to provide a user-centric market-based approach for the system with non-QoS service. According to the user-centric approach, different users might have different goals and preferences, and a user expresses his or her desire for service in a common way (e.g., through currency). The most common market-based model follows the auction-based resource allocation mechanism [74, 72] that has three major entities: users or buyers, system providers or sellers and the resources to be sold. In job scheduling, the resources are mainly
computing resources like processors, memory, storages etc. The user wants to allocate the processor(s) for a specific duration and is willing to pay a certain value for the execution of the job. The system provider is interested selling the resources to the user with an intention to maximize its overall profit. The auction process, which is generally proposed by the system provider, considers the value or bid of all contending users and ultimately awards that to the highest bidder.

Wladspurger, et. al., [74] are the pioneers in using the market-based microeconomic approach for batch scheduling. They utilized the auction process to choose the winner from the bids of different users. Stoics, et. al., [72] also propose an auction-based microeconomic approach for parallel job scheduling. In this scheme, every user has a savings account where he or she receives funds for buying resources for jobs. Also, the user creates an independent expense account for every job and starts transferring funds from his or her savings to a job’s expense account. The rate of this fund transfer determines the priority of the job that ultimately plays a vital role in the auction process.

Two recent publications [21, 39] also have addressed the market-based approach of value-based parallel job scheduling. Both studies rely on a per-job specific utility or value function that provides an explicit mapping of service quality to value. Generally, the value function is a piece-wise linear function that decays as a function of the job completion time. This type of function has a magnitude that shows the value of the job and a rate of decay that reflects the urgency or sensitivity to delay. The fundamental idea of this system is that the user submits the job with a value function along with other job characteristics. Then, the scheduler decides how to schedule the job using the job information and current state of the system. Culler, et. al., [21] adopt the user-centric performance metrics instead of system-centric metrics to evaluate the overall
system performance. They recommend an aggregate utility to measure the satisfaction of users with the resource allocation process. For job selection, the proposed scheme implements a heuristic approach where the job with high value per unit running time gets the preference. Chase, et. al., [39] propose an enhancement to the Culler, et. al., [21] approach where different improved heuristics are utilized. Chase, et. al., [39] also consider the risk of accepting or rejecting a job due to future uncertainty. While a market-based user-centric charging model is a very good concept for computational economics, none of the studies has addressed this concept in the QoS-aware system where services like deadline guarantees and service differentiation are very important.
CHAPTER 3

SYSTEM MODEL AND EVALUATION BASICS

In this chapter, we describe several essential issues that are frequently utilized in subsequent experiments. As there are too many possible variants in the parallel system, in Section 3.1 we strive to portray a widely used system model that is adopted for our research. In Section 3.2, we describe a general set of notations to mathematically represent the different entities of a system. Also, we illustrate a high-level experimental setup adopted to evaluate our proposed schemes. in Section 3.3.

3.1 System Model

Generally, a parallel system can have different properties intended for various purposes resulting in many possible models. For our research, we adopt a representative model that is most commonly used in supercomputer centers. In this section, we describe this model with some reasonable assumptions.

A multi-processor system (M) can be represented by a set of n processors or machines, \( M = \{M_1, M_2, \ldots, M_n\} \) where \( M_i, 1 \leq i \leq n \) is a processor connected with other processors through a specific networking setup. To simplify the system model without loss of generality, it is assumed that all processors are homogeneous in nature where each processor has identical capabilities such as same speed, memory, disk, etc. In
the proposed model, a parallel job arrives to the system independently and dynamically with different attributes including the required number of processors, estimated runtime, deadline, etc.

![Figure 3.1: A simple model of parallel job scheduling](image)

The scheduler maintains the required system information, such as, available (free) and used time slots of each processor. Using these processor information, when a job arrives to the system, the scheduler determines whether the job can be accepted to satisfy its requirements. If the scheduler doesn’t find a feasible schedule, it rejects the job and informs the user accordingly. In the case of acceptance, the scheduler either starts the job execution immediately or stores the jobs in the reservation queue. The scheduler periodically dispatches the job, whose scheduled start time matches the current time, from the reservation queue to the pre-allocated processor(s) to execute. The job that has been dispatched to the processor(s) is executed exclusively or non-preemptively until its completion.
Figure 3.1 shows an instance of such a model that we adopt in our current research. In the figure, due to resource unavailability, four jobs are waiting in reservation queue whereas two multi-processor jobs are running. Depending on the resource requirements of the new job $J_7$, the new job could be placed in reservation queue to be started later or in the running queue to be started immediately. For instance, if $J_7$ needs one processor, the job is placed running queue and it starts the execution in the free processors ($M_5$).

In summary, we can enumerate the key aspects of the system model that is accepted for our research:

- Processors are homogeneous with equal capabilities.
- All jobs are rigid jobs.
- Supports both serial and parallel jobs.
- Jobs arrive dynamically (online systems).
- Jobs are independent.
- Space-shared systems where a job solely executes on required number of processors without sharing the resources.
- Supports only non-preemptive scheduling.

### 3.2 Notations

For analytical purpose, we represent a job $J_i$ as a set of rational parameters, e.g.,

$$J_i = \{ a_i, r_i, d_i, p_i, v_i \},$$

where $a_i$ is the arrival time, $r_i$ specifies the estimated runtime, $d_i$ denotes the deadline, $p_i$ is the number of required processors and $v_i$ denotes the value function or price that the user is willing to pay. A reservation of an accepted job $J_i$ can be formulated as

$$\Pi_i = \{ t_i, \{ M_{j,1}, M_{k,2}, ..., M_{q,p_i} \} \}$$

where $t_i$ denotes the start time of job $J_i$ on all $p_i$ processors ($M_{j,1}, M_{k,2}, ..., M_{q,p_i}$) that
are exclusively allocated from time $t_i$ to completion time $t_i + r_i$. The schedule $S$ of system $M$ is a set of reservations provided to all the active jobs at any instant, and the schedule $S$ of $m$ number of jobs can be expressed as 

$$S = \{\Pi_1, \Pi_2, \ldots, \Pi_m\}$$

For the simplicity of illustration, we define a set of new notations for frequently used terminologies that are shown in Table 3.1.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>RunTime($J_i$)</td>
<td>Estimated runtime of job $J_i$ ($r_i$)</td>
</tr>
<tr>
<td>NumProc($J_i$)</td>
<td>Required number of processors for job $J_i$ ($p_i$)</td>
</tr>
<tr>
<td>Deadline($J_i$)</td>
<td>Deadline of job $J_i$ ($d_i$)</td>
</tr>
<tr>
<td>$ST(S, J_i)$</td>
<td>Start time of job $J_i$ in schedule $S$; $ST(S, J_i) = S[\Pi_i][t_i]$</td>
</tr>
<tr>
<td>$CT(S, J_i)$</td>
<td>Completion time of job $J_i$ in schedule $S$; $CT(S, J_i) = ST(S, J_i) + RunTime(J_i)$</td>
</tr>
<tr>
<td>$Value(S, J_i)$</td>
<td>Value of job $J_i$ in schedule $S$. It is important to note that the revenue of a job, in our model, is a time dependent value. It depends on completion time $CT(S, J_i)$ and the value function $v_i(t)$</td>
</tr>
</tbody>
</table>

Table 3.1: Important Notations

### 3.3 Evaluation Basics

When a new scheme is proposed in the research community, the new technique needs to be tested rigorously in various assumed scenarios before deploying the scheme in the production. Simulation is one such established method of evaluation particularly in the job scheduling area. While testing the same scheme into any production system could take many months, the evaluation of a scheme using simulation approach could be achieved within a few hours. Therefore, we accept the simulation-based assessment
approach as the foundation of our comparative studies of the schemes. Generally, there are three main steps associated with simulation-based experimentation that are shown in Figure 3.2. We illustrate the details of these three steps in this section.

### 3.3.1 Trace pre-processing

A simulator requires an input trace file consisting of a set of jobs that it simulates. We mainly use the real production workload from SDSC [11] and CTC [2] obtained from Feitelson’s archives [29]. Usually these input traces contain the runtime, number of nodes, and queue or arrival time for a set of jobs. Often, a trace file also includes some optional information that some schedulers may or may not need. For instance, user-expected runtime (or wall-clock limit) of a job is often important information that a user supplies during a job’s submission. There have been multiple studies showing the importance of user estimates [60, 48, 47]. Nevertheless, these workload traces lack some important information essential for the evaluation of our schemes. For example, our proposed schemes deal with various QoS-enabled services that require deadline, job class and offered revenue information, which are not available in production. Moreover, we evaluate our algorithms in different assumed loads in addition to the production load. Therefore, we cannot use the production traces without modification. The trace-preprocessing module includes those missing job attributes in a realistic manner and employs some pragmatic techniques for load expansion. In short, the trace pre-processing uses the production traces as input and converts those into our domain-friendly traces that are ultimately delivered to the simulator as input.
3.3.2 Simulator

Simulations are used to design and test emerging scheduling algorithms before development in a production system. Simulations provide an environment where varying scheduling algorithms can be compared in reproducible scenarios, providing comparable results. In addition, months of data can be simulated in minutes or hours, as compared to months or years it would require to run multiple algorithms in a live environment. Therefore, we developed an event-based simulator that takes data in the standard workload format version 2.0 [29]. The simulator then performs the necessary simulation using various schemes and creates corresponding output traces.

3.3.3 Output Post-processing

The main objective of our experiments is to analyze the impact of our proposed algorithms on system providers and users. This analysis is often accomplished through comparisons with other relevant schemes. Also, it is a real challenge to determine which
metrics to evaluate and how to present the results for easy understanding. The output trace generated by simulation contains data necessary to gather metrics and perform post-processing. Output post-processing primarily deals with this problem. In short, output post-processing includes the calculation of different applicable metrics and the presentation of results in graphs or tables.
CHAPTER 4

QOS GUARANTEES IN JOB SCHEDULING

4.1 Introduction

Numerous studies have focused on the problem of scheduling dynamically-arriving independent parallel jobs on a given set of resources. The metrics evaluated include system metrics such as system utilization, throughput [35, 20], etc. and users metrics such as turnaround time, wait time [44, 62, 22, 37, 60, 43], etc. There has also been some recent work in the direction of providing differentiated service to different classes of jobs using statically or dynamically calculated priorities [73, 6] assigned to jobs.

However, there has been no work addressing the provision of Quality of Service (QoS) in Parallel Job Scheduling. In the current job schedulers, the charge for a run is based on the resources used, but is unrelated to the responsiveness of the system. Thus, a 16-processor job that ran for one hour would be charged for 16 CPU-hours irrespective of whether the turn-around time were one hour or one day. Further, on most systems, even if a user is willing to pay more to get a quicker turn-around on an urgent job, there is no mechanism to facilitate that. Some systems, e.g. NERSC [6] offer different queues which have different costs and priorities: in addition to the normal priority queue, a high priority queue with double the usual charge, and a low priority queue with half the usual
charge. Jobs in the high priority queue get priority over the normal queue, until some threshold on the number of serviced jobs is exceeded. Such a system offers some choice to users, but does not provide any guarantee on the response time. It would be desirable to implement a charging model for a job with two components: one based on the actual resource usage, and another based on the responsiveness sought. Thus if two users with very similar jobs submit them at the same time, where one is urgent and the other is not, the urgent job could be provided quicker response time than the non-urgent job, but would be charged more.

We view the overall issue of providing QoS for job scheduling in terms of two related aspects, which however can be decoupled:

- **Cost Model for Jobs**: The quicker the response time sought, the larger should be the charge. The charge will generally be a function of many factors, including the resources used and the load on the system.

- **Job Scheduling with Response-time Guarantees**: If jobs are charged differently depending on the response time demanded by the user, the system must provide guarantees of completion time. Although deadline-based scheduling has been a topic of much research in the real-time research community, it has not been much addressed in the context of job scheduling.

In this chapter, we address the latter issue (Job Scheduling with Response-time Guarantees) by providing Quality of Service (QoS) in the response time given to the end-user in the form of guarantees in the completion time to the submitted independent parallel applications. We do not explicitly consider the cost model for jobs; the way deadlines are associated with jobs in our simulation studies is explained in the subsequent sections.
At this time, the following open questions arise:

• *How practical is a solution to this problem?*

• *What are the trade-offs involved in such a scheme compared to a non-deadline based scheme?*

• *How does the imposition of deadlines by a few jobs affect the average response time of jobs that do not impose any deadlines?*

• *Meeting deadlines for some jobs might result in starvation of other non-deadline jobs. Does making the scheme starvation free by providing artificial deadlines to the non-deadline jobs affect the true deadline jobs?*

As mentioned earlier, the QoS guarantee in the form of response time is relatively a new concept; there is no current scheme that supports the QoS in the parallel job scheduling. We study the feasibility of such an idea by providing a framework, termed as *QoPS* (Standing for **Qo**S for **P**arallel Job **S**cheduling), for providing QoS with job schedulers; we compare the trade-offs associated with it with respect to the existing non-deadline based schemes. We consider modifying the Advance Reservation (AR) technique that is a well-established concept in different computing fields intended for other purposes. Although AR is a very simple concept to comprehend, it is not flexible enough to accomplish better performance to provide deadline guarantees in parallel job scheduling area. In addition, we evaluate adaptations of three existing algorithms - the *Slack-Based (SB) algorithm* [73] and the *Real-time (RT) algorithm* [64], previously proposed in different contexts. The SB algorithm [73] was proposed as an approach to improve the utilization achieved by a back-filling job scheduler. On the other hand, the
RT algorithm [64] was proposed in order to schedule non-periodic real-time jobs with hard deadlines, and was evaluated in a static scheduling scenario for uni-processor jobs. As explained later, we adapted these two schemes to schedule parallel jobs in a dynamic job scheduling context with deadlines.

The remaining part of the chapter is organized as follows. In Section 4.2, we describe different forms of possible QoS service pertinent to Parallel systems. We illustrate the details of different QoS-aware schemes including the adaptation of existing schemes in Section 4.4. In Section 4.5, we discuss the simulation approach to evaluate the schemes is discussed. In Section 4.6, we present results of our simulation studies comparing the various schemes. In Section 4.7, we conclude the chapter and present some possible future work.

4.2 QoS in Job Scheduling:

Quality of Service (QoS) is generally defined by a set of measurable parameters to identify the level of service that a service provider can be held responsible [9]. The primary objective of any service provider is to satisfy its user as much as possible. Since the satisfaction of a person is a subjective issue, various QoS parameters of the offered service are often defined depending on the targeted field to evaluate the satisfaction in a reasonable way. In this section, we discuss different QoS aspects in job scheduling including its various possible forms.

4.2.1 Importance of QoS in Job Scheduling

Although QoS is a very active and dominant area of research in various domains of computer science, QoS becomes synonymous to the computer-networking field because of its comprehensive presence. Accordingly, there are very few works done to
provide the QoS in parallel job scheduling. But, QoS provision can surely boost up the satisfaction of the super computer center's user. For example, a user may need some experimental results to be ready within particular time after executing his or her parallel job. With this level of urgency, currently the user just submits the job to a supercomputer center and waits with great uncertainty for the result to be available at any time. In other words, there is no way for the user of specifying the deadline to the job by which the job should be completed. In addition, it would definitely be convenient to the user, if the user knows in advance whether his or her job will be completed by some specified time. In this condition, if the center cannot accept the job to complete by that deadline, the user can resubmit his or her job to other center. In short, this type of deadline guarantees is a much-needed feature that a super computer center can offer to its users. Considering the exigency of this necessity, we focus our research in this area of job scheduling.

4.2.2 QoS classification

QoS in job scheduling can be modeled in various ways depending on the requirement of users. In this section, we propose three different categories of potential QoS service depending on the level of QoS strictness.

**Best-Effort Service:** Best-effort service ensures minimal or no performance guarantees. In this model, jobs do not get any special service from the scheduler that, in most of the cases, follows the FIFO policy in job scheduling. Thus, user who needs special service does not get any preferential treatment from the scheduler though user is willing to pay more for that service. Because of the simplicity in its design, most of the current schedulers are derived from this model.
**Differentiated Service (DiffServ)**  DiffServ model supports the preferential treatment for certain job during job scheduling. DiffServ, also known as Soft QoS, provides the statistical preference to higher priority of jobs, not a hard and fast guarantee. There are two possible ways of specifying the priority: fine-grained and coarse-grained. In fine-grained job ranking, each job has individual priority assigned by certain way. For example, during the job submission, each job can specify the revenue that the user is willing to pay to the center. The scheduler can consider that as a metric of prioritization where jobs with higher offered revenue gets the better service. On the contrary, in coarse-grained ranking, all the jobs could be categorized into a pre-defined set of classes where each class has its own priority-level allocated for all jobs in that class. As done in QoS in networking, job can be classified in different classes and scheduler needs to be familiar with the proposed relative priority of those classes. Essentially, in this model, there should be a specific way of classifying a job so that scheduler can identify the job and its anticipated treatment in scheduling. In current job scheduling, this doesn’t get much attention with an exception of [6] that implements the differentiated service in a limited scale.

**Guaranteed Service (IntServ)**  Integrate Service (IntServ) provides a fine-grained QoS where the service is guaranteed for jobs. IntServ, also known as hard QoS, can be primarily implemented in job scheduling by providing hard-guarantees for the requested jobs. In this model, there should be an admission control mechanism to accept or reject any job during job submission. When a user submits a job with deadline constraint, the scheduler checks whether it can schedule the job without violating the job’s deadline. If the scheduler accepts any job, it needs to guarantee that the job is completed.
within the specified deadline. The research community also gives very little attention to this model in dynamically arriving parallel job scheduling.

4.3 Quantifying Service Quality

In general, service differentiation ensures distinctive services to various applications based on an individual application’s requirements [79] But, determining the appropriate metric to measure the differentiated service is a real challenge. Although time is the basic metric from the user’s viewpoint, there are various time-based metrics available in parallel job scheduling that have respective advantages and drawbacks. The probable metrics include wait time, response time, and slowdown. We intend to investigate different models for service differentiation with comprehensive feasibility analysis of these three metrics. Though the basic definitions of the metrics are described in Section 2.1.6, we further illustrate, in this section, the various characteristics to be considered in our future QoS-aware models.

Wait time specifies how long a job waits in the queue before execution begins. It is a very simple and popular metric for normal users who often prefer to see their job being started sooner. In contrast, since wait time does not consider the shape of the job, this metric has very little significance from the system viewpoint. For example, in the case of two different jobs (one is short-narrow and the other is long-wide), wait time of one hour for both jobs has different implications. Particularly, in a high load situation, providing the same wait time for a long-wide job is a very difficult task for the scheduler then a short-narrow job. Above all, wait time has little importance in preemptive systems where jobs can start right a way but can be preempted many times and eventually can be completed very late.
Response time indicates how long a job takes to complete after the job’s submission. Since this metric considers the job’s runtime, it could be a suitable metric for use in service differentiation. Likewise, slowdown is another appropriate metric that also includes the job’s runtime. Slowdown specifies the ratio of response time for running a job in a supercomputer center to executing the job in dedicated machines.

4.4 Schemes for QoS guarantees

Currently there is no online scheduler for the independently arriving parallel jobs to provide the QoS in the form of deadline guarantees for space-shared parallel system. In this section, we discuss the various aspects of such a scheme and thus present four schemes meticulously. For an effective scheme with QoS guarantees, the following properties are highly anticipated:

• Provide an admission control mechanism for a newly arrived job with deadline constraint.

• Although the shuffling of already admitted jobs is allowable during acceptance of a new job, the deadline guarantees of those jobs should be maintained as well.

• User can ask to execute a job without specifying the deadline (i.e. non-deadline job). While the scheme accepts those jobs with no guarantees, the non-deadline jobs should not starve too much.

• Important-system metrics such as utilization, throughput should not suffer too much.

Adhering to these properties, we initially consider the existing Advanced Reservation (AR), Slack-Based(SB) [73], Real-Time(RT) [64] that are extensively used in different areas for different purposes. We starts by modifying those concepts to be utilized in parallel job scheduling area to support QoS guarantees. More significantly, in
this section, we propose a new scheduler (QoPS) that is exclusively designed to support the QoS in parallel job scheduling. The QoPS has built-in deadline based scheduling support with elegant admission control mechanism by rationally shuffling the jobs for better performance.

4.4.1 Modified Advanced Reservation (MAR)

Advanced Reservation (AR) is a commonly used concept applied in different domains of computer science. In parallel job scheduling, AR is used to allocate a specific position in a schedule for the job and inflexibly keeps that space reserved. AR scheduling can be adapted to provide the QoS in parallel job scheduling in the form of response time guarantee. In an AR scheduler, when a job is submitted with deadline, the scheduler can accept either the earliest possible schedule or the latest available schedule that does not violate the deadline constraint. Both of the above mentioned notions have their own benefits and drawbacks depending on the way the concept is applied. In a more realistic scenario, where both deadline and non-deadline jobs are common, there are various ways of managing the non-deadline jobs as well. When a non-deadline job arrives in the system, the scheduler can assign an artificial deadline with sufficient flexibility so that the job does not suffer starvation. In the above situation, the scheduler needs to decide how to reserve this job with imitated deadline. The schemes can assign, like other real deadline job, either the latest or earliest possible schedule using this synthetic and relax deadline. So there are following four different strategies that can be pursued in modified AR scheduling in a system with both deadline and non-deadline jobs:

- AR with earliest possible schedule for deadline job and earliest possible schedule for non-deadline job (AREE)
• AR with earliest possible schedule for deadline job and latest possible schedule for non-deadline job (AREL)

• AR with latest possible schedule for deadline job and latest possible schedule for non-deadline job (ARLL)

• AR with latest possible schedule for deadline job and earliest possible schedule for non-deadline job (ARLE).

Algorithm 1 illustrates the pseudo code for AREL as a representative scheme. Rest of the three schemes follows the same idea.

Algorithm 1 Admission control mechanism followed in AREL scheme

1: function AREL($J_i$, $S_{org}$, $S_{new}$)
2:    Input: New job $J_i$, existing schedule $S_{org}$
3:    Output: Updated schedule $S_{new}$
4:    Returns: Whether Job $J_i$ is accepted or rejected
5:    $S_{new} ← S_{org}$
6:    if the job $J_i$ has a deadline assigned then
7:       Find the earliest possible reservation for job $J_i$ in $S_{new}$
8:       if CompletionTime($J_i$) ≤ Deadline($J_i$) then
9:           return ACCEPTED
10:      else
11:         $S_{new} ← S_{org}$
12:         return REJECTED
13:     end if
14:    else
15:       Deadline($J_i$) ← ArrivalTime($J_i$) + max(24 hours, 2 * Runtime($J_i$))
16:       Find the latest possible reservation for job $J_i$ in $S_{new}$
17:       if no such feasible schedule exist then
18:          Find the earliest possible reservation for job $J_i$ in $S_{new}$
19:          Deadline($J_i$) ← CompletionTime($J_i$)
20:      end if
21:      return ACCEPTED
22: end if
23: end function
4.4.2 Modified Slack Based (MSB) Algorithm

The Slack-Based (SB) Algorithm, proposed by Feitelson et. al [73], is a backfilling algorithm used to improve the system throughput and the user response times. In this section we discuss the key characteristics of this algorithm as well as its modifications. The main idea of the algorithm is to allow a slack or laxity for each job. The scheduler gives each waiting job a pre-calculated slack, which determines how long it may have to wait before running: ‘important’ and ‘heavy’ jobs will have little slack in comparison with others. When other jobs arrive, this job is allowed to be pushed behind in schedule time as long as it’s execution is within the initially calculated laxity.

The calculation of the initial slack involves cost functions taking into consideration certain priorities associated with the job. This scheme supports both user selected and administrative priorities, and guarantees a bounded wait time for all jobs.

Though this algorithm has been proposed for improving the system utilization and the user response times, it can be modified to support hard real time deadlines by fixing the slack appropriately. We propose this modified algorithm shows in Figure 2.

Compared to the original SB algorithm, MSB differs in the way the slack is determined for a given job. The original SB algorithm uses weighted user and political priorities to determine the slack. However, in the current scenario, we change this by setting the slack to be as: $\text{Slack} = \text{Deadline} - (\text{ArrivalTime} + \text{RunTime})$.

The rest of the algorithm follows the approach taken by the SB algorithm. When a new job arrives, the jobs currently present in the schedule are arranged in an order decided by a heuristic function (e.g. job arrival time). Once this order is fixed, the new job is inserted in each possible position in this arrangement. Thus, if there are $N$ jobs in the existing schedule, when the new job arrives, there are $N + 1$ possible
Algorithm 2 Admission control mechanism followed in Modified Slack-Based scheme

1: function MSB(J_i, S_{org}, S_{cheap})
2: \hspace{1em} Input: New job J_i, existing schedule S_{org}
3: \hspace{1em} Output: Updated schedule S_{cheap}
4: \hspace{1em} Returns: Whether Job J_i is accepted or rejected
5: \hspace{1em} cheapPrice \leftarrow \infty
6: \hspace{1em} S_{cheap} \leftarrow S_{org}
7: \hspace{1em} S_{new} \leftarrow S_{org}
8: \hspace{1em} for each time slot t_s of S_{org} do
9: \hspace{2em} Remove all waiting jobs from t_s to the end of S_{new} and place them into a Temporary List (TL)
10: \hspace{2em} Schedule the job J_i in S_{new}
11: \hspace{2em} Sort the TL using the scheduled time order
12: \hspace{2em} Add each job from TL into the schedule S_{new}
13: \hspace{2em} price \leftarrow \text{Cost}(S_{new})
14: \hspace{2em} if price < cheapPrice then
15: \hspace{3em} cheapPrice = price
16: \hspace{3em} S_{cheap} \leftarrow S_{new}
17: \hspace{3em} Update slack of all jobs in S_{cheap}
18: \hspace{2em} end if
19: \hspace{1em} end for
20: \hspace{1em} if cheapPrice \neq \infty then
21: \hspace{2em} return ACCEPTED
22: \hspace{1em} else
23: \hspace{2em} S_{cheap} \leftarrow S_{org}
24: \hspace{2em} return REJECTED
25: \hspace{1em} end if
26: end function
schedules. A pre-decided cost function is used to evaluate the cost of each of these \( N+1 \) schedules and the one with the least cost is accepted. We can easily see that MSB is an \( O(N) \) algorithm considering the evaluation of the cost function to be a constant cost. In practice, evaluating the cost function of the schedule depends on the number of jobs in the schedule and thus is a function of \( N \). However, for the sake of comparison between the various algorithms and for ease of understanding, we approximate the evaluation of the cost function to be a constant value. It is to be noted that this approximation does not change the relative difference in the time complexity.

### 4.4.3 Modified Real Time (MRT) Algorithm

The Real Time (RT) Algorithm, proposed by Ramamritham et. al, is an approach to schedule uni-processor static tasks with hard real time deadlines on multi-processor systems. In this section, we describe main features of this algorithm along with its modification to accommodate QoS guarantees.

It has been shown that for dynamic systems with more than one processor, a polynomial-time optimal scheduling algorithm does not exist [58, 57, 59]. The algorithm tries to meet the specified deadlines for the jobs by using heuristic functions. The tasks are characterized by worst case computation times, deadlines and resource requirements. Starting with an empty partial schedule, each step in the search extends the current partial schedule with one of the tasks yet to be scheduled. The heuristic functions used in the algorithm actively direct the search for a feasible schedule i.e., they help choose the task that extends the current partial schedule. Earliest Deadline First and Least Laxity First are examples of such heuristic functions.
In order to accommodate this algorithm into the domain of scheduling dynamically arriving parallel jobs, we have made two modifications to the algorithm. The first one is to allow parallel jobs to be submitted to the algorithm and the other is to allow dynamically arriving jobs. The details of the modified algorithm are shown in Figure 3.

### Algorithm 3 Admission control mechanism followed in Modified Real-Time scheme

1. **function** MRT($J_i$, $S_{org}$, $S_{new}$)
2. **Input**: New job $J_i$, existing schedule $S_{org}$
3. **Output**: Updated schedule $S_{new}$
4. **Returns**: Whether Job $J_i$ is accepted or rejected
5. $S_{new} \leftarrow S_{org}$
6. $backTrackCount \leftarrow 0$
7. Remove all waiting jobs from $S_{new}$ and place them into a Temporary List ($TL$)
8. Add the new job $J_i$ into the $TL$
9. Sort the $TL$ using the appropriate heuristic function (e.g. EDF)
10. $S_{new} \leftarrow \phi(\text{empty})$
11. **for** each job $J$ from $TL$ **do**
12. Determine whether the job $J$ is strongly feasible in current partial schedule $S_{new}$
13. **if** $J$ is strongly feasible **then**
14. Add job $J$ into $S_{new}$
15. Remove job $J$ from $TL$
16. Continue
17. **else**
18. Backtrack to the previous partial schedule of $S_{new}$
19. $backTrackCount \leftarrow backTrackCount + 1$
20. **if** $backTrackCount > MAX\_BACK\_TRACK$ **then**
21. $S_{new} \leftarrow S_{org}$
22. return REJECTED
23. **else**
24. continue
25. **end if**
26. **end if**
27. **end for**
28. **if** all jobs are placed in the schedule **then**
29. return ACCEPTED
30. **else**
31. return REJECTED
32. **end if**
33. **end function**
The RT algorithm assumes that the calculation of certain heuristic function for scheduling a job into a given partial schedule takes constant time. However, this assumption only holds true for sequential (single processor) jobs (which was the focus of the algorithm). However, the scenario we are looking at in this chapter relates to parallel jobs, where holes are possible in the partial schedule. In this situation, such an assumption would not hold true.

The Modified RT algorithm (MRT algorithm) uses the same technique as the RT algorithm but increases the time complexity to accommodate the parallel job scenario. When a new job arrives, all the jobs that have not yet started (including the newly arrived job) are sorted using some heuristic function (such as Earliest Deadline First, Least Laxity First, etc.). Each of these jobs is inserted into the schedule in the sorted order. A partial schedule at any point during this algorithm is said to be feasible if every job in that schedule meets its deadline. A partial schedule is said to be strongly feasible if the following two conditions are met:

- The partial schedule is Feasible
- The partial schedule would remain feasible when extended by any one of the unscheduled jobs

When the algorithm reaches a point where the partial schedule obtained is not feasible, it backtracks to a previous strongly feasible partial schedule and tries to take a different path. A certain number of backtracks are allowed, after which the scheduler rejects the job.

4.4.4 The QoPS Scheduler

Although MAR, MSB, MRT can serve the purpose of providing hard deadline guarantees; utilizing them to achieve QoS results in significant under-utilization of resources,
making them not the best choice as a QoS aware scheme. Therefore, in this section, we propose the new QoPS Scheduler that exclusively deals with parallel job scheduling in hard deadline-based systems.

**Essentials of QoPS**

For dynamic systems with more than one processor, it has been shown that a polynomial-time optimal scheduling algorithm does not exist. So the QoPS scheduling algorithm uses a heuristic approach to try to find feasible schedules for the jobs. The scheduler ideally considers a system where each job arrives with a corresponding completion time deadline requirement. When each job arrives, it attempts to find a feasible schedule for the newly arrived job. A schedule is said to be feasible if it does not violate the deadline constraint for any job in the schedule, including the newly arrived job. However, it does allow a flexibility of reordering the jobs in any order as long as the resultant schedule remains feasible.

As a continuation of notational representation described in Section 3.2, consider there are \( n \) jobs in the current feasible schedule \( S = \{\Pi_1, \Pi_2, \ldots, \Pi_n\} \) where \( \Pi_i \) is the individual reservation for job \( J_i \). In this circumstance, when a job \( J_{n+1} \) is submitted in the system, the scheduler determines whether to accept or reject the job. In theory, there can be \( (n+1)! \) different combinations (valid or invalid) of jobs possible. The required time complexity to verify the acceptance criteria for all of above-mentioned \( (n+1)! \) schedules is exponential (i.e. computationally intractable). Therefore, we suggest a heuristic approach where the new jobs will be placed in \( \log_2 n \) different positions \((0, \frac{n}{2}, \frac{3n}{4}, \ldots, n)\) and only the schedules resulting from these are tested for acceptance; thus making it computationally manageable.
Let’s define a term \( S_{pos} \) as the schedule where the jobs from position \( pos \) to \( n \) in the existing schedule including the new job \( J_{n+1} \) are rearranged according to Earliest Deadline First (EDF) policy. Essentially, there will be a set total \( \log_2 n \) different schedules \( \mathcal{R}^{\text{org}} = \{ S_0, S_2, S_4, ..., S_n \} \) where \( |\mathcal{R}^{\text{org}}| = \log_2 n \) and \( S_i \) is a (valid or invalid) schedule. Amongst these \( \log_2 n \) schedules, using the specific acceptance criteria (described later in this section), a subset of these schedules can be identified as feasible. Furthermore, using certain best-fit determining technique (also described later), the best schedule among those feasible schedules is found.

**Acceptance criteria used for QoPS:**

Let us assume \( S_{\text{old}} \) as the old feasible schedule (before the new job \( J_{n+1} \) arrives) that can be specified for \( n \) jobs as \( S_{\text{old}} = \{ \Pi_1, \Pi_2, ..., \Pi_n \} \) where \( \Pi_i \) is the individual reservation for job \( J_i \). Also, assume the new schedule \( S_{\text{new}} \) of \( n+1 \) jobs is defined as \( S_{\text{new}} = \{ \Pi'_1, \Pi'_2, ..., \Pi'_n \} \) where \( \Pi'_i \) is the new individual reservation of job \( J_i \) and \( S_{\text{new}} \in \mathcal{R}^{\text{org}} \). In this situation, the schedule \( S_{\text{new}} \) will be a feasible schedule and the scheduler will accept the new job \( J_{n+1} \) if and only if:

\[
\bigcap_{i=1}^{n+1} CT(S_{\text{new}}, J_i) \leq \text{Deadline}(J_i) \text{ is true}
\]

In other words, the new job is accepted if the deadline constraints are maintained for all the previously accepted active jobs as well as the newly arrived job. After using this acceptance criteria, we may find some non-feasible schedules in \( \mathcal{R}^{\text{org}} \). We can symbolize the feasible set of schedules of \( \mathcal{R}^{\text{org}} \) as \( \mathcal{R}^{\text{feasible}} = \{ S^i, S^j, ..., S^w \} \) where \( S^i \) is a feasible schedule and \( \mathcal{R}^{\text{feasible}} \subseteq \mathcal{R}^{\text{org}} \) and \( |\mathcal{R}^{\text{feasible}}| \leq \log_2 n \).

**Finding Best schedule for QoPS:**

Firstly, let us define the cost of a schedule, which is determined by the make span of that
schedule and can be mathematically characterize as

\[
Cost(S) = \infty \quad \text{If } S \text{ is not a feasible schedule} \tag{4.1}
\]

\[
= \text{MakeSpan}(S) \quad \text{otherwise} \tag{4.2}
\]

By using the above cost function, we can find out the best schedule (\(S_{\text{best}}\)) from (\(R_{\text{feasible}}\)) with minimum cost. We can mathematically expressed the idea as follows:

\[
S_{\text{best}} = S \quad \text{where } S \in R_{\text{feasible}} \text{ and } Cost(S) = \text{MIN} \{Cost(S^i)\} \quad \text{where } S^i \in R_{\text{feasible}}
\]

### 4.4.5 QoPS Algorithm

The main difference between the QoPS algorithm and other different variations of AR is the flexibility the QoPS algorithm offers in reordering the jobs that have already been scheduled (but not yet started). For example, suppose jobs \(J_1, J_2, \ldots, J_N\) are the jobs which are currently in the schedule but not yet started. In AR-based schemes, this ordering of the jobs are fixed. When a new job \(J_{N+1}\) arrives, the AR-based algorithms try to fit this new job in the given schedule without any change to the initial ordering of the jobs. On the other hand, the QoPS scheduler allows flexibility in the order in which jobs are considered for scheduling.

Algorithm 4 presents the pseudo code for the QoPS. When a new job arrives, it is given \(\log_2(N)\) points in time where its insertion into the schedule is attempted, corresponding to the reserved start-times of jobs \((0, \frac{n}{2}, \frac{3n}{4}, \ldots, n)\) respectively, where \(N\) is the number of jobs currently in the schedule. The interpretation of these options is as follows: For option 1 (corresponding to job at time 0), QoPS starts by removing all the jobs from the schedule and placing them in a temporary list (TL) and then add the
Algorithm 4: Admission control mechanism used in QoPS scheme for FIRST-FIT

1: function QoPS(Jᵢ, S₀, Sₙₑₙ)
2:     Input: New job Jᵢ, existing schedule S₀
3:     Output: Updated schedule Sₙₑₙ
4:     Returns: Whether Job Jᵢ is accepted or rejected
5:     \( Sₙₑₙ \leftarrow S₀ \)
6:     for each pos of S₀ in position \((0, \frac{N}{2}, \frac{3N}{4}, ..., N)\) do
7:         status ← scheduleAtPos(pos, Sₙₑₙ)
8:         if status = TRUE then
9:             return ACCEPTED
10:     end if
11: end for
12:     Sₙₑₙ ← S₀
13: return REJECTED
14: end function

15: function scheduleAtPos(Jᵢ, pos, Sₙₑₙ)
16:     Input: New job Jᵢ, position pos, existing schedule Sₙₑₙ
17:     Output: Updated schedule Sₙₑₙ
18:     Returns: Whether Job Jᵢ is accepted or rejected
19: Remove all waiting jobs from pos to N and place them into a Temporary List (TL)
20: Add the new job Jᵢ into TL
21: Sort TL using any heuristic function (e.g. EDF)
22: ViolationCount ← 0
23: for each job Jₖ in TL do
24:     Add job Jₖ into schedule Sₙₑₙ in the earliest possible position
25:     if CompletionTime(Jₖ) > Deadline(Jₖ) then
26:         ViolationCount ← ViolationCount + 1
27:     if ViolationCount > KFACTOR then
28:         return FALSE
29:     end if
30:     Failedpos ← position where violation occurs
31: Remove jobs of Sₙₑₙ from position \(\frac{pos+Failedpos}{2}\) to Failedpos and add them into TL
32: Sort TL again using the same heuristic used.
33: Add the failed (critical) job Jₖ at the top of TL to make sure it is scheduled first
34: end if
35: end for
36: return TRUE
37: end function
newly arrived job to the list. After that the scheduler sorts TL according to some heuristic function (the heuristic function could be least laxity first, earliest deadline first, etc). Finally, the scheduler tries to place the jobs in the schedule in the order specified in TL. For option 2, QoPS does not start with an empty schedule. Instead, the scheduler only removes the latter $\frac{N}{2}$ jobs in the original schedule, chosen in scheduled start time order, places them in the temporary list $TL$ including the new job $J$, and sorts the temporary list (based on the same heuristic function). QoPS then find schedule for ($\frac{N}{2} + 1$) jobs trying in the order specified in sorted TL. Thus, there would be $log_2(N)$ options of placement.

For each option given to the newly arrived job, the algorithm tries to schedule the jobs based on heuristic order. If a job misses its deadline, this job is considered as a critical job and is pushed to the head of the list (thus altering the temporary schedule). This altering of the temporary schedule is allowed at most 'K' times for any option; after that the scheduler decides that the new job cannot be scheduled while maintaining the deadline for all of the already accepted jobs. In this case, the scheduler stops trying to find the schedule for that position and then try for the next option, if there is any. After trying all $log_2(N)$ options and not finding any suitable schedule, QoPS decides to reject the new job $J$ and keeps the original schedule unchanged. However, if there are some feasible schedules, the scheduler selects the best schedule with minimum makespan and accept that as effective schedule. This results in a time complexity of $O(KN^2 log_2 N)$ for the QoPS scheduling algorithm.
4.5 Evaluation Setup

To study the impact of different schemes that support deadline guarantees, we implement a trace-based simulation approach described briefly in Section 3.3. In this section, we elaborate that approach with details followed in our experiments.

4.5.1 Trace Pre-Processing

Job scheduling strategies are usually evaluated using real workload traces, such as those available at the Parallel Workload Archive [29]. In this section, we discuss issues regarding deadline generation, load expansion and different mixes of jobs in trace generation.

Deadline Inclusion All of our schemes depend on deadline information supplied by user during job submission. But, for our simulation, we use the real job traces from supercomputer centers that have no deadline information. So it is a challenge to generate a realistic job trace with appropriate deadline information. A possible approach to evaluating the QoPS scheduling strategy might be based on the methodology that was used in [64] to evaluate their real-time scheduling scheme. The randomized synthetic job sets was created in such a way that a job set could be packed into a fully filled schedule, say from time $= 0$ to time $= T$, with no holes at all in the entire schedule. Each job was then given an arrival time of zero, and a completion deadline of $(1 + r) \times T$. The value of $r$ represented a degree of difficulty in meeting the deadlines. A larger value of $r$ made the deadlines more lax. The initial synthetic packed schedule is clearly a valid schedule for all non-negative values of $r$. The real-time scheduling algorithm was evaluated for different values of $r$, over a large number of such synthesized task sets. The primary metric was the fraction of cases that a valid schedule for all tasks was found by the
scheduling algorithm. It was found that as \( r \) was increased, a valid schedule was found for a larger fraction of experiments, asymptotically tending to 100% as \( r \) increased.

We first attempted to extend this approach to the dynamic context. We used a synthetic packed schedule of jobs, but unlike the static context evaluated in [64], we set each job’s arrival time to be the start time in the synthetic packed schedule, and set its deadline beyond its start-time by \((1 + r)\) times its runtime. When we evaluated different scheduling algorithms, we found that when \( r \) was zero, all schemes had a 100% success rate, while the success rate dropped as \( r \) was increased! This was initially puzzling, but the reason was quickly apparent - with \( r = 0 \), as each job arrives, the only possible valid placement of the new job corresponds to that in the synthetic packed schedule, and any deadline-based scheduling algorithm exactly tracks the optimal schedule. When \( r \) is increased, other choices are feasible, and the schedules begin diverging from the optimal schedule, and the failure rate increases. Thus, this approach to generating the test workload is attractive in that it has a known valid schedule that meets the deadlines of all jobs; but it leads to the unnatural trend of decreasing scheduling success rate, as the deadlines of jobs are made more relaxed.

Due to the above problem with the evaluation methodology used in [64], we pursued a different trace-driven approach to evaluation. We used a trace from Feitelson’s archive (a 10000-job subset of the CTC and SDSC trace) and first used EASY backfill to generate a valid schedule for the jobs. Deadlines were then assigned to all jobs, based on their completion time on the schedule generated by EASY backfill. A deadline stringency factor determined how much tighter the deadline was to be set, compared to the EASY backfill schedule. With a stringency factor of 0, the deadlines were set to be the completion times of the jobs with the EASY backfill schedule.
With a stringency factor of $S$, the deadline of each job was set after its arrival time by $\max(\text{runtime}, (1 - S) \times EASY\_Schedule\_Response\_Time)$. As $S$ is increased, the deadlines become more stringent. So we would expect the number of successfully scheduled jobs to decrease.

**Load Expansion:** As the demand for the resources are increasing, we need to investigate the effectiveness of our schemes in various high load environments. This necessitates the expansion of the real traces in a rational way. There are mostly two familiar methods that we can utilize in our trace generation: job duplication and job expansion. In job duplication technique, certain number of jobs is duplicated with same or random arrival time by keeping other attributes unchanged. On the other hand, in job expansion method, the expected runtime of each job is expanded as specified by load expansion factor. We mainly use the duplication approach for varying the load on the supercomputer center (number of jobs submitted). Job are selected randomly and duplicated with the same arrival time. For example, we start with a trace and call this the base trace (load = 1.0). To generate a new trace with load = 1.2, we randomly pick 20% of the jobs in the base trace and introduce extra duplicate jobs at the same points in the trace. The deadline of the newly introduced job is retained as the original job’s deadline (obtained from the EASY backfill schedule). We also pay special attention to maintain the subset relationship of jobs across the different loads. For example, the jobs selected for load 1.2 are also selected for load 1.6 with additional 40% new jobs.

**Job Mix:** In a realistic environment, some user might need deadline guarantees and some user might not need such guarantees. In other words, some of the jobs may be urgent and impose deadlines; there would likely be other jobs that are non-urgent, with
the users not requiring any deadlines. We also incorporate this mixed scenario in our experiments. Essentially, we generate traces with different assumed mixtures of deadline and non-deadline jobs.

### 4.5.2 Simulation

Simulation is the core stage of our experiments that execute our proposed schemes. In this section, we discuss how to deal with the non-deadline jobs as well as the list of variations in simulation setup.

**Managing Non-deadline jobs:** As described above, our proposed system can get jobs with or without deadline. In order to evaluate different schemes under this scenario of mixed jobs, some with user-imposed deadlines and others without, the scheduler particularly requires dealing with non-deadline jobs. While the different schemes could be run with an “infinite” deadline for the non-deadline jobs, we do not go along with that in order to avoid starvation of any jobs. Therefore, the scheduler artificially created very lax deadlines for the non-deadline jobs. The artificial turn-around time of each non-deadline job is set to $\max(24\text{hours}, R \times (\text{Earliest Possible ResponseTime})),$ where $R$ is relaxation factor given as simulation parameter. Thus, short non-deadline jobs were given an artificial deadline of one day after its arrival, while long jobs were given a deadline of $R \times \text{Earliest Possible ResponseTime}.$

**Options in Simulation:** We carry out our extensive simulation-based experiments with various presumed practical scenarios for different offered schemes. Those cases are enumerated below:
• **Load variations:** In general, load of supercomputer centers is growing. Therefore, in addition of real base load (load =1.0), we run simulation for various higher loads (e.g., 1.1, 1.2, 1.3, 1.4, 1.5 and 1.6).

• **Stringency variations:** As mentioned in Section 4.5.1, we simulate the experiments both with high stringent ($S = 0.5$) where tight deadline is assigned and with low stringency ($S = 0.2$) where relatively relaxed deadline is allotted.

• **Varying Job-mixes:** We include both deadline and non-deadline jobs in the same trace. We exercise different ratios of this mixture. We start with 100% deadline jobs where no non-deadline job is present. Then, we run with low percent of non-deadline jobs (e.g., 20% non-deadline and 80% deadline jobs) as well as high percent of non-deadline jobs (e.g., 80% non-deadline and 20% deadline jobs).

• **Real trace from various sources:** To further show the resiliency of our schemes across different real workloads gather from different supercomputer centers, we ran the simulation with both SDSC and CTC traces.

• **Runtime Variations:** During job submission, user specifies the expected runtime of the job. Often these estimates are very wrong. As the real workload contains both user estimated runtime and actual runtime of jobs, we simulate our schemes with both runtimes. The differences among various schemes are easy to discern using the actual runtime because the system has one less variable to consider. But, to emulate the exact system, we also carry out the test with real (often inaccurate) user estimate given in the original trace.

### 4.5.3 Output Post-processing

In the output post-processing step, we generate several necessary metrics and draw the corresponding graphs to easily analyze the impact of different schemes on various
metrics. In this section, we illustrate two new metrics used in deadline-based schemes. We also use other most common metrics that are described in Section 2.1.6.

**Metrics Used:** All proposed schemes suggest the admission control mechanism where a new job gets acceptance or rejection response from the scheduler upon submission. None of the commonly used metrics can evaluate the performance of this aspect of job scheduling. We suggest following two metrics in this regard.

**Unadmitted Job Count:** It reveals how many jobs the scheduler rejects due to the deadline violation. This is calculated by counting the number of dropped jobs. Generally, we want the value of this metrics to be as less as possible. Lower value of unaccepted job count means the scheduler is making more people happy through servicing and assuring the deadline constraints of their jobs. However, this metric might be confusing in broader respect of comparing two schemes. For instance, one scheme accepts mostly the smaller jobs by rejecting the large jobs; thus keeping most of the resources unused. While this example scheme satisfies a lot of users with small jobs, it displeases some important users who submit large jobs. Therefore, we introduce the next metric that considers this type of biased acceptance.

**Unadmitted Processor Seconds:** It specifies the total number of processor seconds rejected due to the deadline violation. Lower value in unadmitted processor seconds metric demonstrates the better performance. We can calculate this mathematically for the set of N jobs as follow:

\[
\text{Unadmitted Processor Seconds} = \sum_{i=1}^{N} (\text{Processor}(J_i) \times \text{Runtime}(J_i) \times \text{Rejected}(J_i))
\]

where \( \text{Rejected}(J_i) = 1 \) if \( J_i \) is dropped otherwise 0
4.6 Experimental Results: Comparison of Different Schemes

As mentioned earlier, supporting QoS guarantees in the form of response time is a new concept for space-shared parallel job scheduling. Designing an effective scheme to support deadline guarantees is a significant challenge for the research community. In Section 4.4, we suggest a bunch of such schemes. Firstly, we attempt to extend some of the existing schemes originally intended for different purposes. More importantly, we propose a novel scheme, termed as, QoPS exclusively addressing the deadline guarantees. While new schemes are generally compared against some existing similar scheme, there is no such existing scheme for space-shared system to support deadline guarantees in parallel job scheduling. Therefore, we evaluate our proposed scheme against some modified schemes with respect to various pertinent metrics through simulation-based approach, described in Section 2.1.6 and 4.5.3. In particular, we compare the following seven schemes that are comprehensively illustrated in Section 4.4:

- **MSB** (Modified Slack Based) is an extension of slack-based scheme proposed by Feitelson et. al, [73] to support deadline guarantees.
- **MRT** (Modified Real-Time) is an extended version of deadline-based scheme mainly suggested for real-time system with single processor system by Ramamritham et. al, [64].
- **AREE**: Advanced Reservation with Earliest possible schedule for deadline job and Earliest possible schedule for non-deadline job.
- **AREL**: Advanced Reservation with Earliest possible schedule for deadline job and Latest possible schedule for non-deadline job.
- **ARLL**: Advanced Reservation with Latest possible schedule for deadline job and Latest possible schedule for non-deadline job.
• *ARLE* Advanced Reservation with Latest possible schedule for deadline job and Earliest possible schedule for non-deadline job.

### 4.6.1 Impact on Job acceptance

All the deadline-based schedulers implement some kind of the admission control mechanism. In other words, when a scheduler receives a new job with deadline requirement, the scheduler either accepts or rejects the job for service depending on the availability and convenience. Thus, we evaluate the admittance capacity of various schemes. As mentioned above, we mainly compare the new scheme QoPS with six different modified schemes. Due to the convenience of graph presentation, we split the presentation of same metric into two different graphs. We show the result for QoPS and various forms of Advanced Reservation techniques in one graph. The other graph demonstrates the result of QoPS, MSB and MRT schemes. We analyze various schemes with respect to different loads (low to very high load) and for different stringency factors (moderate to high stringent deadline) to gauge the resilience of concepts. In this section, we compare mainly two different forms of admittance capacity: number of unaccepted load and number of unaccepted jobs described in Section 4.5.3.

**Unadmitted Load:** Unadmitted load metric specifies how much processor seconds are rejected by any scheme. The scheme that dropped less processor second is better regarding this metric. We examine this metric for different mixes of jobs where some job have their deadline assigned and some do not have any deadline. As the real system could have any combinations of job mixes, we also study the behavior of schemes for different mixtures: ranging from low percent (20%) of jobs with deadline to as high as 100% of jobs with deadline. Figure 4.1(a) displays the lost processor-seconds for the
Figure 4.1: Unaccepted Processor Seconds for different loads where 20% jobs have user specified deadline with exact user runtime estimate and Stringency Factor 0.2 for CTC trace

Figure 4.2: Unaccepted Processor Seconds for different loads where 20% jobs have user specified deadline with exact user runtime estimate and Stringency Factor 0.5 for CTC trace
Figure 4.3: Unaccepted Processor Seconds for different loads where 80% jobs have user specified deadline with exact user runtime estimate and Stringency Factor 0.2 for CTC trace

Figure 4.4: Unaccepted Processor Seconds for different loads where all jobs have user specified deadline with exact user runtime estimate and Stringency Factor 0.2 for CTC trace
different forms of Advanced Reservation (AR) and QoPS scheduler, for a stringency factor of 0.2, as the load factor is varied from 1.0 to 1.6 where 20% of the jobs have user requested deadline. QoPS scheme performs significantly better (as high as 41%) compared to other AR-based schemes by rejecting less processor seconds. Figure 4.1(a) also exhibits the unaccepted processor seconds for MSB, MRT and QoPS where QoPS significantly outperforms other two schemes by as high as 47%. The performance difference increases as the load increases because there are more processor seconds in higher load and QoPS can utilize that extra load achieving further improvement. The same overall trend also holds for a higher stringency factor (0.5), as seen in Figure 4.2. However, the performance of QoPS is closer to the other schemes. In general, we find that as the stringency factor increases, the performance of the different strategies tends to converge. This suggests that the additional flexibility that QoPS tries to exploit in rearranging schedules is most beneficial when jobs have sufficient laxity with respect to their deadlines.

Figure 4.3 and Figure 4.4 present the data of lost processor seconds for stringency factor 0.2 where 80% and 100% (all) jobs have user expected deadline respectively. It can be seen that the QoPS scheme provides consistently better performance compared to other schemes, especially at high load. But the differences in value among the schemes diminish as the number of jobs with deadline increases. This again suggests that in scenarios where many jobs have significant flexibility (in Figure 4.1, the non-deadline jobs comprise 80% of jobs and they have significant flexibility in scheduling), the QoPS scheme makes effective use of the available flexibility.

So far, we assume that user exactly estimates the runtime of job. This assumption gives the opportunity to assess the schemes in more predictable way where it hides the
Figure 4.5: Unaccepted Processor Seconds for different loads where 20% jobs have user specified deadline with inexact user runtime estimate and Stringency Factor 0.2 for CTC trace

Figure 4.6: Unaccepted Processor Seconds for different loads where 20% jobs have user specified deadline with inexact user runtime estimate and Stringency Factor 0.2 for SDSC trace
variations in runtime estimate. But, in reality, users are awfully unreliable in predicting the runtime of their job. Therefore, we further conduct our experiments with inexact user estimate gather from the real trace. Figure 4.5 demonstrates the effect of same metric on various schemes with inexact user estimate for stringency factor 0.2 where 20% job has user specified deadline. In this case also, QoPS surpass the performance of other schemes. Clearly the findings substantiate that QoPS performs well regardless in the degree of correctness of user runtime estimation.

Thus far, we discuss all the results generated by running simulation based on CTC trace. Furthermore, we examine the performance of various schemes by using traces collected from another well-know center SDSC. Figure 4.6 shows the unaccepted processor-seconds for different schemes where 20% job has deadline assigned with 0.2 stringency factor. It can be noted that the general trend of the relative performances of the schemes does not change significantly. This result further demonstrates the QoPS resilience across different realistic circumstances.

**Unadmitted Job:** Number of unaccepted job is another way of evaluation the admission capacity of different schemes. According to this metric, the scheme that drops fewer number jobs performs better. This metric specifies how many users are satisfied. In some cases, this metric could misrepresent the exact result. Some schemes could reject most of the heavier jobs in favor of accepting the short and narrow jobs that requires less processor seconds. These schemes generally perform well with respect to this metric. However, these schemes definitely dissatisfy most of the user (often influential) with heavier job. In spite of this disadvantage, unaccepted job count is considered as an associated metric in many situations. Figure 4.7 and Figure 4.8 present the data for
Figure 4.7: Number of unadmitted jobs for different loads where 20% jobs have user specified deadline with inexact user runtime estimate and Stringency Factor 0.2 for CTC trace.

Figure 4.8: Number of unadmitted jobs for different loads where 20% jobs have user specified deadline with inexact user runtime estimate and Stringency Factor 0.2 for SDSC trace.
unaccepted number of jobs where 20% job has deadline assigned with stringency factor 0.2 for CTC and SDSC trace respectively. The figures show that QoPS performs better or comparable with other schemes. Although, QoPS performance differences with other schemes are very discernible in unaccepted processor seconds displayed in Figure 4.1(a), the same is not true regarding the number of unaccepted jobs. This clearly shows that QoPS accepts heavier jobs, while other schemes prefer short-narrow jobs.

4.6.2 Impact on Non-deadline Job

Figure 4.9: Average Response Time for different loads where 20% jobs have user specified deadline with inexact user runtime estimate and Stringency Factor 0.2 for CTC trace

So far, we mainly considered how good is the schemes in job admittance while guaranteeing the deadline of a job. But, in the mixed case where some jobs have user requested deadline and some do not, it is interesting to see how the jobs without deadline are managed in various schemes. As discussed in Section 4.5.2, non-deadline jobs are
Figure 4.10: Average Response Time for different loads where 80% jobs have user specified deadline with inexact user runtime estimate and Stringency Factor 0.2 for CTC trace

Figure 4.11: Average Response Time for different loads where 20% jobs have user specified deadline with inexact user runtime estimate and Stringency Factor 0.2 for SDSC trace
associated with an artificial deadline that provides considerable slack, but prevents starvation. We evaluate the average response time and average slowdown metrics described in Section 2.1.6 only for the job without deadline. However, these metrics are less relevant for jobs with user specified deadline. In addition of comparing average response time and slowdown among various proposed schemes, we also evaluate the EASY back-fill scheme widely used in current super computer center. Inclusion of EASY scheme provides an opportunity to comprehend the probable impacts on current users. Figure 4.9 and Figure 4.10 show the variation of average response time for non-deadline jobs with load for the cases with 20% and 80% of jobs being deadline jobs respectively for stringency factors of 0.2. In addition to the data for the various deadline-based scheduling schemes, data for the EASY back-fill mechanisms is also shown. The average response time can be seen to be lower for most of the deadline-based schemes, when compared to EASY. This is because the delivered load for the non-deadline based schemes is equal to the offered load (the X-axis), whereas the delivered load for the deadline-based scheduling schemes is lower than offered load. In other words, with the non-deadline based schemes, all the jobs are admitted, whereas with the other deadline based schemes, not all deadline jobs are admitted. This also explains the reason why the performance of QoPS appears not always very good compare to other deadline-enabled schemes- as seen from Figure 4.1, the rejected load from the deadline jobs is much higher for other schemes than QoPS. ARLL and AREL schemes suffer very high response time compare to other three schemes. Both ARLL and AREL place the non-deadline jobs at the furthest place after the jobs are assigned with very high slacked deadline by scheduler. So the schemes are designed to penalize the non-deadline jobs with high response time. Figures 4.11 shows the data for similar experiments with SDSC.
trace where 20% job has deadline assigned with stringency factor 0.2 for inexact user estimate. We also present the data for slowdown metrics for 20% deadline and 80%

Figure 4.12: Average Slowdown for different loads where 20% jobs have user specified deadline with inexact user runtime estimate and Stringency Factor 0.2 for CTC trace

deadline jobs in Figure 4.12 and Figure 4.13 respectively with stringency factor 0.2 for inexact user runtime estimate. Figure 4.14 demonstrates the same experiment result but for SDSC trace. It can be seen that QoPS is consistently superior to other modified schemes as well as EASY backfill. Thus despite the constraints of the deadline-jobs, QoPS is able to achieve better average slowdown and response time for the non-deadline jobs when compared to other schemes while accepting a higher load.

4.6.3 Impact on System

In this sub-section, we assess the widely used system metric - utilization (described in Section 2.1.6) to evaluate effectiveness of different schemes. From the supercomputer center’s perspective, it is a very important to see how efficiently the system is being
Figure 4.13: Average Slowdown for different loads where 80% jobs have user specified deadline with inexact user runtime estimate and Stringency Factor 0.2 for CTC trace

Figure 4.14: Average Slowdown for different loads where 20% jobs have user specified deadline with inexact user runtime estimate and Stringency Factor 0.2 for SDSC trace
Figure 4.15: Utilization for different loads where 20% jobs have user specified deadline with inexact user runtime estimate and Stringency Factor 0.2 for CTC trace

Figure 4.16: Utilization for different loads where 80% jobs have user specified deadline with inexact user runtime estimate and Stringency Factor 0.2 for CTC trace
Figure 4.17: Utilization for different loads where all jobs have user specified deadline with inexact user runtime estimate and Stringency Factor 0.2 for CTC trace

Figure 4.18: Utilization for different loads where 80% jobs have user specified deadline with inexact user runtime estimate and Stringency Factor 0.2 for SDSC trace
utilized in spite of offering the QoS guarantees in job scheduling. The achieved utilization for the different schemes as a function of load is shown in Figure 4.15, Figure 4.16, Figure 4.17 for 20%, 80% and 100% job with deadline respectively with stringency factor 0.2 for CTC trace where Figure 4.18 exhibits the same but for SDSC trace. It can be seen that the achieved utilization with QoPS is better than other schemes. As QoPS is frequently accepting more processor seconds compared to other schemes, better utilization in QoPS is expected.

4.7 Conclusions

Scheduling dynamically-arriving independent parallel jobs on a given set of resources is a long studied problem in space-shared parallel scheduling where solutions mainly focus to improve various system and user metrics such as utilization, throughput, turnaround time, etc. However, a solution to the problem of providing Quality of Service (QoS) guarantee for parallel job scheduling has been long overdue. In this chapter, to support deadline guarantee, we suggest several modifications of various schemes originally intended for different purposes. We, more remarkably, propose a new scheme termed as the QoPS to exclusively provide QoS guarantee in job’s response time for space-spared parallel system. We also perform extensively the trace-driven simulation experiments that evidently demonstrates the significantly better performance with respect to all metrics compared to other adapted schemes.
CHAPTER 5

REVENUE MANAGEMENT IN PROVIDER-CENTRIC MODEL

5.1 Introduction

A lot of research has focused on the problem of scheduling dynamically arriving independent parallel jobs on a given set of resources. The metrics evaluated include system metrics such as the system utilization, throughput [20, 35], etc., and user metrics such as the turnaround time, wait time [22, 37, 43, 44, 60, 62], etc. Recently, there has been some work in the direction of providing differentiated service to different jobs. The schemes that provide differentiation can be classified into two broad categories.

The first category comprises of approaches that provide “best effort” relative prioritization for individual jobs or different classes of jobs. Such prioritization may either be assigned statically to the jobs (e.g., jobs from a group of users might be given a higher priority compared to others), or may dynamically vary during the queue time of the job (e.g., if a job has been waiting in the queue for a long time, its priority is increased). The NERSC computing center [6] is an example environment which uses such a scheduler. NERSC offers different queues which have different costs and priorities: in addition to the normal priority queue, a high priority queue which double the usual charge, and a
low priority queue with half the usual charge. Jobs in the high priority queue get priority over the normal queue, until some threshold on the number of serviced jobs is exceeded.

The second category of schemes to provide differentiated service comprises of those which guarantee a certain Quality of Service (QoS) in the turnaround time for the submitted job. With such schemes, the users have an option of specifying the deadline they need with each submitted job. We are unaware of any production job schedulers that implement such a scheme, but our proposed scheme QoPS described in Section ?? and [41] is an example of such a scheme. QoPS implements an admission mechanism for incoming jobs, attempting various schedule rearrangements of previously admitted jobs to make a decision on whether the requested deadline is achievable without violating the deadlines provided for the other admitted jobs. If achievable, QoPS admits the job and guarantees the requested deadline to the job.

The overall issue of providing QoS for job scheduling can be viewed in terms of two related aspects:

- **Charging Model for Jobs**: All the non-QoS based charging model cannot be appropriate for QoS-aware scheduling. Also, there should be some incentive for the provider to offer the QoS guarantees to users. In general, the QoS-aware charging model should be based on the principle that the quicker the sought response time, the larger should be the charge. The charge will generally be a function of many factors, including the resources used and the load on the system.

- **Job Scheduling with Response-time Guarantees**: Users should be able to specify deadlines of their jobs by which they want jobs to be completed. In addition of supporting this guarantees, the QoS-aware scheduler should provide a admission
control mechanism by which a user can get the acceptance decision during job submission.

The QoPS scheduling algorithm presented in Section 8.4 addressed only the second aspect, i.e., how to implement admission control for deadline-based scheduling of parallel jobs, effectively exploiting the deadline flexibility among previously admitted jobs in order to maximize system utilization without violating deadlines of any admitted jobs. But a scheduling mechanism like QoPS, that implements admission control and provides deadline guarantees to admitted jobs, will be ineffective in providing effective differentiated services to users unless a suitable charging model is imposed. If the charge differential for provision of rapid versus slow turn-around for jobs is relatively small, all users might submit jobs demanding very tight deadlines, so that no effective differentiation will be achieved between urgent and non-urgent jobs submitted to the system. In this chapter, we propose extensions to the QoPS algorithm, that are motivated by considerations with respect to the job charging model. We perform evaluations to characterize trends with respect to a QoS cost-component and a resource-usage cost component. The issue of how to effectively combine the QoS component and resource-usage component to form the overall charging function is a difficult and open problem that is beyond the scope of this research. Here, we study two separate cost components for QoS and resource usage (as explained later in Section 3).

The enhancements to QoS-based scheduling that we present in this chapter are summarized below:

- Feedback on earliest feasible completion of unadmitted jobs: In our previous trace-based evaluation of QoPS, we associated deadlines with each job, and only
admitted jobs whose deadlines could be met without violating any prior commitments. Thus, if the requested deadline of a job was not satisfiable, it was simply dropped and not further considered for scheduling. In practice, it is likely that in some circumstances users would be willing to accept a slower response time than their original request, while in other situations they would not - they may choose to submit their job at some other center or choose to submit a different job instead. With a basic QoS-based scheduling scheme like QoPS, that implements admission control and deadline guarantees, users of unadmitted jobs with flexibility may need to iteratively resubmit their jobs with looser and looser deadlines until acceptance (or abandonment if the achievable deadline was unacceptably late). We develop a mechanism in QoPS to provide users of inadmissible jobs with feedback on the earliest guarantee-able deadline for the job.

- **Modeling User Tolerance:** Given the feedback mechanism described above, it is feasible to perform trace-based evaluations of QoS-based scheduling, where some jobs that cannot meet their originally requested deadline are nevertheless re-submitted with a more relaxed deadline. We performed simulations under different assumptions of user tolerance with respect to deadlines. We parameterize our studies with a Tolerance Factor (TF), that specifies the relative increase in response-time that a user is willing to tolerate. The effect on the QoS and resource components of cost are studied as a function of TF. A surprising result is that the total QoS component of revenue of a center does not monotonically increase with increasing tolerance on the part of the user.
• **Incorporation of Artificial Slack:** Although giving a job its best possible deadline would maximize the revenue achievable from that job, this might result in a tight schedule for the rest of the jobs causing many later arriving (and potentially more urgent jobs) to be dropped. On the other hand, providing an artificial slack to some of the jobs (which do not provide too much revenue) might result in later arriving urgent jobs to be admitted, causing an overall increase in the supercomputer center revenue. We study the effect of different degrees of artificial slack for various assumed tolerance factors.

• **Enabling Kill-and-Restart:** Some supercomputer centers implement mechanisms for Kill-and-Restart, where a running job can be killed to enable a different job to be started. Later, the killed job is re-started, but from scratch. We evaluate whether such a mechanism can be utilized to improve the overall profit in the supercomputer centers in the QoS-based scheduling context.

The remaining part of this chapter is organized as follows. Section 5.2 deals with the current cost model used by current supercomputer centers, our proposed cost model and its various components. In Section 5.3, we models the user tolerance and discuss the feedback based QoPS algorithm developed to provide the best possible deadline achievable by the job. The various techniques to maximize revenue is discussed in Section 5.4. Section 5.5 deals with the experimental setup we used in our simulation test-bed. We describe the impacts of user-tolerance and various approaches to counter its negative impacts in Sections 5.6 and present some concluding remarks in Section 5.7.
5.2 Provider-centric Charging Model

In general, Supercomputer center offers charging model that specifies how the price of executing a job is determined. Most of the existing charging models for non-QoS scheduling is a function resources used and usually computed by \((\text{number of processor} \times \text{runtime})\). As mentioned earlier, some systems such as NERSC [6] offer the users some choice by providing three queues: a normal queue, a high priority queue (with double the usual charge) and a low priority queue (with half the usual charge) but don’t give users any guarantee on the response time provided. However, the idea of charging the user based on the service or priority assigned to them is still relevant and critical to the practical applicability of a QoS based scheduler. In this dissertation, we discuss two different charging models for QoS-aware system. Firstly, in this section, we introduce a new charging model that is determined by the provider. Secondly, in next chapter, we present an existing user-centric charging model targeted for non-QoS system that can be adopted to employ in QoS-based system. In this section, we first discuss the charging model in current supercomputer centers. We extend this charging model to incorporate the QoS capabilities of the QoPS scheduler and present the various components associated with such a charging model.

The charging model in current supercomputer centers is mainly based on the resources utilized by the submitted jobs, and is unrelated to the responsiveness of the system. Thus, for example, when two jobs are submitted to the system with same resource but different response time requirements (e.g. job 1 wants the job to be started within an hour and job 2 doesn’t care about the response time), the circumstances need to be managed in economically rational way. It is obvious that the job 1 has to pay much more than the job 2 to get the response time guarantee. Whereas, QoPS guarantees that
response time constraint to the jobs, some charging model requires to deal with that proportionately. Currently, there is no charging model that considers the above cases. But without that every user will start asking to run their job immediately and hence providing the differentiated service to highly paid (or important) users will be unattainable.

In this model, we break up the total charge of a submitted job into two separate components: (i) Resource charge and (ii) QoS charge. The resource charge is similar to that used by current supercomputer centers and is based on the resources requested by the job. In general, this would depend on the various resources provided by the supercomputer center, e.g., CPU, memory, disk space, etc. In our environment, we only consider the CPU resource, i.e., the resource charge for a job would be equal to the product of the processors requested by the job and the time for which the job runs. This idea can easily be extended to other resources too.

The QoS charge, on the other hand, depends on the urgency of the job. For example, if two similar jobs are submitted where one of them is urgent while the other is not urgent, the resource charge for both the jobs would be similar, whereas the QoS charge would be much different. Further, if two different jobs request for similar urgencies, they could still have different QoS charges based on the “difficulty” of the supercomputer center in meeting the requested urgency.

In this chapter we use the slowdown of job as the base metric for deciding the QoS charge for the job. The “difficulty” of the supercomputer center in meeting the requested deadline depends on two components: (i) the current load in the system (the number of queued processor seconds) and (ii) the average slowdown of the category to which the job belongs. For example, typically short-wide jobs (ones which use a lot of processors but run for a small amount of time) have huge slowdown values while long-narrow jobs
have lesser slowdown values. Thus, we categorize short-wide jobs to be more “difficult” to schedule within the requested slowdown as compared to long-narrow jobs. We use the following equations for the resource and QoS charge for jobs:

\[ Resource\text{Charge} = Processors \times Runtime \]

\[ QoS\text{Charge} = \max((\frac{Category\text{Slowdown}}{Requested\text{Slowdown}} - 1) \times Runtime \times NProcs, 0) \]

5.3 Understanding User Tolerance

Since different users have their respective unpredictable requirements, it is difficult to model any user behavior. In this section, we attempt to formulate the users’ reaction toward the missed deadline of their jobs.

5.3.1 Modeling User Tolerance

Until now, we assumed that users are very rigid in their deadline restriction. When a job is submitted with a deadline, a user is unwilling to relax it whether the scheduler accepts or rejects it. But in reality, users are not that firm on their deadline, they might want to slacken their deadline in some extent. The scheduler can try to take this user behavior into consideration to capitalize on the revenue. But it is not easy to model the user behavior to determine whether any user will relax the deadline and if yes by how much. In our experiment, we tried to capture this widely varied user characteristics in a reasonable manner. For instance, we emulate user tolerance in terms of the factor of extension in the requested deadline that the user might be willing to accept in case the scheme fails to accept the job within the initially requested deadline. We quantify this factor by a parameter named Tolerance Factor (TF). The more flexible an user means larger value of TF where TF = 1.0 (minimum possible TF value) means that the user
is rigid in its deadline asking and hence offer no tolerance. Tolerance Factor (TF) is formally expressed as follows: \( TF = \frac{Maximum \ Acceptable \ Turnaround \ Time}{Requested \ Turnaround \ Time} \)

### 5.3.2 Feedback based QoPS Algorithm

As discussed earlier, in our original QoPS algorithm we associated deadline with each job and only admitted jobs whose deadlines is met without violating any prior commitments. Thus, a job whose requested deadline can not be satisfied is dropped and not considered further for scheduling. In practice, however, users might not be so strict about their requested deadline. Ideally, a user likes to have the job done by the requested deadline. If this is not possible, a different and less stringent deadline might also be fine with the user. However, by blindly dropping the job, such possibilities had not been considered previously. The scheduler can try to take this user behavior into consideration to capitalize on the revenue. With the basic QoPS scheme, users of unadmitted jobs with flexibility may need to iteratively resubmit their jobs with looser and looser deadlines until acceptance (or abandonment if the achievable deadline is unacceptably late). In this sub-section, we discuss the provision of providing a negotiation mechanism with the QoPS based algorithm which lets the users know the best possible deadline the system can provide to their job.

The basic idea of the scheme is to first try to provide the requested deadline to the submitted job. If the system is able to admit the job for this requested deadline, it just accepts the job. On the other hand, if the system fails to admit the job right away, the algorithm tries a fixed number of other possible deadlines (logarithmically refining the search for the best viable deadline). Based on these trials, it provides the user with what it thinks is the best possible deadline for the job that the system can provide. In
practice, it is the user discretion to decide whether to accept this proposed deadline. In our simulation, to imitate this user behavior, we make use of Tolerance Factor (described in previous sub-section) as a simulation parameter in our analysis. Figure 5.1 illustrates the pseudo code for this algorithm where original QoPS schedule is considered as a function.

5.4 Techniques for Revenue Maximization

In this section, we explore two independent techniques to further enhance the overall revenue. The first technique is to add an artificial slack to the scheduler-proposed deadline when FQoPS fails to accept a job within the user requested deadline. The second
technique can be applied to the centers where kill and restart mechanism is supported. In this scheme, a running job can be killed and rescheduled in a later time to accommodate another job that would have failed otherwise.

5.4.1 Incorporating Artificial Slack

Using the above-mentioned FQoPS algorithm, if we are unable to accept a job with the user requested deadline, we try to offer the best (and stringent) possible deadline to the job. Though it maximizes the revenue achievable from that job, but this might result in a tight schedule for the rest of the jobs causing many later arriving (and potentially more urgent) jobs to be dropped. In this section, we propose a technique to manage this type of situation by introducing slack in the offered deadline.

Since the scheduler offers this deadline and the scheduler objective is to achieve the overall revenue instead of satisfying any specific job, there is scope for the scheduler to propose different levels of stringent deadline. In this chapter, we provide a certain artificial slack to such jobs and return an even looser deadline to the user. If the user consents to accept the job with this deadline, the supercomputer center would gain more flexibility to admit later arriving jobs. We recognize that this model might increase the possibility of user rejection due to the slackened deadline but this increases our overall revenue outlook. We model this slack with an additional parameter called Slack Factor (SF). The offered deadline to the user is estimated by 
\[
(ArrivalTime + (Earliest\ possible\ deadline - ArrivalTime) \times SF).
\]
For example, SF= 1.0 denotes that there is no extra slack added. SF=2.0 means that there are additional slack equals to the amount of the earliest possible turnaround time. To study how this additional slack effects in overall revenue, we evaluate different values of SF ranging
from no slack to a very large slack. Slack Factor (SF) is mathematically defined as

$$SF = \left( \frac{Provided\ Turnaround\ Time}{Earliest\ Possible\ Turnaround\ Time} \right)$$

5.4.2 Utilizing Kill-And-Restart Mechanism

Some supercomputer centers support the kill-and-restart mechanism where a running job can be killed and restarted as a new job if no permanent files are modified during the run. The model of I/O for such jobs at the Ohio Supercomputer Center is that all input files are first copied into a special temporary directory created for the job, and all output is written into files in that temporary directory during execution of the program; after successful completion of the program, the output files are copied from the temporary directory to the persistent files. If such a job is aborted and then restarted from scratch and run to completion, the final results would be exactly the same as running to completion on the first initiation. In this section, we present an approach to utilize such capabilities provided by the supercomputer center to further improve the profit achievable.

The basic idea of the algorithm is to schedule a new job by killing a running job if the QoPS scheme fails to find a schedule for new job within its deadline. In this approach, instead of trying to find the earliest possible deadline for a job whose initially requested deadline could not be satisfied, we try to satisfy the requested deadline by killing and rescheduling an already running job. If we are able to satisfy the requested deadline of the new job, we try to reschedule the killed job within its guaranteed deadline. If we are able to schedule this killed job too, the final schedule is accepted.

Our scheme uses a heuristic approach to determine the order in which running job should be killed. The running jobs are sorted in the increasing order of used processors
seconds (the processor seconds for which they have already run so far). We then kill the first job in the list and try to schedule the new job. If an acceptable schedule is found for the new job, we try to schedule the killed job again within its guaranteed deadline. If both jobs are thus scheduled without violating their deadlines, we accept the new schedule. Otherwise we keep the first job running and follow the same steps with the second running job from the sorted list. If no schedule is possible after trying all the running jobs, we try to find the acceptable deadline that we can offer and return it to the user.

5.5 Evaluation Approach

For experimental setup of our simulation, we adopted the same idea described in Section 4.5. In this section, we only discuss the portion that is exclusively needed for this experiment.

As mentioned in Section 5.2, the QoS charge of a job is based on the slowdown of the respective category to which this job belongs, i.e., short-wide jobs have a high category-slowdown and hence would have a higher QoS charge for a requested slowdown as compared to long-narrow jobs which have a low category-slowdown.

We classify jobs into different categories by using two parameters: the runtime of the job and the number of processors requested by the job. In particular, we split up both the time and processors into sixteen different categories ranging from short-narrow jobs to long-wide jobs. We run the EASY backfill for different loads with several stringency factors and calculate the corresponding category slowdown for the various types of jobs that is ultimately used to compute the QoS cost.
5.6 Experimental Results

From Supercomputer center perspective, the revenue is a very critical metrics to take into account. While providing the QoS to user is a very fascinating idea to the center, but without any appropriate revenue incentive, the initiative might not be very attractive to the center. In this section, we confront this challenge started with the understanding of user tolerance that the scheduler can manipulate to acquire better revenue. We then explore the effects of two techniques (described in Section 5.4) to further increase the overall revenue.

5.6.1 User Tolerance and Revenue

As discussed earlier, when a new job is submitted, the Feedback based QoPS scheme (FQoPS) tries to find a schedule without violating the deadline constraints of either the new job or the existing jobs in the schedule. If it is able to satisfy the deadline restriction, it accepts the job. However, if it is unable to satisfy the deadline, it doesn’t drop it instantaneously. Instead, it finds the best possible deadline it can provide to the job and returns it to the user. Then it is in the user’s discretion to accept or reject the offered deadline. In the modeling of our scheme, we tried to emulate the most rational behavior of users. For instance, we emulate user tolerance in terms of the factor of extension in the requested deadline that the user might be willing to accept in case the scheme fails to accept the job within the initially requested deadline. We quantify this factor by a parameter named Tolerance Factor (TF). This parameter is completely based on the user characteristics. In our simulations, we show the impact of such tolerance on the part of the user on the overall profits attainable by the supercomputer center by choosing
Figure 5.2: Resource Charge for different loads for Stringency Factor = 0.2: where (a) all jobs have user specified deadline (b) 80% jobs have user specified deadline

different assumed values for this parameter. In this section, we explain with graphs the relationship between user tolerance and earned revenue.

Whenever FQoPS scheme fails to schedule a job within the user requested deadline, it finds a schedule at the earliest possible time. If the earliest possible deadline is within user tolerance given by (TF x Requested Deadline), we assume that the user would accept the offered deadline. Otherwise, we assume that the user would not accept the provided deadline.

In our first set of simulation experiments with FQoPS, we study the behavior of the scheme with respect to resource charge and QoS charge metrics as a function of TF. We assume that all users have the same amount of tolerance (equal TF for all jobs). These experiments try to show critical insights into the real impact of the user tolerance without being diluted by the difference in the tolerance amongst various users. TF defined as the simulation parameter shows the general characteristics of users in this case, for instance, a high value of TF would create a scenario where users are more tolerant while a low value of TF would create a scenario where users are less tolerant.
Figure 5.2 shows the variation of the resource charge of the system with TF for load 1.3 and load 1.6. As the TF value increases, the overall resource charge increases. When the TF value becomes sufficiently large, however, the trend is more intuitive with more and more jobs being accepted. After some point as the TF value increases, the Resource cost does not vary because it reaches to the saturation. In other words, for higher TF value, most of the jobs are accepted and increasing the TF value further does not facilitate anything. It is shown in the graph that this saturation point comes earlier for lower load (1.3) when it is compared with higher load (1.6). Because at higher load there are more jobs to be accepted and hence requires higher TF value to reach to the maximum. The same trend is shown for 80% jobs with user specified deadline with one supplementary observation. The resource cost reaches to saturation for relatively lower value of TF because there are few jobs with deadline.

Figure 5.3 shows the QoS Charge as a function of TF for different loads. This figure shows a counter-intuitive result of a monotonic drop in the QoS charge with user tolerance, i.e., if the users are more tolerant, the supercomputer center gets lesser profit
from the jobs! This is attributed to the per-job profit achieved with increasing user
tolerance. With an increase in the TF value, jobs which would have failed in the original
no-tolerance scheme are now accepted. Due to this, later arriving jobs which would
have been accepted with the user requested deadline are unable to be admitted with
this deadline. However, due to the increased user tolerance, they are still admitted with
a looser deadline, i.e., the later arriving jobs are still admitted, but for a lesser QoS
charge. This effect cascades for later arriving jobs causing an overall decrease in the
QoS charge the supercomputer center can get.

5.6.2 Providing Artificial Slack

As discussed in Section 5.6.1, most of the time, users are not very strict in their
requested deadlines. If the supercomputer center is not able to meet their requested
deadline, we try to find the best possible deadline the center can provide and let the user
decide if this deadline is acceptable. However, as we have seen, though the resource
charge achieved by the supercomputer center increases as the users become more and
more tolerant, the overall QoS charge falls. In this section, we illustrate the effect of
slack on the two components of revenue.

Depending on the ratio of the QoS charge to resource charge imposed by the super-
computer center, a high tolerance might provide a higher or lower overall profit. One
extreme is for the supercomputer center to have no explicit charge for the QoS provided,
in which case the center has to do nothing; the tolerance of the users would automatic-
ically improve the profit achievable by the center. The other extreme is to charge the
users only based on the QoS they requested. In this case, the original QoPS algorithm
would perform the best, since its behavior would be equivalent to that of no user toler-
ance (TF = 1.0). It is to be noted that user tolerance is completely dependent on the user

Figure 5.4: Resource Charge for different Slack Factors with Stringency Factor = 0.2 and load 1.6: (a) all jobs have user specified deadline (b) 80% jobs have user specified deadline

characteristics and is not a parameter under the control of the supercomputer center. So, in the general case where the overall profit depends on both the QoS charge as well as the resource charge, we need to come up with a different approach to try to negate the effect of the user tolerance to any required degree. We consider incorporating a slack to the earliest possible deadline provided by the scheduler to any first time failed jobs. This will give the flexibility to the scheduler for later arriving job though it might slow down the current job in some extent.

We study different cost metrics with different slack factor (SF) and tolerance factor (TF) values. It is worthy to note that the value of Slack Factor could be 1 or more (SF = 1 means no slack). Also for any TF value less than SF, the scheme behaves like our original QoPS, i.e. all the jobs whose initially requested deadline could not be met, are dropped immediately. Further, in general, an increase in the value of SF tends to negate user tolerance, i.e., increasing SF tends to be equivalent to decreasing TF. We evaluate the metrics for both moderate load (1.3) and high load (1.6).
Figure 5.4 shows the variation of the resource charge with the Slack Factor (SF) for different Tolerance Factors. Again, with increasing SF, larger jobs have a higher absolute slack value. This lets more small jobs to be admitted by utilizing this slack. The admittance of these small jobs, however, uses up the slack present in the schedule and forcing later arriving larger jobs to be dropped. So, in general, increasing SF has the inverse effect of increasing TF.

Figure 5.5 shows the variation of the QoS charge with the Slack Factor (SF) for different Tolerance Factors. Again, we see a similar effect as the previous graph: increasing SF has the inverse effect of increasing TF, i.e., QoS charge increases with increasing SF.

### 5.6.3 Kill and Restart

Kill and Restart mechanism is available for some super computer center where a running job can be killed and restarted from the beginning. As discussed in section 5.3.2, we modify our schemes FQoPS by incorporating the Kill and Restart mechanism to further improve the overall revenue. In this section, we describe how kill-and-restart mechanism could assist in improving overall revenue. Figures 5.6 and 5.7 show the
Figure 5.6: Resource Charge with Kill-and-Restart and without Kill-and-Restart for Stringency Factor = 0.2, load = 1.6 and TF = 4.0 (a) all jobs have user specified deadline (b) 80% jobs have user specified deadline

Resource and QoS charges achievable by the supercomputer center for varying slack factors with tolerance factors of 4. In each case, we consider jobs where all jobs have user specified deadlines and 80% jobs have user specified deadlines respectively. The gain in the QoS charge is attributed to the capability of the Kill-and-Restart scheme to admit later arriving urgent jobs by killing one of the already running jobs. On the other hand, the loss in the resource charge is attributed to the wastage of processor-seconds due to the restarting of killed jobs. We don’t observe any difference between the two schemes for a low tolerance factor due to the low amount of flexibility available in the schedule.

In general, if the system has a mix of urgent and non-urgent jobs, Kill-and-Restart allows us to admit more urgent jobs by killing already running non-urgent jobs and increase the overall QoS cost for the system. To further strengthen this argument, we study the impact of the Kill-and-Restart based scheme on workloads where not all jobs require deadlines. This is a more realistic scenario for real supercomputer centers. Some jobs request a hard deadline and pay more (in the form of the QoS charge) while others
Figure 5.7: Resource Charge with Kill-and-Restart and without Kill-and-Restart for Stringency Factor = 0.2, load = 1.6 and TF = 4.0 (a) all jobs have user specified deadline (b) 80% jobs have user specified deadline

don’t request any hard deadline and pay lesser (only the resource charge). We however provide an artificial deadline to even the non-deadline jobs to ensure that there’s no starvation in the system. At the same time, these jobs are given sufficient slack so that they don’t interfere in the admission of true deadline based jobs.

5.7 Conclusions

Although there has been considerable research on the topic of scheduling of parallel jobs, the issue of provision of QoS has received little attention. In this chapter, we extend a previously proposed scheme (QoPS) to provide deadline guarantee; we propose extensions to the algorithm in multiple aspects: (i) a feedback mechanism to provide the best possible deadline for jobs whose requested deadline could not be met, (ii) providing artificial slack to some jobs to maximize the overall profit the supercomputer center can achieve and (iii) utilizing Kill-and-Restart improve the profit attainable.
6.1 Introduction

Recent emerging architectural breakthroughs coupled with the acceptance of parallel concepts in general purpose computing revolutionize the importance of parallel systems in numerous ways. Effective resource management of such huge and demanding systems, as well as satisfying both the resource provider and the user with diversified objectives, is a very challenging problem. Considerable research [35, 60, 44, 66, 73, 41] has already focused on these seemingly orthogonal aspects of parallel job scheduling.

In general, a job scheduler determines when and where to execute a job, given a set of resources and a stream of jobs. In a typical model, when a job arrives in the system, the scheduler tries to allocate the required resources to start the job immediately, if the specified resources are available. Otherwise, the job is queued and scheduled to start at a later time. User satisfaction is often evaluated by response time which is the sum of the waiting time in the job queue (for resources to be available) and the actual runtime after the job starts running. In contrast, a supercomputer center is usually interested in
the overall system utilization that determines what fraction of the resources is actually utilized and what fraction remains idle.

While shorter response time and larger utilization are very appealing features from the supercomputing community, overall revenue maximization with the optimal management of resources is the key motivational factor from a resource provider’s perspective. Revenue computation primarily requires a suitable charging or cost model. There are two different charging models available in literature. In the provider-centric model, the supercomputer center determines the charge required to execute a job. This widely used model is based on the resources utilized and is usually computed by employing the product, \( \text{number of processors} \times \text{runtime} \). On the other hand, in the user-centric model, the user, instead of the system provider, offers the price for running a job. According to the adopted model, user specifies the value of a job depending on the importance of timely delivery. Basically, each user defines a piece-wise linear value or utility function whereupon the charge is calculated as function of completion time of a job.

In this chapter, we adopt the user-centric approach on the lines of the market-based charging model originally proposed by Culler et. al [21] and Chase et al [39] for sequential jobs. Both papers proposed various heuristics to maximize the revenue based on expected unit yields and other risk and reward related parameters. However, they do not provide any theoretical backing of the heuristics they employ and their work lacks any impact analysis on widely used performance metrics. In this chapter, we propose a new scheduling heuristic which aims at revenue maximization in a online multi-processor system. More significantly, we prove the optimality of our approach in a simplified scenario involving a uni-processor system and an offline batch of jobs. Then, we propose sufficient conditions which when true, guarantee optimality, for an online stream.
of jobs on a uni-processor system. Finally, we apply our proposed scheduling scheme in a generic multiprocessor system with parallel jobs.

We present the detailed analysis of the schemes with trace-based simulation using different real workloads of jobs from Feitelson’s archive [29]. Our results demonstrate that the proposed scheme provides significantly higher revenue as compared to existing schemes while achieving the better performance with respect to standard performance metrics such as slowdown and utilization.

### 6.2 Related Work

Several job schedulers such as Portable Batch System (PBS) [8], Moab [5], Load Sharing Facility (LSF) [4], Sun Grid Engine (SGE) [13], etc. have been deployed at shared-resource supercomputer centers to schedule parallel jobs. These schedulers primarily focus on performance by improving utilization, throughput, average turnaround time [35, 20], etc. Even though maximizing revenue is a very desirable feature from a resource provider’s perspective, it has received very little attention from both the industry [10, 3, 6] and the research community [42]. In this section, we review some of the existing work pertaining to the concept of revenue in job scheduling.

Typically, the service provider calculates the revenue by using either the provider-selected charging model or the user-specific charging model. Most supercomputer centers [10, 1, 3] adopt the provider-centric approach, in which a user is charged in proportion to its resource usage. In most of the cases, the basic resource is a processor; the charge is proportional to the product of the required number of processors and the run time of a job. Some supercomputer centers [10, 3] that provide multiple queues for different levels of services determine the charges depending on the resource used.
and quality of services sought. In this model, the charge is generally proportional to \((\text{number of processors} \times \text{runtime} \times \text{queuecharge})\). In addition, we propose a new charging model for QoS-aware scheduling in our recent work [42]. The proposed charging model is based on the notion that quicker the response time sought, the larger the charge should be. In particular, there are two separate charging components associated with resource usage and QoS guarantees respectively. The resource usage component relies mainly on the amount of resources used and the duration of the service. The QoS component of the charge essentially depends on the flexibility of the requested deadline.

Alternatively, there has been some work done to provide user-centric market-based approaches for revenue-aware scheduling [21, 39, 63]. In the user-centric approach, different users with varied goals and preferences express their desire for service in a common way (e.g., through currency). The most common market-based model follows a auction-based resource allocation mechanism that has three major entities: users or buyers, system providers or sellers and the resources to be sold [74, 72]. A user wants to allocate a processor(s) for a specific duration and is willing to pay a certain value for the execution of the job. The system provider is interested in selling the resources to the user with an intent to maximize its overall profit. The auction process, which is generally proposed by the system provider, considers the value or bid of all contending users and ultimately awards it to the highest bidder.

Wladspurger et. al., [74] proposes a market-based microeconomic approach to batch scheduling. They utilize the auction process to choose the winner from the bids of different users. Stoics et. al. [72] also propose an auction-based microeconomic approach for parallel job scheduling. In this scheme, every user has a savings account where he or she receives funds for buying resources for jobs. Also, the user creates an independent
expense account for every job and starts transferring funds from his or her savings to a job’s expense account. The rate of this fund transfer determines the priority of the job that ultimately plays a vital role in the auction process.

Two recent works [21, 39] have also looked at a market-based approach to value-based parallel job scheduling. Both studies rely on a per-job specific utility or value function that provides an explicit mapping of service quality to value. Generally, the value function is a piece-wise linear function that decays as a function of the job completion time. The rate of decay reflects the urgency or sensitivity to delay. The fundamental idea of this model is that the user submits the job with a value function along with other job characteristics. Then, the scheduler decides how to schedule the job using the job information and current state of the system. Culler et. al. [21] adopts user-centric performance metrics instead of system-centric metrics to evaluate the overall system performance. They recommend an aggregate utility function to measure the satisfaction of users with the resource allocation process. For job selection, the proposed scheme implements a heuristic approach where the job with highest value per unit running time gets the preference. Chase et. al. [39] proposes an enhancement to this approach by considering the risk of accepting or rejecting a job due to future uncertainty. Islam at. el. [40] addressed the issue of providing deadline guarantees as well as achieving maximum revenue using a user-centric approach. Their work focused on analyzing the opportunity cost in a QoS-aware scheduler and proposed a history-based predictive technique to estimate the opportunity cost and increase the overall system revenue.
6.3 Value-based Scheduling

In this section, we examine FirstPrice [21], PresentValue, OpportunityCost, FirstReward approaches [39] proposed in the context of value-based scheduling. The key aspect of the job model is that a user specifies the value of a job depending on the importance of timely delivery. Basically, each user defines a piece-wise linear value or utility function (shown in Figure 7.2) whereupon the charge is calculated as a function of the completion time of a job. The value function reflects the urgency of the job as a time-dependent function. For simplicity, the value function has two linear pieces. The first part indicates the maximum value that a user is willing to pay if it is completed in its earliest possible time, which in turn expresses the importance of the job. Likewise, the second part denotes decay or down-slope of the value and shows the sensitivity of the job to further delay in the job’s completion time. It is expected that users with tight deadline jobs will offer a high initial value and steeper slope for the job. Eventually, the aggregate utility,
which is calculated by summing up the individual value earned for all the jobs in the system, can be used to estimate the overall system performance.

Formally, let $S = \langle j_1, j_2, \ldots, j_n \rangle$ be a set of jobs. Let $EST_i$ be earliest start time of job $j_i$, $P_i$ be the processing time of job $j_i$, and $Slope_i$ be the slope of the linearly decreasing value for the job $j_i$ over time. Each job $j_i$ earns a maximum value $MaxRev_i$ if it completes at its minimum completion time $EST_i + P_i$. If the job is delayed, then the value decays linearly at the rate $slope_i$. Since the value starts deteriorating only once the job completion time exceeds the earliest completion time, therefore we focus our analysis on the decreasing portion of the curve. The time varying value $Value_{it}$ in the decreasing portion of the interval is of the form

$$Value_{it} = MaxRev_i - Slope_i \times t$$ \hspace{1cm} (6.1)

### 6.3.1 FirstPrice.

Jobs are ordered for execution in the decreasing order of their expected yield per unit of resource per unit of processing time. In other words, FirstPrice orders the jobs based on the unit gains they offer.

$$FirstPrice_i = \frac{Value_{it}}{P_i}$$ \hspace{1cm} (6.2)

### 6.3.2 PresentValue.

This approach builds upon the idea proposed in the FirstPrice scheduling criterion. It is based on the concept of present value in finance. The key idea is that, if there are two jobs with the same unit gains and slopes, then its preferable to run the shorter job first. This is because a shorter job reduces the risk of delaying a highly urgent or highly
valuable job which arrives later. In other words, executing the shorter job first makes the scheduling more risk-aware. The present value of a job $PV_i$ is defined as follows.

$$PV_i = \frac{Value_{it}}{1 + DiscountRate \times P_i}$$  \hspace{1cm} (6.3)

Here, $DiscountRate$ is a parameter which is used to decide the weightage to be given for future gains.

### 6.3.3 OpportunityCost.

The $OpportunityCost$ models the loss incurred by a job $j_k$ whose execution is delayed in order to execute another queued job $j_i$. A job’s urgency cost depends upon the urgency of other queued jobs. The opportunity cost $OC_i$ to start a job at some point is equal to the aggregate decline in the yield of all other competing jobs. Formally, it is defined as follows.

$$OC_i = \sum_{\forall k, k \neq i} slope_k \times P_i$$  \hspace{1cm} (6.4)

### 6.3.4 FirstReward.

The $FirstReward$ heuristic tries to balance the competing effects of a job’s present value as well as its opportunity cost. It does so by introducing a tunable parameter alpha as follows.

$$FirstReward_i = \frac{\alpha \times PV_i - (1 - \alpha) \times OC_i}{P_i}$$  \hspace{1cm} (6.5)
6.4 Normalized Urgency

In this section, we present a new approach, namely, *NormalizedUrgency*, to solve the aforesaid problem. The objective function is to maximize the total value earned by the jobs. The problem is how to schedule the jobs so as to maximize the overall value earned. In other words, the goal is to maximize $\text{Rev}$ which is defined as follows.

$$
\text{Rev} = \sum_{i=1}^{n} \text{Value}_{it} \quad (6.6)
$$

For each job, we define an ordering function $\text{NormalizedUrgency}_i$, defined as follows:

$$
\text{NormalizedUrgency}_i = \frac{\text{slope}_i}{P_i} \quad (6.7)
$$

Jobs are ordered for execution in the decreasing order of their $\text{NormalizedUrgency}_i$ values. In order to provide an intuition for our approach, we first look at certain restricted versions of the general problem and propose optimal solutions or sufficient conditions to guarantee optimality for them.

6.4.1 Batch of jobs: Offline scenario

In this section, we look at the following restricted version of the problem. All jobs are associated with the same earliest start times. In other words, we are focusing on an offline version of the problem where all jobs are available to be executed at time $t=0$. The problem is how to schedule these jobs on a uniprocessor system to achieve our goal function as mentioned in (Eq. 6.6).
**Theorem 1.** An optimal sequence can be obtained by ordering the jobs in non-increasing order of their respective NormalizedUrgency values.

**Proof.** Consider two jobs $j_i$ and $j_k$ such that the optimal sequence is to execute job $j_i$ before job $j_k$ and let us consider that $\text{NormalizedUrgency}_i < \text{NormalizedUrgency}_k$. The total revenue attributed to these jobs, $\text{Rev}_{ik}$ is:

$$
\text{Rev}_{ik} = \text{MaxRev}_i - \text{slope}_i \times P_i + \\
\text{MaxRev}_k - \text{slope}_k \times (P_i + P_k) 
$$

(6.8)

Now swap the two jobs so that job $j_k$ now executes before job $j_i$. The revenue attributed to the two jobs in this case, $\text{Rev}(ki)$ is:

$$
\text{Rev}_{ki} = \text{MaxRev}_k - \text{slope}_k \times P_k + \\
\text{MaxRev}_i - \text{slope}_i \times (P_k + P_i) 
$$

(6.9)

The difference in the overall revenue earned is:

$$
\text{Rev}_{ki} - \text{Rev}_{ik} = \text{slope}_k \times P_i - \text{slope}_i \times P_k 
$$

(6.10)

The right hand side in (Eq. 6.10) is positive since $\text{NormalizedUrgency}_i < \text{NormalizedUrgency}_k$. Therefore, swapping the two jobs to obey a non-increasing order of the NormalizedUrgency values leads to an increase in the overall revenue earned. Therefore, the former solution is not optimal. A sequence of such interchanges yields a sequence which satisfies the order specified in the claim and is optimal.

□

107
6.4.2 A dynamically arriving stream of jobs (with same slope): Online scenario

In this section, we look at a more relaxed version of the problem discussed in section 6.4.1. All jobs are associated with the possibly different earliest start times $EST$. In other words, we are focusing on an online version of the problem where all jobs arrive over time. However, we assume the knowledge of earliest start times for jobs arriving in future. The optimal solution for such a problem can act as an upper bound on the total revenue earned in a truly online scenario where jobs arrive over time and the scheduler does not have any knowledge of the arrival times of future arriving jobs. For simplicity sake, we assume that all the jobs have the same decay rate $slope$.

The earliest completion times $ECT_i$ of each job is calculated dynamically based on the previously scheduled jobs. In the initial state, when no job has been scheduled, the earliest completion time $ECT_i$ of each job is defined as follows:

$$ECT_i = EST_i + P_i$$ (6.11)

**Theorem 2.** Two conditions, which when true together, guarantee optimality. Condition 1 states that jobs should be scheduled in a non-decreasing order of their earliest start times. Condition 2 says that the jobs should be scheduled in a non-decreasing order of earliest completion times. If a schedule obeys both these conditions, it is guaranteed to give an optimal solution.

**Proof.** We first prove this in the context of a two job scenario where in there are only two competing jobs in the system.

Consider two jobs $j_i$ and $j_k$. Without loss of generality, Let us consider that $ECT_i < ECT_k$. There are three possible scenarios.
1. **Scenario 1:** \( EST_i < EST_k < EST_i + P_i < EST_k + P_k \)

If job \( j_i \) executes before job \( j_k \), the total revenue earned, \( Rev_{ik} \) is computed as follows:

\[
Rev_{ik} = MaxRev_i - \text{slope} \times (EST_i + P_i) \\
+ MaxRev_k - \text{slope} \times (ECT_i + P_k) \quad (6.12)
\]

If job \( j_k \) executes before job \( j_i \), the revenue \( P_{ki} \) earned is:

\[
Rev_{ki} = MaxRev_k - \text{slope} \times (EST_k + P_k) \\
+ MaxRev_i - \text{slope} \times (ECT_k + P_i) \quad (6.13)
\]

The difference in the overall revenue earned is:

\[
Rev_{ki} - Rev_{ik} = \text{slope} \times (EST_i + ECT_i \\
- EST_k - ECT_k) \quad (6.14)
\]

In this scenario, \( EST_i < EST_k < ECT_i < ECT_k \), therefore the right hand side in (Eq. 6.14) is negative. Therefore, the overall value decreases if job \( j_k \) executes before job \( j_i \).

2. **Scenario 2:** \( EST_k < EST_i < EST_i + P_i < EST_k + P_k \)

If job \( j_i \) executes before job \( j_k \), the revenue \( P_{ik} \) earned is:
\[ \text{Rev}_{ik} = \text{MaxRev}_i - \text{slope} \ast (\text{EST}_i + P_i) + \text{MaxRev}_k - \text{slope} (\text{ECT}_i + P_k) \]  
\[ (6.15) \]

If job \( j_k \) executes before job \( j_i \), The revenue \( P_{ki} \) earned is:

\[ \text{Rev}_{ki} = \text{MaxRev}_k - \text{slope} \ast (\text{EST}_k + P_k) + \text{MaxRev}_i - \text{slope} (\text{ECT}_k + P_i) \]  
\[ (6.16) \]

The difference in the overall revenue earned is:

\[ \text{Rev}_{ki} - \text{Rev}_{ik} = \text{slope} \ast (\text{EST}_i + \text{ECT}_i - \text{EST}_k - \text{ECT}_k) \]  
\[ (6.17) \]

In this scenario, two conditions are sufficient to make the right hand side of the (Eq. 6.17) negative. The two conditions are: \( \text{EST}_i < \text{EST}_k \) and \( \text{ECT}_i < \text{ECT}_k \).

If these two conditions are true, then the order \( ik \) is certainly better than the order \( ki \). However, the condition \( \text{EST}_i < \text{EST}_k \) violates the assumptions of Scenario 2 that is \( \text{EST}_k < \text{EST}_i \).

3. **Scenario 3:** \( \text{EST}_i < \text{EST}_i + P_i < \text{EST}_k < \text{EST}_k + P_k \)

In this scenario, it is quite obvious that job \( j_i \) should be executed before job \( j_k \). This is also reflected by the fact that both the two sufficient conditions for the
ordering $ik$ to be better than the ordering $ki$: $EST_i < EST_k$ and $ECT_i < ECT_k$ are true in this scenario.

From these three possible scenarios, we can safely conclude that there are two conditions which are sufficient to guarantee optimality. The conditions are $EST_i < EST_k$ and $ECT_i < ECT_k$. In other words, a sequence which satisfies a non-decreasing order of earliest start times as well as the earliest completion times is optimal for a 2 job scenario.

\[\square\]

6.4.3 A dynamically arriving stream of jobs (different slopes): Online scenario

In this section, we look at an extended version of the problem discussed in Section 6.4.2. All jobs are associated with the possibly different earliest start times $EST_i$ as in Section 6.4.2. However, now we relax the assumption that all jobs have same decay rates. Each job $j_i$ is associated with a decay rate $slope_i$.

**Theorem 3.** Two conditions, which when true together, guarantee optimality. Condition 1 states that jobs should be scheduled in a non-decreasing order of their respective $EST \times slope$ values. Condition 2 says that the jobs should be scheduled in a non-decreasing order of their respective $ECT_{slope}$ ratios. If a schedule obeys both these conditions, it is guaranteed to give an optimal solution.

Consider two jobs $j_i$ and $j_k$. Let us consider that $ECT_i < ECT_k$. There are three possible scenarios, of which we look at one in this section.

$EST_i < EST_k < EST_i + P_i < EST_k + P_k$
If job $j_i$ executes before job $j_k$, the component of the total revenue earned which is attributed to these two jobs, $Rev_{ik}$, is computed as follows:

$$
Rev_{ik} = \text{MaxRev}_i - \text{slope}_i \times (\text{EST}_i + P_i) \\
+ \text{MaxRev}_k - \text{slope}_k (\text{ECT}_i + P_k)
$$

(6.18)

If job $j_k$ executes before job $j_i$, the revenue earned is:

$$
Rev_{ki} = \text{MaxRev}_k - \text{slope}_k \times (\text{EST}_k + P_k) \\
+ \text{MaxRev}_i - \text{slope}_i \times (\text{ECT}_k + P_i)
$$

(6.19)

The difference in the overall revenue earned is:

$$
Rev_{ki} - Rev_{ik} = \text{slope}_i \times \text{EST}_i - \text{slope}_k \times \text{EST}_k \\
+ \text{slope}_k \times \text{ECT}_i - \text{slope}_i \times \text{ECT}_k
$$

(6.20)

In order for the right hand side in Equation 6.20 to be negative, there are two sufficient conditions: $\text{slope}_i \times \text{EST}_i < \text{slope}_k \times \text{EST}_k$ and $\text{slope}_k \times \text{ECT}_i < \text{slope}_i \times \text{ECT}_k$. Therefore, we again have two possibly conflicting conditions. Ordering the jobs according to one condition may violate the another.

\[\square\]

6.5 Experimental Results

In this section, we experimentally compare the performance of our proposed scheme $\text{NormalizedUrgency}$ against the existing schemes such as $\text{First Price}$ [21], $\text{Present Value}$,
Opportunity Cost and First Reward [39]. We employ an event-based simulator that essentially takes data in the standard workload format version 2.0 [29], simulates the scheduling model and creates an output trace containing data necessary to gather metrics and perform post processing.

![Revenue Improvement](image)

(a) 50% Urgency Job

(b) 20% Urgency Job

Figure 6.2: Revenue improvement relative to FirstPrice assuming exact runtime estimate at different offered loads

We primarily compare the overall revenue gained from different schemes. In addition, other performance metrics such as slowdown, response time and system utilization are also evaluated to analyze the correlated impact of improving the revenue. Moreover, since the parallel system with dynamically arriving parallel jobs could produce varied scenarios, we further study the robustness of the schemes by varying the offered loads and also looking at inexact user estimates.
Figure 6.3: Revenue improvement relative to FirstPrice assuming exact runtime estimate at different offered loads with 80% urgent job

6.5.1 Evaluation Approach

The strategies evaluated in this chapter are simulated using real workload traces, such as those available at the Parallel Workload Archive [29]. These traces include information such as the jobs runtime, the number of nodes each job used, the submission time, and a user estimated runtime limit (wall-clock limit). In this work, we used two such real workload trace of 10,000 jobs subset of the SDSC SP-2 and CTC SP-2 workload trace.

**Runtime Estimates:** Runtime estimates are critical when evaluating parallel job schedulers. Therefore, two sets of simulations are performed. The first set of simulations
takes an idealistic view of the traces and assumes that users are able to perfectly estimate their job’s runtime; this allows us to concentrate on the capabilities of our algorithms without being affected by other noise in the traces. The second set of simulations uses the actual runtime estimates given in the workload trace; this allows us to evaluate our algorithms in more realistic environments.

**Job Submission Load:** As the demand for resources is increasing, we need to investigate the effectiveness of our scheme in various high load environments. This necessitates the expansion of the real traces in a rational way. We mainly use the duplication approach for varying the load on the supercomputer center (number of jobs submitted). Jobs are selected randomly and duplicated with the same arrival time keeping other attributes value unchanged. For example, we start with a trace and call this the base trace (load = 1.0). To generate a new trace with load = 1.2, we randomly pick 20% of the jobs in the base trace and introduce extra duplicate jobs at the same points in the trace. We also pay special attention to maintain the subset relationship of jobs across the different loads. For example, the jobs selected for load 1.2 are also selected for load 1.3 with additional 10% new jobs. In summary, for CTC trace, we start with base offer load of 74% and increased the load to 80% and 88%. For SDSC trace, the offered loads are 82% (base trace), 90% and 104%.

**Urgency and Job Cost:** None of the available workload traces contains any information about urgency requirements for the job or the amount the user is willing to pay for the job, an aspect which is essential to our model. In our model, we assume that each job specifies a maximum cost the user is willing to pay for the job, and a linearly decaying cost function (as shown in figure 7.2). Hence, for our evaluations, we randomly select a fraction $U$ of the jobs as urgent. The cost of non-urgent jobs is fixed at 0.1 units
Figure 6.4: Response time for different schemes assuming exact runtime estimate at different offered loads

Figure 6.5: Response time and Slowdown for different schemes assuming exact runtime estimate at different offered loads with 80% urgent job
per processor-second of the job. The cost of urgent jobs is set to be higher than that of non-urgent jobs by a factor $C$. In our experiments, we used values of 20%, 50% and 80% for $U$ and value of 100 for $C$.

![Graphs showing Slowdown for different schemes assuming exact runtime estimate at different offered loads](image)

(a) 50% Urgency Job  
(b) 20% Urgency Job

Figure 6.6: Slowdown for different schemes assuming exact runtime estimate at different offered loads

### 6.5.2 Revenue Improvement for Different Schemes

Revenue is an important metric to Supercomputer center to evaluate the performance of the center. In this section, we measure the revenue achieved using each scheme for the same workload. Overall revenue of a system of $n$ jobs can be estimated using the Eq. 6.6 in the user-centric model (Figure 7.2).

Although our main focus is parallel jobs in a multi-processor system, we initially study the schemes in the context of only sequential jobs. In Table 6.1, we present the revenue improvement of our proposed scheme $\text{NormalizedUrgency}$ with respect to the best previously proposed scheme $\text{FirstReward}$. For this experiment, the sequential
Figure 6.7: Achieved utilization for different schemes assuming exact runtime estimate at different offered loads

Table 6.1: Percentage Improvement of NormalizedUrgency over FirstReward for sequential job traces.
job trace is synthetically generated using the utility from [29]. The results are presented for various offered loads and for two different job-mixes. The data displays that the new scheme *NormalizedUrgency* earns as high as 40% more revenue than *FirstReward*. More detailed results are shown for different schemes in Table 6.2 and Table 6.3.

Figure 6.2 shows the revenue improvements for different schemes with respect to *FirstPrice* [21] for a multi-processor system. Figure 6.2(a) displays the revenue where urgent and non-urgent jobs are equal in load and Figure 6.2(b) shows the revenue where the number of urgent jobs are fewer (20% urgent jobs). In both graphs, we vary the system load from the actual load (74%) of CTC trace to higher loads upto 88%. The graphs

**Table 6.2:** Percentage Revenue Improvement of different schemes over *FirstReward* for sequential jobs with 20% urgent job

<table>
<thead>
<tr>
<th>Offered Load</th>
<th>First Price</th>
<th>Normalized Urgency</th>
<th>Present Value</th>
<th>Opportunity Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>59%</td>
<td>-79</td>
<td>7</td>
<td>-81</td>
<td>-11</td>
</tr>
<tr>
<td>65%</td>
<td>-140</td>
<td>13</td>
<td>-139</td>
<td>-20</td>
</tr>
<tr>
<td>72%</td>
<td>-314</td>
<td>36</td>
<td>-309</td>
<td>-24</td>
</tr>
<tr>
<td>78%</td>
<td>-452</td>
<td>40</td>
<td>-445</td>
<td>-53</td>
</tr>
</tbody>
</table>

**Table 6.3:** Percentage Revenue Improvement of different schemes over *FirstReward* for sequential jobs with 50% urgent job

<table>
<thead>
<tr>
<th>Offered Load</th>
<th>First Price</th>
<th>Normalized Urgency</th>
<th>Present Value</th>
<th>Opportunity Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>59%</td>
<td>-89</td>
<td>8</td>
<td>-87</td>
<td>-11</td>
</tr>
<tr>
<td>65%</td>
<td>-137</td>
<td>14</td>
<td>-136</td>
<td>-20</td>
</tr>
<tr>
<td>72%</td>
<td>-280</td>
<td>19</td>
<td>-276</td>
<td>-39</td>
</tr>
<tr>
<td>78%</td>
<td>-476</td>
<td>43</td>
<td>-469</td>
<td>-56</td>
</tr>
</tbody>
</table>
clearly demonstrate that our proposed scheme $NormalizedUrgency$ achieves the highest revenue among all the schemes (nearly 45% improvement over the second best scheme). This revenue improvement is expected because the $NormalizedUrgency$, unlike other schemes, is based on the job’s urgency relative to its length. It does not look at the maximum revenue offered by a job. Our optimality proofs and sufficiency criteria for the restricted versions of the problem validate the idea of normalized urgency. In addition, we observe that the revenue improvements are much higher at high load as compared to the original load (74%) scenario. This is because, though the job mix is the same as the original load case, the absolute number of urgent jobs is higher in the high-load case allowing more opportunity to start more high revenue jobs earlier. The results for traces with mostly urgency jobs (80%) also exhibits the same trends as shown in Figure 6.3.

6.5.3 Impact on System Performance

Although revenue maximization is an important aspect for a supercomputer center, it is significant to study the impact on other widely used performance metrics such as slowdown, response time and utilization. In this section, we study the response time and slowdown metrics for various schemes at different load. We did not include the utilization results due to the space constraints.

**Response Time:** The response time of a job is the sum of the time for which it has to wait in the job queue (for resources to be available) and the actual runtime after the job starts running. Figure 6.4 shows the graphs for response time as the load varies for two different job mixes. The graphs demonstrate that $FirstPrice$ performs worst whereas
Table 6.4: Slowdown (First Price) for offered load 88%

<table>
<thead>
<tr>
<th></th>
<th>1 Proc</th>
<th>2-8 Procs</th>
<th>9-32 Procs</th>
<th>&gt; 32 Procs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-10min</td>
<td>137.8</td>
<td>61.28</td>
<td>91.47</td>
<td>267.16</td>
</tr>
<tr>
<td>10m-1hr</td>
<td>3.09</td>
<td>7.02</td>
<td>17.37</td>
<td>36.36</td>
</tr>
<tr>
<td>1hr-8hr</td>
<td>1.8</td>
<td>1.98</td>
<td>4.99</td>
<td>5.59</td>
</tr>
<tr>
<td>&gt; 8hr</td>
<td>1.17</td>
<td>1.29</td>
<td>1.84</td>
<td>2.04</td>
</tr>
</tbody>
</table>

Table 6.5: Response Time in hour (First Price) for offered load 88%

<table>
<thead>
<tr>
<th></th>
<th>1 Proc</th>
<th>2-8 Procs</th>
<th>9-32 Procs</th>
<th>&gt; 32 Procs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-10min</td>
<td>0.96</td>
<td>1.63</td>
<td>4.13</td>
<td>15.91</td>
</tr>
<tr>
<td>10m-1hr</td>
<td>1.80</td>
<td>2.80</td>
<td>6.14</td>
<td>12.87</td>
</tr>
<tr>
<td>1hr-8hr</td>
<td>6.79</td>
<td>6.44</td>
<td>12.40</td>
<td>16.06</td>
</tr>
<tr>
<td>&gt; 8hr</td>
<td>15.72</td>
<td>17.68</td>
<td>22.68</td>
<td>22.95</td>
</tr>
</tbody>
</table>

Table 6.6: Slowdown (Present Value) for offered load 88%

<table>
<thead>
<tr>
<th></th>
<th>1 Proc</th>
<th>2-8 Procs</th>
<th>9-32 Procs</th>
<th>&gt;32 Procs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-10min</td>
<td>14.47</td>
<td>23.85</td>
<td>55.92</td>
<td>234.58</td>
</tr>
<tr>
<td>10m-1hr</td>
<td>1.42</td>
<td>2.7</td>
<td>8.13</td>
<td>35.54</td>
</tr>
<tr>
<td>1hr-8hr</td>
<td>1.27</td>
<td>1.45</td>
<td>3.36</td>
<td>4.78</td>
</tr>
<tr>
<td>&gt; 8hr</td>
<td>1.11</td>
<td>1.15</td>
<td>1.92</td>
<td>2.2</td>
</tr>
</tbody>
</table>

the proposed *NormalizedUrgency* performs the best or comparable to the second best.

Figure 6.5(a) displays the response time where most of the jobs are urgent (80%).

**Slowdown:** Slowdown of a job measures how much slower the system appears to the user compared to a dedicated machine. It is calculated as the ratio of the response time to the runtime of a job. Figure 6.6 shows the average slowdown for various schemes at different loads. In addition, Figure 6.5(b) displays the slowdown where
most of the jobs are urgent (80%). The graphs demonstrate that our proposed scheme NormalizedUrgency significantly outperforms the other schemes in terms of slowdown. In general, the shorter jobs contribute most to the average slowdown metric. In other words, the scheme, which prefers the short jobs to long jobs, exhibits a lower slowdown value. Since the proposed scheme also favors the short jobs, the improvement is anticipated. This hypothesis is further supported by the size-wise results in the table 6.4 to table 6.13. We categorize the jobs into sixteen categories based on the runtime and number of processors requested. The tables present the slowdown and response time for each category of job for various schemes. The first row of table 6.8 and 6.12 show that NormalizedUrgency schemes serves the short jobs (runtime between 0 to 10 minutes) better as compared to FirstReward, resulting in a lower slowdown. However, the slowdown value for longer jobs are similar for both the schemes, as expected.
<table>
<thead>
<tr>
<th></th>
<th>1 Proc</th>
<th>2-8 Procs</th>
<th>9-32 Procs</th>
<th>&gt;32 Procs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-10min</td>
<td>0.09</td>
<td>0.24</td>
<td>0.81</td>
<td>1.95</td>
</tr>
<tr>
<td>10m-1hr</td>
<td>0.87</td>
<td>0.98</td>
<td>2.11</td>
<td>3.06</td>
</tr>
<tr>
<td>1hr-8hr</td>
<td>5.21</td>
<td>4.89</td>
<td>9.70</td>
<td>11.69</td>
</tr>
<tr>
<td>&gt;8hr</td>
<td>15.56</td>
<td>15.78</td>
<td>27.16</td>
<td>29.74</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1 Proc</th>
<th>2-8 Procs</th>
<th>9-32 Procs</th>
<th>&gt;32 Procs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-10min</td>
<td>4.83</td>
<td>9.83</td>
<td>28.31</td>
<td>85.18</td>
</tr>
<tr>
<td>10m-1hr</td>
<td>1.42</td>
<td>2.05</td>
<td>6.53</td>
<td>14.21</td>
</tr>
<tr>
<td>1hr-8hr</td>
<td>1.23</td>
<td>1.33</td>
<td>3.55</td>
<td>5.79</td>
</tr>
<tr>
<td>&gt;8hr</td>
<td>1.07</td>
<td>1.11</td>
<td>1.73</td>
<td>2.5</td>
</tr>
</tbody>
</table>

Table 6.9: Response Time in hour (Normalized Urgency) for offered load 88%.

Table 6.10: Slowdown (Opportunity Cost) for offered load 88%.

**Utilization:** Utilization is the ratio of resources used by the jobs to the resources offered. The utilization metric is very important to the supercomputer centers. Figure 6.7 shows the achieved utilization for different schemes. The achieved utilization for the schemes are nearly identical, and are a function of the scheduler’s ability to tightly pack the submitted jobs in the 2D chart.

### 6.5.4 Impact of User Runtime Inaccuracy

In our previous experiments, we assumed that the user accurately estimated runtime of a job at submission. The assumption was made to reduce the number of variables and thereby aid the process of understanding the behavior of schemes in a more predictable scenario. However, the user runtime estimates are inherently inaccurate. Most of the time, the user over-estimates the runtime of a job. Therefore, in this section, we investigate the impact of such inaccuracy in user runtime estimation for different schemes.
Table 6.11: Response Time in hour (Opportunity Cost) for offered load 88%.

<table>
<thead>
<tr>
<th></th>
<th>1 Proc</th>
<th>2-8 Procs</th>
<th>9-32 Procs</th>
<th>&gt; 32 Procs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-10min</td>
<td>0.12</td>
<td>0.30</td>
<td>1.37</td>
<td>5.89</td>
</tr>
<tr>
<td>10m-1hr</td>
<td>0.89</td>
<td>0.95</td>
<td>2.32</td>
<td>4.89</td>
</tr>
<tr>
<td>1hr-8hr</td>
<td>4.96</td>
<td>4.66</td>
<td>9.83</td>
<td>16.58</td>
</tr>
<tr>
<td>&gt; 8hr</td>
<td>14.48</td>
<td>15.35</td>
<td>21.42</td>
<td>28.57</td>
</tr>
</tbody>
</table>

Table 6.12: Slowdown (First Reward) for offered load 88%.

<table>
<thead>
<tr>
<th></th>
<th>1 Proc</th>
<th>2-8 Procs</th>
<th>9-32 Procs</th>
<th>&gt; 32 Procs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-10min</td>
<td>5.82</td>
<td>9.73</td>
<td>29.64</td>
<td>72.57</td>
</tr>
<tr>
<td>10m-1hr</td>
<td>1.39</td>
<td>2.24</td>
<td>6.88</td>
<td>16.17</td>
</tr>
<tr>
<td>1hr-8hr</td>
<td>1.21</td>
<td>1.24</td>
<td>3.33</td>
<td>3.97</td>
</tr>
<tr>
<td>&gt; 8hr</td>
<td>1.06</td>
<td>1.11</td>
<td>1.7</td>
<td>2.54</td>
</tr>
</tbody>
</table>

Figure 6.8 shows the revenue improvement in a real setting where the inaccurate estimated runtime from an actual trace is utilized in scheduling decision. The results show similar trends to the ones shown in Figure 6.2 where exact runtime estimation was assumed. However, the absolute revenue improvements are lower in all cases (compared to Figure 6.2) and this disparity is highest in the case where we employ the lowest percentage of urgent jobs (Figure 6.8(b)). The main reason for this is the over-estimation of a job’s runtime and the usage of overestimated runtime in defining value function (Figure 7.2). As previously described, our value function has two region: a flat region where revenue is constant and a sloped region where the revenue drops as the time increases. Since the runtime is usually over-estimated, the flat region becomes longer and most of the jobs get completed within the flat region resulting in no differences in revenue. Figure 6.9 displays the same overall trend where most of the jobs (80%) are
Table 6.13: Response Time in hour (First Reward) for offered load 88%.

<table>
<thead>
<tr>
<th></th>
<th>1 Proc</th>
<th>2-8 Procs</th>
<th>9-32 Procs</th>
<th>&gt; 32 Procs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-10min</td>
<td>0.14</td>
<td>0.31</td>
<td>1.39</td>
<td>4.77</td>
</tr>
<tr>
<td>10m-1hr</td>
<td>0.87</td>
<td>1.03</td>
<td>2.52</td>
<td>5.76</td>
</tr>
<tr>
<td>1hr-8hr</td>
<td>4.82</td>
<td>4.51</td>
<td>9.27</td>
<td>12.42</td>
</tr>
<tr>
<td>&gt; 8hr</td>
<td>14.39</td>
<td>15.35</td>
<td>20.74</td>
<td>28.96</td>
</tr>
</tbody>
</table>

Figure 6.8: Revenue improvement relative to FirstPrice assuming inexact runtime estimate at different offered loads

(a) 50% Urgency Job  
(b) 20% Urgency Job

urgent. In summary, although the absolute revenue gains of the schemes are not very good, our proposed scheme still outperforms all other schemes at all loads and at all job mixes exhibiting the improved adaptability of our scheme.

So far, in our experiments, we used the CTC trace as the base trace. To demonstrate the robustness of our schemes across different workloads, we illustrate the revenue improvement using the SDSC workload. Figure 6.10 shows the revenue improvement for
Figure 6.9: Revenue improvement relative to FirstPrice assuming inexact runtime estimate at different offered loads with 80% urgent job

various loads and two different job mixes where user estimated runtime is actual but inaccurate. Although the trends are consistent with the CTC workload, the improvement is far better for SDSC workload. This is attributed to high original offered load (82%) for SDSC compared to low original offered load (74%) in CTC. At high load, there are more urgent jobs which the schemes can leverage to improve the overall revenue.

### 6.6 Conclusions

In this chapter, we explored the problem of market-based batch scheduling for parallel jobs running on supercomputer centers, with a view to maximize the overall revenue earned by the resource provider. We proposed a new scheduling heuristic called
Figure 6.10: Revenue improvement relative to FirstPrice assuming inexact runtime estimate at different offered loads in SDSC trace

NormalizedUrgency which is based on the notion of prioritizing jobs based on their urgency and their processing times. We prove the optimality of our approach for certain restricted versions of the problem. Finally, we experimentally compare our schemes against the existing schemes like FirstPrice, FirstReward etc. using real supercomputer center workloads. Our results demonstrate that the proposed scheme leads to significantly higher revenue as compared to existing schemes while achieving better performance with respect to standard performance metrics such as slowdown and utilization.
7.1 Introduction

Batch job schedulers are commonly used to schedule parallel jobs at shared-resource supercomputer centers such as the Ohio Supercomputer Center (OSC) [7], San Diego Supercomputer Center (SDSC) [11], Cornell Theory Center (CTC) [2], etc. The standard working model for these supercomputer centers is to allocate resources (e.g., processors, memory) to a job on submission, if available. If the requested resources are not currently available, the job is queued and scheduled to be started at a later time (when the resources become available). The turnaround time or response time of a job is the sum of the time for which it has to wait in the job queue (for resources to be available) and the actual runtime after the job starts running. Users are typically charged as a function of the total resources used (resources × runtime).

Together with the standard working model described above, there has also been a lot of recent interest in Quality of Service (QoS) capabilities in job scheduling in terms of guarantees in the job’s turn-around time. Such QoS capabilities are useful in several instances. For example, a scientist can submit a job before leaving work in the evening and request a deadline in the job’s turn-around time for 8:00am the next morning, i.e.,
she needs the job to complete and the results to be ready by the time she is back the next morning. Currently, the only mechanism most supercomputer centers provide to achieve this is based on *advance reservations*.

With advanced reservations, when the user submits a job, she can request the needed resources during a specific time window (within her deadline). If the requested resources are available in this time window, the job is accepted and is statically assigned to be executed in that time window. If the resources are not available in this time window, the job is rejected and the user (or some software agent on behalf of the user) can try a different time window within the deadline period. While *advance reservation* capabilities are widely available in supercomputing centers, utilizing them to achieve QoS results in significant under-utilization of resources, making them not the best choice as a QoS aware scheme. Therefore, we proposed a new scheme, *QoPS* described in Section 8.4 and [41, 42], to provide QoS guarantees without statically assigning a time window to execute the job, i.e., QoPS allows jobs to be moved in the schedule while being bound by the requested deadline constraint. This results in significant improvements in resource efficiency as well as the number of accepted jobs.

The basic QoPS implementation attempts to accept as many jobs as possible and does not analyze the costs of the jobs that are being submitted. This issue is illustrated in figure 7.1. Let us consider a situation where there are a set of jobs that are already running in the system and there are four idle processors available. Now, let us say a 4-processor job $J_4$ arrives, requests a deadline in 4 hours and offers to pay $100. In this situation, QoPS checks that it can accommodate this job into the system and accepts it. Immediately after this job is accepted, another 4-processor job $J_5$ arrives, requests the same deadline (4 hours) and offers to pay $200 (a higher price than $J_4$). Since $J_4$ has
Figure 7.1: Expensive job $J_5$ can not be scheduled because identical but cheaper job $J_4$ is already accepted.

already been accepted, the system cannot accept $J_5$ and hence has to forgo the more profitable job. This example demonstrates that it may not always be beneficial to admit all revenue generating jobs, because of consequent potential future loss of revenue, i.e., *opportunity cost*. In other words, while each job pays a explicit price to the system for running it, the system may also be viewed as paying an implicit opportunity cost by accepting the job. Accordingly, accepting the job is profitable for the system when the job’s price is higher than its opportunity cost.

Formally, the opportunity cost of a job is defined as the difference between the highest revenue possible for the entire workload, with and without accepting the job. If the opportunity cost of a job is known up front, the system can easily derive the potential benefits in accepting the job. However, knowing the opportunity cost of a job up front is impossible. Thus, this chapter addresses the analysis of impact such opportunity cost can have on the overall revenue of the supercomputer center and attempting to minimize it through predictive techniques. Specifically, we first present an extension of QoPS, named Value-aware QoPS (VQoPS), that is aware of the different prices of different
jobs and analyzes it with various statically assumed opportunity cost values for the jobs in the system. As we will demonstrate in the later sections, no single statically assumed opportunity cost value does well for all kinds of job mixes.

In the second part of this chapter, we introduce Dynamic Value-aware QoPS (DVQoPS), which is a self learning variant of VQoPS to analyze past jobs and predict the opportunity costs for future arriving jobs. However, a simple history-based scheme has several disadvantages. For example, if the opportunity cost is decided based on a very long history, the mechanism will lose its sensitivity with respect to adapting to small changes in the pattern of the jobs being submitted. At the same time, if the opportunity cost is decided based on a very short history, the sample set of previous jobs might be too small and the results obtained might be noisy and unstable. To add to the complexity, the optimal history length to be considered might be different for different days (e.g., there are more jobs when there is a paper deadline, and hence short histories might be sufficient) or for different parts of the day (e.g., there are more jobs in the day time as compared to nights or weekends). Thus, the length of the history also needs to be dynamically adapted to balance sensitivity (not too long a history) and stability (not too short a history).

It is to be noted that analyzing the opportunity costs of different jobs and trying to maximize revenue also has a much desired side-effect of service differentiation, i.e., if an expensive job arrives slightly after a cheaper job arrives, a good scheme can provide appropriate service differentiation between these two jobs and give a higher chance of acceptance to the more expensive job though both jobs do not arrive at the same time.
We present the detailed analysis of both VQoPS and DVQoPS with simulation based on different real workloads of jobs from Feitelson’s workload archive (San Diego Supercomputing Center (SDSC) trace from 2000 and Cornell Theory Center (CTC) trace from 1997. Our results provide various insights into the impact of opportunity costs in QoS-aware job schedulers and demonstrate that DVQoPS can provide several factors higher system revenue capabilities as compared to QoPS. Further, we also demonstrate that DVQoPS can achieve good service differentiation between high-paying urgent and low-paying non-urgent jobs.

The remaining part of this chapter is organized as follows. In section 7.2, we describe the experimental setup and simulation approach that will be utilized in subsequent sections. We discuss basic concepts of user centric charging model in Section 7.3. In Section 7.4, we introduce the concept of opportunity cost in job scheduling. We present in Section 7.5 a value-aware modification of the QoPS scheduling algorithm, targeted at a job submission environment where users provide time-varying pricing offers along with their jobs. We also develop the dynamic, adaptive, value-aware scheduling scheme that uses recent history in determining key parameters in Section 7.6. Simulation results based on the SDSC and CTC workloads are provided in section 7.7. Section 7.8 concludes the chapter.

7.2 Evaluation Approach

The strategies evaluated in this chapter are simulated using real workload traces, such as those available at the Parallel Workload Archive [29]. These traces include information such as the jobs runtime, the number of nodes each job used, the submission time, and a user estimated runtime limit (wall-clock limit). In this work, we used
two such real workload trace of 10,000 jobs subset of the SDSC SP-2 and CTC SP-2 workload trace.

**Deadline Information:** None of the workload traces available contain any information about requested job deadlines, as hard deadline guarantees are not supported by any supercomputer center. Therefore, we synthetically generate these deadlines. For this work we assume that users will assign deadlines based on the runtime of their job, i.e., longer jobs have longer deadlines and shorter jobs have shorter deadlines. Each job’s deadline is assigned to be 5 times the users estimated runtime ($\text{deadline}_j = 5 \times \text{user runtime estimate}_j$).

**Runtime Estimates:** Runtime estimates are critical when evaluating parallel job schedulers. Therefore, two sets of simulations are performed. The first set of simulations takes an idealistic view of the traces and assumes that users are able to perfectly estimate their job’s runtime; this allows us to concentrate on the capabilities of our algorithms without being affected by other noise in the traces. The second set of simulations uses the actual runtime estimates given in the workload trace; this allows us to evaluate our algorithms in more realistic environments.

**Job Submission Load:** In most supercomputer centers, the number of jobs submitted within a given time varies from hour-to-hour (high in the day time and low in the nights), between days (high during the week days and low in the weekends) or even between weeks (high during submission deadlines when researchers are looking for measurement results). In order to capture such different scenarios, we carry out simulations with two different kinds of offered loads for each traces. The first is the original load as measured in the SDSC trace subset (total processor seconds submitted correspond to about 89% of the overall system capacity). The second is a high load emulation that is
achieved by randomly duplicating roughly 40% of the jobs (total processor seconds submitted correspond to about 125% of the overall system capacity). The same technique is used for CTC trace with original load (74%) and high load (104%). Duplicated jobs have the same submission time, runtime estimate, runtime, and deadline as the original job.

**Urgency and Job Cost:** Like deadline information, none of the workload traces available contains any information about urgency requirements for the job or the amount the user is willing to pay for the job. Hence, for our evaluations we randomly mark a fraction \( U \) of the jobs as urgent. The cost of non-urgent jobs is fixed at 0.1 units per processor-second of the job. The urgent jobs’ cost is set to be higher than that of non-urgent jobs by a factor \( C \). In our experiments, we used values of 20%, 50% and 80% for \( U \) and values of 10, 5 and 2 for \( C \). Further, as shown in figure 7.2, we assume that each job specifies a maximum cost the user is willing to pay for the job, and a linearly decaying cost function. The value of the job becomes zero at the requested deadline of the job. This model is quite similar to that used in previous papers [21, 39], except for deadline guarantees, which were not previously considered.

### 7.3 User-centric Charging Model

Revenue is an important metric to Supercomputer center to evaluate the performance of the center. Though, it is a significant objective of the centers to make most of the customers satisfied, the primary and prevalent goal of the centers is to maximize the overall revenue of the centers. Our proposed QoPS scheduler provides satisfaction to the Supercomputer center users by maintaining the QoS guarantee in the form of deadline. Conversely, revenue maximization appeals mainly to the Supercomputer center.
a suitable charging model is the first step of characterizing the revenue. There are two possible ways of expressing the charge of a job in parallel system. We have already described the provider-centric approach in chapter 5. In this section, we introduce the user-centric charging model used extensively in market-based economy models.

In user-centric charging model, the user, instead of system provider, offers the price for running a job. We adopt the user-centric model utilized in [21, 39, 63] mainly for non-QoS job scheduler. According to the adopted model, user specifies the value of a job depending on the importance of timely delivery. Basically, each user defines a piece-wise linear value or utility function (shown in Figure 7.2) whereupon the charge is calculated as function of completion time of a job. The value function reflects the urgency of the job as a time-dependent function. For simplicity, the value function has two linear pieces. The first part indicates the maximum value that a user wants to pay if it is completed in earliest possible time, which in turn expresses the importance of the job. Likewise, the second part denotes decay or down-slope of the value and shows the sensitivity of the job to further delay. It is obvious that, for example, a user who has very close paper deadline and wants the result sooner, offers high initial value and steeper slope for the job. Conversely, the user whose paper deadline has few more weeks away, propose low value and smoother slope value function. Eventually, the aggregate utility, which is calculated by summing up the individual value earned for all the admitted jobs in the system, can be used to estimate the overall system performance. This aggregate value metric can be used as a measure of total value served to all users or as overall revenue obtained for the system. Mathematically, the aggregate value for a system of n jobs (using Figure 7.2) can be estimated using the following set of equations:

\[
\text{Slope of job } J_i, \quad \text{Slope}_i = \frac{\text{Max Revenue}_i}{(\text{Deadline}_i - \text{Runtime}_i)}
\]
Revenue of job $J_i$, $Revenue_i = MaxRevenue_i - Slope_i \times (CompletionTime_i - Runtime_i)$

Total system value or revenue $AggregateRevenue = \sum_{i=1}^{n} Revenue_i$

### 7.4 Opportunity Cost in Job Scheduling

Opportunity cost in parallel job scheduling gets very little attention in the research community. We broadly study the opportunity cost concerning the revenue maximization in QoS-aware system. Generally, the opportunity cost of a job depends on a number of parameters, both local to each job as well as global to the entire workload of jobs. In this section, we extensively analyze the various aspects of opportunity cost in maximizing the overall system revenue.

#### 7.4.1 Fundamentals of Opportunity Cost

In an online model where a job is associated with a hard deadline and a value, its acceptance could sometimes cause the rejection of some much profitable later arriving jobs. In other words, there is always some opportunity cost involved in the acceptance of
a newly arrived job at any instant of time. In this section, we elucidate the basic concept of opportunity cost in parallel job scheduling.

Considering the objective of maximizing overall revenue gain, the acceptance decision of any newly arrived job, should be made quite judiciously taking this opportunity cost into account. But, the estimation of opportunity cost is a real challenge as it depends on various unknown factors in an online job-scheduling scenario where job arrives dynamically over time. To study the OC in job scheduling, we first consider a relatively easy and predictable model of offline system where all jobs information including the job’s arrival is known at the beginning.

Let us assume a set $J = \{J_1, J_2, \ldots, J_n\}$ of n jobs with corresponding arrival time of $\{t_1, t_2, \ldots, t_n\}$ in an offline system that needs to be scheduled. $S_{tn}(J)$ is a feasible schedule at time $t_n$ for the set of all input jobs $J$. In $S_{tn}(J)$, not all the jobs in list $J$ are accepted; some jobs may be rejected too (i.e. $AcceptedJobs(S_{tn}(J)) \subseteq J$). In other words, we can state that $AcceptedJobs(S_{tn}(J)) + RejectedJobs(S_{tn}(J)) = J$

**Definition:** The overall revenue of a schedule depends only on the accepted jobs in that schedule. It is calculated by summing up the individual revenue gathered from all the accepted jobs. For the simplicity, we assume that the offered value of a job (i.e. $Value(J_i)$) is time independent. $Revenue(S_{tn}(J)) = \sum_{J_i \in AcceptedJobs(S_{tn}(J))} Value(J_i)$

Suppose that, at time $t_i$, job $J_i \in J$ arrives and the scheduler can accommodate the job $J_i$ within its deadline. The admittance decision for Jobs $\{J_1, J_2, \ldots J_{i-1}\}$ are already finalized by time $t_i$, and all the accepted jobs are in schedule $S_{t_{i-1}}(\{J_1 \ldots J_{i-1}\})$. At this instant, accepting the job $J_i$ can only influence the admittance of some of the future
arriving jobs in \(\{J_{i+1}...J_n\}\). As the scheduler knows all the future jobs at time \(t_i\), it can run forward what-if simulation from time \(t_i\) to time \(t_n\) for two different scenarios: where \(J_i\) is definitely accepted and where \(J_i\) is definitely rejected. Both schedules are resulted from the same state where acceptance decision of \(J_{i-1}\) is completed. Let’s denote these two schedule as \(S_{tn}(\{J_i...J_n\})\) and \(S_{tn}(\{J_{i+1}...J_n\})\).

**Claim 1:** \(\text{AcceptedJobs}(S_{tn}(\{J_i...J_n\})) - J_i \neq \text{AcceptedJobs}(S_{tn}(\{J_{i+1}...J_n\}))\)

**Proof:** Since it is assumed earlier that \(J_i\) can fulfill the deadline constraint and considered as accepted in \(S_{tn}(\{J_i...J_n\})\), we can state \(J_i \in \text{AcceptedJobs}(S_{tn}(\{J_i...J_n\}))\). At the same time, it is obvious that \(J_i \notin \text{AcceptedJobs}(S_{tn}(\{J_{i+1}...J_n\}))\), because \(J_i\) is not considered in this schedule. If the job \(J_i\) is scheduled in \(S_{tn}(\{J_i...J_n\})\) such that it doesn’t prevent any later arriving job \(\{J_{i+1}...J_n\}\) from admitting, this equality is true: \(\text{AcceptedJobs}(S_{tn}(\{J_i...J_n\})) - \{J_i\} = \text{AcceptedJobs}(S_{tn}(\{J_{i+1}...J_n\}))\). But the above equality does not hold if the acceptance of job \(J_i\) in \(S_{tn}(\{J_i...J_n\})\) prevents other later arriving jobs in the schedule. For instance, let’s consider that if job \(J_i\) is accepted in the first schedule, it forces the job \(J_{i+1}\) to reject, but if job \(J_i\) is rejected in first schedule, it allows the job \(J_{i+1}\) to accept. In this circumstances, the following inequality is always valid \(\text{AcceptedJobs}(S_{tn}(\{J_i...J_n\})) - \{J_i\} \neq \text{AcceptedJobs}(S_{tn}(\{J_{i+1}...J_n\}))\).

\[\square\]

**Definition:** The opportunity cost (OC) at time \(t_i\) for accepting the job \(J_i\) can be expressed mathematically: \(OC(t_i, J_i) = \text{Revenue}(S_{tn}(\{J_i...J_n\})) - \text{Revenue}(S_{tn}(\{J_{i+1}...J_n\}))\)

Essentially, the opportunity cost is estimated as the global revenue difference obtained from the two different schedules: when the scheduler accepts the job \(J_i\) and when it doesn’t accept the job \(J_i\).
Claim 2: If a job can be admitted in a schedule without deadline violation, the maximum overall revenue could be earned when the following acceptance criterion is considered for job $J_i$ at time $t_i$: $OC(t_i, J_i) > 0$

Proof: The proof of the acceptance criterion depends on the rather intuitive notion that if the scheduler doesn’t loss any money in the long run due to the admission of the new job, it accepts the job. In other words, if accepting a job $J_i$ yields more revenue globally than the rejecting it, the scheduler should accept it to maximize the revenue. As all jobs’ information is known apriori (offline system), this type of global decision is possible with extra computational overhead due to forward simulation.

But in an online scheduling scenario which is our intended research area, the exact estimation of OC is not possible as there is no scope to know the job list $\mathcal{J}$ in advance in a system where the job arrives dynamically. The OC can be approximately anticipated using the predictive technique based on either stochastic or history based analysis.

7.4.2 Impact of Job and Workload Characteristics on Opportunity Cost

As described in section 7.4.1, opportunity cost for a given job identifies the amount of potential future revenue that a job loses by accepting this job. This potential future revenue depends on a number of aspects, which can be classified into two broad categories: local characteristics of the job and global characteristics of the workload. In this section, we briefly describe these two aspects.

Local Characteristics of the Job: Two primary characteristics of a job play a role in impacting its opportunity cost, viz., job shape and QoS requirements of the job.
1. **Job shape (processors and wall-clock time requested):** The shape of a job can affect how many later jobs must be dropped. For example, if a large job (which requests many processors and/or a long wall-clock time) is accepted, it uses up more of the available resources in a schedule, potentially requiring more later arriving jobs to be rejected. If on the other hand, a smaller job is accepted, it might be possible to accept more later arriving jobs. Thus, the opportunity cost of large jobs is likely higher than a small job.

2. **QoS requirements of the job:** The QoS requirement of the job can determine how stringent the schedule is, i.e., if a job has a tight deadline, it cannot be moved at all. On the other hand, if a job has a looser deadline, it might be possible to move in the schedule and thus allow other jobs to be accepted more easily. Thus, the opportunity cost of stringent jobs is likely higher.

**Global Characteristics of the Workload:** Three primary characteristics of the workload play a role in impacting a job’s opportunity cost, viz., offered load, job mix and job pricing.

1. **Offered load:** When the number of jobs in the system is very less (i.e., offered load is low), the acceptance of a non-urgent job is less likely to prevent the admittance of a future urgent job. Thus, the opportunity cost would typically be lower than when the number of jobs in the system is high (i.e., offered load is high).

2. **Job mix (% of Urgent Jobs):** If all jobs had the same urgency and pricing, the opportunity cost of a job is essentially zero, since it is not possible for a later job to have better pricing. But when there are some urgent (and high-paying) and some non-urgent (and low-paying) jobs in the system, opportunity cost is no
longer zero. Further, if the percentage of urgent high-paying jobs in the job trace is very high, the opportunity cost of admitting a non-urgent job is high, since there is a high probability that its admittance could prevent admission of a future high-paying job.

3. *Job pricing*: The relative premium paid by urgent jobs is also an important factor. The higher the premium paid by an urgent job relative to a non-urgent job, the greater the cost of losing an urgent job, and thus greater the opportunity cost of non-urgent jobs.

### 7.5 Value-aware QoPS (VQoPS)

The QoPS algorithm does not perform any differentiation between jobs. It assumes that all jobs have the same price value assigned to them. However, in an environment that allows users to offer price based on responsiveness, some jobs in the system will offer more revenue than others. Similarly, some jobs will have tighter deadlines than others. In this section, we describe the essentials of VQoPS, the extended version of QoPS, which incorporate the value of a job during job scheduling.

For an algorithm that is expected to improve the overall revenue of the system, the following two properties are desirable:

1. During backfilling, the new algorithm should reorder the job queue so as to give a higher priority for more urgent jobs and attempt to reduce their turnaround time, thus increasing revenue.
2. The new algorithm should attempt to maximize the overall revenue of the system during job acceptance by considering both the explicit revenue benefit to the system and the implicit loss of opportunity for the system.

Considering these characteristics for revenue maximization, we can mathematically formulate the admission control mechanism for VQoPS using the notations described in section 4.4.4.

Acceptance criteria used for VQoPS: For VQoPS, there are two different acceptance criteria; both of them required to be satisfied simultaneously to accept any new job.

- **Acceptance criteria 1**: The same criteria used for QoPS and described in section 4.4.4.

- **Acceptance criteria 2**: Let’s first define the possible revenue earned from a schedule \( (S) \) of \( m \) jobs as

\[
Revenue(S) = \begin{cases} 
-\infty & \text{If } S \text{ is not a feasible schedule} \\
\sum_{i=1}^{n} Value(S, J_i) & \text{otherwise}
\end{cases}
\]  
(7.1)

Also assume \( S_{old} \) is the existing feasible schedule and \( S_{new} \in \mathbb{R}^{org} \) is a probable new schedule. The following condition checks whether the revenue-gain due to the acceptance of the new job \( (J_{n+1}) \) is worthy enough for the system considering the opportunity cost. \( Revenue(S_{new}) - Revenue(S_{old}) \geq OC_{Factor} * RunTime(J_{n+1}) * NumProc(J_{n+1}) \) Also the parameter, \( OC_{Factor} \), which determines the estimation of opportunity cost, is specified statically or dynamically during scheduling.
Finding Best schedule for VQoPS: When there is a set of feasible schedules \( \mathbb{S}^{feasible} \), we can find out the best schedule \( S_{best} \) by identifying the schedule with maximum revenue and can be expressed as follows:

\[
S_{best} = S \quad \text{where } S \in \mathbb{S}^{feasible} \text{ and }
Revenue(S) = \max_{S' \in \mathbb{S}^{feasible}} Revenue(S')
\]
7.5.1 Design of VQoPS

This section presents a new design known as Value-aware QoPS (VQoPS) that incorporates the properties described in previous section. Specifically, it utilizes some of the local information about a job, i.e., the shape of the job, to identify whether the job is likely to have a high opportunity cost or not. Accordingly, an opportunity cost that is proportional to the size of the job (number of processor-seconds of the job) is statically assumed, i.e., the opportunity cost is assumed to be \( \text{OC-Factor} \times \text{job size} \), where OC-Factor is a static system parameter.

Figure 7.3 shows the high-level VQoPS flowchart based on the properties described above. VQoPS utilizes QoPS as a pluggable module to check the acceptance of a new job (passed as an argument). Since the original QoPS design just provided a recommendation about whether a job should be accepted or rejected, instead we adapt it to return the entire list of acceptable schedules. Particularly, when a new job arrives in an existing schedule of \( N \) accepted jobs, QoPS tries to push the new job in \( \log_2(N) \) different positions \((0, \frac{n}{2}, \frac{3n}{4}, \ldots, n)\) of the queue resulting in as many as \( \log_2(N) \) different schedules. Among these log schedules, modified QoPS module only gives back the list of deadline-feasible schedule.

When a new job arrives, this job together with the OC-Factor is passed as input to VQoPS. Admission control for this job is done in two stages. In the first stage, VQoPS uses, as mentioned above, the QoPS module to check for feasible schedules such that the newly arriving jobs meets its deadline without violating the deadline constraint of any of the already accepted jobs. In the second stage, VQoPS weighs the statically assumed opportunity cost of the new job with its price and decides whether the job
should be accepted. In particular, if the revenue gain obtainable can offset the opportunity cost, VQoPS accepts the job. It is worthy to note that the OC-Factor input in VQoPS is constant (static) throughout the job processing and expected to be defined by an administrator as a system parameter. The time complexity of our approach is $\theta(K.N^2.log(N))$.

7.5.2 Impact of Workload Characteristics on VQoPS

In this section, we analyze VQoPS by varying the different characteristics of the workload, such as the relative urgency cost, percentage of urgent jobs and the offered load on the system.

Table 7.1 shows the improvement in revenue generated (compared to QoPS) for the different workload characteristics and different OC-Factors (0.0, 0.05, 0.1, 0.2, and 0.4). Several insights can be drawn from these results. Primarily, though the VQoPS algorithm outperforms QoPS by up to 160% in some cases, we notice that the improvement achieved is highly dependent on the trace characteristics. That is, no single OC-Factor can consistently provide a good performance for all kinds of workloads. In fact, the same OC-Factor that provided 160% improvement in one trace, can provide worse performance than even QoPS by almost 100% in another trace.

The impact of the workload characteristics on VQoPS can be noticed in several aspects. First, as the relative cost of the urgent jobs decreases, we see the smaller OC-Factor values perform better. This is expected, since the opportunity cost of turning away a future urgent job decreases as the cost of the urgent job decreases. In other words, if QoPS makes a bad decision about accepting a job and ends up accepting a normal job instead of an urgent job, the opportunity cost it would pay would decrease as...
Table 7.1: Revenue improvement for varying OC-Factors

<table>
<thead>
<tr>
<th>Relative Urgent Cost</th>
<th>% Urgent Jobs</th>
<th>Offered Load</th>
<th>OC-Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.05</td>
</tr>
<tr>
<td>10X</td>
<td>80%</td>
<td>Orig</td>
<td>21%</td>
</tr>
<tr>
<td>5X</td>
<td>80%</td>
<td>Orig</td>
<td>20%</td>
</tr>
<tr>
<td>2X</td>
<td>80%</td>
<td>Orig</td>
<td>19%</td>
</tr>
<tr>
<td>10X</td>
<td>80%</td>
<td>Orig</td>
<td>21%</td>
</tr>
<tr>
<td>10X</td>
<td>50%</td>
<td>Orig</td>
<td>23%</td>
</tr>
<tr>
<td>10X</td>
<td>20%</td>
<td>Orig</td>
<td>26%</td>
</tr>
<tr>
<td>10X</td>
<td>80%</td>
<td>Orig</td>
<td>21%</td>
</tr>
<tr>
<td>10X</td>
<td>80%</td>
<td>High</td>
<td>63%</td>
</tr>
</tbody>
</table>

the cost of the urgent job decreases and hence the amount it would lose would reduce. Accordingly, the OC-Factor providing the best performance decreases as well.

Second, as the number of urgent jobs decreases, smaller OC-Factor values perform better. This is again an expected trend since as the number of urgent jobs decreases, though the worst case opportunity cost for QoPS would not reduce, the average case opportunity cost would reduce, and accordingly the OC-Factor.

Third, with load increase we see that the revenue improvement increases for all OC-Factor values. As the fraction of dropped jobs increases with increasing load (since the system cannot accommodate all the jobs while meeting their requested deadlines), the algorithm gets very selective and picks only the high-paying jobs. QoPS, on the other hand, picks all the jobs that it can meet the deadlines without considering their prices. This, accordingly, reflects as higher revenue improvement for all OC-Factor values at high loads.
7.6 DVQoPS: A Self-learning Variant of Value-aware QoPS

As described in section 7.5, while Value-aware QoPS is able to achieve revenue improvements, the behavior of the algorithm heavily depends on the characteristics of the trace. Thus, it is desirable to develop a more sophisticated algorithm that has all the benefits of VQoPS, but is able to adapt the OC-Factor based on the characteristics of the trace. In this section, we describe a variant of VQoPS which aims to achieve this.

7.6.1 Dynamic Value-aware QoPS (DVQoPS)

The VQoPS algorithm presented in the previous section requires that a good OC-Factor be determined - say by a system administrator. In order to properly tune the OC-Factor, various dynamically changing variables must be taken into account. The offered load must be accounted for. In lightly loaded situations more normal jobs should be accepted, and a lower OC-Factor should be used. As the load increases, the OC-Factor should be increased. If the expected price difference between urgent and non-urgent jobs is small, the scheduler should be more aggressive and accept more of the less expensive jobs. As the expected revenue difference increases, the OC-Factor should be increased to not accept jobs that only increase revenue modestly. Finally, as the percent of urgent or expensive jobs increases, the OC-Factor should also increase. When there are only a few urgent jobs, it is not desirable to reject a large number of normal jobs waiting for an urgent job to arrive. However, as it becomes more likely an urgent job will arrive soon, the OC-Factor should be increased.

It is very difficult to manually take these considerations and their interactions into account. It becomes even more difficult as these criterion may change over time. Thus, it is highly desirable for the scheduler to automatically generate and adjust the VQoPS
OC-Factor. The approach we pursue is that of performing a number of what-if simulations over a limited backward window in time, called the rollback window. The essential idea is to periodically adjust the OC-Factor by looking backwards and performing simulations to estimate the revenue that would have been realized for various choices of OC-Factors. A set of such simulations is feasible since the history of arrived jobs (both actually accepted and those rejected in the real schedule) is available. For each choice of OC-Factor, simulation over the rollback window is used to estimate the revenue for each job accepted by the simulated schedule, and thereby the total estimated revenue over the rollback period is estimated. The OC-Factor giving the best estimated revenue over the window is chosen as the actual OC-Factor to be used for the immediate future (until the next OC-Factor change event).

We next discuss the issue of the choice of rollback window duration. The basic premise of the adaptive OC-Factor selection procedure is that the best OC-Factor depends on various characteristics of the trace, which vary over time. For example, the load in supercomputer centers tends to be bursty, with some periodic patterns. At federal supercomputer centers, job arrival rate during working hours tends to be higher than during the night, and arrival rate during weekdays tends to be higher than during weekends. If the DVQoPS scheme is to be effective in adapting to such load variations, clearly the rollback window must not be too large, e.g., one month, because the time variance in load will get averaged out over the long rollback window. At the other extreme, there is a different problem if the rollback window is made extremely small in order to be very responsive to the time variation of trace characteristics. When the
number of jobs submitted over the window is very small, the results of the what-if simulations may be extremely sensitive and not robust. In the next subsection we discuss this issue further.

### 7.6.2 Balancing Sensitivity and Stability in DVQoPS

In this section, we illustrate the issues of determining the rollback window interval in a dynamic system. The choice of rollback window often involves a judicious balance between two competing factors:

1. **Sensitivity to Trace Variation**: The rollback window should be short enough to capture the effect of changes in input trace characteristics, such as load and job mix characteristics.

2. **Stability**: The rollback window should not be so short that specific jobs affect the best what-if OC-Factor, rather than the aggregate characteristics of the jobs in the window.

In order to assess these factors, we carried out studies using different traces. For assessing the effect of the rollback window to trace variation, the average offered load was computed over segments of duration equal to the rollback window, and the variance of these averages was computed. At one extreme, if the rollback window size is the entire trace duration, the variance is zero since a single average load is computed. At the other extreme, when the rollback window is minimized, the variance of the average loads is maximized. As the rollback window size increases, the load variance is expected to decrease.

To understand the impact of rollback window on stability of OC-Factor choice, consider the following example. Suppose that a set of $N$ consecutively arriving jobs were
Table 7.2: Effects of rollback windows size for exact (perfect) user estimates, relative urgent job cost = 10X, and 50% urgent jobs at a high offered load

<table>
<thead>
<tr>
<th>Rollback Window Size</th>
<th>Avg. OC-Factor Variance (+10&lt;sup&gt;-5&lt;/sup&gt;)</th>
<th>Load Variance</th>
<th>Revenue</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.15</td>
<td>10.60</td>
<td>473631718</td>
</tr>
<tr>
<td>4</td>
<td>6.18</td>
<td>2.89</td>
<td>508341077</td>
</tr>
<tr>
<td>16</td>
<td>7.84</td>
<td>0.62</td>
<td>555062813</td>
</tr>
<tr>
<td>32</td>
<td>2.99</td>
<td>0.34</td>
<td>692266945</td>
</tr>
<tr>
<td>48</td>
<td>1.36</td>
<td>0.24</td>
<td>715606095</td>
</tr>
<tr>
<td>64</td>
<td>1.03</td>
<td>0.17</td>
<td>715733110</td>
</tr>
<tr>
<td>128</td>
<td>1.13</td>
<td>0.04</td>
<td>701476009</td>
</tr>
</tbody>
</table>

Considered for the what-if simulations in a rollback window, and the best OC-Factor choice determined. Let us then slightly change the front end of the window to exclude the latest of these N jobs, but move back the rear end of the rollback window to include the last arriving job outside the currently chosen window. If the best choice of OC-Factor for the two rollback windows is very different, it implies that the choice procedure is very unstable.

Table 7.2 shows results that study the impact of rollback window choice on the variance of OC-Factor choice for adjacent window groups, variance of the average offered load and overall revenue, for the scenario of exact user estimates, relative urgent job cost = 10X, and 50% urgent jobs at a high offered load. By varying the length of the rollback window from 1 to 128 hours, the revenue varies from 414M units to 716M units. This shows that choosing the correct rollback window size will have a large impact on the overall revenue. However, setting the rollback window size is not trivial. If the rollback window is too small, the OC-Factor will change very erratically. The second column of the table shows the average of the variance of each set of 5 consecutive OC-Factor
choices. A higher average variance shows that the OC-Factor changes more erratically and thus will be unable to reach a good value. Because with a small rollback window, a small change in the exact jobs considered in the window will result in a very different OC-Factor, implying that OC-Factor for the new job was incorrect. However, if the value is too large, critical changes in the workload (e.g., load, percentage of urgent jobs, relative job values) may be missed. The third column shows the variance of the average offered load over the window size. As the variance in average offered load decreases, important variations in the load are being missed and the historical simulation will not be able to “see” the variations. The rollback window size needs to be large enough to have a low average OC-Factor variance (so the historical OC-Factor has meaning), but small enough to capture significant workload differences. Therefore, each scenario may require a different rollback window size that may vary over time.

Thus, a max rollback window (64 hours for this experiment) is used to dynamically vary the rollback window size. Each max rollback window hours the scheduler runs historical simulations (using each of the following rollback window sizes: 1 hour, 2 hours, 4 hours, 8 hours, 16 hours, 32 hours, 64 hours) to determine what rollback window would have yielded the best revenue over the last max rollback window hours. For the next max rollback window hours, the scheduler uses the new rollback window.

7.6.3 DVQoPS Algorithm

In this section, we describe the actual VQoPS algorithm. Figure 7.4 shows the DVQoPS flowchart encompassing the aspects described above. Figure 7.4(a) illustrates
the main flow of the DVQoPS algorithm, which passes the dynamically calculated OC-Factor to the VQoPS algorithm (used as a module). DVQoPS also asynchronously evaluates the rollback interval and OC-Factor after every fixed interval. The details of calculating these two system variables are presented in figures 7.4(b) and 7.4(c) respectively.

Figure 7.4(b) demonstrates the basic steps used to determine the best rollback interval. This is used in figure 7.4(c) when evaluating the best OC-Factor.

For each candidate rollback interval, DVQoPS runs the simulation starting candidate rollback interval hours in the past. The rollback interval is set to the candidate rollback interval that would have produced the best revenue. This best rollback interval is used for the next max rollback window hours. The OC-Factor is set by running the what-if simulation for different values of candidate OC-Factors. For each candidate OC-Factor, DVQoPS runs a what-if simulation starting rollback window hours in the past. The current OC-Factor is set to the candidate OC-Factor that yields the maximum revenue. DVQoPS uses the new OC-Factor for the next rollback window hours.

Since DVQoPS dynamically determines the best OC-Factor out of a set of $T$ OC-Factors and the best rollback window out of a set of $R$ rollback windows, its time complexity would increase to $\theta(T.R.K.N^2.log(N))$. While the time complexity of the scheme seems to be high, in practice the time taken for a scheduling event is not too much. For example, during our experiments, the scheduling event took an average of 0.9 seconds for each job. Given that job arrival times are typically in the order of several minutes in most supercomputer centers, this is not a concern.
Figure 7.4: DVQoPS Algorithm Flow Chart
7.7 Experimental Results

In this section, we use a simulation approach to evaluate the capabilities of the proposed schemes. Specifically, we compare the performance of VQoPS and DVQoPS against QoPS to study the benefits in overall revenue and service differentiation that each of these schemes can achieve.

As described in section 7.4, there are three workload characteristics that affect the opportunity cost of jobs and hence the system revenue: (i) urgent job mix, (ii) cost of urgent jobs and (iii) the load on the system. In this section, we vary the above characteristics to analyze the behavior of the different schemes. Further, we also measure the impact of user inaccuracy in runtime estimates on the performance of our schemes. As mentioned in sections 7.5 and 7.6, the time complexities of the VQoPS and DVQoPS schemes are $\theta(K.N^2.log(N))$ and $\theta(T.R.K.N^2.log(N))$ respectively. In practice the time taken for a scheduling event using DVQoPS is less than a second (as measured in our experiments) on a 2.4GHz system. Further, most of the processing of these scheduling systems can easily be processed with near linear speedup on a parallel machine, with up to $(T \times R)$ processors using a master-slave setup.

7.7.1 Impact of Urgent Job Mix on Performance

Figure 7.5(a) illustrates the percentage improvement in revenue (as compared to QoPS) for VQoPS (with different static OC-Factors) and DVQoPS for SDSC trace. The figure shows that VQoPS achieves about 20%-45% improvement in performance for all OC-Factor values. Also, depending on the trace characteristics (i.e., what percentage
Figure 7.5: Revenue improvement and Accepted load relative to QoPS assuming exact estimates at the original offered load (89%) for SDSC trace. Cost of urgent jobs is 10X compared to normal jobs. In (b), 50% urgent jobs are used.

Figure 7.6: Revenue improvement and Accepted load relative to QoPS assuming exact estimates at the original offered load (74%) for CTC trace. Cost of urgent jobs is 10X compared to normal jobs. In (b), 50% urgent jobs are used.
of the jobs are urgent jobs), different OC-Factor values perform well – there is no consistently superior OC-Factor value. DVQoPS, on the other hand, consistently achieves within 5% of the best VQoPS implementation for all traces.

Together with revenue maximization, it is important to provide service differentiation as well. Figure 7.5(b) shows the service differentiation capability of the different schemes. As shown in the figure, QoPS accepts the highest overall percentage of the workload. However, it does not differentiate between urgent and normal jobs. This inability to differentiate is what allows for the increase in revenue for the VQoPS and DVQoPS scheduling policies. As the OC-Factor is increased, the VQoPS scheduler rejects more jobs that could have been accepted by the standard QoPS algorithm, reducing the overall acceptance. However, as the OC-Factor is increased (up to a certain value) the percentage of accepted urgent jobs is increased. Very few normal jobs are accepted at OC-Factors greater than 0.05, even when the revenue is high. The increase in revenue is caused by the ability to achieve a higher revenue from urgent jobs. The DVQoPS algorithm is able to achieve competitive revenues, but accepts many more normal jobs; it even accepts more urgent jobs in scenarios where the static OC-Factor is greater than 0.05, where the revenue is often the best. In summary, the DVQoPS algorithm is able to tune the percentage of urgent and non-urgent jobs it is able to accept and to dynamically adapt itself to the trace characteristics.

To demonstrate the robustness of our schemes across different workloads, we illustrate the revenue improvement using the CTC workload in Figure 7.6. It is evident from the figure that, the trend of the results is similar in the CTC workload as well. However, the improvements are lesser compared to the corresponding SDSC workload. This is attributed to the low original offered load (74%) for CTC workload as compared to the
Figure 7.7: Revenue improvement relative to the QoPS assuming exact estimates for original and high offered load for SDSC trace

SDSC workload (89%), i.e., if the number of jobs are less, opportunity cost would be less too; thus the difference between the schemes will not be as prominent.

### 7.7.2 Impact of Cost of Urgent Jobs and System Load

Figure 7.7(a) illustrates the performance of the different schemes as we vary the cost of the urgent jobs SDSC workload. The first thing we observe is that when the cost of the urgent jobs is very low, high static OC-Factors actually perform worse as compared to QoPS. This is expected, since high static OC-Factors aim to be extremely picky about jobs by anticipating that picking cheap jobs might result in a high opportunity cost and hence a loss of revenue. However, when the urgent jobs are not very expensive, the potential opportunity cost is low; thus not accepting the cheaper jobs would in fact hurt revenue as it does in the above figure. Again, DVQoPS shows a consistent performance with less than 5% difference as compared to the best static OC-Factor.
Figure 7.8: Revenue improvement relative to the QoPS assuming exact estimates for original and high offered load of CTC trace

Figure 7.7(b) illustrates a similar trend, but for a high-load scenario. In this case, we can observe that the revenue improvements are much higher as compared to the original load scenario. This is because, though the job mix is the same as the original load case, the absolute number of urgent jobs is higher in the high-load case. Since all the schemes shown tend to pick the urgent jobs and drop the non-urgent jobs, this allows them to improve their revenue further. The overall trend, however, is still the same, with high static OC-Factors performing worse than QoPS when the cost of urgent jobs is not too high.

We illustrate the revenue improvement using the CTC trace in Figure 7.8 relative to QoPS. The revenue improvement is around 5% for original offered load (74%) and as high as 100% for higher offered load (103%). Though it is obvious from the graphs that the performance trend is same for the proposed schemes for both CTC and SDSC trace, but the performance improvement for SDSC trace is better due to higher available load.
in SDSC trace. It also substantiates that our schemes can earn further benefit from extra offered load.

### 7.7.3 Impact of User Runtime Estimate Inaccuracy

Figure 7.9(a) shows the performance of the schemes for different urgent job mixes in the case where the original inaccurate user estimates of jobs are used with SDSC workload. Compared to a scenario with exact user estimates, we notice that the overall revenues are lower in all cases (especially when the percentage of urgent jobs is low). The main reason for this is the functional difference between a service differentiation scheme and a scheme that maintains deadline guarantees. Specifically, a scheme that maintains deadline guarantees has to be conservative with respect to its estimates about a job’s runtime. For example, if a one-minute job estimates its runtime as one-hour, the scheme has to assume that its going to run for the entire one-hour and try to find a feasible schedule before it can accept the job. Because of this conservative nature of the deadline guarantee scheme, the number of jobs that are accepted are much lesser than what the system could potentially accept. Further, for all the accepted jobs, if most jobs terminate early and thus pay the maximum amount they had promised (figure 7.2), there is no real differentiation that can be achieved for these jobs. Only for the jobs that are accepted and have to wait in the queue, can we provide efficient mechanisms to improve the overall revenue of the system. Since these kind of jobs are lesser with inaccurate estimates, we see that the overall revenue is lesser as well. In general, with inexact user estimates, it may appear that little profit can be made, but due to jobs completing early, more profit can actually be made by running jobs earlier and the effect of the opportunity
Figure 7.9: Revenue improvement and Accepted load relative to QoPS assuming actual inexact estimates at the original offered load (89%) for SDSC trace. Cost of urgent jobs is 10X compared to normal jobs. In (b), 50% urgent jobs are used.

cost is reduced. Therefore, it is more often better to be more lax with the OC-Factor and accept jobs that appear to only modestly increase revenue.

Also, for low percent of urgent jobs, high OC-Factor values actually perform worse than QoPS. The reason for this is, when the static OC-Factor is high, the scheme is rejecting all the normal jobs; since the number of urgent jobs is very low, the system is being left highly under-utilized as compared to schemes with lower static OC-Factor values. This reflects in a lower revenue than even basic QoPS in some cases.

Figure 7.9(b) shows the service differentiation capabilities of the different schemes. The trends for the inexact case are pretty similar to that of the exact case. This is expected since the inaccuracy in estimation only affects the overall revenue of the system, but not the kind of jobs each scheme can accept. Like performance for SDSC trace in Figure 7.9, Figure 7.10 exhibits the same overall trend for CTC trace. Again, the improvement is less compare to SDSC due to less overall load in CTC trace.
Figure 7.10: Revenue improvement and Accepted load relative to QoPS assuming actual inexact estimates at the original offered load (74%) for CTC trace. Cost of urgent jobs is 10X compared to normal jobs. In (b), 50% urgent jobs are used.

Figures 7.11(a) and 7.11(b) illustrate the revenue improvement capability of the different schemes while varying the cost of urgent jobs for original offered load as well as high offered load for SDSC workload. The overall drop in revenue improvement is especially noticeable when the cost of urgent jobs is less. Since there are not many jobs to achieve a revenue improvement from in the inaccurate estimate case, if the urgent job cost is low, the overall revenue would get hurt as well. For a higher load, though the improvement is a little better, compared to exact estimates, there is a drastic degradation.

Figure 7.12 presents the performance of our proposed schemes for CTC trace for the same scenarios. The general tendency of improvement remains the same as SDSC in Figure 7.11 and thus demonstrates the robustness of our schemes independent of traces.

Overall, it appears that inaccuracy in user estimates has a significant impact on the revenue improvements VQoPS and DVQoPS can achieve. The service differentiation aspect is not affected, as expected. However, for a deadline guarantee scheme, we expect the supercomputer centers to provide a dual charging model, i.e., resource usage
cost and deadline guarantee cost. Resource usage cost could be the same as what we
currently have (resources × runtime). For the deadline guarantee cost, since requesting
for a deadline guarantee means that potential later jobs could be rejected, we expect
supercomputer centers to charge the user based on the estimated runtime rather than the
actual runtime. This, hopefully, would require users to improve their runtime estimates
and in turn improve the revenue gains VQoS and DVQoS can achieve.

7.8 Conclusions

In this chapter, we propose two extensions to our previous QoS-aware job schedul-
ing mechanism, QoPS, in order to analyze and minimize the impact of opportunity cost
in job scheduling. Specifically, for each job accepted, the supercomputer center may
relinquish its capability to accept some future arriving (and potentially more expensive)
(a) Original offered load (74%)  

(b) High offered load (103%)

Figure 7.12: Revenue improvement relative to the QoPS assuming inexact estimates for actual and high offered load for CTC trace

jobs, i.e., it can viewed as paying an implicit opportunity cost. Accordingly, in this chapter, we use the predictive techniques to identify such opportunity costs and attempted to minimize their impact on the overall system revenue. We analyze our designs and evaluate them in a simulation environment with different real workloads of jobs that had run on the San Diego Supercomputing Center (SDSC) in 2000 and the Cornell Theory Center (CTC) in 1997. Our results demonstrate that we not only achieve several factors improvement in system revenue, but also good service differentiation as a much desired side-effect.
CHAPTER 8

PROPORTIONAL SERVICE DIFFERENTIATION IN JOB SCHEDULING

8.1 Introduction

Supercomputer centers routinely use job schedulers as a mechanism to arbitrate access for many users on a limited number of resources. Users typically request these resources using a script containing information regarding the requested resources (e.g., number of processors, amount of memory) and the duration that the resources are needed. If the requested resources are available, the job scheduler allocates them to the job for the requested amount of time. If not, the job is queued and scheduled to be started at a later time (e.g., when the resources become available). The turnaround time (or response time) of a job is the sum of the time that the job spent waiting in the queue (for resources to be available) and the job's actual runtime.

Since the job scheduler is responsible for managing the resources available in the system, the approaches it utilizes can significantly impact the overall performance of the jobs (e.g., average turn-around time) and/or that of the system (how well the resources in the system are utilized). Accordingly, there has been a considerable amount of research
that has attempted to improve these metrics in various environments and with various trade-offs.

In addition, providing quality of service (QoS) guarantees to users has recently received some attention [71, 41, 6, 73, 6] but it is far from the expectations. The QoS in job scheduling can be classified into two broad categories: hard QoS and soft QoS. According to hard QoS, for example, a user can specify a hard deadline by which a job must be completed. The scheduler that offers hard QoS should ensure this hard guarantee once a job is accepted. Recently a substantial amount of research [71, 77, 41, ?, 40] has studied the provision of hard QoS in the form of deadline guarantees.

On the other hand, a scheduler with soft QoS functionality only supports the preferential treatment for certain jobs during job scheduling. Soft QoS, also known as service differentiation, provides the statistical preference to higher priority jobs, not a hard and fast guarantee. There are two basic ways of providing the service differentiation: absolute service differentiation (ASD) and proportional service differentiation (PSD). In ASD, a user can submit jobs with different priorities depending on the importance of the job. An ASD-enabled scheduler would always execute the jobs from the higher priority, if there is any. But there are several disadvantages of basic ASD that often follow the strict priority policy: (1) if the job from a higher priority is consistently backlogged, the job from a lower priority may experience starvation; (2) the performance spacing between two different priorities of jobs is load-dependent that eventually introduces the pricing inconsistency. For instance, there are two jobs $J_1$ and $J_2$ with charge rate $R_1$ and $R_2$ respectively where $R_1 > R_2$. It is expected that the service received by job $J_1$ is proportionally better than that of job $J_2$. In other words, the service rendered to $J_1$ is ideally $\frac{R_1}{R_2}$ times the service received by $J_2$. The ASD-aware scheduler certainly can’t
guarantee this aspect. Proportional service differentiation (PSD) could effectively address the above shortcomings of ASD. According to PSD, the higher priority job would get preferential treatment over the lower priority job as long as the service ratio is maintained. When a higher priority job is relatively over-served, the lower priority job will get the preference in scheduling over the higher priority job. Based on the above discussion, a suitable PSD-aware scheduler should preferably have the following two essential features:

- **Consistency**: A supercomputer center user expects that if she submits a job with higher priority, she would achieve better service compared to submitting the job with lower priority. This relative service differentiation should be consistent irrespective of load value and burstiness, relative to offered loads of various priority classes as well as the shape of a job.

- **Controllability**: A supercomputer center administrator should have the flexibility of specifying and adjusting the relative performance spacing among different priorities, based on some pricing or policy constraints.

Although PSD concepts received substantial attention [50, 49, 52, 51, 25] in the networking community, to the best of our knowledge, there is no research performed regarding the provision of PSD in job scheduling. In this paper, we systematically address this issue by introducing the following related questions:

- How important or relevant is the issue for the present computing centers?

- How practical is the solution to this problem?

- What are the characteristics of such a PSD-aware system?
- What would be the impact of preemption and non-preemption on the proposed solution?

- What are the trade-offs involved between PSD-aware and non-PSD scheduling?

- How are other metrics such as utilization and response time affected by this new service?

We address the obvious challenges of providing PSD service in job scheduling by first specifying a pioneer PSD-aware model. Then we design three new schemes to study PSD, ensuring minimal impact on other important metrics. We compare the proposed schemes against an EASY scheduler that is very popular but unaware of any priority or service differentiation. For the first scheme, we modify the EASY scheduler in which the jobs are scheduled in any of FCFS \[53\], WJF \[28\] or SJF \[19, 28\] order. In contrast, the proposed priority scheduler sorts the jobs according to the service provided to each job at that moment. In other words, the job that is most underserved would get the highest priority. The result shows better proportional service differentiation for the proposed scheduler compared to the PSD-unaware EASY scheduler. However, the achieved performance is still way of the expected value because the proposed non-preemptive scheduler has few decision points to prioritize the jobs.

Since the non-preemptive scheme has fundamental limitations to adjust the relative performances, we study the impact of adopting preemptive scheduling, where stopping and restarting a job at any time is possible. Particularly, a preemptive scheduler would have more flexibility to adjust the proportionality of service. The proposed scheduler essentially evaluates the relative performance of each job in every predefined decision point. When the scheduler identifies any deviation from the expected service ratio, it
takes the appropriate action by stopping the over-served jobs and starting the under-
served jobs. Accordingly, in our second scheme, we design a local preemption-based
scheme in which the suspended jobs must be restarted in the same node. Although the
scheme provides very good PSD, it is likely to generate a lot of holes in the schedule
due to the node constraints. This might eventually result in lower system utilization and
higher response time. To circumvent this problem, we propose another scheme based
on migratable preemption, in which a preempted job could be restarted in any node. In
this third scheme, we reasonably estimate the overhead due to the job migration and
consider that in subsequent scheduling. For migratable preemption, we essentially con-
sider a system with resource virtualization capability. The migratable preemption-based
scheme exhibits very good PSD without sacrificing other common metrics.

We present the detailed analysis of all the schemes with trace-based simulation using
different real workloads of jobs from Feitelson’s archive [29] and from the Ohio Super-
computer Center (OSC) [7]. Our results substantiate that the three proposed schemes
provide better PSD compared to EASY, both with respect to service differentiation ca-
pabilities and absolute performance. The results also demonstrate that preemption tech-
niques ensure better proportional service differentiation than the non-preemption tech-
nique. Furthermore, the detailed simulation data establishes the migratable preemption-
based scheme as the best, considering all metrics for the system with high data transfer
rate.

8.2 Job Scheduling Background

In this section, we give a brief background on job scheduling in general and some of
the existing models within job scheduling.
8.2.1 Overview of Job Scheduling

Parallel job scheduling is generally viewed as a space-sharing problem in a two-dimensional chart, with time along the horizontal axis and the number of processors along the vertical axis.\(^1\) In this representation, a rectangle symbolizes a job in which the required number of processors and the runtime of the job determine the height and width of the rectangle.

Most job schedulers use the First Come First Served (FCFS) policy where jobs are considered for scheduling in the order they arrive [69, 28]. Backfilling [53, 47] is a popular optimization used for FCFS and other scheduling schemes that allow the scheduler to improve system utilization without violating fairness. It allows small jobs to move ahead in the schedule and run on processors that would otherwise remain idle [53].

The primary challenge faced by most job schedulers is satisfying users and system providers at the same time. User satisfaction is often evaluated by response time which is the sum of the waiting time in the job queue (for resources to be available) and the actual runtime after the job starts running. In contrast, a supercomputer center is usually interested in the overall system utilization that determines what fraction of the resources are actually utilized and what fraction remains idle.

8.2.2 Preemption in Job Scheduling

Most job schedulers deal with jobs in a rigid form; that is, once a job has started, it has to run till completion. Job preemption is an alternative model used in some supercomputers where the scheduler can suspend a job to allow another job to use the

\(^1\)Though time-sharing variants of job-scheduling are also available, they are not as common, and more importantly are not relevant to this chapter.
resources provided to the first job. The first job is resumed when the resources become available again. There are three different types of job preemption [35]:

**No Preemption:** In this scheme, jobs are never preempted. Every job runs till completion before freeing its resources. Thus, there is no additional overhead in implementing this scheme, but it is coarse grained and has low flexibility.

**Local Preemption:** This approach allows the job scheduler to preempt jobs, but stores the state of the preempted job on the local node itself (in memory or as an image on the local disk using check-pointing facilities). This means that the job can only be restarted on the same node. On the positive side, this approach allows some flexibility with respect to scheduling jobs in a more fine-grained manner without the overheads associated with data movement across nodes.

**Migratable Preemption:** Migratable preemption is the most flexible of the preemption strategies, where a job is preempted to disk and the associated image is stored on a shared file-system. Thus the job can be restarted on any combination of nodes and does not have to be done on the same node from which it was preempted. On the flip side, however, this scheme requires the image to be moved to the shared file-system over the network and from the file-system to the new node on which it is rescheduled. This adds additional overhead whenever a job is preempted and restarted.

### 8.3 Proportional Service Differentiation Model

Proportional Service Differentiation (PSD) in parallel job scheduling is a relatively new topic. PSD provides QoS to all classes of jobs, attempting to ensure that higher priority jobs receive a proportionally larger share of the system then lower priority jobs.
The goal is to control the relative resource assignments such that neither class of jobs receive too many resources.

### 8.3.1 Properties of a Parallel Job Scheduler

The model assumes a multi-processor system, with a set of $N$ identical processors. A job, $j$, is described as a set of rational parameters, i.e., $J = a, r, n, p$ where $a$ is the arrival time, $r$ specifies the estimated runtime, $n$ is the required number of processors and $p$ denotes the priority of the job. In this model, jobs arrive to the system independently. Likewise, job attributes such as the priority, runtime and processor requested are independent of any other job and the system load. Generally jobs are submitted to the job scheduler which determines when each job will run on what processor(s).

### 8.3.2 Job Prioritization

In this model each user specifies a desired priority for each job submitted to the system. The system administrator must set a relative weight and an associated pricing for each job priority, such that each users has a mechanism to “pay” more to have jobs receive additional resources. For instance, jobs that are submitted with priority 2 may receive two times as many resources as jobs with priority 1 and may pay twice as much. A key attribute to a PSD system is that the user is paying extra to receive more “time slices” compared to a lower priority job. The goal of an ideal PSD scheduler is to assign resources such that over a time quantum, each job of a given priority is to receive the same number of time slices, and each job from another priority levels receives the specified fractional resource assignments.

This ideal does not consider the length or width of the submitted job, but rather only the number of jobs in the system at any instant. Longer (and/or wider) jobs pay extra.
simply by using more resources, it is assumed that the system charges per processor-second used. Therefore, ideally, during any time period (starting at $t_a$ and ending at $t_b$) where the number of jobs in the system are unchanged (i.e., no job completed or arrived) the resource should be shared in accordance with the relative weights ($w_i$), i.e.,

$$\text{expected}_{\text{runtime}}_i = \sum_{j \in \text{AllJobs}} \frac{w_j}{(w_j)} \times (t_b - t_a).$$

However, in a real system, strict adherence to this system may not affect the performance experience by each user. As long as each job receives the proper proportional service during the entire lifetime (from arrival to completion), the user will be mostly unconcerned (and usually unaware) whether or when the instantaneous QoS was ever violated, i.e., $\text{runtime}_i < \int_{\text{arrival}}^{\text{completion}} \text{expected}_{\text{runtime}}_i$.

Adhering to this QoS ensures that a user who pays for a high priority will receive more resources, and that lower priority jobs will still make sufficient progress in the system. However, at the end of the day most users are more concerned with when their job’s completion time rather than whether some priority metric is met, therefore it is important to consider the effects of PSD QoS on the relative response time.

### 8.3.3 Some Consequences of the Model

In general, it will be impossible to ensure that each job can meet its QoS in a strict space-shared system. In any non-preemptive system all jobs are either running or waiting for the entire time quantum. Therefore, it is impossible for a general scheduler to meet the QoS of all jobs, without placing additional restrictions on the system such as the fixing the width of all jobs. However, as mentioned in the previous subsection it may not be necessary to meet the QoS in all time quantum, as long as the QoS can be
met over the length of each job. Regardless, preemption will allow further flexibility to enable the scheduler to meet QoS specifications.

Also, when the system is in saturation, e.g. there is a non-linear relationship between submitted workload and response time, increasing the load of any priority-class results in a higher response time for all jobs. The new job should meet its proportional service in each time window, reducing the $expected_{runtime}$ of all other jobs in the time window. However, if the system were not in saturation and the new job could be assigned to otherwise idle resources, the above property will not be valid. This also implies that if there are no job waiting for service during a time quantum, the scheduler cannot guarantee the service differentiation for that duration. In that case, the service ratio for all jobs would be $1$ regardless of their priorities.

Similarly, submitting a higher priority job has a greater affect on the response time of existing jobs then a similar lower priority job. The $expected_{runtime}$ for all existing jobs is affected less when lower priority jobs arrive.

Additionally, increasing the weighted priority of any priority class improves the performance of jobs in that class at the expense of jobs in other classes. Job $expected_{runtime}$ of jobs in the changed priority class would increase, while jobs in all other priority classes would experience a decrease in their $expected_{runtime}$.

### 8.4 PSD-aware Schemes

As previously mentioned, there is no existing scheme that supports proportional service differentiation in job scheduling. In this section, we meticulously describe three new schemes with their respective strengths and limitations.
8.4.1 Simple PSD-aware Scheme

Algorithm 5 During backfill event of a Simple PSD-aware scheme

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>function BACKFILL(WaitingJobs)</td>
</tr>
<tr>
<td>2</td>
<td>Input: All waiting jobs</td>
</tr>
<tr>
<td>3</td>
<td>Output: None</td>
</tr>
<tr>
<td>4</td>
<td>call sortJobList(WaitingJobs)</td>
</tr>
<tr>
<td>5</td>
<td>for each job ( J_i \in \text{WaitingJobs} ) in above sorting order do</td>
</tr>
<tr>
<td>6</td>
<td>Schedule job ( J_i )</td>
</tr>
<tr>
<td>7</td>
<td>if Schedule found for job ( J_i ) to start now then</td>
</tr>
<tr>
<td>8</td>
<td>Remove the job from ( \text{WaitingJobs} ) and start it</td>
</tr>
<tr>
<td>9</td>
<td>else</td>
</tr>
<tr>
<td>10</td>
<td>Leave the job ( J_i ) in the WaitingJobs Queue</td>
</tr>
<tr>
<td>11</td>
<td>end if</td>
</tr>
<tr>
<td>12</td>
<td>end for</td>
</tr>
<tr>
<td>13</td>
<td>end function</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>function SORTJOBLIST(WaitingJobs)</td>
</tr>
<tr>
<td>15</td>
<td>Input: Job List to be sorted</td>
</tr>
<tr>
<td>16</td>
<td>Output: Sorted job list</td>
</tr>
<tr>
<td>17</td>
<td>for each job ( J_i \in \text{WaitingJobs} ) do</td>
</tr>
<tr>
<td>18</td>
<td>( J_i,\text{InstantaneousPriority} \leftarrow 1 - \frac{1}{J_i,\text{WaitingTime} \times \text{ServiceDiffParameter}[J_i,\text{priority}]} )</td>
</tr>
<tr>
<td>19</td>
<td>end for</td>
</tr>
<tr>
<td>20</td>
<td>Sort ( \text{WaitingJobs} ) using following keys in order</td>
</tr>
<tr>
<td>21</td>
<td>a. 1st Key: Instantaneous Priority, descending</td>
</tr>
<tr>
<td>22</td>
<td>b. 2nd Key: Service Differentiation Parameter of job, descending</td>
</tr>
<tr>
<td>23</td>
<td>c. 3rd Key: Runtime, descending</td>
</tr>
<tr>
<td>24</td>
<td>d. 4th Key: Requested processor count, descending</td>
</tr>
<tr>
<td>25</td>
<td>e. 5th Key: First Come First Serve</td>
</tr>
<tr>
<td>26</td>
<td>end function</td>
</tr>
</tbody>
</table>

In a typical scheduler, there are two main scheduling events: job’s arrival and job’s completion. When a new job arrives, the scheduler first determines if there are enough idle resources to immediately start the job. If there are not, the job is placed in the waiting queue. In a “work conserving” scheduler, there is no scope for incorporating QoS metrics during job arrival, either there are enough resources or there are not. However, during completion events, the scheduler searches the waiting queue to determine which
job to start next, often called backfilling. This is where a non-preemptive job scheduler can attempt to improve QoS metrics.

Algorithm 5 describes the details of a non-preemptive Simple PSD-aware scheme. In particular, at any backfill event, the scheduler first sorts the waiting jobs using multiple keys sequentially. The first key, called instantaneous priority, considers each job’s original priority as well as the waiting time. This key weights the wait time by the ServiceDifferentiationParameter of \( job_i \), i.e., the assigned weight for \( job_i \). Essentially, assuming equal priority, a job with a larger wait time will get the resources first. In addition, if two jobs are waiting for the same time but they have different priorities, the job with higher priority will get the preference. In short, it is the normalized waiting time with original priority of each job as the basis for instantaneous priority evaluation. When two jobs have the same instantaneous priority, the job’s original priority is considered as the secondary key.

Moreover, there are other keys such as the job’s estimated runtime and number of processor requested to give the longer and wider job preference, while the job’s arrival time is used as the last key.

8.4.2 Preemption-Based Schemes

Although the non-preemption based scheme is simple and easy to incorporate into any existing scheduler, it is limited in its ability to adjust the relative services for different jobs. In particular, since a running job can not be preempted, the scheduler has few options (only at job completion) to reorganize the jobs. Therefore, a preemption-enabled scheme, where a job can be preempted at any time, would allow the scheduler more flexibility in meeting the PSD QoS metric. In this subsection, we describe two
preemption based schemes where preemption runs at a predefined interval to ensure proportional service differentiation.

**Local Preemption Based**

In the Local preemption based scheme illustrated in Algorithm 6, all jobs that are either running or waiting in the queue are considered for service adjustment, while only the waiting jobs were considered in the non-preemption based scheme. At first, all jobs are temporarily removed from the schedule. Then the removed jobs are sorted primarily based on the historical service provided to each job. In particular, the job that received relatively less service would ultimately get preference in the next scheduling attempt. In this regard, we calculate the instantaneous priority based on the service rendered to a job, the job’s waiting time and the job’s original priority. When the instantaneous priorities of two jobs are equal, we favor the job which has higher original priority. We have used other keys to reduce the number of unnecessary job preemption.

While the jobs are sorted, the scheduler initially selects a list of jobs from the sorted list that could be scheduled in an empty schedule. Since the node-restricted jobs (currently running or previously preempted) are difficult to schedule, the scheduler attempts to assign those jobs from the selected job list first. Then the scheduler tries to schedule other non-restricted jobs.

**Migratable Preemption Based**

Local preemption enabled scheme provides the flexibility to meet the proportional service differentiation by allowing the preemption at certain intervals. However, since local preemption needs to restart the preempted job on the originally assigned nodes, it
Algorithm 6 PSD-aware Local Preemption based scheme to be run at every preemption interval

1: function LOCALPREEMPTION
3: call sortJobList(Active.Jobs)
4: call scheduleRestrictedJobs(Active.Jobs)
5: call scheduleUnRestrictedJobs(Active.Jobs)
6: end function

7: function SCHEDULERESTRICTEDJOBS(Active.Jobs)
8: Input: List of existing active jobs
9: Output: None
10: for each job \( J_i \in Active.Jobs \) do
11: if \( J_i \) is running or previously preempted then
12: Find a schedule for job \( J_i \) using the same nodes used initially
13: if schedule found for job \( J_i \) to start now then
14: Remove the job from WaitingJobs
15: else if \( J_i \) is running then
16: Preempt job \( J_i \) and put in the waiting queue
17: end if
18: end if
19: end for
20: end function

21: function SCHEDULEUNRESTRICTEDJOBS(Active.Jobs)
22: Input: List of existing active jobs
23: Output: None
24: for each job \( J_i \in Active.Jobs \) do
25: if \( J_i \) was not running at any time then
26: Find a schedule for job \( J_i \)
27: if schedule found for job \( J_i \) to start now then
28: Put the job \( J_i \) in the waiting queue
29: end if
30: end if
31: end for
32: end function

33: function sortJobList(Active.Jobs)
34: Input: List of existing active jobs
35: Output: Sorted active joblist
36: for each job \( J_i \in Active.Jobs \) do
37: \[ achievedService \leftarrow \frac{Already.Executed.Runtime}{Current.Time - Job's.Arrival.Time} \]
38: \[ J_i.Instantaneous.Priority \leftarrow 1 - \frac{achievedService}{Service.Diff.Parameter[J.priority]} \]
39: end for
40: Sort Active.Jobs using fololowing keys in order
41: a. 1st Key: Instantaneous Priority, descending
42: b. 2nd Key: Service Differentiation Parameter of job, descending
43: c. 3rd Key: Running job would get preference over non-running jobs
44: d. 4th Key: Runtime, descending
45: e. 5th Key: Requested processor count, descending
46: f. 6th Key: First Come First Serve
47: end function
Algorithm 7 PSD-aware Migratable Preemption based scheme to be run at every preemption interval

1: function MIGRATABLEPREEMPTION
2:     Input: None
3:     Output: None
4:     ActiveJobs ← WaitingJobs + RunningJobs
5:     call sortJobList(ActiveJobs)
6:     for each job $J_i \in$ ActiveJobs in the sorted order do
7:         Find a schedule for job $J_i$
8:         if schedule found for job $J_i$ to start now AND $J_i$ then
9:             Remove the job from WaitingJobs and start it
10:        else if $J_i$ is running then
11:            Expand the job’s estimated runtime using $J_i.EstimatedRuntime + = migrationOverhead$
12:            Preempt the job $J_i$ and put in the waiting queue
13:        else
14:            Leave the job $J_i$ in the waiting queue
15:        end if
16:     end for
17: end function

would generate “holes” in the schedule (from fragmentation) resulting in lower utilization and higher response time. Therefore, a migratable preemption based scheme could be a better approach. Migration allows the scheduler to restart any job on any set of nodes. However, the job’s runtime is expanded due to job’s migration cost. Migratable preemption offers the flexibility of job rearrangement in every preemption interval while ensuring the efficient usages of resources.

Algorithm 7 depicts the high-level algorithm for a Migratable preemption enabled scheme. It uses the same sorting scheme used in Local preemption, except there are no node-restriction of jobs. Another difference between the local preemption and the migratable algorithm is the incorporation of the migration cost due to the overhead of reading and writing the job’s image.
8.5 Experimental Results

In this section, we evaluate the three variants of PSD discussed in Section 8.4. We perform tests comparing five different schemes: (a) simple PSD, (b) preemptable PSD, (c) migratable PSD, (d) traditional job schedulers which do not provide any QoS (EASY) and (e) previously proposed coarse-grained service differentiation mechanisms (ASD).

8.5.1 Evaluation Setup

The comparison of different schemes is performed using a simulation-based approach using a recent real workload of 10000 jobs that had actually been submitted to the San Diego Supercomputing Center (SDSC) SP2 system. The workload contains information about the job runtime, number of nodes, and arrival time for the jobs.

**Job Priorities:** Since the SDSC SP2 does not perform any prioritization of jobs, the workload does not contain any such information either. Hence, this information was artificially added in by classifying the jobs into three classes of jobs—high priority (P3), medium priority (P2) and low priority (P1).

**Load Variation:** SDSC SP2 workload is a collection of jobs that were actually submitted to the system. Accordingly, the demand on the resources (referred to as “load”) is fixed. However, depending on what part of overall trace we pick, the instantaneous load might be quite different. For example, the load can be expected to be high during the day and lower at night. Similarly, high load closer to paper deadlines is quite common. To emulate such differences in load without diluting the workload by picking completely different jobs from the trace, we use an approach known as *load expansion*. In this approach the runtime of each job is expanded as specified by load expansion factor, while
keeping the arrival times fixed. For our evaluations, we present offered loads of 82%, 90%, 97%, and 117%

### 8.5.2 Performance Analysis of Migratable PSD

As we had described earlier, while migratable PSD has the maximum flexibility with respect to both preemption of jobs as well as restarting them at any location, it suffers from the disadvantage of data movement. Specifically, when a job is preempted, the associated image is stored on a shared file-system; similarly, when the job is restarted, it is read from the shared file-system. Clearly, the performance achieved by the file-system is critical to the overhead of migratable PSD and consequently its performance. In this section, we study the performance of migratable PSD with different total bandwidths provided by the file-system. We used file-system bandwidths of 8 GBps, 12 GBps, 16 GBps and infinite. The first three values (8, 12, 16 GBps) correspond (approximately) to our measured performance of the PVFS2 file-system with 16, 24 and 32 I/O servers respectively. The last value (infinite) refers to an ideal case environment.

Figure 8.1 shows the performance of migratable PSD with different file-system bandwidths. The x-axis shows different offered loads (described in Section 8.5.1) while the y-axis shows the ratio of the response times achieved by the low-priority jobs to that achieved by the high-priority jobs. In our experiments, the high priority jobs are promised three times better service than the low priority jobs and medium priority jobs are promised two times better service than the low priority jobs; thus, the ratio of the response times should be around three or two respectively, as well.
Figure 8.1: Provided relative service (response time) to different priorities of jobs for Migratable scheme for different file transfer rate while offered load varies.

We notice that a file-system bandwidth of about 12 GBps achieves the best performance and is within 5% of the theoretical peak (infinite bandwidth). For very high file-system bandwidth, we see a slight drop in performance. However, this is not counter-intuitive, as the y-axis is the ratio of performance achieved by the high and the low priority jobs. A drop in this value only means that the low priority jobs achieved a slightly higher improvement in performance as compared to the high priority jobs.

Figure 8.2 shows the absolute response time of the migratable PSD approach with different file-system bandwidths. These results demonstrate an expected trend in that overall response time drops as we increase file-system bandwidth and increases as we increase load. We also notice that a 12 GBps file-system bandwidth achieves within 5-10% of the theoretical peak (infinite bandwidth).
Based on the above results, we pick a file-system bandwidth of 12 GBps for the remaining experiments.

8.5.3 Comparing different PSD Schemes with respect to Service Differentiation Capability

In this section, we compare the service differentiation capability of the different PSD schemes. As mentioned in Section 8.5.2, we use a file-system bandwidth of 12 GBps for the migratable PSD scheme.

Figure 8.3 illustrates the service differentiation capability of the three PSD schemes compared to that of a traditional job scheduler that does not provide any service differentiation (legend: EASY). The figure represents service differentiation capability using
(a) Response time ratios of high and low priority jobs

(b) Response time ratios of medium and low priority jobs

Figure 8.3: Provided relative service (response time) to different priorities of jobs for different schemes while offered load varies. Ideal ratios are $\frac{\text{low}}{\text{high}} = 3$ and $\frac{\text{low}}{\text{medium}} = 2$

the ratio of the response times of the low priority jobs to that of the (a) high priority jobs and (b) the medium priority jobs. The high priority jobs are promised three times better priority and medium priority are promised two times better priority, so the ideally expected ratio is around three (or two for medium jobs) as well.

We notice two trends in this figure. First, the simple non-preemptive PSD scheme achieves the worst service differentiation of the three schemes and performs very similar to EASY. This shows that the coarse-grained nature of the simple non-preemptive PSD approach renders it largely unusable for service differentiation at low and moderate loads. At very high loads, however, the performance of simple PSD improves as compared to EASY. This is expected, as an increase in the number of waiting jobs allows a non-preemptive scheduler more flexibility in choosing which priority needs to have a job started from upon each job completion.
The second trend we notice is that while both the local preemption and migratable preemption approaches achieve better service differentiation than EASY or simple PSD, their relative performance is not the same for all loads. For light loads, local PSD achieves better service differentiation than migratable PSD. This is due to the additional load added by migratable PSD in writing the process image to the shared file-system when the job is preempted and reading it back when the job is restarted. However, for heavier loads, migratable PSD achieves better service differentiation since when the number of jobs are much higher than what the system can handle, the additional flexibility it can provide can be very beneficial, reflecting in better service differentiation capability for it.

8.5.4 Comparing different PSD Schemes with respect to Absolute Performance

In this section, we compare the absolute performance achieved by the different PSD schemes.

Figure 8.4(a) and 8.4(b) shows the absolute performance achieved by the three PSD schemes as compared to EASY. Unlike the service differentiation comparison (Figure 8.3), the response time for local preemption performs very poorly as compared to the other schemes. To understand this behavior further, we analyze the system utilization of the four schemes in Figure 8.5. Conversely, the slowdown for the Local preemption simulations is quite good. This is because small jobs dominate the slowdown, while long jobs dominate the response time, and small jobs will be able to backfill more easily into the fragmented schedule created by local preemption. This hypothesis is supported by the size-wise results in the tables in Figure 8.6. Migratable is consistently superior regarding this metric as well.
Figure 8.4: Response time and slowdown of jobs for different schemes for load=97%

Figure 8.5: Utilization achieved in different schemes for various offered loads
As shown in Figure 8.5, local preemption achieves an awfully low system utilization as compared to the other schemes. This means that while local preemption is able to maintain the service differentiation, it does so by delaying jobs significantly, resulting in many holes in the schedule (i.e., fragmentation of the schedule) and unutilized resources in the system.

<table>
<thead>
<tr>
<th></th>
<th>1 Proc</th>
<th>2-8 Procs</th>
<th>9-32 Procs</th>
<th>&gt;32 Procs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>0-10 min</strong></td>
<td>27.67</td>
<td>75.67</td>
<td>80.36</td>
<td>411.53</td>
</tr>
<tr>
<td><strong>10m-1hr</strong></td>
<td>5.31</td>
<td>7.79</td>
<td>10.55</td>
<td>9.87</td>
</tr>
<tr>
<td><strong>1hr-8hr</strong></td>
<td>0.23</td>
<td>3.44</td>
<td>7.18</td>
<td>0</td>
</tr>
<tr>
<td><strong>&gt;8hr</strong></td>
<td>0.13</td>
<td>1.62</td>
<td>2.02</td>
<td>0</td>
</tr>
</tbody>
</table>

a) Slowdown (EASY)

<table>
<thead>
<tr>
<th></th>
<th>1 Proc</th>
<th>2-8 Procs</th>
<th>9-32 Procs</th>
<th>&gt;32 Procs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>0-10 min</strong></td>
<td>74.91</td>
<td>20.67</td>
<td>48.76</td>
<td>514.72</td>
</tr>
<tr>
<td><strong>10m-1hr</strong></td>
<td>1.32</td>
<td>2.14</td>
<td>4.7</td>
<td>46.39</td>
</tr>
<tr>
<td><strong>1hr-8hr</strong></td>
<td>0.21</td>
<td>1.33</td>
<td>3.79</td>
<td>5.29</td>
</tr>
<tr>
<td><strong>&gt;8hr</strong></td>
<td>0.13</td>
<td>1.11</td>
<td>1.72</td>
<td>0</td>
</tr>
</tbody>
</table>

b) Response Time (EASY)

c) Slowdown (Non-Preemption)

<table>
<thead>
<tr>
<th></th>
<th>1 Proc</th>
<th>2-8 Procs</th>
<th>9-32 Procs</th>
<th>&gt;32 Procs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>0-10 min</strong></td>
<td>2.4</td>
<td>2.33</td>
<td>2.51</td>
<td>5.77</td>
</tr>
<tr>
<td><strong>10m-1hr</strong></td>
<td>1.25</td>
<td>2.94</td>
<td>5.81</td>
<td>6.25</td>
</tr>
<tr>
<td><strong>1hr-8hr</strong></td>
<td>0.31</td>
<td>7.99</td>
<td>10.69</td>
<td>13.82</td>
</tr>
<tr>
<td><strong>&gt;8hr</strong></td>
<td>0.49</td>
<td>8.92</td>
<td>13.48</td>
<td>0</td>
</tr>
</tbody>
</table>

d) Response Time (Non Preemption)

e) Slowdown (Local Preemption)

<table>
<thead>
<tr>
<th></th>
<th>1 Proc</th>
<th>2-8 Procs</th>
<th>9-32 Procs</th>
<th>&gt;32 Procs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>0-10 min</strong></td>
<td>18.22</td>
<td>5.59</td>
<td>5.53</td>
<td>9.67</td>
</tr>
<tr>
<td><strong>10m-1hr</strong></td>
<td>1.64</td>
<td>1.73</td>
<td>1.81</td>
<td>2.23</td>
</tr>
<tr>
<td><strong>1hr-8hr</strong></td>
<td>0.15</td>
<td>2.08</td>
<td>2.14</td>
<td>2.29</td>
</tr>
<tr>
<td><strong>&gt;8hr</strong></td>
<td>0</td>
<td>2.14</td>
<td>2.29</td>
<td>0</td>
</tr>
</tbody>
</table>

e) Slowdown (Local Preemption)

<table>
<thead>
<tr>
<th></th>
<th>1 Proc</th>
<th>2-8 Procs</th>
<th>9-32 Procs</th>
<th>&gt;32 Procs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>0-10 min</strong></td>
<td>35.0</td>
<td>429</td>
<td>583</td>
<td>1011</td>
</tr>
<tr>
<td><strong>10m-1hr</strong></td>
<td>2970</td>
<td>2807</td>
<td>3529</td>
<td>4065</td>
</tr>
<tr>
<td><strong>1hr-8hr</strong></td>
<td>0</td>
<td>2104</td>
<td>2798</td>
<td>2191</td>
</tr>
<tr>
<td><strong>&gt;8hr</strong></td>
<td>0</td>
<td>89032</td>
<td>126649</td>
<td>163727</td>
</tr>
</tbody>
</table>

e) Slowdown (Migratable Preemption)

Figure 8.6: Category-wise slowdown and response time for various schemes (Offered Load = 97%)

### 8.5.5 PSD vs. ASD

In this section, we compare the service differentiation capability of the migratable PSD scheme with the existing coarse-grained service differentiation approach (ASD).
As shown in Figure 8.7, ASD achieves a better absolute performance than PSD for high priority jobs, but worse absolute performance for low priority jobs. This is expected since ASD essentially gives infinite priority to high priority jobs and does not consider the low priority jobs for scheduling till all the high priority jobs are scheduled. Thus, it is clear that ASD would outperform PSD for high priority jobs, but at the cost of worse performance for the low priority jobs. Further, ASD does not allow for “fine-tuning” of the priorities, while in PSD the relative weight for each priority can be modified to meet the needs of the system and users.

### 8.5.6 Load Adaptation

So far, we study the impact of various offered load on different metrics. But there are other scenarios where the offered load can vary from various perspectives while the overall system load is unchanged. For instance, the load could fluctuate over time.
Further relative offered load with respect to different priority classes could also change. In this section, we explore the effect of such load variations with respect to the relevant metrics.

**Burstiness**

![Graph showing response time ratios for different priority classes](image)

(a) Response time ratios of high and low priority jobs  
(b) Response time ratios of medium and low priority jobs

Figure 8.8: Provided relative service (response time) to different priorities of jobs for different schemes with exact user runtime estimate, same offered load for each priority class and burstiness factor = 0.5 where ideally expected ratios are $\frac{\text{low}}{\text{high}} = 3$ and $\frac{\text{low}}{\text{medium}} = 2$

The offered load of system varies over time depending the time of day, day of week etc. Even the load in one hour could be very low but, in the next hour, it is very high. To study this issue in a reasonable manner, we introduce the burstiness factor that identifies the fraction by which the load fluctuates. For example, burstiness factor value 0.5 means that if the load in one interval is halved from the original load, the load on the second
Figure 8.9: Response time and slowdown of jobs for different schemes for load=97\% with exact runtime estimate, same offered load for each priority class and burstiness factor = 0.5

Figure 8.10: Utilization achieved in different schemes for various offered loads with exact runtime estimate, same offered load for each priority class and burstiness factor = 0.5
interval would be doubled. In our experimentation, without loss of generality, we use the interval of one hour and burstiness factor = 0.5.

Figure 8.8 to Figure 8.10 show the effect regarding relevant metrics including the achieved relative service and absolute performance. The result demonstrate the same trend shown in the non-bursty case (Figure 8.3–8.5). This substantiates that our proposed schemes are stable with respect to the possible load burstiness of a system.

**Priority-level Load Variations**

![Graphs showing response time ratios of high and low priority jobs](image)

(a) Response time ratios of high and low priority jobs  
(b) Response time ratios of medium and low priority jobs

Figure 8.11: Provided relative service (response time) to different priorities of jobs for different schemes with exact user runtime estimate, offered load ratio for priority class high:medium:low = 4:2:1 and burstiness factor = 0.0 where ideally expected ratios are $\frac{low}{high} = 3$ and $\frac{low}{medium} = 2$.

Until now, we consider the offered load for different priority classes are same. But in reality the relative load could change where, for instance, there could be more high priority job in the system than the lower priority job. We investigate this impact of this
Figure 8.12: Response time and slowdown of jobs for different schemes for load=97% with exact runtime estimate, offered load ratio for priority class high:medium:low = 4:2:1 and burstiness factor = 0.0

Figure 8.13: Utilization achieved in different schemes for various offered loads with exact runtime estimate, offered load ratio for priority class high:medium:low = 4:2:1 and burstiness factor = 0.0
Figure 8.14: Average response time and service ratio for different grouping of jobs comparing PSD, ASD based schemes (Local and Migratable) at offered load(97%) with exact runtime estimate, offered load ratio for priority class high:medium:low = 4:2:1 and burstiness factor = 0.0

variation by experimenting with a modified trace where the load of higher priority job is higher (i.e. high: medium: low = 4:2:1).

Figure 8.11 to Figure 8.14 display the result of concerned metrics. The graphs reveal that the schemes can adjust where the relative load of different priority classes change. Figure 8.11 shows that while there are more high priority jobs, the relative service is preserved. Figure 8.14 exhibits that, in ASD, the lower priority jobs are penalized further, since there are more high priority jobs to schedule. However, the PSD aware schemes are adaptable to this situation.

8.5.7 Inaccuracy in User Estimate

In our previous experimentations, we assumed that the user accurately estimated runtime of a job during job.s submission. The assumption was made to reduce the
Figure 8.15: Provided relative service (response time) to different priorities of jobs for different schemes with inexact user runtime estimate, same offered load for each priority class and burstiness factor = 0.0 where ideally expected ratios are $\frac{low}{high} = 3$ and $\frac{low}{medium} = 2$

Figure 8.16: Response time and slowdown of jobs for different schemes for load=97% with inexact user runtime estimate, same offered load for each priority class and burstiness factor = 0.0
Figure 8.17: Utilization achieved in different schemes for various offered loads with inexact runtime estimate, same offered load for each priority class and burstiness factor = 0.0.

The number of variables in understanding the behavior of schemes in a better predictable scenario. However, the user runtime estimate is awful in practice. Most of the time, the user over-estimates the runtime of a job. Therefore, in this section, we investigate the impact of such inaccuracy in user runtime estimation for different schemes.

Figure 8.15 to Figure 8.17 show the result of pertinent metrics where the inaccurate estimated runtime from actual trace is applied in scheduling decision. The graphs clearly exhibits the general trend is consistent with the result shown in Figure 8.3 to 8.5 where exact runtime estimation was assumed. In summery, our proposed schemes are adaptable to the situation of incorrect user runtime estimation.
8.6 Related Work

Preemption is extensively and successfully used at the operating system level in a single processor system and on shared-memory multiprocessor system [14]. There has also been some research towards creating preemptive scheduling, however much of this work [24, 70, 61] assumes malleable jobs, where the required number of processors can change over time. However, this is not a realistic model for most current supercomputers. In this chapter, we assume rigid jobs (where the required number of processors remains the same during the job execution) which is a more widely used model for real supercomputers. There has also been work in preemption techniques to improve user and system metrics [61, 45]. However, neither of these papers consider service differentiation at all.

Hard QoS in job scheduling is a recently introduced concept that has been the focus area for a lot of recent research [6, 10, 3, 73, 71, 76]. However, these approaches either consider hard QoS guarantees (where a job is only admitted if the system can guarantee the requested turn-around time), which are not the focus area for this chapter. QoS with respect to service differentiation (which is the focus area for this chapter) has been widely investigated in the networking community [80, 51, 52]. However, in job scheduling literature, there has been no research done to provide proportional service differentiation. In practice, however, a few supercomputer centers [6] provide differentiated service using absolute service differentiation (ASD) as discussed and compared against in this chapter.

In summary, this chapter presents a novel approach for proportional service differentiation and proposes various schemes that provide different tradeoffs in providing service differentiation vs. achieving the best absolute performance.
8.7 Concluding Remarks

While there have been some previously proposed schemes that aim at providing service differentiation with respect to different job priorities, they are quite naive and coarse-grained with respect to the kind of differentiation they can provide. In this chapter, we presented PSD—a proportional service differentiation scheme to allow for fine grained service differentiation in parallel job scheduling. We proposed three variants of PSD (simple PSD, locally preemptive PSD and migratable PSD) and evaluated them against each other and existing coarse-grained schemes, both with respect to service differentiation capabilities as well as with respect to absolute performance. Our experimental results demonstrate that our schemes not only significantly outperform existing approaches, but are also capable of achieving such differentiation without hurting overall performance.
CHAPTER 9

CONCLUSIONS AND FUTURE WORKS

In this chapter, we summarize the works performed for this dissertation and their associated importance. In addition, we propose a list of topics that could be studied to further realize the QoS in job scheduling.

9.1 Research Contributions

A significant volume of research has been carried out to schedule dynamically arriving parallel jobs in a space-shared parallel system. These studies are mainly focused on improving performance with respect to various system metrics such as utilization, throughput, average response time, average slowdown, etc. There has also been some limited research performed to provide differentiated service to various classes of jobs. However, there has been no work done to provide QoS service in the form of deadline guarantee. Likewise, the extent of studies performed regarding service differentiation is far from the expectations of current Super Computer centers. Moreover, there is no QoS-aware revenue model available based on job urgency. In this dissertation proposal, the above-mentioned essential issues are systematically addressed that can be summarized as follows:
• **Job scheduling scheme with QoS guarantees:** We propose a pioneer scheme called QoPS to provide the QoS guarantees in the form of response time for the consumer without overly penalizing other common metrics. Essentially, QoPS provides an admission control mechanism to newly-arrived jobs while maintaining the deadline guarantees of the already-accepted jobs. Our simulation results demonstrate the superiority of QoPS compared to other modified schemes that are initially proposed for different purposes.

• **PSD-aware schemes:** We address the obvious challenges of providing proportional service differentiation (PSD) in job scheduling by first specifying a pioneer PSD-aware model to allow for fine-grained service differentiation in parallel job scheduling. We proposed three variants of PSD (simple PSD, locally preemptive PSD and migratable PSD) and evaluated them against each other and existing coarse-grained schemes, both with respect to service differentiation capabilities as well as with respect to absolute performance. Our experimental results demonstrate that our schemes not only significantly outperform existing approaches, but are also capable of achieving such differentiation without hurting overall performance.

• **Propose a new provider-centric charging model:** We suggest a novel and effective charging model, based on the urgency sought in the response time of a job. In this model, the provider decides the price by splitting it into two distinct components: resource charge and QoS charge. The resource charge component is mainly based on the resources utilized by the submitted jobs, and is unrelated to the responsiveness of the system. The QoS charge, on the other hand, depends on the urgency of the job. For example, if two similar jobs are submitted where one
of them is urgent while the other is not urgent, the resource charge for both the jobs would be similar, whereas the QoS charge would be much different.

• **Revenue maximization in provider-centric model:** Using the provider-centric pricing model, we explore the influence of user behavior towards missed deadlines and consequently offer various techniques to counteract the negative effect of user behavior in revenue gain. Essentially, we extend a previously proposed scheme (QoPS) to provide deadline guarantee; we propose extensions to the algorithm in multiple aspects: (i) a feedback mechanism to provide the best possible deadline for jobs whose requested deadline could not be met, (ii) providing artificial slack to some jobs to maximize the overall profit the supercomputer center can achieve and (iii) utilizing Kill-and-Restart to improve the profit attainable.

• **Revenue maximization in user-centric model:** We propose a new scheduling heuristic to improve the system revenue in a user-centric model with non-QoS aware scheduler. More significantly, we prove the optimality of our approach in a simplified scenario involving a uni-processor system and an offline batch of jobs. Then, we propose sufficient conditions which when true, guarantee optimality, for an online stream of jobs on a uni-processor system. Finally, we apply our proposed scheduling scheme in a generic multiprocessor system with parallel jobs.

• **Revenue management considering the opportunity cost:** In the user-centric approach where user specifies the charge, we adopt the market-based charging model used in QoS-aware scheduling. In this context, we meticulously analyze the impact of opportunity cost in revenue and later suggest a history-based predictive approach called DVQoPS to maximize the revenue while maintaining the
deadline guarantees as well as ensuring the service differentiation. Our trace-based simulation results evidently substantiate the effectiveness of our proposed schemes.

The studies that have been carried out thus far are a major achievement in quantifying the QoS feature in parallel job scheduling as an important research topic. It is anticipated that the already-completed work along with the planned work will encourage future researchers to concentrate on QoS aspect in the broader scheduling area.

9.2 Continuing Work

The studies carried out thus far are a major stride towards quantifying the QoS feature in parallel job scheduling as an important research topic. Essentially, we address the overall issues in two broader perspectives: providing QoS services and the QoS-aware revenue model. For QoS services, we investigate the feasibility of QoS guarantees in the form of response time and eventually propose a novel and effective scheme, called QoPS, focusing on deadline guarantees. We also pursue associated revenue aspects by suggesting two new charging models based on provider and user, respectively. In both cases, we carry out extensive experimentations to understand the inherent correlation between offering guaranteed service to users and ensuring the higher revenue to providers. Additionally, we propose several effective techniques to improve overall system revenue.

9.2.1 SLA-based Service Differentiation

Service Level Agreement (SLA) is a negotiated service contract between service provider and consumer to formally defined the expected service quality [12]. We intend
to introduce this widely used SLA concept into parallel job scheduling as a way of defining the desired QoS service quality. In this section, we briefly discuss different probable variations of technical SLA that are applicable to job scheduling.

Although SLA encompasses many different aspects ranging from business to technical [79], we plan primarily to understand the impact of SLA introduction on job scheduling. We also want to study the feasibility of supporting SLA into schedulers. Fundamentally, we have to separate the scheduler from the policy. In other words, the policy can change over time by using SLA negotiation, but the scheduler should be independent of policy changes. Thus far, we have tentatively formulated a catalog of possible SLA statements along with cursory details of proposed solutions related to scheduling. We intend to further study the feasibility of some or all of these types of facilities and propose some measurable solutions.

- The average response time of jobs from class X should not exceed Y seconds. To accommodate these types of functionalities, the scheduler requires the provision of an admission control mechanism. The admission control needs to accept or reject the job depending on whether it can assure the expected service quality without violating any previous service assurance.

- Maximum allowable wait time for job J is Z seconds. We can redesign this problem by assigning a job’s deadline as job’s arrival time + job’s runtime + Z. Afterward, we can use any deadline-based scheme like QoPS for scheduling.

- The relative slowdown ratios of successive four classes (e.g.) are X, Y, Z. The scheduler requires the assurance that the average slowdown of class 1 is not more than X times the average slowdown of class 2. Likewise, average slowdown of class 2 should be less than the Y times average slowdown of class 3 and so on.
9.2.2 Quantifying the Opportunity Cost

The opportunity cost of accepting a job in parallel job scheduling signifies whether admitting a job will ultimately be profitable considering the future system conditions. Ideally, opportunity cost of a job is calculated as the difference of two revenues: estimated overall revenue of accepting a job and estimated overall revenue of rejecting a job. In online systems, these estimations are very challenging due to the uncertainty in the future system status regarding load and jobs’ criteria. In our recent work, described in section 7.4.1, we attempt to analyze the impact of different factors on opportunity cost as well as the associated job acceptance decision. We also propose a history-based dynamic approach called DVQoPS to estimate the opportunity cost. Although our scheme demonstrates superior performance in system revenue earnings, further understanding of opportunity cost is very important. We intend to further investigate the feasibility of an analytical approach to formulate the opportunity cost and eventually measure the accuracy of our estimation.
BIBLIOGRAPHY


