LIMITED PREEMPTIVE SCHEDULING IN REAL-TIME SYSTEMS

Abhilash Thekkilakattil

2016
LIMITED PREEMPTIVE SCHEDULING IN REAL-TIME SYSTEMS

Abhilash Thekkilakattil

Akademisk avhandling

som för avläggande av teknologie doktorsexamen i datavetenskap vid Akademin för innovation, design och teknik kommer att offentligen försvaras fredagen den 27 maj 2016, 13.15 i Gamma, Mälardalens högskola, Västerås.

Fakultetsopponent: Associate Professor Reinder Bril, Eindhoven University of Technology
Abstract

Preemptive and non-preemptive scheduling paradigms typically introduce undesirable side effects when scheduling real-time tasks, mainly in the form of preemption overheads and blocking, that potentially compromise timeliness guarantees. The high preemption overheads in preemptive real-time scheduling may imply high resource utilization, often requiring significant over-provisioning, e.g., pessimistic Worst Case Execution Time (WCET) approximations. Non-preemptive scheduling, on the other hand, can be infeasible even for tasksets with very low utilization, due to the blocking on higher priority tasks, e.g., when one or more tasks have WCETs greater than the shortest deadline. Limited preemptive scheduling facilitates the reduction of both preemption related overheads as well as blocking by deferring preemptions to favorable locations in the task code.

In this thesis, we investigate the feasibility of limited preemptive scheduling of real-time tasks on uniprocessor and multiprocessor platforms. We derive schedulability tests for global limited preemptive scheduling under both Earliest Deadline First (EDF) and Fixed Priority Scheduling (FPS) paradigms. The tests are derived in the context of two major mechanisms for enforcing limited preemptions, viz., defer preemption for a specified duration (i.e., Floating Non-Preemptive Regions) and defer preemption to the next specified location in the task code (i.e., Fixed Preemption Points). Moreover, two major preemption approaches are considered, viz., wait for the lowest priority job to become preemptable (i.e., a Lazy Preemption Approach (LPA)) and preempt the first executing lower priority job that becomes preemptable (i.e., an Eager Preemption Approach (EPA)). Evaluations using synthetically generated tasksets indicate that adopting an eager preemption approach is beneficial in terms of schedulability in the context of global FPS. Further evaluations simulating different global limited preemptive scheduling algorithms expose runtime anomalies with respect to the observed number of preemptions, indicating that limited preemptive scheduling may not necessarily reduce the number of preemptions in multiprocessor systems. We then theoretically quantify the sub-optimality (the worst-case performance) of limited preemptive scheduling on uniprocessor and multiprocessor platforms using resource augmentation, e.g., processor speed-up factors to achieve optimality. Finally, we propose a sensitivity analysis based methodology to control the preemptive behavior of real-time tasks using processor speed-up, in order to satisfy multiple preemption behavior related constraints. The results presented in this thesis facilitate the analysis of limited preemptively scheduled real-time tasks on uniprocessor and multiprocessor platforms.
Abstract

Preemptive and non-preemptive scheduling paradigms typically introduce undesirable side effects when scheduling real-time tasks, mainly in the form of preemption overheads and blocking, that potentially compromise timeliness guarantees. The high preemption overheads in preemptive real-time scheduling may imply high resource utilization, often requiring significant over-provisioning, e.g., pessimistic Worst Case Execution Time (WCET) approximations. Non-preemptive scheduling, on the other hand, can be infeasible even for tasksets with very low utilization, due to the blocking on higher priority tasks, e.g., when one or more tasks have WCETs greater than the shortest deadline. Limited preemptive scheduling facilitates the reduction of both preemption related overheads as well as blocking by deferring preemptions to favorable locations in the task code.

In this thesis, we investigate the feasibility of limited preemptive scheduling of real-time tasks on uniprocessor and multiprocessor platforms. We derive schedulability tests for global limited preemptive scheduling under both Earliest Deadline First (EDF) and Fixed Priority Scheduling (FPS) paradigms. The tests are derived in the context of two major mechanisms for enforcing limited preemptions, viz., defer preemption for a specified duration (i.e., Floating Non-Preemptive Regions) and defer preemption to the next specified location in the task code (i.e., Fixed Preemption Points). Moreover, two major preemption approaches are considered, viz., wait for the lowest priority job to become preemptable (i.e., a Lazy Preemption Approach (LPA)) and
preempt the first executing lower priority job that becomes preemptable \(i.e.,\) an Eager Preemption Approach (EPA)). Evaluations using synthetically generated tasksets indicate that adopting an eager preemption approach is beneficial in terms of schedulability in the context of global FPS. Further evaluations simulating different global limited preemptive scheduling algorithms expose runtime anomalies with respect to the observed number of preemptions, indicating that limited preemptive scheduling may not necessarily reduce the number of preemptions in multiprocessor systems. We then theoretically quantify the sub-optimality (the worst-case performance) of limited preemptive scheduling on uniprocessor and multiprocessor platforms using resource augmentation, \(e.g.,\) processor speed-up factors to achieve optimality. Finally, we propose a sensitivity analysis based methodology to control the preemptive behavior of real-time tasks using processor speed-up, in order to satisfy multiple preemption behavior related constraints. The results presented in this thesis facilitate the analysis of limited preemptively scheduled real-time tasks on uniprocessor and multiprocessor platforms.
Faculty Opponent:
  • Reinder Bril, Technische Universiteit Eindhoven, Netherlands

Examiners:
  • Giorgio Buttazzo, Scuola Superiore Sant’Anna di Pisa, Italy
  • Gerhard Fohler, Universität Kaiserslautern, Germany
  • Liliana Cucu-Grosjean, INRIA Paris-Rocquencourt, France

Reserve:
  • Damir Isovic, Mälardalens Högskola, Sweden

Supervisors:
  • Sasikumar Punnekkat, Mälardalens Högskola, Sweden
  • Radu Dobrin, Mälardalens Högskola, Sweden
“Lokah Samastah Sukhino Bhavantu. ”

“May all beings everywhere be happy and free, and may the thoughts, words, and actions of my own life contribute in some way to that happiness and to that freedom for all. ”

This Sanskrit verse is an expression of the universal spirit found in the ancient Indian scriptures of Vedas.
“Lokah Samastah Sukhino Bhavantu.”

“This Sanskrit verse is an expression of the universal spirit found in the ancient Indian scriptures of Vedas.
Acknowledgements

The dream begins with a teacher who believes in you, who tugs and pushes and leads you to the next plateau, sometimes poking you with a sharp stick called “truth.”

This section is my humble attempt at thanking several people who have contributed both academically and personally towards the realization of this thesis. First, let me thank my supervisors Radu Dobrin and Sasikumar Punnekkat for their unflinching support throughout the last several years that made this thesis possible. Thank you very much for believing in me, and for providing me with opportunities that positively affected the course of my studies and life. No words are enough to thank Prof. Sasi’s last minute efforts 7 years ago that gave me the opportunity to come to Sweden. Radu, it has been a privilege to be your student, and to be able to travel with you all these years. Even though I still lock my car, at least I have learned not to mistrust “not so good looking” second hand cars.

In addition to my supervisors several other people contributed towards the development of the work presented in this thesis. My sincere thanks to Sanjoy Baruah for giving me an opportunity to visit his outstanding research group at UNC Chapel Hill, as well as for providing me with a starting point into multiprocessor scheduling. Many thanks to Jim Anderson for mentoring my interests towards more

1This quote is widely attributed to Dan Rather, an American media personality.
Acknowledgements

The dream begins with a teacher who believes in you, who tugs and pushes and leads you to the next plateau, sometimes poking you with a sharp stick called “truth”.¹

This section is my humble attempt at thanking several people who have contributed both academically and personally towards the realization of this thesis. First, let me thank my supervisors Radu Dobrin and Sasikumar Punnekkat for their unflinching support throughout the last several years that made this thesis possible. Thank you very much for believing in me, and for providing me with opportunities that positively affected the course of my studies and life. No words are enough to thank Prof. Sasi’s last minute efforts 7 years ago that gave me the opportunity to come to Sweden. Radu, it has been a privilege to be your student, and to be able to travel with you all these years. Even though I still lock my car, at least I have learned not to mistrust “not so good looking” second hand cars.

In addition to my supervisors several other people contributed towards the development of the work presented in this thesis. My sincere thanks to Sanjoy Baruah for giving me an opportunity to visit his outstanding research group at UNC Chapel Hill, as well as for providing me with a starting point into multiprocessor scheduling. Many thanks to Jim Anderson for mentoring my interests towards more

¹This quote is widely attributed to Dan Rather, an American media personality.
“implementable” research. My visit at UNC also gave me an opportunity to meet Fred Brooks, who graciously accepted my request for a meeting, listened to my research, as well as gave me a few tips; many thanks to you!

I am deeply grateful to Giorgio Buttazzo for providing me with the opportunity to visit the legendary RETIS lab at Pisa. My sincere thanks to Enrico Bini and Lui Sha for allowing me to attend their courses as well as for the discussions around my research. I am also extremely thankful to Alan Burns for the feedback on mixed-criticality systems, and for the discussions around real-time systems research in general.

Rob Davis contributed to my understanding of limited preemptive scheduling by providing me with exceptional feedback. Your technical acumen and attention to details have been inspiring. Many thanks to Marko Bertogna who, as my session chair for ECRTS’14, highlighted the issue with respect to handling preemptions in multiprocessor limited preemptive scheduling, resulting in one of the papers that forms a significant contribution in this thesis.

I owe special thanks to Luis Almeida for reminding me about the connection between limited preemptive scheduling and scheduling in real-time networks, and Gordana Dodig-Crnkovic for the feedback on the research methods section of this thesis. I am extremely grateful to my friends at RETIS lab and UNC Chapel Hill for all the fun during my stay. Many conference trips were wonderful mostly because of the company of Dorin, Cristian, Youcheng, Alessandra, Alessandro, Kecheng, Zhishan and Bryan—thank you for your friendship.

Of course, the wonderful colleagues and friends at IDT (who are too many to list) deserve much appreciation for making life so much fun with the coffee breaks, movies and badminton sessions. Much appreciations also go to the administrative wing at IDT, particularly Carola, Sofia and Jenny, for making the administration activities of MRTC truly “real-time”. Many thanks to Kaj for being available whenever I had some questions on safety critical systems.

Life in Sweden wouldn’t have been so much fun without the nice company of several friends in Västerås, Örebro and Fågersta. Many
thanks to my friends Rohan and Sreekanth for their friendship, as well as for the occasional phone calls. No amount of acknowledgements would be enough to thank Mirel who helped me with my car on numerous occasions. The otherwise boring weekends were unusually pleasant just because of the football gang (who are again too many to list). No words are enough to thank Prof. Sasi, Sunitha chechi, Kannan and Kunju for being the family away from home, and making sure that we don’t miss the homely food. Many thanks to Vipin Sir and Jessy chechi for motivating me to pursue higher studies, which has been the starting point for everything.

I would also like to thank my parents for making sure that their children get the best education, and my sister for being a nice sister of a rather forgetful brother. Also, many thanks to my in-laws for being such a friendly and caring lot. Lastly, my dear wife Meera, no words are enough to describe your love and affection, and the happiness that you have brought into my life. Thank you very much for being who you are, and for putting up with my almost endless trips.

Yes, that was a pretty long acknowledgements section! However, this thesis would remain incomplete without telling the story behind it.

Abhi,
Västerås, May, 2016

*The work presented in this thesis was partially supported by Vetenskapsrådet (The Swedish Research Council) under the project CONTESSE (2010-4276), and Mälardalen University.*
Publications that form the basis of this thesis


Publications

Publications that form the basis of this thesis


Publications relevant to this thesis

1. Quantifying the Exact Sub-Optimality of Non-Preemptive Scheduling, Rob Davis, Abhilash Thekkilakattil, Oliver Gettings, Radu Dobrin and Sasikumar Punnekkat, The 36th Real-Time Systems Symposium, IEEE, San Antonio, Texas, United States 2015


**Other publications**

There was also a possibility to work on other interesting topics, a few of which have been accepted.


8. Optimizing Fault Tolerant Capabilities of Distributed Real-time systems, Abhilash Thekkilakattil, Radu Dobrin, Sasikumar
Punnekkat and Huseyin Aysan, Work in progress session, The 14th International Conference on Emerging Technologies and Factory Automation, IEEE, Palma Mallorca, Spain, September, 2009
List of Figures

1.1 Overview of the contributions to global limited preemptive scheduling. . . . . . . . . . . . . . . . . . . 4
1.2 Overview of the contributions to sub-optimality of uniprocessor scheduling. . . . . . . . . . . . . . . . . . 6
1.3 Overview of the research method adopted. . . . . . . . . 14
2.1 Real-time task attributes . . . . . . . . . . . . . . . . . 19
2.2 Preemption overheads for preemptions at a different time instant. ....................... 22
2.3 Deadline miss under non-preemptive scheduling. . . . . 25
2.4 Example schedule illustrating floating non-preemptive regions. .......................... 28
2.5 Example schedule illustrating fixed preemption points. . 29
2.6 Controlling blocking and preemption overheads using limited preemptive scheduling. . . . . . . . . . . . . 30
2.7 Example schedule illustrating push through blocking. . . 31
2.8 Example schedule illustrating lazy preemptions. . . . . . 34
2.9 Example schedule illustrating eager preemptions. . . . . 35
3.1 An illustration of the forced forward demand bound function (FF-DBF) ..................... 42
3.2 Lower bound on the work done in any time interval \([t_i, t_i - 1]\). .......................... 43
3.3 An illustration of the jobs given by algorithm 1. . . . . . 45
List of Figures

1.1 Overview of the contributions to global limited preemptive scheduling. .................. 4
1.2 Overview of the contributions to sub-optimality of uniprocessor scheduling. .............. 6
1.3 Overview of the research method adopted. ............ 14

2.1 Real-time task attributes ................................. 19
2.2 Preemption overheads for preemptions at a different time instants. ......................... 22
2.3 Deadline miss under non-preemptive scheduling. .......... 25
2.4 Example schedule illustrating floating non-preemptive regions. ....................... 28
2.5 Example schedule illustrating fixed preemption points. .... 29
2.6 Controlling blocking and preemption overheads using limited preemptive scheduling. ...... 30
2.7 Example schedule illustrating push through blocking. ... 31
2.8 Example schedule illustrating lazy preemptions. ......... 34
2.9 Example schedule illustrating eager preemptions. ........ 35

3.1 An illustration of the forced forward demand bound function (FF-DBF). .................. 42
3.2 Lower bound on the workdone in any time interval $[t_i, t_{i-1})$. ......................... 43
3.3 An illustration of the jobs given by algorithm 1. ...... 45
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Task with <em>carry-in</em> in an interval of length $t$.</td>
<td>55</td>
</tr>
<tr>
<td>4.2</td>
<td>Task without <em>carry-in</em> in an interval of length $t$.</td>
<td>56</td>
</tr>
<tr>
<td>4.3</td>
<td>An illustration of the critical scenario.</td>
<td>65</td>
</tr>
<tr>
<td>4.4</td>
<td>Weighted schedulability under varying number of tasks.</td>
<td>72</td>
</tr>
<tr>
<td>4.5</td>
<td>Weighted schedulability under varying NPR lengths (large NPRs).</td>
<td>74</td>
</tr>
<tr>
<td>4.6</td>
<td>Weighted schedulability under varying NPR lengths (small NPRs).</td>
<td>75</td>
</tr>
<tr>
<td>4.7</td>
<td>Weighted schedulability under varying number of processors.</td>
<td>76</td>
</tr>
<tr>
<td>5.1</td>
<td>Deadline miss under lazy preemption.</td>
<td>84</td>
</tr>
<tr>
<td>5.2</td>
<td>No deadline miss under eager preemption.</td>
<td>85</td>
</tr>
<tr>
<td>5.3</td>
<td>Deadline miss under eager preemption.</td>
<td>86</td>
</tr>
<tr>
<td>5.4</td>
<td>No deadline miss under lazy preemption.</td>
<td>87</td>
</tr>
<tr>
<td>5.5</td>
<td>Average number of preemptions per 100 time units under varying utilizations.</td>
<td>92</td>
</tr>
<tr>
<td>5.6</td>
<td>Weighted number of preemptions under varying number of processors.</td>
<td>94</td>
</tr>
<tr>
<td>5.7</td>
<td>Weighted number of preemptions under varying number of tasks.</td>
<td>95</td>
</tr>
<tr>
<td>5.8</td>
<td>Weighted number of preemptions under varying NPRs.</td>
<td>97</td>
</tr>
<tr>
<td>5.9</td>
<td>Preemptions under Fully Preemptive Scheduling.</td>
<td>100</td>
</tr>
<tr>
<td>5.10</td>
<td>Preemptions under Eager Preemption Approach.</td>
<td>101</td>
</tr>
<tr>
<td>5.11</td>
<td>Preemptions under Lazy Preemption Approach.</td>
<td>102</td>
</tr>
<tr>
<td>5.12</td>
<td>Preemptive behavior of G-LP-EDF.</td>
<td>104</td>
</tr>
<tr>
<td>5.13</td>
<td>Preemptive behavior of G-LP-FPS.</td>
<td>105</td>
</tr>
<tr>
<td>6.1</td>
<td>The schedule for example 18</td>
<td>119</td>
</tr>
<tr>
<td>6.2</td>
<td>Speed vs. feasibility.</td>
<td>127</td>
</tr>
<tr>
<td>6.3</td>
<td>Speed vs. LP feasibility.</td>
<td>128</td>
</tr>
<tr>
<td>7.1</td>
<td>Speed vs. G-LP-EDF feasibility under floating NPRs.</td>
<td>144</td>
</tr>
<tr>
<td>8.1</td>
<td>Methodology overview</td>
<td>155</td>
</tr>
</tbody>
</table>
List of Figures

8.2 Limited preemption requirement to guarantee preemptions only at optimal preemption points . . . . . . . . . . . 158
8.3 Limited preemption requirement to guarantee critical section execution inside non-preemptive regions . . . . . 159
8.4 Required average speed-up that guarantees non-preemptive feasibility. . . . . . . . . . . . . . . . . . . . . . . 163
8.5 Required maximum speed-up that guarantees non-preemptive feasibility. . . . . . . . . . . . . . . . . . . . . . 164

9.1 Number of preemptions under varying utilization. . . . . 191
9.2 Weighted number of preemptions under varying number of processors. . . . . . . . . . . . . . . . . . . . . . . . 192
9.3 Weighted number of preemptions under varying NPR lengths. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 193
Contents
1 Introduction 1
   1.1 Motivation and Problem Formulation . . . . . . . . . . . 3
      1.1.1 Schedulability Analysis for Global Limited Preemptive Scheduling . . . . . . . . . . . . . . 3
      1.1.2 Sub-optimality of Limited Preemptive Scheduling 5
      1.1.3 Research Goals and Research Questions . . . . . 7
   1.2 Overview of the Thesis Contributions . . . . . . . . . . 9
      1.2.1 Schedulability tests for Global Limited Preemptive Scheduling . . . . . . . . . . . . . . 9
      1.2.2 Investigation of the Runtime Preemptive Behavior of Global Limited Preemptive Scheduling .................... 10
      1.2.3 Resource Augmentation Bounds for Limited Preemptive Scheduling . . . . . . . . . . . . . . 11
      1.2.4 Preemption Control Approach using Resource Augmentation ................... 12
   1.3 Research Methodology . . . . . . . . . . . . . . . . . 13
   1.4 Organization of the Thesis . . . . . . . . . . . . . . . 15
2 Background 17
   2.1 Real-time Systems .................... 17
   2.2 Schedulability and Feasibility . . . . . . . . . . . . . . 19
   2.3 Real-time Scheduling Algorithms . . . . . . . . . . . 20
   2.4 Preemptive Real-time Scheduling . . . . . . . . . . . 21
xxi
# Contents

1 Introduction .......................... 1

1.1 Motivation and Problem Formulation .................. 3

1.1.1 Schedulability Analysis for Global Limited Preemptive Scheduling .................. 3

1.1.2 Sub-optimality of Limited Preemptive Scheduling .................. 5

1.1.3 Research Goals and Research Questions .................. 7

1.2 Overview of the Thesis Contributions .................. 9

1.2.1 Schedulability tests for Global Limited Preemptive Scheduling .................. 9

1.2.2 Investigation of the Runtime Preemptive Behavior of Global Limited Preemptive Scheduling .................. 10

1.2.3 Resource Augmentation Bounds for Limited Preemptive Scheduling .................. 11

1.2.4 Preemption Control Approach using Resource Augmentation .................. 12

1.3 Research Methodology .................. 13

1.4 Organization of the Thesis .................. 15

2 Background .......................... 17

2.1 Real-time Systems .................. 17

2.2 Schedulability and Feasibility .................. 19

2.3 Real-time Scheduling Algorithms .................. 20

2.4 Preemptive Real-time Scheduling .................. 21
### Contents

2.4.1 Preemption Related Overheads ............................................ 23

2.5 Non-Preemptive Real-time Scheduling ................................. 25
  2.5.1 Blocking in Non-Preemptively Scheduled Systems ............... 26

2.6 Limited Preemptive Real-time Scheduling ............................ 27
  2.6.1 Floating Non-Preemptive Regions ............................... 28
  2.6.2 Fixed Preemption Points ........................................ 29

2.7 Blocking and Preemption Overheads in Limited Preemptive Scheduling ................................................................. 30
  2.7.1 Push Through Blocking ........................................... 31

2.8 Managing Preemptions in Global Limited Preemptive Scheduling ............................. 32
  2.8.1 The Lazy Preemption Approach ................................. 33
  2.8.2 The Eager Preemption Approach ................................. 34

2.9 Resource Augmentation .................................................... 36

2.10 Chapter Summary ......................................................... 37

3 Global Limited Preemptive EDF ........................... 39
  3.1 Models and Notations .................................................... 40
    3.1.1 Task and Processor Model .................................... 40
    3.1.2 G-LP-EDF with Lazy Preemptions Model ................... 41
    3.1.3 Definitions ..................................................... 41

3.2 Schedulability Analysis for G-LP-EDF with Lazy Preemptions ............................. 43

3.3 Chapter Summary ......................................................... 51

4 Global Limited Preemptive FPS ........................................ 53
  4.1 Models and Notations .................................................... 54
    4.1.1 Task and Processor Model .................................... 54
    4.1.2 Global Preemptive Fixed Priority Scheduling ............ 55
    4.1.3 G-LP-FPS with Eager Preemptions Model .................. 57
    4.1.4 Definitions ..................................................... 57

4.2 Link-based Scheduling for Lazy Preemptions .......................... 59

4.3 Schedulability Analysis for G-LP-FPS with Eager Preemptions ............................. 60

4.4 Evaluation ................................................................. 70
  4.4.1 Experimental Setup ............................................... 71
  4.4.2 Experimental Results ............................................. 72

4.5 Chapter Summary ......................................................... 77

5 Preemptive Behaviors in Limited Preemptive Scheduling ....... 79
  5.1 Models and Notations .................................................... 80
    5.1.1 Task Model ....................................................... 80
    5.1.2 Definitions ..................................................... 81

5.2 Schedulability under Global LP Scheduling .......................... 82
    5.2.1 Prior Work ....................................................... 82
    5.2.2 Lazy Preemption Approach .................................... 83
    5.2.3 Eager Preemption Approach .................................... 84
    5.2.4 Schedulability under Eager and Lazy Preemption Approaches ................................................................. 86

5.3 Evaluation of Preemptive Behavior under Global LP Scheduling ............................. 89
    5.3.1 Experimental Setup ............................................... 89
    5.3.2 Results for Varying Total Utilization ....................... 92
    5.3.3 Results for Varying Number of Processors .................. 93
    5.3.4 Results for Varying Number of Tasks ......................... 94
    5.3.5 Results for Varying Lengths of Non-preemptive Regions ................................................................. 96
    5.3.6 Observations ..................................................... 97

5.4 Analysis of the Experimental Results ................................. 99
    5.4.1 Preemptive Scheduling vs. Limited Preemptive Scheduling with Eager Preemptions ................................................................. 99
    5.4.2 Lazy Preemption Approach vs. Eager Preemption Approach ................................................................. 102
    5.4.3 Global Limited Preemptive FPS vs. Global Limited Preemptive EDF ................................................................. 103

5.5 Discussions .............................................................. 105
    5.5.1 Preemption Overhead Accounting ................................ 105
    5.5.2 Energy Consumption ............................................... 106
## Contents

4.4 Evaluation ........................................... 70  
4.4.1 Experimental Setup ................................. 71  
4.4.2 Experimental Results ............................... 72  
4.5 Chapter Summary ..................................... 77  

5 Preemptive Behaviors in Limited Preemptive Scheduling 79  
5.1 Models and Notations .................................. 80  
5.1.1 Task Model ........................................ 80  
5.1.2 Definitions ........................................ 81  
5.2 Schedulability under Global LP Scheduling ............... 82  
5.2.1 Prior Works ........................................ 82  
5.2.2 Lazy Preemption Approach ......................... 83  
5.2.3 Eager Preemption Approach ......................... 84  
5.2.4 Schedulability under Eager and Lazy Preemption Approaches ............... 86  
5.3 Evaluation of Preemptive Behavior under Global LP Scheduling ............... 89  
5.3.1 Experimental Setup ................................ 89  
5.3.2 Results for Varying Total Utilization ............... 92  
5.3.3 Results for Varying Number of Processors ............ 93  
5.3.4 Results for Varying Number of Tasks ............... 94  
5.3.5 Results for Varying Lengths of Non-preemptive Regions ............... 96  
5.3.6 Observations ....................................... 97  
5.4 Analysis of the Experimental Results ............... 99  
5.4.1 Preemptive Scheduling vs. Limited Preemptive Scheduling with Eager Preemptions ............... 99  
5.4.2 Lazy Preemption Approach vs. Eager Preemption Approach ............... 102  
5.4.3 Global Limited Preemptive FPS vs. Global Limited Preemptive EDF ............... 103  
5.5 Discussions ........................................... 105  
5.5.1 Preemption Overhead Accounting ............... 105  
5.5.2 Energy Consumption .............................. 106
## Contents

8.1.1 Task and Processor model ............................................. 154
8.2 Methodology .............................................................. 155
  8.2.1 Specification of Task-level Limited Preemption
       Requirements ......................................................... 156
  8.2.2 Sensitivity Analysis for Preemption Control ................ 160
  8.2.3 Example ............................................................. 161
8.3 Evaluation ................................................................. 162
8.4 Chapter Summary .......................................................... 164

9 Conclusions and Future Work ........................................... 167
  9.1 Summary of the Contributions ...................................... 168
  9.2 Conclusions .............................................................. 171
  9.3 Future Work .............................................................. 173
    9.3.1 Minimizing Preemption Related Overheads by
          exploiting Time Triggered Scheduling ..................... 173
    9.3.2 Preemption Overhead Accounting in
          Multiprocessor Systems ......................................... 174
    9.3.3 Probabilistic Schedulability Analysis for Mixed
          Criticality Systems with Fixed Preemption Points ....... 174
    9.3.4 Preemption Thresholds for Preemption Points
          to Improve Global Limited Preemptive
          Scheduling .......................................................... 175
    9.3.5 A Practical Approach to Quantifying the Sub-
          optimality of Scheduling Algorithms ......................... 176

Bibliography ................................................................. 177

Appendix A ................................................................. 191

Populärvetenskaplig svensk sammanfattning .................................. 195

Popular Science Summary in English .......................................... 197
Chapter 1
Introduction

There was a man who drowned crossing a stream with an average depth of six inches.

Real-time computer systems are being ubiquitously deployed in many mission and safety critical applications, and are increasingly becoming the backbone of most modern cyber-physical systems, e.g., autonomous vehicles. They are typically based on contemporary uniprocessor and multiprocessor platforms which support performance enhancing hardware, e.g., caches and instruction pipelines to pre-fetch data and instructions, that significantly improve average system performance. Preemptively scheduling hard real-time tasks on such platforms typically imply non-negligible preemption and migration related overheads, potentially causing deadline misses. Consequently, the deployment of such modern processors in real-time systems requires a careful analysis of the resulting hardware-software ecosystem. High preemption and migration related overheads are considered to be an emerging problem in many real-time applications.  

1This quote that highlights the significance of real-time scheduling theory is found in the book on real-time systems by Giorgio Buttazzo [1] and has been attributed to John Stankovic's notes.
Chapter 1

Introduction

There was a man who drowned crossing a stream with an average depth of six inches.\(^1\)

Real-time computer systems are being ubiquitously deployed in many mission and safety critical applications, and are increasingly becoming the backbone of most modern cyber-physical systems, \textit{e.g.}, autonomous vehicles. They are typically based on contemporary uniprocessor and multiprocessor platforms which support performance enhancing hardware, \textit{e.g.}, caches and instruction pipelines to pre-fetch data and instructions, that significantly improve average system performance. Preemptively scheduling hard real-time tasks on such platforms typically imply non-negligible preemption and migration related overheads, potentially causing deadline misses. Consequently, the deployment of such modern processors in real-time systems requires a careful analysis of the resulting hardware-software ecosystem. High preemption and migration related overheads are considered to be an emerging problem in many real-time applications.

\(^1\)This quote that highlights the significance of real-time scheduling theory is found in the book on real-time systems by Giorgio Buttazzo \cite{Buttazzo} and has been attributed to John Stankovic’s notes.
Chapter 1. Introduction

[2][3][4]. e.g., in autonomous vehicles where data intensive operations, such as image processing for vehicular vision systems, form a critical part of the software. On the other hand, as pointed out by Short [5], non-preemptive scheduling is often favored for applications with severe resource constraints due to its low memory requirements and simple implementation. However, non-preemptive scheduling has received less attention as compared to preemptive scheduling since the works by [6][7] and [4]. The facts that non-preemptive scheduling can be infeasible even at arbitrarily low processor utilizations due to blocking on higher priority tasks [8][5], and the strict domination of preemptive scheduling over non-preemptive scheduling [9] may have contributed towards the limited efforts in addressing the feasibility of non-preemptive scheduling.

Limited preemptive scheduling offers the advantage of both preemptive and non-preemptive scheduling by limiting preemptions in the schedule, consequently enabling control of preemption related overheads as well as blocking. There are many different mechanisms for limiting preemptions (see [10]), of which we consider:

- **Floating Non-Preemptive Regions (floating NPRs)** in which preemptions are deferred by a specified time duration [9].

- **Fixed Preemption Points (FPP)** in which preemptions are restricted to specified locations in the task code [11][12][13][14].

Moreover, global limited preemptive scheduling on multiprocessor systems, especially when using Fixed Preemption Points (FPPs), brings in an additional challenge with respect to determining which of the lower priority tasks to preempt. Two principal choices with respect to the preemption approach exists:

- **Lazy Preemption Approach (LPA)** in which the scheduler waits for the lowest priority job to become preemptable.

- **Eager Preemption Approach (EPA)** in which the scheduler preempts the first job, among the lower priority ones, that becomes preemptable.
In this thesis, we investigate limited preemptive scheduling of real-time tasks on both uniprocessor and multiprocessor platforms, under Earliest Deadline First (EDF) and Fixed Priority Scheduling (FPS) paradigms. The main contributions include:

**C1:** Schedulability analysis techniques for global limited preemptive scheduling under EDF and FPS, considering eager and lazy preemption approaches, in the context of floating NPR and fixed preemption point scheduling models.

**C2:** Empirical investigation of the choice of the preemption approach and the scheduling algorithm on the number of observed preemptions at runtime.

**C3:** Resource augmentation bounds for limited preemptive scheduling of real-time tasks on uniprocessor and multiprocessor platforms.

**C4:** A methodology that enables fine grained control of the preemptive behavior of real-time tasks using processor speed-up.

### 1.1 Motivation and Problem Formulation

In this section, we briefly discuss the motivation and concretely define the challenges addressed.

#### 1.1.1 Schedulability Analysis for Global Limited Preemptive Scheduling

Limited preemptive scheduling has been extensively investigated in the context of uniprocessor systems, and there exists many research results for analyzing limited preemptively scheduled real-time tasks on uniprocessors [10]. However, similar advances for multiprocessor systems are still in their infancy.

Recently, works by Davis et al. [15][16] and Marinho et al. [17] investigated limited preemptive scheduling on multiprocessor systems.
4 Chapter 1. Introduction

Earlier, link based scheduling [18] [3] was proposed for multiprocessor scheduling of real-time tasks using shared resources, that can as well be applied to the context of global limited preemptive scheduling. All of these works considered fixed priority scheduling, with the exception of link-based scheduling that also applies to global EDF. Moreover, the above mentioned works mainly focused on limited preemptive scheduling with LPA, and did not consider EPA in detail. Lastly, the focus of most of these efforts was on tasks with FPPs, and the floating Non Preemptive Region (floating NPR) model was not explicitly considered. Therefore, extending the results to global limited preemptive earliest deadline first scheduling, considering the context of floating NPR scheduling and deriving techniques for schedulability analysis for the eager preemption approach remain as important open problems. Finally, different scheduling paradigms viz. EDF and FPS, implementing different approaches for managing preemptions viz. EPA and LPA, imply many combinations, each of which may have a different runtime behavior in achieving the actual goal of reducing the effects of preemptions. Hence, an empirical
evaluation of the different combinations with respect to the observed preemptive behavior at runtime enables system designers to choose appropriate combinations. For example, for specific systems such as mixed-criticality and energy constrained systems that depend on the slack available in the schedule to improve system performance.

An overview of the contributions of this thesis to global limited preemptive scheduling is presented in Figure 1.1.

1.1.2 Sub-optimality of Limited Preemptive Scheduling

It is known that EDF is a uniprocessor optimal scheduling algorithm [19]. This means that if there exists a uniprocessor feasible real-time taskset, then EDF can schedule it. FPS on the other hand is not uniprocessor optimal, i.e., there are uniprocessor feasible tasksets that FPS cannot possibly schedule. Consequently, many research efforts [20][21][22][23][24][25] focused on quantifying the sub-optimality of uniprocessor FPS when compared to optimal uniprocessor scheduling algorithms like EDF. These works quantified the sub-optimality of FPS when compared to EDF using 1) empirical methods like break down utilization [24], 2) optimality degree [25], and 3) resource augmentation [20][21][22][23]. Resource augmentation, first proposed by Kalyanasundaram and Pruhs [26], is a widely used method for quantifying the (worst case) performance of scheduling algorithms in terms of the extra resources, e.g., processor speed, required to achieve optimality. The goal of resource augmentation analysis is to investigate the benefits of having additional resources in the system, and to find bounds on the extra resources required to achieve a certain specified system behavior.

Inspite of the fact that the preemptive paradigm strictly dominates the non-preemptive paradigm in feasibly scheduling real-time tasks on uniprocessors [9], none of the prior research efforts were directed towards quantifying the sub-optimality of non-preemptive scheduling on uniprocessors. This would give a quantitative bound on the inefficiency of uniprocessor non-preemptive scheduling when compared to an optimal (preemptive) scheduling algorithm. Buttle
Figure 1.2: Overview of the contributions to sub-optimality of uniprocessor scheduling.

[27], in his keynote given at the Euromicro Conference on Real-time Systems in 2012, indicated that limited preemptive scheduling is widely favored in the automotive industry where data-intensive real-time tasks consisting of non-preemptable runnables are required to be cooperatively scheduled. In this case, the schedulability test must determine whether these non-preemptable runnables can be scheduled without causing deadline misses. If the taskset is unschedulable, it may mean that at least one of the non-preemptable runnables causes significantly high blocking leading to deadline misses in the schedule. Consequently, quantifying the sub-optimality of limited preemptive scheduling that guarantees a specified limited preemptive behavior on uniprocessors and multiprocessors provides for measuring the efficiency loss in terms of the (extra) computational resources required to guarantee that behavior.

The existing techniques for implementing limited preemptive scheduling do not provide for a fine grained ability to control the preemptive behavior of real-time tasks, e.g., to guarantee a user specified length of non-preemptive regions (such as in the case of automotive tasks [27]). To guarantee mutual exclusion during critical section execution, it is essential that the largest possible non-preemptive region of each task is at least as large as the largest critical section. Similarly, to minimize preemption related costs, the
length of the largest possible non-preemptive region of any task must be no less than the execution time between any two consecutive optimal preemption points (where the preemption overhead is affordable). If the length of the non-preemptive region does not guarantee a specified preemptive behavior, a fully preemptive or non-preemptive schedule may not be a choice. For example, if the preemptions are not possible at optimal preemption points, it may increase the task execution time by upto 33% [28], potentially causing deadline misses. **Augmenting limited preemptive scheduling schemes with the possibility of specifying a desired limited preemptive behavior can further enhance its applicability** in real-time systems that are based on modern processing platforms, since it enables greater flexibility for the control of preemption overheads. This could, for example, be done by the addition of extra resources.

An overview of the contributions of this thesis to the sub-optimality landscape of limited preemptive scheduling is presented in Figure 1.2.

### 1.1.3 Research Goals and Research Questions

This thesis aims to investigate limited preemptive scheduling of real-time tasks on uniprocessor and multiprocessor platforms. The two main goals of this thesis are enumerated in the following:

**G1**: Extend limited preemptive scheduling theory to the context of multiprocessors.

**G2**: Quantify the sub-optimality of limited preemptive scheduling that guarantees a specified limited preemptive behavior.

These two main goals are further subdivided into 4 sub-goals, as enumerated in the following:

**SG1**: Propose schedulability analyses for real-time tasks scheduled using Global Limited Preemptive Earliest Deadline First (G-LP-EDF) scheduling and Global Limited Preemptive Fixed
Chapter 1. Introduction

Priority Scheduling (G-LP-FPS), implementing different mechanisms to limit preemptions (Floating NPRs and FPPs), under different approaches to preemption (EPA and LPA).

**SG2:** Empirically study the effects of choosing eager and lazy preemption approaches on EDF and FPS based schedulers, with respect to the number of preemptions, when limited preemptively scheduling real-time tasks on multiprocessors.

**SG3:** Quantify the sub-optimality of uniprocessor and multiprocessor limited preemptive scheduling in terms of the widely used notion of resource augmentation.

**SG4:** Investigate methods to perform a fine grained control of the preemptive behavior of real-time tasks by trading resources in order to achieve an efficient design.

Finally, we present the research questions formulated to address the identified sub-goals. These are enumerated as follows:

**Q1:** How can we determine the schedulability of real-time tasks scheduled using G-LP-EDF with LPA assuming floating NPRs?

**Q2:** How can we determine the schedulability of real-time tasks scheduled using G-LP-FPS with EPA assuming fixed preemption points?

**Q3:** How does the number of preemptions vary with the choice of the scheduling algorithm *viz.* G-P-EDF, G-P-FPS, G-LP-EDF with EPA, G-LP-EDF with LPA, G-LP-FPS with EPA and G-LP-FPS with LPA?

**Q4:** What is the sub-optimality of uniprocessor limited preemptive EDF when compared to uniprocessor preemptive EDF in terms of the widely used notion of resource augmentation?

**Q5:** What is the sub-optimality of G-LP-EDF with LPA assuming floating NPRs when compared to an optimal multiprocessor
scheduling algorithm in terms of the widely used notion of resource augmentation?

**Q6:** How can we guarantee that various constraints such as, non-preemptive execution of critical sections, possibility of deferring preemptions to optimal preemption points and guaranteed upper-bounds on the number of preemptions, can be satisfied for greater schedulability by a fine grained control of the preemptive behavior using resource augmentation?

In the above, questions Q1 and Q2 address sub-goal SG1, question Q3 addresses sub-goal SG2, questions Q4 and Q5 corresponds to sub-goal SG3 and finally question Q6 helps in achieving sub-goal SG4. These research questions have been addressed in the form of scientific papers published in reputed peer reviewed journals and conferences.

### 1.2 Overview of the Thesis Contributions

In this thesis, we make four main contributions in the area of limited preemptive scheduling of real-time tasks. The contributions address significant open problems in the real-time scheduling theory, as well as enable formulation of guidelines for choosing specific variants of limited preemptive schedulers on multiprocessor platforms.

#### 1.2.1 Schedulability tests for Global Limited Preemptive Scheduling

We make two sub-contributions in the context of EDF and FPS scheduling respectively. Note that we consider floating NPR and fixed preemption point scheduling respectively.

1. We derive a schedulability test for G-LP-EDF with lazy preemptions assuming floating NPRs. This contribution advances the research on limited preemptive scheduling on
multiprocessors by considering EDF in conjunction with floating NPRs, and addresses research question Q1. To our knowledge this is the first such test for global LP EDF with floating NPRs.

2. We derive a schedulability test for G-LP-FPS with eager preemptions assuming fixed preemption points, addressing research question Q2. We then compare the derived test for eager preemptions against the test [29] for link-based scheduling [18], that implements lazy preemptions, using synthetic tasksets. The test for link-based scheduling used for the experiments is the state of the art response time analysis [30] after inflating the WCETs of the tasks with the largest lower priority NPR value (as proposed by Brandenburg and Anderson [29]). The work further advances the area of limited preemptive scheduling since, to the best of our knowledge, this is the first such work in the area of G-LP-FPS with eager preemptions for tasks with fixed preemption points.

Note that, even though we have considered specific combinations of the scheduling algorithm, techniques to enforce limited preemptions and the approach to preemption, our methods are general enough to be applied to all the combinations.

1.2.2 Investigation of the Runtime Preemptive Behavior of Global Limited Preemptive Scheduling

In this contribution, we perform an empirical investigation of the runtime preemptive behavior of eager and lazy approaches for global limited preemptive scheduling, similar to the comparison made for the uniprocessor case by Buttazzo [31]. Our experiments using synthetic tasksets reveal a number of observations, many of which are counter-intuitive and do not generalize from uniprocessor systems.

1. In particular, we observe that global limited preemptive scheduling does not necessarily reduce the number of
1.2 Overview of the Thesis Contributions

preemptions at run-time; in most cases, under EPA, it generates more preemptions than fully preemptive scheduling.

2. We observe that the well known property regarding the preemptive behavior of EDF on uniprocessors generalizes to the case of multiprocessors; Global Preemptive Earliest Deadline First (G-P-EDF) suffers from fewer preemptions than Global Preemptive Fixed Priority Scheduling (G-P-FPS).

3. However, we show that, this does not generalize to global limited preemptive scheduling on multiprocessors; Global Limited Preemptive Earliest Deadline First (G-LP-EDF) generates more preemptions than Global Limited Preemptive Fixed Priority Scheduling (G-LP-FPS).

4. Our experiments show that G-LP-FPS with LPA suffers from the least number of preemptions.

Lastly, the practical implications of the experimental results with respect to real-time embedded systems design are discussed. This contribution addresses research question Q3.

1.2.3 Resource Augmentation Bounds for Limited Preemptive Scheduling

We make two main contributions in the context of the sub-optimality of limited preemptive scheduling on uniprocessors and multiprocessors (respectively addressing Q4 and Q5):

1. We derive resource augmentation bounds to quantify the sub-optimality of limited preemptive scheduling on uniprocessors. This also provides us with the sub-optimality of uniprocessor non-preemptive scheduling. We build on the optimality of limited preemptive EDF and non-preemptive EDF for uniprocessors under a non-idling scheme. For example, we show that if the entire execution time of the tasks scales linearly
with the processor speed, the speed-up bound $S$ that guarantees the non-preemptive execution of all uniprocessor feasible tasks for a duration no greater than $L_{\text{max}}$ on a uniprocessor is given by

$$S \leq 2 \max \left( 1, \frac{L_{\text{max}}}{D_{\text{min}}} \right)$$

Note that we subsequently improve upon the bounds in the context of a simplified execution time model in a later publication [32].

2. We derive resource augmentation bounds for G-LP-EDF with lazy preemptions under floating NPRs when compared to an optimal multiprocessor scheduling algorithm. This also provides us with the sub-optimality of G-NP-EDF. We show that if the entire execution time of the tasks scales linearly with the processor speed, the speed-up bound required to guarantee G-LP-EDF feasibility of all $m$-processor feasible tasksets under the floating NPR model, such that all tasks can execute non-preemptively for a duration $L$, is given by,

$$S \leq 2 \max \left( 1, \frac{L}{D_{\text{min}}} \right) \left( 2 - \frac{1}{m} \right)$$

The technique that we used to derive the speed-up factors in this thesis has been re-used in other works to improve upon our bounds (e.g., [32][33][34]).

1.2.4 Preemption Control Approach using Resource Augmentation

Lastly, we propose a sensitivity analysis based methodology to achieve a fine grained control of the preemptive behavior to enable satisfaction of multiple preemption related constraints. These constraints typically include:

- Guarantee specified upper-bounds on the number of preemptions on any given task.
Chapter 1. Introduction

with the processor speed, the speed-up bound $S$ that guarantees the non-preemptive execution of all uniprocessor feasible tasks for a duration no greater than $L_{\text{max}}$ on a uniprocessor is given by

$$S \leq 2 \max\left(1, L_{\text{max}} D_{\text{min}}\right)$$

Note that we subsequently improve upon the bounds in the context of a simplified execution time model in a later publication [32].

2. We derive resource augmentation bounds for G-LP-EDF with lazy preemptions under floating NPRs when compared to an optimal multiprocessor scheduling algorithm. This also provides us with the sub-optimality of G-NP-EDF. We show that if the entire execution time of the tasks scales linearly with the processor speed, the speed-up bound required to guarantee G-LP-EDF feasibility of all $m$-processor feasible tasksets under the floating NPR model, such that all tasks can execute non-preemptively for a duration $L_{\text{max}}$, is given by,

$$S \leq 2 \max\left(1, L_{\text{max}} D_{\text{min}}\right)\left(2 - \frac{1}{m}\right)$$

The technique that we used to derive the speed-up factors in this thesis has been re-used in other works to improve upon our bounds (e.g., [32][33][34]).

1.2.4 Preemption Control Approach using Resource Augmentation

Lastly, we propose a sensitivity analysis based methodology to achieve a fine grained control of the preemptive behavior to enable satisfaction of multiple preemption related constraints. These constraints typically include:

• Guarantee specified upper-bounds on the number of preemptions on any given task.

• Guarantee that preemptions occur only at locations where the associated preemption related overheads are minimum.

This contribution provides an answer to research question Q6. In our two step methodology:

1. We first translate the preemption related constraints in to a set of task level limited preemption requirements (as the required length of the maximum non-preemptive regions).

2. We then leverage on the speed-up bounds derived previously to perform a sensitivity analysis on the processor speed that guarantees the derived limited preemption requirements.

Even though we present the sensitivity analysis in the context of uniprocessors, the technique can be extended to multiprocessor floating NPR scheduling as well.

1.3 Research Methodology

According to Shaw's classification of research products [35][36], we propose techniques that facilitate the analysis of limited preemptively scheduled real-time tasks on multiprocessors. According to Shaw [35], the associated research approach or method involve:

“Invent new ways to do some tasks, including procedures and implementation techniques. Develop a technique to choose among alternatives.”[35]

We have also developed qualitative or descriptive models that describe the preemptive behavior of multiprocessor preemptive and limited preemptive scheduling algorithms, as well as the extra resources required to guarantee a specified limited preemptive behavior.
According to Shaw, the methods adopted to develop *qualitative or descriptive model* involve:

“Organize & report interesting observations about the world. Create & defend generalizations from real examples. Structure a problem area; formulate the right questions. Do a careful analysis of a system or its development.” [35]

The above mentioned *qualitative or descriptive model* that we developed may also be classified as an *empirical predictive model* since the methods adopted to develop an *empirical predictive model* include:

“Develop predictive models from observed data.”[35]

We have adopted a combination of widely used techniques to develop and validate the research presented in this thesis. The general method adopted involved a mix of literature survey, identification of the problem, solving the problem by building on the state of the art, formal proofs of correctness and evaluation of the results, followed by peer-review [37][38]. An overview of the research methodology adopted in this paper is presented in Figure 1.3. It roughly follows the terminology adopted by Holz et al. [37].

In the first phase of our research, we performed a literature survey to understand the state of the art techniques in real-time scheduling and identify open problems that may be of interest. In the second phase,
1.4 Organization of the Thesis

The rest of this thesis is organized as follows:

- **Chapter 2** contains a background of real-time scheduling in general and limited preemptive scheduling in particular. It also contains a background on resource augmentation, that is widely used to quantify the performance of scheduling algorithms.

- **Chapter 3** describes the solution to the first of our research question, namely Q1. It describes the schedulability analysis technique for Global Limited Preemptive EDF with lazy preemption approach, using floating NPRs.

- **Chapter 4** describes the solution to the second research question Q2 and contains the schedulability analysis technique for Global Limited Preemptive FPS with eager preemption approach, using fixed preemption points.

- **Chapter 5** describes our experiments simulating Global Preemptive EDF and FPS, as well as, Global Limited Preemptive
EDF and FPS with eager and lazy preemption approaches for real-time tasks with fixed preemption points. It addresses research question Q3.

- **Chapter 6** derives resource augmentation bounds for uniprocessor limited preemptive scheduling that guarantees a specified *limited preemption behavior* in the schedule, addressing research question Q4.

- **Chapter 7** addresses research question Q5 and contains resource augmentation bounds for multiprocessor limited preemptive scheduling in the context of G-LP-EDF for floating NPR scheduling.

- **Chapter 8** addresses research question Q6 and presents a two step sensitivity analysis based technique for fine grained preemption control using resource augmentation.

- **Chapter 9** presents the conclusions of this thesis and describes potential areas for future work.
Chapter 2

Background

If the past is any guide to the future, the availability of more computing power will only open up real-time applications requiring greater functionality, thus exacerbating the timing problems.¹

2.1 Real-time Systems

Computing systems are being ubiquitously deployed in critical applications, e.g., in the transportation domain, to improve safety, efficiency and user-experience. Real-time systems form a major part of many of these, what may be called, computing ecosystems, and are increasingly becoming the backbone of most safety and mission critical applications. Real-time systems are computer systems where the correctness of the system depends not only on the functional correctness of the computations performed, but also on the timeliness of these computations.

¹This quote from the 1988 article by John Stankovic [39], in which he dispels many misconceptions about real-time systems, anticipates the severity of some of the challenges faced by real-time computing today e.g., problems associated with caches.
Example 1. The airbag control unit found in modern cars is a typical example of a real-time system. The airbag control unit detects a crash, evaluates its severity and, if needed, triggers the deployment of the airbags. The deployment of the airbags must happen within a specified time interval for it to be effective in minimizing injury (i.e., before the impact of the crash on the passengers). This means that the different computations happening in the system, e.g., sensing, determining the severity of the crash and the deployment of the airbags, must all happen in a timely manner (before the impact).

The events occurring in the environment in which a real-time system operates, which are typically periodic, sporadic or aperiodic, are mapped to a set of real-time tasks that are expected to perform a specified computation (and in most cases perform an actuation) within a specified bounded time interval. In this thesis, we focus on periodic and sporadic real-time tasks. Each real-time task has a set of task attributes that are used to specify the associated timing requirements. The attributes of a periodic or a sporadic real-time task typically consist of its Worst Case Execution time (WCET), the minimum or an exact inter-arrival time (also called period), and a relative deadline with respect to its release time.

- The *Worst Case Execution Time (WCET)* of a task is defined as the upper-bound on the time that the task could take to complete its execution without any preemptions (note that some works assume preemption overheads to be a part of the task execution time).

- The *inter-arrival time (or period)* of a task is defined as the minimum time duration between any two consecutive arrivals of the jobs of the task.

- The *relative deadline* of a task is defined as the time duration with respect to its release time within which the task must complete its execution.

Figure 2.1 illustrates the attributes of a real-time task (note that this is a very basic model of a real-time task). Each task in the taskset generates
Example 1. The airbag control unit found in modern cars is a typical example of a real-time system. The airbag control unit detects a crash, evaluates its severity and, if needed, triggers the deployment of the airbags. The deployment of the airbags must happen within a specified time interval for it to be effective in minimizing injury (i.e., before the impact of the crash on the passengers). This means that the different computations happening in the system, e.g., sensing, determining the severity of the crash and the deployment of the airbags, must all happen in a timely manner (before the impact).

The events occurring in the environment in which a real-time system operates, which are typically periodic, sporadic or aperiodic, are mapped to a set of real-time tasks that are expected to perform a specified computation (and in most cases perform an actuation) within a specified bounded time interval. In this thesis, we focus on periodic and sporadic real-time tasks. Each real-time task has a set of task attributes that are used to specify the associated timing requirements. The attributes of a periodic or a sporadic real-time task typically consist of its Worst Case Execution time (WCET), the minimum or an exact inter-arrival time (also called period), and a relative deadline with respect to its release time.

- The Worst Case Execution Time (WCET) of a task is defined as the upper-bound on the time that the task could take to complete its execution without any preemptions (note that some works assume preemption overheads to be a part of the task execution time).
- The inter-arrival time (or period) of a task is defined as the minimum time duration between any two consecutive arrivals of the jobs of the task.
- The relative deadline of a task is defined as the time duration with respect to its release time within which the task must complete its execution.

Figure 2.1 illustrates the attributes of a real-time task (note that this is a very basic model of a real-time task). Each task in the taskset generates a potentially infinite sequence of jobs, depending on the associated physical events, such that any two consecutive jobs are separated by the exact/minimum inter-arrival time. Depending on the consequences of a deadline miss, the real-time system can be classified as either a hard real-time system or a soft real-time system. In hard real-time systems, e.g., a flight control system, deadline misses can potentially cause catastrophic consequences such as loss of lives or property. In soft real-time systems, on the other hand, deadline misses cause a mere degradation of the service levels, e.g., telecommunication and multimedia systems.

### 2.2 Schedulability and Feasibility

One of the main goals of real-time systems design is to provide temporal guarantees for the real-time tasks. This is typically achieved using a schedulability and/or a feasibility test.

- A schedulability test determines whether or not, for any given taskset and a specified scheduler, deadlines will be missed in the schedule generated by that scheduler.

- A feasibility test determines the existence of a valid real-time schedule for any given taskset, independent of the scheduling algorithm.
If a taskset is deemed to be feasible using a suitable feasibility test, the scheduling algorithm that can schedule the taskset still needs to be found. There exists several utilization based [40], response time based [41][42] [30] and demand bound based [43] [44] schedulability tests for major uniprocessor and multiprocessor scheduling algorithms. In the uniprocessor case, the well known feasibility tests [43] [45] are derived building on the uniprocessor optimality of several scheduling algorithms e.g., EDF on uniprocessors.

2.3 Real-time Scheduling Algorithms

The real-time tasks have to be executed on a computing platform with one or more processors while satisfying the hard or soft timing requirements. The number of processors are typically fewer in number than the number of tasks, and hence appropriate real-time scheduling algorithms are required to guarantee the timeliness of the real-time tasks sharing the processors. Real-time scheduling algorithms can be classified as either an online or an offline algorithm depending on when the scheduling decision is made. In offline scheduling, the scheduling decisions are made offline and the schedule is stored, e.g., in a table. In this case, the advantage is that, much of the complexities concerning the schedule generation can be handled offline using very complicated tools and techniques, while ensuring a simple runtime mechanism for dispatching the tasks. However, the main disadvantage is that the schedule needs to be recomputed every time there is a change in the taskset. In online scheduling, on the other hand, scheduling decisions happen during runtime using a suitable criteria, e.g., priorities. The main advantage of using an online scheduling algorithm is that the scheduler has the possibility of adapting to changing factors e.g., tasks executing for less than their worst case. On the other hand, since the scheduling decisions are made at runtime, it can potentially increase overheads in the schedule.

Typically, online scheduling algorithms are priority driven, and hence can be classified as fixed task priority, fixed job priority and
If a taskset is deemed to be feasible using a suitable feasibility test, the scheduling algorithm that can schedule the taskset still needs to be found. There exist several utilization based [40], response time based [41][42] and demand bound based [43][44] schedulability tests for major uniprocessor and multiprocessor scheduling algorithms. In the uniprocessor case, the well-known feasibility tests [43][45] are derived building on the uniprocessor optimality of several scheduling algorithms, e.g., EDF on uniprocessors.

### 2.3 Real-time Scheduling Algorithms

The real-time tasks have to be executed on a computing platform with one or more processors while satisfying the hard or soft timing requirements. The number of processors are typically fewer in number than the number of tasks, and hence appropriate real-time scheduling algorithms are required to guarantee the timeliness of the real-time tasks sharing the processors. Real-time scheduling algorithms can be classified as either an online or an offline algorithm depending on when the scheduling decision is made. In offline scheduling, the scheduling decisions are made offline and the schedule is stored, e.g., in a table. In this case, the advantage is that much of the complexities concerning the schedule generation can be handled offline using very complicated tools and techniques, while ensuring a simple runtime mechanism for dispatching the tasks. However, the main disadvantage is that the schedule needs to be recomputed every time there is a change in the taskset. In online scheduling, on the other hand, scheduling decisions happen during runtime using a suitable criteria, e.g., priorities. The main advantage of using an online scheduling algorithm is that the scheduler has the possibility of adapting to changing factors, e.g., tasks executing for less than their worst case. On the other hand, since the scheduling decisions are made at runtime, it can potentially increase overheads in the schedule.

Typically, online scheduling algorithms are priority driven, and hence can be classified as fixed task priority, fixed job priority and dynamic priority scheduling depending on the priority assignment policy. In fixed task priority scheduling, the task priorities are assigned offline and all the jobs of any given task execute with the same priority, e.g., Rate Monotonic (RM) scheduling. In fixed job priority scheduling, different jobs of any given task can have different priorities, however, the priority of any given job does not change during its execution, e.g., Earliest Deadline First (EDF) scheduling. In dynamic priority scheduling, on the other hand, the job priorities are recomputed online and the same job of any given task can have different priorities at different time instants, e.g., least laxity first scheduling.

In the context of a processing platform with at least two processors (i.e., a multiprocessor system), the scheduling algorithms can also be classified as either partitioned or global scheduling. In partitioned scheduling, the tasks are partitioned onto individual processors using a suitable bin-packing strategy offline, and then the tasks are scheduled using suitable uniprocessor scheduling algorithms, e.g., partitioned EDF. In global scheduling, the tasks are despatched to the processors from a global queue according to the specified priority assignment strategy, e.g., Global Preemptive EDF.

Real-time scheduling algorithms can also be classified depending on whether the currently executing task can be suspended and replaced with a higher priority task. If a preemptive scheduler is employed, then the currently executing task can be suspended and replaced with a higher priority task. On the other hand, under non-preemptive scheduling, if a higher priority task is released during the execution of a lower priority task, the scheduler waits for the lower priority task to complete before allowing the higher priority task to execute.

### 2.4 Preemptive Real-time Scheduling

In real-time scheduling, preemptions are an important mechanism that enables efficient utilization of the computing platform. Being able to stop the currently executing task and allow another task (that may have an earlier deadline or higher priority) to execute is necessary to
22 Chapter 2. Background

Preemption related overheads

(a) Preemption overheads in real-time scheduling.

(b) Deadline miss due to preemptions at different time instants.

Figure 2.2: Preemption overheads for preemptions at a different time instants.

guarantee that urgent tasks are executed as soon as possible. Enabling preemptions in operating systems require special mechanisms to safely stop the currently executing task, save its current context so that it can continue its execution from the same point, and replace it with the preempting task. These mechanisms that enable preemptions, together with performance enhancing features such as caches, typically introduce overheads in the schedule, that in turn delay the completion of the real-time tasks, potentially affecting their timeliness.

Example 2. Consider a task $\tau_j$ executing on a processor as shown in Figure 2.2a. If a more urgent task $\tau_i$ (having an earlier deadline) is released as shown in Figure 2.2a, then the scheduler suspends $\tau_j$ and allows $\tau_i$ to execute in its place. However, as shown in the figure, when $\tau_j$ resumes its execution, the preemption related overheads may increase its response time. This is interesting since, at a different point of preemption, the preemption related overheads could be higher, consequently leading to an even larger response time potentially resulting in deadline misses (as shown in Figure 2.2b).

The above example illustrates the need for safe and efficient accounting of preemption overheads in the schedule since the points of preemptions determine the overheads, that in turn influence the task response times. For example, it has been shown that, in the case of
uniprocessor systems the critical instant may not necessarily give the worst case response times in the presence of preemption related overheads [46]. This is because, if the tasks are preempted at different locations (different from the scenario given by the critical instant), the preemption overheads in the schedule could be higher than that given by the critical instant, leading to larger task response times. Lastly, the above example illustrates the need to limit preemptions in the schedule to improve predictability of real-time systems.

2.4.1 Preemption Related Overheads

Preemption (and migration) related overheads typically result from the mechanisms that enable preemptions. In modern processing platforms, it may also arise from special hardware features, such as caches, that are used to significantly improve average system performance. The most commonly considered preemption related overheads are:

- **Context Switch Related Overheads:** Whenever a lower priority task is preempted by a higher priority task, the context of the preempted task is saved, and needs to be reloaded, in order to resume its execution from the same point at a later time. The delay incurred in performing this context switch manifests as a temporal overhead in the schedule, potentially causing deadline misses on real-time tasks.

- **Cache Related Preemption and Migration Delays:** Preemptions may also displace data from the cache memory that is currently used by the preempted task. This data may need to be reloaded when the preempted task resumes its execution. The delay involved in reloading the evicted cache lines causes temporal overheads which can vary with the point of preemption [46]. The cache related preemption delays can be of the order of hundreds of micro-seconds for a single preemption [28], and in several cases are in excess of one milli-second [2]. Moreover, the CPMDs can be ill-defined if there is a high contention for the
shared caches in the system. Consequently, the cumulative temporal overhead on a task, due to preemptions, can be very high depending on the number of preemptions and the points at which the preemptions occur. Hence, reducing the number preemptions, as well as restricting them to only specified points in the task code is highly desirable.

- **Pipeline Related Overheads**: Preemption on a lower priority task flushes the instruction pipelines in order to load the instructions of the higher priority task. When the preempted task resumes its execution, the pipeline has to be refilled. This flushing and refilling of the instruction pipeline manifest as a temporal overhead in the system, leading to potential deadline misses.

The fundamental feasibility and schedulability analysis techniques (see [1] and [47] for details) of preemptive real-time tasks typically assume negligible preemption related overheads. Whenever the overheads are not negligible, they are assumed to be (pessimistically) accounted for in the worst case execution times of the tasks. Extending some of these fundamental techniques, many methods were proposed to efficiently account for the preemption related overheads in the schedulability analysis (e.g., [48] [49] [50] [51] [52] [53]). However, all the preemptions considered by the above mentioned works may not occur in the actual schedule leading to analysis pessimism, that typically results in an over-provisioned design, affecting efficiency. Reducing preemption and migration related overheads, and safely accounting them, is an important challenge in many real-time applications. One example is the vehicular vision system used in autonomous vehicles which contain data intensive operations, such as image processing, that can significantly increase e.g., context switch overheads.
2.5 Non-Preemptive Real-time Scheduling

On the other hand, under non-preemptive scheduling, the scheduler does not allow the currently executing task to be preempted. Any higher priority tasks released during the execution of a task has to wait for it to complete its execution. This means that the scheduler need not implement or execute mechanisms to suspend the currently running task, save its context or reload the context when it resumes execution, implying negligible runtime overheads when compared to a preemptive scheduler.

![Diagram](image)

(a) Blocking in non-preemptive scheduling.
(b) Deadline miss due to blocking under non-preemptive scheduling.

Figure 2.3: Deadline miss under non-preemptive scheduling.

**Example 3.** Consider the job of task $\tau_j$ executing on a processor as shown in Figure 2.3a. Under non-preemptive scheduling, even if a more urgent task $\tau_i$ (having an earlier deadline) is released as shown in Figure 2.3a, the scheduler allows $\tau_j$ to complete its execution only after which it allows $\tau_i$ to execute. Since $\tau_j$ has an execution time that is larger than the deadline of $\tau_i$, in general, it is not possible to schedule the tasks using a non-preemptive scheme. If the task release happens as shown in Figure 2.3b, then $\tau_i$ will miss its deadline due to the blocking from $\tau_j$.

The absence of preemption related overheads makes non-preemptive scheduling the preferred scheduling algorithm for applications with severe resource constraints [5]. Howell and
Venkatrao [54] showed that no optimal online scheduling algorithm that uses inserted idle times exists for non-preemptively scheduling sporadic real-time tasks. Jeffay et al. [4] showed that, under a non-idling scheme, non-preemptive EDF is optimal on a uniprocessor for sporadic task systems. Putting these two results together, when it comes to non-preemptively scheduling sporadic real-time tasks on uniprocessors, non-preemptive EDF is optimal, i.e., if there exists an algorithm that can non-preemptively schedule a sporadic real-time taskset, non-preemptive EDF can schedule it. However, note that non-preemptive EDF is not an optimal uniprocessor scheduling algorithm, unlike preemptive EDF. To our knowledge there exists no optimal non-preemptive scheduling algorithm on multiprocessors.

2.5.1 Blocking in Non-Preemptively Scheduled Systems

The above example on non-preemptive scheduling demonstrates what is called the long task problem [5]: if any task in a given taskset has a deadline that is shorter than the largest execution time in the taskset, then in general it is not possible to schedule that taskset using the non-preemptive paradigm. Preemptive scheduling strictly dominates non-preemptive scheduling, a major reason of which is the long task problem [9]. The possibility of having tasksets with the long task problem makes non-preemptive scheduling infeasible even at arbitrarily low processor utilizations [5]. We illustrate this using the following example.

**Example 4.** Consider two tasks $\tau_1$ and $\tau_2$ such that $\tau_1$ has an execution time= 1, deadline= 2 and a period= 1000, and $\tau_2$ has an execution time= 5, period=deadline= 5000. In this case the total utilization of the taskset is:

$$U_{tot} = \frac{1}{1000} + \frac{5}{5000} = 0.001 + 0.001 = 0.002$$

However, if $\tau_1$ is released at an arbitrarily small time instant after $\tau_2$ starts its execution, then $\tau_1$ will miss its deadline.

Even though the long task problem is less of a concern in multiprocessor systems [8], it still significant. Therefore, we can say
that the applicability of a non-preemptive scheduling scheme is limited to only a small fraction of the feasible task sets. Note that, in this thesis, we consider non-idling non-preemptive scheduling algorithms.

2.6 Limited Preemptive Real-time Scheduling

The limited preemptive scheduling paradigm addresses the blocking related infeasibility under non-preemptive scheduling, while enabling control of the preemption related overheads. Many techniques have been proposed in the literature to limit preemptions in the schedule e.g., preemption threshold scheduling [55] [56], floating non-preemptive regions [9] and fixed preemption points [11] [12] [13]. A detailed survey of limited preemptive scheduling is available in [10]. In this thesis, we focus on two of the major mechanisms for implementing the limited preemptive scheduling paradigm. The preemptive behavior of real-time tasks are controlled by using either:

- Floating Non-Preemptive Region (floating NPR) scheduling which defers preemptions for a specified time duration.
- Fixed Preemption Points (FPP) scheduling which restricts preemptions to predetermined points in the program code.

The need to limit preemptions in real-time systems is well recognized by both academia and industry. In the keynote given by Darren Buttle [27], he indicates that limited preemptive scheduling is widely favored in the automotive industry where data-intensive real-time tasks consisting of non-preemptable runnables are required to be cooperatively scheduled.

Note that limited preemptive scheduling has also been applied in the context of real-time networks to transmit multiple packets of the same message. Bartolini et al. [57] considered scheduling multiple packets of the same message in event triggered network systems, in particular, the Controller Area Network (CAN). Almeida and Fonseca [58] considered scheduling message packets in the context of
synchronous systems by exploiting the possibility of inserting idle times. The FTT-CAN framework [59] exploits the possibility of limited preemptive scheduling of messages over the network by breaking down messages into packets. Ashjaei et al. [60] presented an optimization based method to derive optimum lengths of the packets in Ethernet based networks. Many of these techniques implemented in the context of real-time networks are similar to uniprocessor limited preemptive scheduling using fixed preemption points in which the non-preemptive regions are determined offline.

2.6.1 Floating Non-Preemptive Regions

In the floating non-preemptive region scheduling model [9], if a higher priority task is released during the execution of a lower priority task, the lower priority task switches to a non-preemptive mode for specified upper-bounded duration, before allowing the higher priority task to preempt.

![Floating non-preemptive region](image_url)

Figure 2.4: Example schedule illustrating floating non-preemptive regions.

**Example 5.** Consider a lower priority task executing on the processor as shown in Figure 2.4. If a higher priority task is released then the lower priority task will start executing its NPR (for a specified maximum duration), only after which the higher priority task is allowed to preempt.
Chapter 2. Background

Synchronous systems by exploiting the possibility of inserting idle times. The FTT-CAN framework [59] exploits the possibility of limited preemptive scheduling of messages over the network by breaking down messages into packets. Ashjaei et al. [60] presented an optimization based method to derive optimum lengths of the packets in Ethernet based networks. Many of these techniques implemented in the context of real-time networks are similar to uniprocessor limited preemptive scheduling using fixed preemption points in which the non-preemptive regions are determined offline.

2.6.1 Floating Non-Preemptive Regions

In the floating non-preemptive region scheduling model [9], if a higher priority task is released during the execution of a lower priority task, the lower priority task switches to a non-preemptive mode for specified upper-bounded duration, before allowing the higher priority task to preempt.

Example 5. Consider a lower priority task executing on the processor as shown in Figure 2.4. If a higher priority task is released then the lower priority task will start executing its NPR (for a specified maximum duration), only after which the higher priority task is allowed to preempt.

The lower priority task executes non-preemptively for the specified maximum duration every time a higher priority task is released during its execution (i.e., regardless of the number and points of release of high priority tasks).

2.6.2 Fixed Preemption Points

In the fixed preemption point scheduling model [11] [12] [13], every task consists of a set of fixed preemption points in the task code. These points may be explicitly specified by the programmer [14] or, in the case of many automotive systems where non-preemptive runnables are grouped together to form real-time tasks [27], the preemption points are given by the finish time of each non-preemptive runnable.

![Fixed Preemption Points](image)

Figure 2.5: Example schedule illustrating fixed preemption points.

Example 6. Consider a lower priority task having a set of fixed preemption points executing on a processor as shown in Figure 2.5. If a higher priority task is released and the lower priority task has not reached a preemption point, the higher priority task will wait for the next available preemption point.

The lower priority task executes non-preemptively only until the next preemption point (i.e., shorter the time to the next preemption point, shorter the duration of the non-preemptive execution of the lower priority task).
Chapter 2. Background

2.7 Blocking and Preemption Overheads in Limited Preemptive Scheduling

Limited preemptive scheduling is an attractive solution to obtain the best of both preemptive and non-preemptive scheduling. In the following example, we illustrate how limited preemptive scheduling can be exploited to control preemption related overheads while keeping the blocking “affordable”.

![Diagram of blocking and preemption overheads](image)

Figure 2.6: Controlling blocking and preemption overheads using limited preemptive scheduling.

**Example 7.** Consider the same two tasks $\tau_i$ and $\tau_j$ used in Examples 2 and 3 for preemptive (Section 2.4) and non-preemptive scheduling (section 2.5) respectively. By deferring the preemption on $\tau_j$ to a more favourable location with respect to preemption overheads, while ensuring affordable blocking on the higher priority task $\tau_i$, it is possible to guarantee the timeliness of both the tasks. This is illustrated in Figure 2.6.

In other words, by appropriately provisioning the non-preemptive regions (e.g., using appropriate placement of preemption points), taskset schedulability can be guaranteed even when both preemptive and non-preemptive scheduling algorithms fail to find a valid schedule due to preemptions overheads and blocking.
Chapter 2. Background

2.7 Blocking and Preemption Overheads in Limited Preemptive Scheduling

Limited preemptive scheduling is an attractive solution to obtain the best of both preemptive and non-preemptive scheduling. In the following example, we illustrate how limited preemptive scheduling can be exploited to control preemption related overheads while keeping the blocking "affordable".

Example 7. Consider the same two tasks $\tau_i$ and $\tau_j$ used in Examples 2 and 3 for preemptive (Section 2.4) and non-preemptive scheduling (section 2.5) respectively. By deferring the preemption on $\tau_j$ to a more favourable location with respect to preemption overheads, while ensuring affordable blocking on the higher priority task $\tau_i$, it is possible to guarantee the timeliness of both the tasks. This is illustrated in Figure 2.6.

In other words, by appropriately provisioning the non-preemptive regions (e.g., using appropriate placement of preemption points), taskset schedulability can be guaranteed even when both preemptive and non-preemptive scheduling algorithms fail to find a valid schedule due to preemptions overheads and blocking.

2.7.1 Push Through Blocking

When limited preemptively scheduling real-time tasks, the final NPRs of some jobs of the tasks may *push through* some higher priority interference to its own next job, thereby delaying the completion of the next job. This higher priority interference pushed through by a previous job of the same task is referred to as *push through blocking*. Bril et al. [12][61][62][63] were the first to identify this phenomenon and apply it, *e.g.*, in the context of Controller Area Networks (CAN) [62], consequently correcting the previous analysis. In the following, we revisit the example given by Davis *et al.* [62] that illustrates *push through blocking*.

<table>
<thead>
<tr>
<th>Message</th>
<th>Priority</th>
<th>Transmission Time</th>
<th>Deadline</th>
<th>Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>High</td>
<td>1 ms</td>
<td>2.5 ms</td>
<td>2.5 ms</td>
</tr>
<tr>
<td>B</td>
<td>Medium</td>
<td>1 ms</td>
<td>3.25 ms</td>
<td>3.5 ms</td>
</tr>
<tr>
<td>C</td>
<td>Low</td>
<td>1 ms</td>
<td>3.25 ms</td>
<td>3.5 ms</td>
</tr>
</tbody>
</table>

Table 2.1: Example message set

![Figure 2.7: Example schedule illustrating push through blocking.](image)

Example 8. Consider a set of CAN messages given in Table 2.1 that is scheduled on the network. Suppose all messages are released together
at time $t = 0$, then the messages are transmitted in priority order, i.e., $A$ followed by $B$, followed by $C$. However, when the second instance of $A$, i.e., $A_2$ is released, as shown in Figure 2.7, it is blocked by the first instance of $C$ (i.e., $C_1$ pushes $A_2$). As a result, $A_2$ begins its transmission a bit late and consequently pushes $B_2$, which is the second instance of message $B$, thereby delaying its completion. At time $t = 5$, when $B_2$ finishes its transmission, since the third instance of $A$ is released, $C_2$ still cannot start its transmission. Finally, $C_2$ can begin transmission at $t = 6$, and hence misses its deadline. Note that in this case, the deadline miss on $C_2$ was due to the interference pushed through by its first instance $C_1$. This interference that is pushed through by the previous instance of the same message (or task in case of scheduling on processors) is referred to as push through blocking.

As shown by Davis et al. [15], the effect of push through blocking is not only limited to single processor systems, but is also significant in the context of multiprocessor systems. Push through blocking needs to be explicitly accounted for in some cases, depending on how the higher priority interference is calculated in the schedulability analysis. For example, pessimistically accounting for higher priority interference, e.g., as done by [64], automatically factors in the effect of push through blocking, while more precise analyses, e.g., [30], do not. We discuss this in detail in Chapter 4.

2.8 Managing Preemptions in Global Limited Preemptive Scheduling

The infeasibility of obtaining higher computing power on uniprocessors, without prohibitive increase in power and thermal constraints, has triggered a multicore revolution [65]. While limited preemptive scheduling is quite straightforward in uniprocessor systems, and there exists many research results for analyzing limited preemptively scheduled real-time tasks on uniprocessors (see [10] for a survey), similar advances for multiprocessor systems are still in their
infancy. Moreover, global limited preemptive scheduling on multiprocessor systems, especially when using FPPs, brings in an additional challenge with respect to determining which of the lower priority tasks to preempt. Two principal choices with respect to the preemption approach exists:

- The scheduler waits for the lowest priority job to become preemptable (i.e., a lazy approach)
- The scheduler preempts the first job, among the lower priority ones, that becomes preemptable (i.e., an eager approach).

In the following, these two approaches to preemptions are illustrated by using examples.

### 2.8.1 The Lazy Preemption Approach

When scheduling real-time tasks on an $m$ processor platform using global limited preemptive scheduling with the Lazy Preemption Approach (LPA), the first $m$ tasks having the highest priorities are scheduled. When a high priority task is released and the lower priority tasks are executing their NPRs, the scheduling decision is deferred until the lowest priority task finishes executing its NPR. In the following, we present an example illustrating lazy preemptions.

**Example 9.** Consider the scenario in Figure 2.8 where 4 tasks $\tau_1$, $\tau_2$, $\tau_3$ and $\tau_4$ (in decreasing priority order), with fixed preemption points, are scheduled on two processors. Assume that tasks $\tau_3$ and $\tau_4$ are currently executing on the processor at time instant $t_1$. Suppose that $\tau_1$ and $\tau_2$ are released together at time instant $t_1$, and the scheduling policy uses LPA. In this case, $\tau_1$ starts executing only at time $t_4$ even though a lower priority task ($\tau_3$) was available to be preempted earlier (at $t_2$). Moreover, task $\tau_2$ is blocked 3 times by $\tau_3$ since $\tau_4$ has to be preempted first. At time $t_4$, when the scheduler is invoked, $\tau_4$ is preempted by $\tau_1$. However, since task $\tau_3$ is not preemptable at $t_4$, $\tau_2$ still cannot start executing. The total number of such priority
34 Chapter 2. Background

![Figure 2.8: Example schedule illustrating lazy preemptions.]

inversions could be arbitrarily large if \( \tau_4 \) has a large non-preemptive region.

### 2.8.2 The Eager Preemption Approach

When scheduling real-time tasks on an \( m \) processor platform using global limited preemptive scheduling with the *eager preemption approach* (EPA), like in the case of LPA, the first \( m \) tasks having the *highest priorities* are scheduled. However, unlike in the lazy approach, when a high priority task is released and the lower priority tasks are executing their NPRs, the scheduling decision is deferred until any one of the lower priority tasks finishes executing its NPR (and not necessarily the lowest priority task).

In the following, we illustrate how G-LP-FPS with eager preemptions schedules the taskset presented in Example 9.

**Example 10.** Consider the same scenario presented in Figure 2.8 where \( \tau_1 \) and \( \tau_2 \) are released at time instant \( t_1 \) and are blocked due to the
2.8 Managing Preemptions in Global Limited Preemptive Scheduling

non-preemptive regions of $\tau_3$ and $\tau_4$. If the scheduler implements EPA, when $\tau_3$ finishes executing its non-preemptive region, $\tau_1$ preempts it and starts executing on processor 1 even though $\tau_3$ is not the lowest priority executing task. When $\tau_4$ finishes executing its non-preemptive region, $\tau_2$ is scheduled on processor 2 and can start its execution, as shown in Figure 2.9.

Example 9 showed that, with LPA, higher priority tasks cannot be scheduled even if one or more lower priority tasks finish executing their NPRs, unless they have the lowest priority. EPA enables the medium priority task $\tau_2$ to be allocated to the processor earlier than in the case with lazy preemptions, as illustrated in Example 10. Unlike LPA, as seen from Example 10, with EPA, tasks may suffer from priority inversions even after they start executing on the processor. This is because, under EPA, a task can be preempted if it finishes executing its NPR, while lower priority are executing (their NPRs) on the other processors (essentially blocking it).
Resource augmentation analysis was first introduced by Kalyanasundaram and Pruhs [26] as an alternative to competitive analysis, which was then a widely used method to compare the performance of (scheduling) algorithms. In resource augmentation, the scheduler under study is given extra resources, the most common being processor speed, and its performance with respect to an optimal, all powerful adversary is investigated. The extra resources required to achieve optimality should be reasonably bounded for the algorithm to be useful in practice. In other words, resource augmentation analysis investigates the bound on the minimum increase in resources required to achieve optimality.

Resource augmentation provides insights into the parameters that affect the satisfaction of the specified goal by the algorithm, giving greater flexibility to the system designer while designing the system. Put differently, resource augmentation can also be used to determine the performance of scheduling algorithms for tasksets with a certain pathological structure. For example, in the case of liquid tasks proposed by Abdelzaher et al. [66] that have very small execution times compared to their deadlines, non-preemptive scheduling becomes very favourable with only a minor increase in processor speed or a minor relaxation of deadlines and time periods [67] [68].

As observed by Kalyanasundaram and Pruhs [26] in their seminal paper:

“Resource augmentation analysis can suggest practical ways to improve performance besides buying more or better resources. For example, in the best-effort effort real-time scheduling problem, our resource augmentation analysis suggest that the online scheduler can perform well if the laxity of every job is at least a constant fraction of its length.” [26]

Moreover, resource augmentation provides an interesting method for
comparing the performance of online scheduling algorithms, particularly when the scheduler does not have all the information regarding job arrivals. As observed by Phillips et al. [69]:

“We suggest, though, that the ultimate decision of whether this sort of analysis is meaningful will depend on the sorts of algorithms the analysis recommends and whether or not it yields interesting and new distinctions between algorithms. When evaluated from this Perspective, our results provide powerful evidence that “extra-resource” analysis is a useful tool in the analysis of online scheduling problems.” [69]

In this thesis, we derive resource augmentation bounds that guarantee the feasibility of a specified limited preemptive behavior, for any given real-time taskset, on both uniprocessor and multiprocessor platforms.

2.10 Chapter Summary

In this chapter, we introduced real-time systems, and some basic mechanisms that enable predictable behavior of the real-time tasks. We discussed the challenges associated with preemptive and non-preemptive real-time scheduling, particularly with respect to the associated overheads. We illustrated how limiting preemptions in the schedule can enable exploitation of the best-of both preemptive and non-preemptive paradigms, as well as the challenges with respect to accounting for blocking in limited preemptively scheduled systems. Finally, we introduced resource augmentation, which is a widely used methodology to quantitatively describe the performance of scheduling algorithms in terms of the additional resources required to achieve optimality.

As discussed earlier in Chapter 1, limited preemptive scheduling theory in the context of multiprocessors is still in its infancy. This thesis makes significant contributions in the area of global limited preemptive scheduling on multiprocessors under EDF and FPS, which are two of the, what may be called, major scheduling “philosophies”.
The thesis considers different mechanisms to enable preemptions, viz. floating NPRs and fixed preemption points. Moreover, the thesis considers different approaches to preemption viz. EPA and LPA. Lastly, the worst case performance of limited preemptive scheduling is quantified using the widely used notion of resource augmentation, enabling trade-offs between resources and limited preemptive feasibility.
Chapter 3

Global Limited Preemptive EDF

It is nice to know the dictionary definition for the adjective “elegant” in the meaning “simple and surprisingly effective”.  

Earliest Deadline First (EDF) scheduling is a major paradigm for scheduling real-time tasks in a timely manner, and is one of the most studied paradigms in the real-time systems community [1] [47]. In this chapter, we explore the possibility of limiting preemptions under global EDF without compromising schedulability. We propose a schedulability analysis technique for Global Limited Preemptive Earliest Deadline First (G-LP-EDF) scheduling of sporadic real-time tasks under the lazy preemption approach. We assume the floating non-preemptive regions paradigm in which the non-preemptive regions are assumed to be floating in the task execution, and is triggered by the release of a higher priority task.

\footnote{This quote is from Edsger Dijkstra’s manuscript number 896, and is used here to highlight the simplicity and effectiveness of deadline based scheduling in real-time systems.}
In particular, in this chapter, we derive a sufficient condition which guarantees that every task in the taskset can execute non-preemptively for a known upper-bounded length, whenever a high priority task is released. This test is then trivially extended to determine the global Non-Preemptive EDF (G-NP-EDF) feasibility of sporadic real-time tasks.

3.1 Models and Notations

In this section, we introduce the notations used in this chapter, including the task and processor model and the scheduling model.

3.1.1 Task and Processor Model

We consider a set of $n$ sporadic real-time tasks $\Gamma = \{\tau_1, \tau_2, ... \tau_n\}$ executing on $m$ identical processors. Each $\tau_i$ is characterized by a minimum inter-arrival time $T_i$, a worst case execution requirement $C_i$, and a relative deadline $D_i \leq T_i$. The tasks are ordered according to their increasing deadlines, i.e., $D_i \leq D_{i+1}$, $1 \leq i < n$, and $D_{\min}$ is used to denote $D_1$. Similarly, the largest execution time is denoted by $C_{\max} = \max \forall \tau_i \in \Gamma C_i$. Without loss of generality, we assume that the default speed of all processors is $s = 1$. In common with [26][23][22][21][69][44], we make the simplifying assumption that task execution times scale linearly with the processor speed. In other words we assume that when a processor that is two times faster is used, the worst case execution time becomes $\frac{C_i}{2}$, $\forall \tau_i \in \Gamma$. The model also allows us to use the terms ‘processor speed-up factor’ and ‘processor speed’ interchangeably. Changing the processor speed from $s = 1$ to $s = a$, is equivalent to speeding up the processor by a factor ‘$a$’.

We assume that every task $\tau_i$ can execute non-preemptively for a duration given by $L = \max \forall \tau_i \in \Gamma (L_i)$. Each of these tasks in $\Gamma$ generates a sequence of jobs $J$ where a job in $J$ is represented by $J_k$. The density
of a task $\tau_i$ is defined as $\delta_i = \frac{C_i}{D_i}$ and the maximum density is defined as

$$\delta_{max} = \max_{\forall \tau_i \in \Gamma} \left\{ \frac{C_i}{D_i} \right\}$$

We also define

$$\hat{\delta}_{max} = \max_{\forall \tau_i \in \Gamma} \left\{ \frac{C_i}{D_i - L} \right\}$$

### 3.1.2 G-LP-EDF with Lazy Preemptions Model

We assume a deadline based scheduler: in any time interval $[t_a, t_f)$, first $m$ jobs having the earliest deadlines are assigned to the $m$ processors, with ties broken arbitrarily. Whenever a higher priority job $J_i$ is released and all $m$ processors are busy, with at least one processor executing a lower priority job, all the lower priority jobs begin executing non-preemptively for at most $L$ time units. After at most $L$ time units, the scheduler reschedules the entire set of tasks. In other words, $J_i$ preempts the lowest priority executing job after getting blocked for at most $L$ time units, or is allocated to the first processor that becomes idle. This model is the floating Non-Preemptive Region (f-NPR) scheduling model for multiprocessor systems, under global scheduling.

### 3.1.3 Definitions

The demand of a sequence of jobs $J$ over a time interval of length $t$ is defined as the cumulative execution time of all the jobs in $J$ scheduled in that interval. The minimum demand of a given sequence of jobs generated over an interval of length $t$ is defined as the minimum amount of execution that the sequence of jobs could require within $t$ in order to meet all its deadlines. This concept has been extended to sporadic task systems, where a task $\tau_i$’s maxmin demand over an interval of length $t$ is defined as the largest minimum demand by any sequence of jobs that could be legally generated by $\tau_i$ in $t$ [44].

Baruah et al. [44] introduced the Forced Forward Demand Bound Function (FF-DBF) that generalized the above concepts on a set of
speed-σ processors (see an illustration in Figure 3.1). The FF-DBF of any task \( \tau_i \) over a time interval of length \( t \) is defined as [44]:

\[
FF-DBF(\tau_i, t, \sigma) = q_i C_i + \begin{cases} 
C_i & \text{if } r_i \geq D_i \\
C_i - (D_i - r_i)\sigma & \text{if } D_i > r_i \geq D_i - \frac{C_i}{\sigma} \\
0 & \text{otherwise}
\end{cases}
\]

where, \( \sigma \) is a positive real-number and,

\[
q_i \overset{\text{def}}{=} \left\lfloor \frac{t}{T_i} \right\rfloor
\]

and

\[
r_i \overset{\text{def}}{=} t \mod T_i
\]

The FF-DBF of the taskset, denoted by \( FF-DBF(\Gamma, t, \sigma) \) is given by,

\[
FF-DBF(\Gamma, t, \sigma) = \sum_{\forall \tau_i} FF-DBF(\tau_i, t, \sigma)
\]

Consequently, the \( FF-DBF(\tau_i, t, \sigma) \) can be seen as the \textit{maxmin} demand of \( \tau_i \) over an interval of length \( t \), where the execution outside the interval occurs on a speed-σ processor.
Figure 3.2: Lower bound on the work done in any time interval $[t_i, t_{i-1})$.

3.2 Schedulability Analysis for G-LP-EDF with Lazy Preemptions

Phillips et al. [69] observed that in any time interval $[t_i, t_{i-1})$ as shown in Figure 3.2, if a job $J_i$ executes for a duration $x_i$, then the total work done in that interval is at least $m(t_{i-1} - t_i - x_i) + x_i$ (as shown by the dotted box in Figure 3.2). This observation forms the basis of a large portion of subsequent research into the schedulability analysis of global scheduling algorithms. In [44], Baruah et al. build on the observation that a deadline miss can only follow a time interval during which a significant amount of job executions have occurred, and derived a condition for which a minimal set of real-time jobs scheduled by a work conserving scheduling algorithm misses a deadline. They then derived the workload generated by G-P-EDF for the same set of jobs. It follows that, if the workload generated by G-P-EDF does not satisfy the derived condition for deadline miss, then the taskset is schedulable by G-P-EDF. We follow a method similar to the one used by Baruah et al. [44].

Let $A$ denote a work conserving limited preemptive algorithm that misses a deadline while scheduling some legal collection of jobs
generated by the taskset $\Gamma$. We derive a condition for this to be true by examining $A$’s behavior on some minimal legal collection of jobs generated by $\Gamma$ on which it misses a deadline. Let $t_0$ denote the time instant at which a deadline miss occurred due to the limited preemptive execution of some job, and let $J_1$ be the job that missed the deadline. The arrival time of $J_1$ is denoted as $t_1$. Let $s$ denote a constant $s = \hat{\delta}_{\text{max}} \leq s \leq 1$. Since $J_1$ misses a deadline due to a blocking of duration no greater than $L$, it must be the case that $J_1$ has executed for strictly less than $(t_0 - t_1 - L) \times s$.

Consider a sequence $J'$ of jobs $J_i$, time instants $t_i$, and an index $q$ according to the pseudo-code presented below (see Figure 3.3 for an illustration).

**Algorithm 1:** Pseudo-code for job selection.

1. for $i \leftarrow 2, 3, \ldots$ do
2.   Let $J_i$ denote a job that
3.   - arrives at some time instant $t_i < t_{i-1}$;
4.   - has a deadline after $t_{i-1}$;
5.   - has not completed execution by $t_{i-1}$;
6.   - has executed for strictly less than $(t_{i-1} - t_i) \times s$ time units over the interval $[t_i, t_{i-1})$;
7. if there is no such job then
8.   $q = (i - 1)$;
9.   break out of for loop;

We define a subset $J \subset J'$ comprising of $k$ jobs: Let $J_k$ represent the last job among the jobs in $J'$ that is blocked by a lower priority job—so that we can analyze the effect of blocking on the deadline miss at $t_0$. Then the subset $J$ is composed of the jobs $J_i, i = k, \ldots, 1$. We then calculate the total executions that occur in the schedule over the interval $[t_k, t_0)$, that lead to a deadline miss at $t_0$.

Let $x_i$ denote the total length of the time intervals in $[t_i, t_{i-1})$ during
45 Chapter 3. Global Limited Preemptive EDF

generated by the taskset $\Gamma$. We derive a condition for this to be true by examining $A$'s behavior on some minimal legal collection of jobs generated by $\Gamma$ on which it misses a deadline. Let $t_0$ denote the time instant at which a deadline miss occurred due to the limited preemptive execution of some job, and let $J_1$ be the job that missed the deadline. The arrival time of $J_1$ is denoted as $t_1$. Let $s$ denote a constant satisfying $\hat{\delta}_{\text{max}} \leq s \leq 1$. Since $J_1$ misses a deadline due to a blocking of duration no greater than $L$, it must be the case that $J_1$ has executed for strictly less than $(t_0 - t_1 - L) \times s$.

Consider a sequence $J_i'$ of jobs $J_i$, time instants $t_i$, and an index $q$ according to the pseudo-code presented below (see Figure 3.3 for an illustration).

**Algorithm 1:** Pseudo-code for job selection.

1. for $i \leftarrow 2, 3, \ldots$
2. Let $J_i$ denote a job that -arrives at some time instant $t_i < t_i - 1$;
3. -has a deadline after $t_i - 1$;
4. -has not completed execution by $t_i - 1$;
5. -has executed for strictly less than $(t_i - 1 - t_i) \times s$ time units over the interval $[t_i, t_i - 1)$;
6. if there is no such job then
7. $q = (i - 1)$;
8. break out of for loop;

We define a subset $J \subset J_i'$ comprising of $k$ jobs: Let $J_k$ represent the last job among the jobs in $J_i'$ that is blocked by a lower priority job—so that we can analyze the effect of blocking on the deadline miss at $t_0$.

Then the subset $J$ is composed of the jobs $J_i$, $i = k, \ldots, 1$. We then calculate the total executions that occur in the schedule over the interval $[t_k, t_0)$, that lead to a deadline miss at $t_0$.

Let $x_i$ denote the total length of the time intervals in $[t_i, t_i - 1)$ during $[t_k, t_0)$. We define a subset $J \subset J_i'$ comprising of $k$ jobs: Let $J_k$ represent the last job among the jobs in $J_i'$ that is blocked by a lower priority job—so that we can analyze the effect of blocking on the deadline miss at $t_0$.

Then the subset $J$ is composed of the jobs $J_i$, $i = k, \ldots, 1$. We then calculate the total executions that occur in the schedule over the interval $[t_k, t_0)$, that lead to a deadline miss at $t_0$.

Let $x_i$ denote the total length of the time intervals in $[t_i, t_i - 1)$ during $[t_k, t_0)$. We define a subset $J \subset J_i'$ comprising of $k$ jobs: Let $J_k$ represent the last job among the jobs in $J_i'$ that is blocked by a lower priority job—so that we can analyze the effect of blocking on the deadline miss at $t_0$.

Then the subset $J$ is composed of the jobs $J_i$, $i = k, \ldots, 1$. We then calculate the total executions that occur in the schedule over the interval $[t_k, t_0)$, that lead to a deadline miss at $t_0$.

Let $x_i$ denote the total length of the time intervals in $[t_i, t_i - 1)$ during $[t_k, t_0)$. We define a subset $J \subset J_i'$ comprising of $k$ jobs: Let $J_k$ represent the last job among the jobs in $J_i'$ that is blocked by a lower priority job—so that we can analyze the effect of blocking on the deadline miss at $t_0$.

Then the subset $J$ is composed of the jobs $J_i$, $i = k, \ldots, 1$. We then calculate the total executions that occur in the schedule over the interval $[t_k, t_0)$, that lead to a deadline miss at $t_0$.

Let $x_i$ denote the total length of the time intervals in $[t_i, t_i - 1)$ during $[t_k, t_0)$. We define a subset $J \subset J_i'$ comprising of $k$ jobs: Let $J_k$ represent the last job among the jobs in $J_i'$ that is blocked by a lower priority job—so that we can analyze the effect of blocking on the deadline miss at $t_0$.

Then the subset $J$ is composed of the jobs $J_i$, $i = k, \ldots, 1$. We then calculate the total executions that occur in the schedule over the interval $[t_k, t_0)$, that lead to a deadline miss at $t_0$.

Observation 3.2.1.

$$\sum_{i=1}^{k} x_i < (t_0 - t_k - L) \times s$$

**Proof.** According to our assumption, for each job $J_i$, $2 \leq i \leq k$,

$$x_i < (t_{i-1} - t_i) \times s$$

Summing up over all $2 \leq i \leq k$, we get:

$$\sum_{i=2}^{k} x_i < (t_1 - t_k) \times s$$

Remember that $J_1$ has executed for

$$x_1 < (t_0 - t_1 - L) \times s$$

Figure 3.3: An illustration of the jobs given by algorithm 1.
Therefore,

\[ \sum_{i=2}^{k} x_i + x_1 < (t_1 - t_k) \times s + (t_0 - t_1 - L) \times s \]

Hence,

\[ \sum_{i=1}^{k} x_i < (t_0 - t_k - L) \times s \]

Let \( W \) denote the amount of execution that occurs in the schedule over the interval \([t_k, t_0]\). In \([t_k, t_0]\) lower priority job executions may interfere with the jobs in \( J \) leading to a deadline miss at \( t_0 \)— we refer to these executions as the blocking executions. Let \( \hat{W} \) denote the amount of execution that occurs in the schedule over the interval \([t_k, t_0]\), excluding the blocking executions that interfere with \( J \). Similarly, let \( W_i \) denote the amount of execution that occurs over the interval \([t_i, t_{i-1}]\), excluding the blocking executions that interfere with \( J \), and hence

\[ \hat{W} = \sum_{i=1}^{k} W_i \]

Finally, let \( Y_i \) denote the amount of blocking executions during the interval \([t_i, t_{i-1}]\), that interfere with \( J \), leading to a deadline miss at \( t_0 \).

In the following, we derive a lower bound on \( W \), which is the sum of \( \hat{W} \) and \( \sum_{i=1}^{k} Y_i \).

**Observation 3.2.2.**

\[ W > (m - (m - 1) \times s) (t_0 - t_k - L) + mL \]

**Proof.** We have assumed that \( J_i \) has not completed its execution by time instant \( t_{i-1} \). Over \([t_i, t_{i-1}]\), all \( m \) processors must be executing i) \( J_i \), ii) jobs having higher priority than job \( J_i \) or iii) blocking executions from
3.2 Schedulability Analysis for G-LP-EDF with Lazy Preemptions

lower priority tasks. Therefore, whenever $J_i$ or blocking executions are not using the processors, all $m$ processors must be executing higher priority jobs (see Figure 3.2 for an illustration).

The lower bound for $W_i$, $2 \leq i \leq k$, is given by:

$$W_i \geq m(t_{i-1} - t_i - x_i) + x_i - Y_i$$

Therefore, the duration for which higher priority jobs execute in $[t_k, t_0)$ is obtained by summing up all $W_i$s:

$$\hat{W} \geq m(t_0 - t_k) - (m - 1) \sum_{i=1}^{k} x_i - \sum_{i=1}^{k} Y_i$$

Adding and subtracting $mL$ to the right hand side of the equation, we get:

$$\hat{W} \geq m(t_0 - t_k) - mL + mL - (m - 1) \sum_{i=1}^{k} x_i - \sum_{i=1}^{k} Y_i$$

Which gives:

$$\hat{W} \geq m(t_0 - t_k - L) - (m - 1) \sum_{i=1}^{k} x_i + mL - \sum_{i=1}^{k} Y_i$$

Substituting for $\sum_{i=1}^{k} x_i$ from Observation 3.2.1, we obtain:

$$\hat{W} > m(t_0 - t_k - L) - (m - 1)(t_0 - t_k - L) \times s + mL - \sum_{i=1}^{k} Y_i$$

Which gives

$$\hat{W} > (m - (m - 1) \times s)(t_0 - t_k - L) + mL - \sum_{i=1}^{k} Y_i$$
We get $W$ by adding $\hat{W}$ and $\sum_{i=1}^{k} Y_i$:

$$W > (m - (m - 1) \times s) (t_0 - t_k - L) + mL - \sum_{i=1}^{k} Y_i + \sum_{i=1}^{k} Y_i$$

$$\Rightarrow W > (m - (m - 1) \times s) (t_0 - t_k - L) + mL$$

$\square$

In the following, we derive an upper-bound on $Y_i$ that denotes the amount of blocking executions happening during the interval $[t_i, t_{i-1})$ interfering with job $J_1$, leading to its deadline miss.

**Observation 3.2.3.**

$$\sum_{i=1}^{k} Y_i \leq mL$$

**Proof.** According to our assumption $J_k$ is the only job in $J$ that is blocked. This means that when job $J_k$ is released (at time instant $t_k$), $p \geq 1$ processors are executing lower priority jobs. Therefore, the total blocking executions happening in the interval $[t_k, t_0)$, over all $m$ processors, is $pL$.

When all $m$ processors are executing lower priority jobs at the release time of $J_k$, all these jobs execute non-preemptively for a duration $L$, and no more blocking executions occur after this (remember our assumption that $J_k$ is the only job in $J$ that is blocked). Therefore, the maximum blocking executions happening in the interval $[t_k, t_0)$ over all $m$ processors is at most $mL$.  

We now find an upper-bound on $W$ in the following observation.

**Observation 3.2.4.** If the work-conserving algorithm is G-LP-EDF, then,

$$W \leq FF-DBF(\tau, (t_0 - t_k), s) + mL$$
3.2 Schedulability Analysis for G-LP-EDF with Lazy Preemptions

Proof. We are analyzing a minimal unschedulable collection [44] of jobs \( J \), where a deadline miss occurred due to a limited preemptive execution of at most \( L \) time units. Since the algorithm is G-LP-EDF, and the jobs in \( J \) do not block each other, they execute according to their deadlines. All the jobs in \( J \) (i.e., the jobs that are released in \( [t_k, t_0) \)) have their deadlines within the interval \( [t_k, t_0) \).

The amount of execution that jobs of any task \( \tau_i \) contribute to \( W \) is bounded from above by the scenario in which a job of \( \tau_i \) has its deadline at the end of the interval \( t_0 \), and the prior jobs arrive as early as possible. In this scenario, the jobs of \( \tau_i \) that contribute to \( W \) include:

1. at least \( q_i \) \( \text{def} = \left\lfloor \frac{(t_0 - t_k)}{T_i} \right\rfloor \) jobs of \( \tau_i \) with releases and deadlines entirely in \( [t_k, t_0) \), and

2. (perhaps) an additional job that has its deadline at time instant \( t_k + r_i \), where \( r_i \) \( \text{def} = (t_0 - t_k) \mod T_i \)

In the case 2 above, there are two sub-cases:

2.a \( r_i \geq D_i \): In this case, the additional job with deadline at \( t_k + r_i \) arrives at or after \( t_k \) and hence its contribution is \( C_i \).

2.b \( r_i < D_i \): In this case, the additional job with deadline at \( t_k + r_i \) arrives prior to \( t_k \). Hence this job should have completed at least \((D_i - r_i) \times s\) units of execution prior to time instant \( t_k \). Therefore, the contribution of this job to \( W \) is at most \( \max(0, (D_i - r_i) \times s) \).

From 1 and 2 above, we can conclude that the total contribution of \( \tau_i \) to \( W \) is upper-bounded by FF-DBF\((\tau_i, t_0 - t_k, s)\). Summing up the total contributions of all such \( \tau_i \in \Gamma \), we get,

\[
\sum_{\forall \tau_i \in \Gamma} \text{FF-DBF}(\tau_i, (t_0 - t_k), s) = \text{FF-DBF}(\Gamma, (t_0 - t_k), s)
\]

Finally, according to observation 3.2.3, the total duration for which blocking executions execute in \( [t_k, t_0) \) is upper-bounded by \( mL \). Hence,

\[
W \leq \text{FF-DBF}(\Gamma, (t_0 - t_k), s) + mL
\]
Observation 3.2.5. Suppose that a constrained deadline sporadic task system $\Gamma$ is not G-LP-EDF feasible upon $m$ unit-speed processors. For each $s, s \geq \hat{\delta}_{\text{max}}$, there is an interval of length $t$ such that

$$FF-DBF(\Gamma, t, s) > (m - (m - 1)s)(t - L)$$

Proof. The proof follows by chaining the lower bound on $W$ of Observation 3.2.2 with the upper-bound of Observation 3.2.4, where $t = (t_0 - t_k)$.

The contraposition of the Observation 3.2.5 above represents a global limited preemptive EDF schedulability condition.

Theorem 3.2.1. A taskset $\Gamma$, where every $\tau_i \in \Gamma$ executes non-preemptively for a duration of at most $L_i$ time units, is G-LP-EDF feasible if $\exists \sigma: \sigma \geq \hat{\delta}_{\text{max}}$ such that $\forall t \geq 0$,

$$FF-DBF(\Gamma, t, \sigma) \leq (m - (m - 1)\sigma)(t - L)$$

where $L = \max_{\forall \tau_i \in \Gamma}(L_i)$

Proof. The proof follows from the fact that we take the contrapositive of Observation 3.2.5.

The intuition behind the above condition is that, in any time interval of length $t$, the total amount of execution that has to necessarily happen in that interval must be no greater than $t - L$, so that even if the tasks are blocked (for a duration $\leq L$), the tasks complete their execution no later than $t$. Instantiating the above theorem in the context of a fully non-preemptive scheduler, we get the following result for g-NP-EDF scheduling.

Corollary 3.2.1. A taskset $\Gamma$ is G-NP-EDF feasible if $\exists \sigma: \sigma \geq \hat{\delta}_{\text{max}}$ such that $\forall t \geq 0$

$$FF-DBF(\Gamma, t, \sigma) \leq (m - (m - 1)\sigma)(t - C_{\text{max}})$$

where $C_{\text{max}} = \max_{\forall \tau_i \in \Gamma}(C_i)$
We have thus obtained sufficient conditions for global limited preemptive and non-preemptive EDF scheduling of sporadic real-time tasks. We can now use an algorithm similar to the one presented in [44] to determine the G-LP-EDF schedulability of a given taskset.

### 3.3 Chapter Summary

In this chapter, we derived a sufficient condition for G-LP-EDF feasibility of sporadic real-time tasks. Our test builds on the well known G-P-EDF feasibility test presented by Baruah et al. [44]. If a taskset is deemed G-LP-EDF feasible by our method, then every task in the taskset has a slack of length $L$. In the context of a limited preemptive scheduler, this slack of length $L$ enables the limited preemptive execution of lower priority tasks for a duration of at most $L$. Note that, we consider the floating NPR model in which the re-scheduling decision is deferred by at most a duration $L$, whenever a higher priority task is released and at least one processor is executing a lower priority task. Such a type of scheduler can be useful in the context of a best effort preemption overhead reduction strategy. That is, the scheduler defers the re-scheduling decision to a later point in time where a preemption is more favourable, and in the worst case enables preemptions after at most $L$ time units.

**Note:** This chapter is based on the paper *The Global Limited Preemptive Earliest Deadline First Feasibility of Sporadic Real-time Tasks*, Abhilash Thekkilakattil, Sanjoy Baruah, Radu Dobrin and Sasikumar Punnekkat, The 26th Euromicro Conference on Real-time Systems, IEEE, Madrid, Spain, July, 2014.

---

2 Many thanks goes to Björn Brandenburg for providing insights into the applicability of floating NPR schedulers.
Fixed Priority Scheduling (FPS) is again one of the major paradigms for scheduling real-time tasks, along with EDF, that is widely studied. In this chapter, we address the problem of Global Limited Preemptive Fixed Priority Scheduling (G-LP-FPS) of tasks with fixed preemption points, and evaluate how the choice of either an eager or a lazy approach to preemption affects schedulability. We introduce a response time based schedulability test for real-time tasksets with fixed preemption points scheduled using G-LP-FPS with eager preemptions. To the best of our knowledge, this is the first such test in the context of G-LP-FPS with fixed preemption points. We then present our empirical evaluations comparing G-LP-FPS with eager preemptions (using the new schedulability test), and G-LP-FPS with lazy preemptions (using the test for link-based scheduling). This quote from the coat of arms of the Danish Order of the Elephant given to Neils Bohr means "Opposites are complementary," and is used here to highlight FPS's position as a complement to EDF.
Chapter 4

Global Limited Preemptive FPS

Contraria Sunt Complementa.\footnote{This quote from the coat of arms of the Danish Order of the Elephant given to Neils Bohr means “Opposites are complementary”, and is used here to highlight FPS’s position as a complement to EDF.}

Fixed Priority Scheduling (FPS) is again one of the major paradigms for scheduling real-time tasks, along with EDF, that is widely studied \cite{1} \cite{47}. In this chapter, we address the problem of Global Limited Preemptive Fixed Priority Scheduling (G-LP-FPS) of tasks with fixed preemption points, and evaluate how the choice of either an eager or a lazy approach to preemption affects schedulability. We introduce a response time based schedulability test for real-time tasksets with fixed preemption points scheduled using G-LP-FPS with eager preemptions. To the best of our knowledge, this is the first such test in the context of G-LP-FPS with fixed preemption points. We then present our empirical evaluations comparing G-LP-FPS with eager preemptions (using the new schedulability test), and G-LP-FPS with lazy preemptions (using the test for link-based scheduling \cite{18}).
evaluations show that G-LP-FPS with eager preemptions outperforms link-based scheduling (that implements G-LP-FPS with lazy preemptions) for a wide range of settings.

### 4.1 Models and Notations

In this section, we present the system model, terminology and definitions assumed in the rest of the chapter.

#### 4.1.1 Task and Processor Model

We consider a set of $n$ sporadic real-time tasks $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ scheduled on $m$ identical processors. The tasks in $\Gamma$ are indexed according to their decreasing unique priorities i.e., $\tau_1$ has the highest priority and $\tau_n$ the lowest, and each task generates an infinite number of jobs. Let $hp(i)$ denote the subset of tasks with priorities greater than $\tau_i$ and $lp(i)$ denote the subset of tasks with priorities lower than $\tau_i$. Each task $\tau_i$ is characterized by a minimum inter-arrival time $T_i$, and a relative deadline $D_i \leq T_i$, and is assumed to contain $q_i \geq 0$ optimal preemption points [14]. The start of the task execution is referred to as the $0^{th}$ preemption point while the end of the task execution is referred to as the $q_i + 1^{th}$ point; however, preemption of the task is of course not possible at these points.

Let $b_{i,j}$, $j = 1...q_i + 1$ denote the worst case execution time of task $\tau_i$ between its $j - 1^{th}$ and $j^{th}$ preemption points (The calculation of the WCET between preemption points is an important problem on multicores; but this is not our focus. Instead we focus on the related problem of the prohibitive increase of WCETs due to preemption related overheads). We use the notation $b_{i,j}$ to also refer to the corresponding Non-Preemptive Region (NPR). In any time interval of length $t$, each task $\tau_i$ can be preempted by a higher priority task at most $h_i$ times where:

$$h_i = \sum_{\forall \tau_j \in hp(i)} \left\lceil \frac{t}{T_j} \right\rceil$$
Chapter 4. Global Limited Preemptive FPS evaluations show that G-LP-FPS with eager preemptions outperforms link-based scheduling (that implements G-LP-FPS with lazy preemptions) for a wide range of settings.

4.1 Models and Notations

In this section, we present the system model, terminology and definitions assumed in the rest of the chapter.

4.1.1 Task and Processor Model

We consider a set of $n$ sporadic real-time tasks $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ scheduled on $m$ identical processors. The tasks in $\Gamma$ are indexed according to their decreasing unique priorities i.e., $\tau_1$ has the highest priority and $\tau_n$ the lowest, and each task generates an infinite number of jobs. Let $hp(i)$ denote the subset of tasks with priorities greater than $\tau_i$ and $lp(i)$ denote the subset of tasks with priorities lower than $\tau_i$.

Each task $\tau_i$ is characterized by a minimum inter-arrival time $T_i$, and a relative deadline $D_i \leq T_i$, and is assumed to contain $q_i \geq 0$ optimal preemption points [14]. The start of the task execution is referred to as the $0$th preemption point while the end of the task execution is referred to as the $q_i+1$th point; however, preemption of the task is of course not possible at these points.

Let $b_{i,j}$, $j=1 \ldots q_i+1$ denote the worst case execution time of task $\tau_i$ between its $j-1$th and $j$th preemption points (The calculation of the WCET between preemption points is an important problem on multicores; but this is not our focus. Instead we focus on the related problem of the prohibitive increase of WCETs due to preemption related overheads). We use the notation $b_{i,j}$ to also refer to the corresponding Non-Preemptive Region (NPR). In any time interval of length $t$, each task $\tau_i$ can be preempted by a higher priority task at most $h_i$ times where:

$$h_i = \sum_{\forall \tau_j \in hp(i)} \lceil \frac{t}{T_j} \rceil$$

Therefore, an upper-bound on the number of preemptions $p_i$ of task $\tau_i$ in an interval of length $t$ is given by:

$$p_i = \min(q_i, h_i) \quad (4.1)$$

The Worst Case Execution Time (WCET) of each task $\tau_i$ can be calculated as

$$C_i = \sum_{j=1}^{q_i+1} b_{i,j}$$

Note that the preemption related overheads can be integrated into the above equation, since we are interested in comparing lazy and eager preemption mechanisms and these overheads are the same in each case, we omit their specific consideration. Similar to Davis et al. [15], we also define $C_i^*$ for each $\tau_i$, where

$$C_i^* = C_i - b_{i,q_i+1} + 1$$

This is because, when using $C_i^*$ in the calculations instead of $C_i$, the resulting response time implies the completion of $C_i - b_{i,q_i+1} + 1$ units of execution and hence gives the start time of the final NPR (the +1 ensures start of the final NPR).

4.1.2 Global Preemptive Fixed Priority Scheduling

In this section, we review some notations from the state-of-the-art schedulability analysis for global preemptive fixed priority scheduling.

Figure 4.1: Task with carry-in in an interval of length $t$. 

Therefore, an upper-bound on the number of preemptions $p_i$ of task $\tau_i$ in an interval of length $t$ is given by:

$$p_i = \min(q_i, h_i) \quad (4.1)$$

The Worst Case Execution Time (WCET) of each task $\tau_i$ can be calculated as

$$C_i = \sum_{j=1}^{q_i+1} b_{i,j}$$

Note that the preemption related overheads can be integrated into the above equation, since we are interested in comparing lazy and eager preemption mechanisms and these overheads are the same in each case, we omit their specific consideration. Similar to Davis et al. [15], we also define $C_i^*$ for each $\tau_i$, where

$$C_i^* = C_i - b_{i,q_i+1} + 1$$

This is because, when using $C_i^*$ in the calculations instead of $C_i$, the resulting response time implies the completion of $C_i - b_{i,q_i+1} + 1$ units of execution and hence gives the start time of the final NPR (the +1 ensures start of the final NPR).
Bertogna and Cirinei [64] derived an upper-bound on the interference generated by any task $i_j$ on a lower priority task $i_k$ over an interval of length $t$, under Global Preemptive Fixed Priority Scheduling (G-P-FPS) and used this to derive a response time analysis. Even though this analysis has subsequently been improved upon by others, we recall this specific result because of its seminal nature. The upper-bound on the interference generated by any task $i_j$ on a lower priority task $i_k$ over an interval of length $t$ is given by:

$$I_i(t, \tau_j) = \min(I'_i(t, \tau_j), t - C_i + 1) \tag{4.2}$$

where,

$$I'_i(t, \tau_j) = N_j(t)C_j + \min(C_j, t + D_j - C_j - N_j(t)T_j) \tag{4.3}$$

In the above equation, $N_j(t) = \left\lfloor \frac{t + D_j - C_j}{I_j} \right\rfloor$. Consequently, any task $\tau_i$ is schedulable if,

$$R_i = C_i + \left\lfloor \frac{1}{m} \sum_{i_j \in hp(i)} I_i(R_i, \tau_j) \right\rfloor \leq D_i \tag{4.4}$$

Baruah [70] observed that the pessimism in the interference calculation can be attenuated by considering a larger time interval. Baruah also observed that at the time instant earlier than the release of $i_i$ where at least one processor is idle, the number of jobs that have carry-in workload is at most $m - 1$ (see figure 4.1 and 4.2 for an illustration of
workloads with and without carry-in). The response time analysis of Bertogna and Cirinei, was improved by Guan et al. [30] by instantiating these observations in the context of G-P-FPS to limit the carry-in interference. Later Sun et al. [71] identified and fixed some anomalies in the test for the arbitrary deadline case.

In order to determine the schedulability of tasks with fixed preemption points, similar to [15], we define:

\[ I_i(t, \tau_j) = \min(I_i^l(t, \tau_j), t - C_i^* + 1) \]  \hspace{1cm} (4.5)

where \( I_i^l(t, \tau_j) \) is given by (4.3).

### 4.1.3 G-LP-FPS with Eager Preemptions Model

We consider Global Limited Preemptive Fixed Priority Scheduling (G-LP-FPS) with eager preemptions in which higher priority tasks preempt the lower priority task that first finishes executing its Non-Preemptive Region (NPR). In any time interval \([t_a, t_f]\), first \(m\) tasks having the highest priorities are assigned to the \(m\) processors, with ties broken arbitrarily. Whenever a higher priority task \(\tau_i\) is released and all \(m\) processors are busy, with at least one processor executing a lower priority job, and all the lower priority jobs are executing their NPRs, \(\tau_i\) preempts the first lower priority job that finishes executing its NPR. Since after a preemption the task can resume either on the same processor or on a different processor, for convenience, whenever we refer to a preemption we mean preemptions and/or migrations.

### 4.1.4 Definitions

According to our model every task is composed of a set of non-preemptive regions with specified lengths. This allows us to define the preemptability of a task.

**Definition 1.** Any task \(\tau_i \in \Gamma\) is defined to be preemptable at time instant \(t\) if and only if time instant \(t\) corresponds to a preemption point \(k, 1 \leq k < q_i\), in the progress of its execution.
An unfinished task is defined to be ready if it is not currently executing (and hence it is assumed to be in the ready queue). Non-preemptive regions within the lower priority tasks can cause priority inversions on higher priority tasks. We enumerate three conditions that are necessary for a single priority inversion to occur due to an NPR of a lower priority task under limited preemption scheduling.

**Definition 2.** A priority inversion occurs on an arbitrary task $\tau_i$ whenever the scheduler is invoked and the following conditions hold.

1. **C1:** The task $\tau_i$ is ready to execute (but not executing).
2. **C2:** At least one processor is executing a lower priority task.
3. **C3:** All the lower priority jobs are not preemptable.

We differentiate the following types of interference for any task $\tau_i$.

**Definition 3.** The higher priority interference on a task $\tau_i$ is defined as the cumulative executions of all tasks having a higher priority than $\tau_i$ that prevent $\tau_i$ from executing on the processor.

**Definition 4.** The lower priority interference on a task $\tau_i$ is defined as the cumulative executions of all tasks having a lower priority than $\tau_i$ that prevent $\tau_i$ from executing on a processor.

We use $I_i(t, \tau_j)$ to denote the interference on $\tau_i$ from a higher priority task $\tau_j$ in the time interval of length $t$ and $I_i(t)$ to denote the total higher priority interference. Finally, we note that the final NPR of the jobs of any $\tau_i$ delays the start of higher priority tasks released during its execution, which may in turn interfere with the next release of $\tau_i$. This interference pushed through by any job of a task on to its next job is defined as push through blocking (see section 1.4 of [62] for an example).
4.2 Link-based Scheduling for Lazy Preemptions

In this section, we recapitulate the schedulability analysis for Global Limited Preemptive Fixed Priority Scheduling (G-LP-FPS) with lazy preemptions using link-based scheduling as proposed by Block et al. [18]. Davis et al. [15][16] enumerated a number of interesting observations regarding the blocking introduced by the priority inversions that occur due to non-preemptive execution of lower priority tasks:

- The number of priority inversions is not limited to the number of processors $m$, as is the case with Global Non Preemptive Fixed Priority Scheduling (G-NP-FPS).
- More than one job of the same (lower priority) task can cause priority inversion on a higher priority task.
- More than one non-preemptive region of the same job of each lower priority task can cause priority inversions.

**Link based scheduling:** Many of these challenges are addressed by link-based scheduling developed by Block et al. [18] in the context of resource sharing, which is equally applicable to the problem of limited preemptive scheduling with NPRs. Link-based scheduling implements a form of lazy preemptions whereby a newly released high priority task is *linked* to the processor executing the lowest priority task. Even though link-based scheduling was originally developed in the context of NPRs present in otherwise preemptable tasks, it can be used in the context of fixed preemption points scheduling by considering the task to be composed of many such NPRs.

**Link based scheduling analysis:** The analysis for link-based scheduling presented by Block et al. [18] and Brandenburg and Anderson [29] provides a simple generic means of accounting for blocking due to NPRs. It uses an *inflation based* method, in which the WCETs of higher priority tasks are inflated to account for blocking
from lower priority tasks. Specifically, $\forall \tau_i \in \Gamma$, the associated WCET is inflated as follows:

$$C_i = C_i + \max_{\tau_j \in lp(i)} b_{j,k}, \quad k = 1, \ldots, q_i^1$$

Brandenburg and Anderson [29] proved that the resulting taskset $\Gamma'$ obtained by inflating the WCETs of all the tasks in $\Gamma$ with the maximum blocking that they can suffer, is a safe hard real-time approximation [3]. This means that if there is a deadline miss with link-based scheduling and NPRs, then there is also guaranteed to be a deadline miss in $\Gamma'$ under fully preemptive scheduling. Consequently, a fully preemptive schedulability analysis such as that given in [30] can be applied to determine schedulability.

### 4.3 Schedulability Analysis for G-LP-FPS with Eager Preemptions

In this section, we present the main contributions of this chapter. Specifically, we examine G-LP-FPS with eager preemptions in which the highest priority task is allowed to preempt the first lower priority job that becomes preemptable. We show that the number of priority inversions on any task $\tau_i$ is upper-bounded by the associated upper-bound on the number of preemptions $p_i$ (defined in (4.1)). Building on this observation, we derive a response time analysis based test for schedulability under G-LP-FPS with eager preemptions.

First, we derive a bound on the number of priority inversions under G-LP-FPS with eager preemptions.

**Lemma 4.3.1.** The number of priority inversions on any task $\tau_i$ that is inserted into the ready queue under G-LP-FPS with eager preemptions is at most 1 before it can again start executing.

**Proof.** Consider the time instant when $\tau_i$ is inserted into the ready queue and all the processors are executing the NPRs of lower priority tasks. Let $\tau_k$ be the lower priority executing task having the longest duration
to the next preemption point from among the executing tasks. When one of the lower priority tasks reaches its preemption point during its execution, the scheduler is invoked and the next highest priority task from the ready queue is scheduled. When the next preemption point of \( \tau_k \) is reached, either \( \tau_i \) will be (or would have been) scheduled or all processors will be executing tasks having priority higher than \( \tau_i \). No new non-preemptive regions of lower priority tasks can start executing since there are higher priority tasks waiting in the ready queue. Therefore, the number of priority inversions on \( \tau_i \) is at most 1 whenever it is inserted into the ready queue. 

We can upper-bound the number of priority inversions on any task using knowledge of the number of preemption points and an upper-bound on the number of preemptions it can suffer.

**Corollary 4.3.1.** The number of priority inversions on any task \( \tau_i \) under G-LP-FPS with eager preemptions in any time interval of length \( t \) is at most \( p_i \) after it starts executing, where \( p_i \) is defined in (4.1).

*Proof.* According to Lemma 4.3.1, \( \tau_i \) suffers from at most 1 priority inversion every time it is inserted into the ready queue. Recall that each task \( \tau_i \) can be preempted at most \( p_i \) times during any interval of length \( t \). Therefore, \( \tau_i \) can be inserted into the ready queue most \( p_i \) times after it starts, i.e., whenever it is preempted. 

**Task execution dynamics on multiprocessors:** Phillips et al. [69] noted that, under preemptive scheduling, in any time interval of length \( t \) between the release time and deadline of a task \( \tau_i \), whenever \( \tau_i \) is not executing, all processors are executing higher priority jobs. Similar ideas were subsequently used to derive schedulability analysis techniques for G-P-EDF and G-P-FPS (e.g., [64] [72]). On the other hand, when tasks are composed of non-preemptive regions, Phillips et al. ’s observation needs to be modified as follows: In any time interval of length \( t \), whenever a task \( \tau_i \) is ready and is not executing, the processor is executing higher priority tasks, or lower priority NPRs of tasks blocking \( \tau_i \). This observation can be applied to obtain the time at
which any non-preemptive region of task $\tau_i$ can start executing, which in turn gives us the corresponding response time.

In the following two lemmas, we present an upper-bound on the lower priority interference on 1) the first NPR of any task $\tau_i$ and 2) any other NPR of $\tau_i$.

The lower priority interference on the first NPR of $\tau_i$ needs to account for the push through blocking which is the interference pushed through by the final NPR of the previous job of $\tau_i$. However, as noted by Davis et al. [15], when calculating the higher priority interference using Bertogna et al.’s [64] method, since the higher priority tasks are assumed to be executing as late as possible, the effect of push through blocking is already accounted for. We therefore, obtain the following upper-bound on the blocking on the first NPR of any $\tau_i$.

**Lemma 4.3.2.** The lower priority interference without accounting for the push through blocking on the first NPR $b_{i,1}$ of a task $\tau_i$ over all $m$ processors under G-LP-FPS with eager preemptions is upper-bounded by

$$\Delta_i^m = \sum_{\tau_j \in lp(i)} \max_{1 \leq k \leq q_{j+1}} b_{j,k}$$

where, the $\sum_{\tau_j \in lp(i)} \max_{\tau_j \in lp(i)} \max_{1 \leq k \leq q_{j+1}}$ term denotes the sum of the $m$ largest values among the NPR’s of all $\tau_j \in lp(i)$.

**Proof.** This follows from the fact that at most $m$ tasks can be executing at any given time instant, and eager preemptions guarantee that the first preemptable lower priority task is preempted by a higher priority task.

In the worst case, when $\tau_i$ is released, all the $m$ processors have just started executing the $m$ largest lower priority NPRs. Consequently, with eager preemptions $\tau_i$ needs to wait until these $m$ largest NPRs of lower priority tasks complete their execution before all the processors are busy executing either tasks having higher priority than $\tau_i$ (in which case, there is no more priority inversion) or $\tau_i$ itself. \qed
4.3 Schedulability Analysis for G-LP-FPS with Eager Preemptions

Lemma 4.3.3. The lower priority interference on the $p^{th}$ non-preemptive region $b_{i,p}$ of any task $\tau_i$ over all $m$ processors under G-LP-FPS with eager preemptions is upper-bounded by

$$\Delta_i^{m-1} = \sum_{\tau_j \in lp(i)} \max_{1 \leq k \leq q_j+1} b_{j,k}$$  \hspace{1cm} (4.7)

where, $2 \leq p \leq q_i + 1$ and $\sum_{\tau_j \in lp(i)} \max_{1 \leq k \leq q_j+1} b_{j,k}$ denotes the sum of the $m - 1$ largest values among all $\tau_j \in lp(i)$.

Proof. When $\tau_i$ is executing, then at most $(m - 1)$ processors are executing lower priority tasks. Suppose that there exists a time instant between the start time and finish time of $\tau_i$ when all the processors are executing lower priority tasks. Let $t$ denote the earliest such time instant. This means that at time instant $t$, the scheduler scheduled a new low priority job (i.e., an $m^{th}$ lower priority task) even though $\tau_i$ was waiting in the ready queue or was executing. We get a contradiction because of our assumption of an global fixed priority based scheduler. \hfill $\Box$

Lemma 4.3.4. An arbitrary task $\tau_i$ can start executing its final NPR $q_i + 1$, in any time interval of length $t$ under G-LP-FPS with eager preemptions if $\tau_i$ is ready at the beginning of the interval and,

$$C_{i}^* + \left[ \frac{1}{m} \left( \Delta_i^m + p_i \times \Delta_i^{m-1} + \sum_{\tau_j \in hp(i)} I_i(t, \tau_j) \right) \right] \leq t$$  \hspace{1cm} (4.8)

where, $p_i$ is given by (4.1), $\Delta_i^m$ is given by (4.6), $\Delta_i^{m-1}$ is given by (4.7) and $I_i(t, \tau_j)$ is given by (4.5).

Proof. Recall that $\Delta_i^m$ denotes the sum of the largest lower priority NPRs that can block $\tau_i$ over all $m$ processors and $I_i(t, \tau_j)$ gives the worst case interference in the interval $t$. We know from Lemma 4.3.1 that the each NPR of $\tau_i$ can be blocked at most once. Moreover, we know from lemmas 4.3.2 and 4.3.3 that the first NPR of $\tau_i$ can be
blocked by at most $\Delta_i^m$, and that each of the remaining NPRs of $\tau_i$ can be blocked at most $\Delta_i^{m-1}$ over all the $m$ processors. Moreover, the number of preemptions on $\tau_i$ is upper-bounded by $p_i$. In the worst case, whenever $\tau_i$ resumes its execution after a preemption, it is blocked by lower priority NPRs. Therefore the upper-bound on the blocking experienced by $\tau_i$ is given by $\Delta_i^m + p_i \times \Delta_i^{m-1}$.

Proof follows from the fact that $\tau_i$ has completed $C_i^* = C_i - b_i, q_i, + 1$ units of execution, after incurring the worst case higher and lower priority interference, implying that the final NPR has already started its execution.

In the following, we present a schedulability test by observing that any task $\tau_i$ can be blocked only when it is preempted, and the number of preemptions on $\tau_i$ is at most $p_i$.

**Theorem 4.3.1.** If for any task $\tau_i$, suppose $t'$ denotes the smallest $t$ for which equation (4.8) is satisfied, the response time of $\tau_i$ under G-LP-FPS with eager preemptions is given by,

$$R_i = C_i + \left[ \frac{1}{m} \left( \Delta_i^m + p_i \times \Delta_i^{m-1} + \sum_{\tau_j \in hp(i)} I_i(t', \tau_j) \right) \right]$$

where $\Delta_i^m$ and $\Delta_i^{m-1}$ are defined in (4.6) and (4.7) respectively.

**Proof.** The proof follows from the fact that, at $t'$, the final NPR of $\tau_i$ has started its execution.

The smallest $t$ that satisfies (4.8) can be obtained by first setting $t = C_i^*$ and performing a fixed point iteration on (4.8) until the condition is satisfied or until a value greater than $D_i - b_i, q_i + 1 + 1$ is obtained, in which case the task is unschedulable.

The test presented above is, however, pessimistic since it assumes that all the higher priority tasks have carry-in jobs that interfere with $\tau_i$. In the following, we build on Baruah’s observations [70], and identify a critical scenario that gives the worst case behavior under
limited preemptive scheduling while limiting the number of tasks with carry-in jobs. Baruah [70] observed that the worst case scenario that leads to a deadline miss on any job of $i_t$, under preemptive scheduling, is such that there exists a busy interval prior to the release of $\tau_i$ where all the processors are executing higher priority jobs and extends to the time instant when the job of $\tau_i$ can start executing. The start time of such a busy period is assumed to be the earliest time before the release of $i_t$ such that at least one processor is idle and no processors are idle between the start of the busy period and the release of $i_t$. We build on this observation by identifying that, in the case of limited preemptive scheduling, the NPRs of some lower priority jobs could influence the busy period and consider the following scenario.

**Critical Scenario:** Consider an arbitrary job $J_i$ of a task $i_t$ released at time instant $r_i$ that has started its execution, such that all jobs of all tasks released prior to $r_i$ are schedulable. We consider the earliest time instant $t_1$ before the release time $r_i$ at which a higher priority task is released, and is blocked by NPRs of some tasks having a lower priority than $\tau_i$, as illustrated in Figure 4.3. If there are no higher priority job releases prior to $r_i$, we set $t_1 = r_i$ (*i.e.*, assume $J_i$ is the task that is blocked). Let the number of processors executing lower priority NPRs at time instant $t_1$
be $x$, $1 \leq x \leq m$. Let $t_0$ denote the earliest among the start times of these $x$ lower priority NPRs that block the higher priority task released at $t_1$. If there is no such time instant, we set $t_0$ to be the earliest time instant before $r_i$ at which at least one processor is idle and no processor is idle in $(t_0, r_i)$ (there is always such an instant e.g., at the start of the schedule). Since we consider a sporadic task system, there is always a possibility of lower priority tasks being released, and being executed, at $t_0$ that may potentially block the execution of $\tau_i$.

**Observation 4.3.1.** At most $m - 1$ higher priority tasks are active at time instant $t_0$.

According to our identified critical scenario, at least one processor is starting to execute a lower priority task at time instant $t_0$. This means that except for the higher priority tasks currently executing on at most $m - 1$ processors no other higher tasks are active. If more than $m - 1$ higher priority tasks were active, lower priority tasks could not have started executing at $t_0$.

The above observation allows us to limit the number of carry-in tasks to at most $m - 1$, rather than assuming that all the tasks have carry-in executions. Consequently, we can exploit the recent advances [30] in efficiently accounting for the carry-in interference to determine the schedulability of real-time tasks under G-LP-FPS with eager preemptions. The workload generated by any higher priority task $\tau_j$ having carry-in interference in any time interval of length $t$ is given as follows [30] (see Figure 4.1 for an illustration):

$$W_j^{CI}(t) = C_j + \left\lfloor \frac{\max((t - C_j), 0)}{T_j} \right\rfloor \times C_j +\quad (4.9)$$

$$\min ([t - C_j]_0 \mod T_j - (T_j - R_j)]_0, C_j)$$

In the above equations, $[A]_B$ means $\max(A, B)$. On the other hand, the workload generated by any higher priority task $\tau_j$ that does not carry-in interference in any time interval of length $t$ is given as follows [30] (see
4.3 Schedulability Analysis for G-LP-FPS with Eager Preemptions

Figure 4.2 for an illustration):

\[ W_j^{NC}(t) = \left\lfloor \frac{t}{T_j} \right\rfloor \times C_j + \min(t \mod T_j, C_j) \quad (4.10) \]

Therefore, in any time interval of length \( t \) and for any \( \tau_j \), we can calculate the upper-bound on the associated carry-in as follows [30]:

\[ W_j^{diff}(t) = W_j^{CI}(t) - W_j^{NC}(t) \]

A conservative upper-bound on the amount of carry-in by higher priority tasks that interferes with any task \( \tau_i \) can be obtained from the \((m-1)\) largest \( W_j^{diff} \) s:

\[ I_i^{CI}(t) = \sum_{\tau_j \in hp(i)} \max_{1 \leq j \leq m-1} W_j^{diff}(t) \]

Similarly, a conservative upper-bound on the interference generated by higher priority tasks on \( \tau_i \), that do not carry-in, in any time interval of length \( t \) is given by,

\[ I_i^{NC}(t) = \sum_{\tau_j \in hp(i)} W_j^{NC}(t) \]

Therefore, the total higher priority interference on any task \( \tau_i \) in any time interval of length \( t \) can be calculated using:

\[ I_i(t) = I_i^{CI}(t) + I_i^{NC}(t) \quad (4.11) \]

However, when restricting the number of carry-in jobs to at most \( m - 1 \), the tasks are not assumed to be executing as late as possible. Consequently, as noted by Davis et al. [15], the effects of push through blocking is not automatically accounted for. In the following we show that, for any \( \tau_i \), only at most one final NPR of a previous job of \( \tau_i \) can contribute to the push through blocking and the push through blocking can affect only its first NPR.

We consider the critical scenario described above and investigate the push through blocking on the job \( J_i \) of \( \tau_i \). We show that the push through blocking comes from at most the final NPR of the previous job of each task.
Lemma 4.3.5. The push through blocking on \( \tau_i \) comes from at most the final NPR of the previous job of \( \tau_i \), under G-LP-FPS with eager preemptions.

Proof. According to the critical scenario, all jobs released prior to the release time of the job \( J_i \) of \( \tau_i \) are schedulable, including any previous jobs of \( \tau_i \). This means that the final NPR of the previous job of \( \tau_i \) had started executing. Therefore, the push through blocking from its previous NPR must have ended. Therefore, the blocking on \( J_i \) comes from only at most the previous job of \( \tau_i \) and its final NPR. \( \square \)

Lemma 4.3.6. Only the first NPR of \( \tau_i \) can be affected by the push through blocking, under G-LP-FPS with eager preemptions.

Proof. Follows from the fact that the push through blocking needs to end before the first NPR of \( \tau_i \) can start executing, and hence does not affect any subsequent NPRs. \( \square \)

Lemmas 4.3.5 and 4.3.6 allows us to upper-bound the lower-priority interference, including any push through blocking, on the first NPR of any \( \tau_i \) (the blocking on other NPRs of \( \tau_i \) remains the same as in (4.7).

Lemma 4.3.7. The lower priority interference on the first NPR \( b_{i,1} \) of a task \( \tau_i \) over all the \( m \) processors under G-LP-FPS with eager preemptions can be upper-bounded by

\[
\Delta_i^m = \sum_{\tau_j \in lp(i)} \max_{\beta_i} \beta_i
\]

where, set \( \beta_i = \{ \tau_j \in lp(i), \max_{1 \leq k \leq q_j + 1} b_{j,k} \} \cup \{ b_{i,q_i + 1} \} \), and \( \sum_{\tau_j \in lp(i)} \max_{\beta_i} \beta_i \) term denotes the sum of the \( m \) largest values.

Proof. At the release time of \( \tau_i \), when calculating blocking, we have two cases:

1. All \( m \) processors are executing lower priority NPRs.
4.3 Schedulability Analysis for G-LP-FPS with Eager Preemptions

In this case, there is no push through blocking since it must have ended for all the processors to be executing lower priority NPRs. The worst case blocking in this case is given by the $m$ largest lower priority NPRs.

2. At least one processor is executing a higher priority task.

In this case, the higher priority tasks may bring in push through blocking. Therefore, the worst case blocking on $\tau_i$ happens when $m - 1$ processors are executing the $m - 1$ largest NPRs and the $m^{th}$ processor is executing the highest priority job that brings in a push through blocking. The push through blocking is at most $b_{i,q_i+1}$ according to lemmas 4.3.5 and 4.3.6

Therefore, the worst case blocking on the first NPR of any $\tau_i$ is obtained by taking the maximum of the two cases described above. \qed

The Lemma 4.3.4 can be modified as follows to compute the start time of the final NPR.

**Lemma 4.3.8.** An arbitrary task $\tau_i$ can start executing its final NPR $q_i + 1$, in any time interval of length $t$ under G-LP-FPS with eager preemptions if $\tau_i$ is ready at the beginning of the interval and,

$$C_i^* + \left[ \frac{1}{m} \left( \Delta_i^m + p_i \times \Delta_i^{m-1} + I_i(t) \right) \right] \leq t \quad (4.13)$$

where, $p_i$ is given by (4.1), $\Delta_i^m$ is given by (4.12), $\Delta_i^{m-1}$ is given by (4.7) and $I_i(t)$ is given by (4.11).

Consequently, the response time of any $\tau_i \in \Gamma$ can be calculated by modifying Theorem 4.3.1, as follows:

**Theorem 4.3.2.** For any task $\tau_i$, suppose $t'$ denotes the smallest $t$ for which equation (4.13) is satisfied, the response time of $\tau_i$ under G-LP-FPS with eager preemptions is given by,

$$R_i = C_i + \left[ \frac{1}{m} \left( \Delta_i^m + p_i \times \Delta_i^{m-1} + \sum_{\tau_j \in hp(i)} I_i(t') \right) \right]$$
where $\Delta^m_i$ and $\Delta_{i-1}^m$ are defined in (4.12) and (4.7) respectively and $I_i(t')$ is defined in (4.11).

While it may seem that accounting for worst case lower priority interference per preemption on each task is pessimistic, it is sufficient to guarantee the absence of scheduling anomalies. For example, it may happen that some higher priority tasks execute for less than their worst case execution time and a lower priority task starts executing a large NPR in the resulting slack.

### 4.4 Evaluation

In this section, we report the results of an experimental evaluation of the performance of G-LP-FPS with eager and lazy preemptions using weighted schedulability [2]. For this purpose, we used the test derived in this chapter for the eager preemption approach (EPA). The schedulability under lazy preemption approach (LPA) was determined for link-based scheduling [18] using the inflation based approach [29] in conjunction with the schedulability test for G-P-FPS given by Guan et al. [30]. We calculated the weighted schedulability variations with respect to 1) number of tasks per taskset 2) the NPR lengths and 3) number of processors. Specifically, we varied:

1. the number of tasks keeping the number of processors and NPR lengths constant
2. the NPR lengths keeping the number of processors and the number of tasks constant for tasks with 1) long NPRs relative to their WCET and 2) short NPRs relative to their WCET
3. the number of processors keeping the number of tasks and NPR lengths constant

A higher weighted schedulability implies a better scheduling algorithm since the schedulability is weighted against taskset utilizations. For reference, we also included the weighted schedulabilities under
G-P-FPS assuming no overheads and G-NP-FPS; the performance of G-P-FPS will significantly decrease relative to the other algorithms when overheads are included since the pessimism associated with overhead accounting is much higher. Detailed evaluation via analysis including overheads and measurements from a real implementation are future work. Moreover, the behavior of G-P-FPS can be obtained using G-LP-FPS by allowing preemptions after every unit of execution, with ties broken using task priority, and that of G-NP-FPS can be obtained by having no preemption points.

### 4.4.1 Experimental Setup

We used the \textit{UUniFast-Discard} algorithm proposed by Davis and Burns [73] to generate task utilizations. The minimum separation times (periods) were uniformly generated between 50 and 500 (note that the time period ranges are changeable). Deadlines were set equal to periods (implicit deadlines), although the schedulability tests also apply to constrained-deadline tasksets. The largest NPR values for each $\tau_i \in \Gamma$ were set as a percentage of its WCET denoted by $\mathcal{P}$ (ceiling function was applied to get integer values). We assumed that all NPRs of $\tau_i$ have the same length equal to the largest value, except for the first that can be smaller (depending on the WCET). Note that prior work [16] shows that larger final NPRs give improved schedulability. The utilization of the tasksets ranged from a minimum of $U_{\text{tot}}^{\text{min}}$ to a maximum of $U_{\text{tot}}^{\text{max}} = m$, where $m$ is the number of processors. In the experiments, we set $U_{\text{tot}}^{\text{min}} = 2.4$ since we are more interested in scheduling tasks with larger utilizations to effectively use the platform. We assumed Deadline Monotonic Priority ordering (DMPO) [74]; although better priority assignment algorithms exists for G-P-FPS DMPO serves our purpose of comparison since we use the same priority assignment for all the considered scheduling algorithms. The test in [30] was adopted as reference for G-P-FPS, and for G-NP-FPS, we used our test after setting the largest NPR length equal to the task computation times.
4.4.2 Experimental Results

In the following, we present our experimental results comparing eager and lazy preemptions varying the number of tasks, size of NPRs and the number of processors.

Varying number of tasks: In the first experiment, we examined the performance of G-LP-FPS for varying numbers of tasks for \( m = 4 \) processors and \( P = 5\% \). The results of the experiments are illustrated in Figure 4.4. The experiment indicates that for high utilization tasksets with large numbers of tasks, G-LP-FPS with eager preemptions performs better than with lazy preemptions. This is due to the inherent pessimism in the inflation based technique, which is amplified for large tasksets and large utilizations. We also observe that the performance of

![Figure 4.4: Weighted schedulability under varying number of tasks.](image)

with large numbers of tasks, G-LP-FPS with eager preemptions performs better than with lazy preemptions. This is due to the inherent pessimism in the inflation based technique, which is amplified for large tasksets and large utilizations. We also observe that the performance of
G-NP-FPS improved significantly with an increasing number of tasks. The main reason is that when the number of tasks is large, the individual task utilizations are very small, hence most tasks have relatively small computation times in relation to their deadlines and so are more amenable to G-NP-FPS [8]. By contrast, the presence of preemption points introduces additional blocking (from lower priority interference occurring after the start of the execution) leading to reduced schedulability. Evaluations for larger numbers of processors (specifically m=6 and m = 8), varying the number of tasks showed similar behavior.

**Varying NPR lengths:** We performed a set of two experiments to investigate the consequences of changing NPR lengths, and consequently the number of preemption points, on schedulability. Increasing NPR lengths increases pessimism in link-based scheduling because of the inflation of WCETs. However, it can have beneficial effects in our test for G-LP-FPS under eager preemption. In the first experiment we varied the size of NPRs as a percentage of the corresponding WCETs, between 5% and 100% (approximated using a ceiling function). The results are presented in Figure 4.5. As can be seen the schedulability varies in a saw-tooth manner until the NPR lengths reach 50% (i.e., the number of preemption points reduces to 1). This saw-tooth variation in schedulability between 5% and 50% demonstrates the futility of increasing the largest NPR lengths without reducing the number of preemption points. For example, as we increase the largest NPR lengths from 25% to 30%, there is no further decrease in the number of preemption points (which is 3 for both cases). On the other hand, the NPR lengths increase leading to increased lower priority interference that occurs after the start of the task executions, reducing schedulability. On further increasing the largest NPR lengths to 35%, the number of preemption points decreases to 2, consequently improving schedulability due to reduction of lower priority interference after the start time of the tasks. The same reasoning explains the decrease in schedulability when increasing the
largest NPR lengths from 35% to 45% since the number of preemptions remain unchanged at 2. When the NPR lengths increase to 50%, schedulability increases because the number of preemptions decrease to 1. Once past 50%, the schedulability continues to decrease until the tasks are almost fully non-preemptive, after which the schedulability starts to increase again. This is because, once the NPR lengths go past 50% of WCET, there is no further reduction in the number of preemption points; however, the length of the largest NPRs continues to increase, consequently increasing blocking on higher priority tasks (that may have a single preemption point). Finally, when the tasks become fully non-preemptive, schedulability increases since the lower priority interference decreases due to the reduction in the
Figure 4.6: Weighted schedulability under varying NPR lengths (small NPRs).

number of preemption points from one to zero.

In the second experiment we varied the size of NPRs from 0.5% to 5% (approximated using a ceiling function). The results are presented in Figure 4.6. Link based scheduling fared better compared to G-LP-FPS with eager preemptions for very small NPR lengths as shown in Figure 4.6. This is due to the fact that small NPRs imply large number of preemption points that in turn imply higher lower priority interference after the tasks start executing. This increased lower priority interference leads to unschedulability. In a few cases, our experiment identified some tasksets as schedulable using G-LP-FPS with eager preemptions but unschedulable under any other approach. However, this was not observed with link-based scheduling since it is
Varying number of processors: Finally, we varied the number of processors keeping the number of tasks and NPR lengths constant, at 30 and 5% respectively (reported in Figure 4.7). We observe that link-based scheduling has a higher schedulability when the number of processors increases. This is due to the fact that availability of more processors implies better schedulability under link-based scheduling for low utilization tasksets with a fixed number of tasks. The same trend is seen in the left end of Figure 4.4 where the number of tasks compared to the available number of processors is small. However, for tasksets with a large number of tasks and high utilizations, the eager
preemption approach is the most effective (Figure 4.4).

4.5 Chapter Summary

Limiting preemptions to predetermined points within real-time tasks is an effective means of reducing preemption and migration related overheads. However, it introduces an interesting question of how best to manage preemption. At one extreme, the scheduler can choose *eager* preemption of the first executing lower priority task that becomes preemptable, while at the other extreme, it can restrict preemption to only the lowest priority executing task, when it becomes preemptable, referred to as *lazy* preemption. Each strategy has a different effect in terms of the number of priority inversions in the schedule, that in turn affects schedulability.

In this chapter, we made the following contributions:

1. We derived a schedulability analysis for G-LP-FPS with eager preemptions building on the observation that blocking happens only when a task resumes execution. To the best our knowledge, this is the first such test for G-LP-FPS with fixed preemption points.

2. We evaluated the new schedulability test by comparing it with a test for G-LP-FPS with lazy preemption. The test used was the state-of-the-art test for G-P-FPS [30] supplemented by the inflation-based approach of accounting for blocking. [18].

Our evaluations showed that G-LP-FPS with eager preemptions outperforms link-based scheduling in the context of fixed preemption points. However, note that, there exists no dominance relation between eager and lazy approaches. Generally, eager preemptions are beneficial when higher priority tasks have short deadlines with respect to their execution times. On the other hand, lazy preemption is typically beneficial when medium priority tasks have a large number of preemption points. Moreover, the preemptive behavior of both the
approaches differs when it comes to actually reducing the runtime preemptions. This is further investigated in the following chapter.

Note: This chapter is based on the paper *Multiprocessor Fixed Priority Scheduling with Limited Preemption*, Abhilash Thekkilakattil, Rob Davis, Radu Dobrin, Sasikumar Punnekkat and Marko Bertogna, The 23rd International Conference on Real-Time Networks and Systems, ACM, Lille, France, November, 2015.
Chapter 5

Preemptive Behaviors in Limited Preemptive Scheduling

The activity of the intuition consists in making spontaneous judgements which are not the result of conscious trains of reasoning.¹

In this chapter, we investigate the run-time preemptive behavior of multiprocessor EDF and FPS under preemptive and limited preemptive paradigms. Specifically, which approach among EPA and LPA generates the least number of preemptions at run-time? Also, which scheduling paradigm among G-LP-FPS and G-LP-EDF generates the least number of preemptions at run-time? These questions are interesting because preemptions have a significant effect on, among others, energy consumption [76] and reliability [5][77][78]. Note that for tasksets in which the optimal fixed preemption points are already known, the runtime overheads depend on whether or not a preemption

¹This quote is from Alan Turing’s PhD dissertation [75], and is used here to highlight the counter-intuitive findings presented in this chapter.
actually occurs at these points. In other words, since preemptions are possible only at these optimal preemption points, the system performance will depend on the actual number of preemptions instead.

We show that, limited preemptive scheduling does not necessarily reduce the number of preemptions on multiprocessor platforms, and hence does not intuitively generalize from uniprocesors; in fact, under an eager preemption approach, the number of preemptions can be higher when compared to fully preemptive scheduling. We also show that the well known observation regarding the preemptive behavior of EDF on uniprocessors generalizes to multiprocessors; G-P-EDF generates fewer preemptions than G-P-FPS. The above, however, does not generalize to global limited preemptive scheduling; with respect to global limited preemptive scheduling, EDF generates more preemptions than FPS. We observe that Global LP FPS with LPA is the most effective with respect to reducing the number of preemptions at run-time. Finally, we conclude by presenting a discussion of the application of the above observations in a wide range of contexts, e.g., energy aware and mixed-criticality scheduling.

5.1 Models and Notations

In this section, we present the system model, terminology and definitions assumed in the rest of the chapter.

5.1.1 Task Model

We consider a set of $n$ sporadic real-time tasks $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ scheduled on $m$ processors. Each task $\tau_i$ is characterized by a minimum inter-arrival time $T_i$, and a relative deadline $D_i \leq T_i$, and is assumed to contain $q_i \geq 0$ optimal preemption points specified by the designer/programmer. Let $b_{i,j}$, $j = 1 \ldots (q_i + 1)$ denote the worst case execution time of task $\tau_i$ between its $(j - 1)^{th}$ and $j^{th}$ preemption points (The calculation of the WCET between preemption points is an important problem on multicores; but this is not our current focus.
Instead we focus on the related problem of the prohibitive increase of WCETs due to preemption related overheads). We use the notation $b_{i,j}$ to also refer to the corresponding Non-Preemptive Region (NPR). The Worst Case Execution Time (WCET) of each task $\tau_i$ can be calculated as $C_i = \sum_{j=1}^{q_i+1} b_{i,j}$.

### 5.1.2 Definitions

According to our model, every task is composed of a set of non-preemptive regions with specified lengths. This allows us to define the *preemptability* of a task.

**Definition 5.** Any task $\tau_i \in \Gamma$ is defined to be preemptable at time instant $t$ if and only if time instant $t$ corresponds to a preemption point $k$, $1 \leq k < q_i$, in the progress of its execution.

An unfinished task is defined to be *ready* if it is not currently executing (and hence it is assumed to be in the ready queue). Non-preemptive regions within the lower priority tasks can cause priority inversions on higher priority tasks. We enumerate three conditions that are necessary for a single priority inversion to occur due to an NPR of a lower priority task under limited preemption scheduling.

**Definition 6.** A *priority inversion* occurs on an arbitrary task $\tau_i$ whenever the scheduler is invoked and the following conditions hold.

- **C1:** The task $\tau_i$ is ready to execute (but not executing).

- **C2:** At least one processor is executing a task with a lower priority than $\tau_i$.

- **C3:** None of the executing lower priority jobs are preemptable.

We differentiate the following types of *interference* for any task $\tau_i$. The *higher priority interference* on a task $\tau_i$ is defined as the cumulative executions of all tasks with a higher priority than $\tau_i$ that
prevent \( \tau_i \) from executing. The **lower priority interference** on a task \( \tau_i \) is defined as the cumulative executions of all tasks with a lower priority than \( \tau_i \) that prevent \( \tau_i \) from executing.

When scheduling real-time tasks on multiprocessors under a limited preemptive paradigm, the choice of the preempted task *viz.* EPA and LPA influences the number of priority inversions on the tasks, consequently affecting the lower priority interference on them.

### 5.2 Schedulability under Global LP Scheduling

In this section, we recapitulate the two approaches to preemption under global LP scheduling *viz.* the eager and lazy preemption approaches. These examples serve to illustrate conditions under which eager and lazy preemption approaches are effective in terms of schedulability. We specifically consider how preemptive behavior influences schedulability.

### 5.2.1 Prior Works

Davis *et al.* [15] [16] proposed schedulability analysis techniques for global FPS with deferred preemptions. They considered task sets with tasks having only a single NPR at the end of their execution, which was generalized by Marinho *et al.* [17]. Analysis techniques for resource sharing protocols, such as link based scheduling [18], can be applied in the context of limited preemptive scheduling. We [79] presented a schedulability analysis technique for G-LP-FPS with EPA, and compared it with G-LP-FPS with LPA using link based scheduling. Chattopadhyay and Baruah [80] and Thekkilakkattil *et al.* [81] presented techniques to determine schedulability of real-time tasks scheduled using global limited preemptive EDF. All of the above works focused on schedulability.

Marinho [82] presented some observations regarding the preemptive behavior under eager and lazy preemption approaches; however, did not present detailed empirical comparisons. Previously, a preliminary study of the preemptive behavior of G-LP-FPS [83] and
that of G-LP-EDF [84] was conducted with respect to the number of preemptions under both EPA and LPA. In this chapter, we perform a comprehensive comparison of the preemptive behavior using a weighted metric similar to weighted schedulability [2] and vary different parameters that may have an impact on the number of preemptions. Moreover, we compare the preemptive behavior of G-LP-EDF with that of G-LP-FPS under both eager and lazy approaches. To our knowledge, this is the first such effort towards investigating the runtime preemptive behavior of limited preemptive scheduling on multiprocessors.

We illustrate eager and lazy preemption approaches using an example, and then identify some interesting scenarios occurring at run-time, that influence schedulability (note that schedulability was empirically investigated in the previous chapter after presenting a schedulability test).

### 5.2.2 Lazy Preemption Approach

In the Lazy Preemption Approach (LPA), if a higher priority task is released and all lower priority jobs are executing their NPRs, the scheduler waits for the lowest priority job to complete its NPR (i.e., to become preemptable) before allowing the higher priority job to preempt. We illustrate this approach using the following example.

**Example 11.** Consider four tasks $\tau_1$, $\tau_2$, $\tau_3$ and $\tau_4$, where $\tau_1$ has the highest priority and $\tau_4$ has the lowest, scheduled on 2 processors $P_1$ and $P_2$. Consider the scenario in figure 5.1 in which $\tau_1$ and $\tau_2$ are released during the execution of NPRs of $\tau_3$ and $\tau_4$ as shown. If the scheduling algorithm used is G-LP-FPS with lazy preemptions, $\tau_1$ and $\tau_2$ will be blocked until $\tau_4$ finishes executing its NPR, after which $\tau_1$ is scheduled on $P_2$. However, $\tau_2$ cannot be scheduled because $\tau_3$ has already started executing its NPR at this point. Once $\tau_3$ completes execution of its NPR, $\tau_2$ is allowed to execute on $P_1$. Although we have considered tasks with fixed priorities, it can be easily seen from the example that the same conclusions can be made about the preemptive behavior under G-LP-
EDF with lazy preemption.

The above example demonstrates that any higher priority task can suffer from priority inversions many times before it starts its execution and suffers from lower priority interference only before the start. In the above example, even though $\tau_3$ was available to be preempted twice, $\tau_1$ had to wait for the NPR of $\tau_4$ to complete. Similarly, $\tau_2$ had to wait for $\tau_1$ to be scheduled.

5.2.3 Eager Preemption Approach

On the other hand, under the Eager Preemption Approach (EPA), if a higher priority task is released and all lower priority executing jobs are executing their NPRs, the scheduler preempts the first lower priority that becomes preemptable. We illustrate this approach using the following example.

**Example 12.** Consider the same four tasks $\tau_1$, $\tau_2$, $\tau_3$ and $\tau_4$ presented in example 11, and the same scenario in which $\tau_1$ and $\tau_2$ are released
The above example demonstrates that any higher priority task can suffer from priority inversions many times before it starts its execution and suffers from lower priority interference only before the start. In the above example, even though $\tau_3$ was available to be preempted twice, $\tau_1$ had to wait for the NPR of $\tau_4$ to complete. Similarly, $\tau_2$ had to wait for $\tau_1$ to be scheduled.

### 5.2.3 Eager Preemption Approach

On the other hand, under the Eager Preemption Approach (EPA), if a higher priority task is released and all lower priority executing jobs are executing their NPRs, the scheduler preempts the first lower priority that becomes preemptable. We illustrate this approach using the following example.

**Example 12.** Consider the same four tasks $\tau_1$, $\tau_2$, $\tau_3$ and $\tau_4$ presented in example 11, and the same scenario in which $\tau_1$ and $\tau_2$ are released during the execution of NPRs of $\tau_3$ and $\tau_4$. If the scheduling algorithm used is G-LP-FPS with eager preemptions, the scheduler will allow $\tau_1$ to preempt $\tau_3$ rather than wait for $\tau_4$ since it is the first available opportunity to preempt (see figure 5.2). Once $\tau_4$ completes its NPR, $\tau_2$ is scheduled on $P_2$. Although we have considered tasks with fixed priorities, the same conclusions can be made about the preemptive behavior under G-LP-EDF with lazy preemption.

Note that, an eager approach to preemption implies that a job suffers from at most one priority inversion when it is released, unlike in the case of the lazy preemption approach shown in example 11. However, jobs can suffer from priority inversions after they have started their execution leading to lower priority interference after they starts their execution. In figure 5.2, $\tau_3$ is preempted by $\tau_1$ and is inserted into the ready queue, while a lower priority task $\tau_4$ is executing on $P_2$.

**Global Preemptive and Non-Preemptive Scheduling:** In the above
5.2.4 Schedulability under Eager and Lazy Preemption Approaches

Davis et al. [15] made some interesting observations regarding the task parameters suitable for eager and lazy preemption approaches, after noting that eager and lazy approaches to preemptions does not dominate one another with respect to schedulability. The observations are made in context of a specific scheduler that implements LPA called link based scheduling [18]:

1. LPA under link-based scheduling is expected to perform better if there are many tasks with many NPRs of similar size.

2. EPA performs better if only a few low priority tasks have long
Figure 5.3: Deadline miss under eager preemption.

Figure 5.4: No deadline miss under lazy preemption.

We take these observations a step further by taking a closer look at the other task parameters that potentially maximizes schedulability.

**Example 13.** Consider the scenarios in examples 11 and 12. As seen from figure 5.1, when $\tau_1$ is released and $\tau_3$ and $\tau_4$ are executing their NPRs, $\tau_1$ will wait for task $\tau_4$ to complete its NPR, consequently missing its deadline. On the other hand, as seen from figure 5.2, if $\tau_1$ preempts $\tau_3$, there is more slack in the schedule for $\tau_1$ (as well as for the other tasks) and there are no deadline misses.

The above described scenario indicates that lazy preemption approach is most likely to be ineffective when the tasksets consists of:

1. Higher priority tasks with comparatively large execution times relative to their deadlines.

2. Lower priority tasks with several long NPRs.
Higher priority tasks with little slack need to be scheduled as early as possible, \textit{i.e.}, using an eager approach, to ensure absence of deadline misses.

The main challenge with the eager preemption approach is the potential \textit{lower priority interference} that may occur \textit{after} the start of the task execution. Consequently, the lazy preemption approach is better in situations in which the number of preemption points on higher priority jobs is significantly large, and lower priority jobs have very large non-preemptive regions. This implies that repeated preemptions on higher priority tasks can cause significant blocking on them potentially leading to deadline misses.

\textbf{Example 14.} Consider the scenario in figure 5.3 in which $\tau_3$ and $\tau_4$ are executing on two processors and $\tau_2$ is released. In this case, under EPA, $\tau_2$ preempts $\tau_3$ instead of $\tau_4$, causing a priority inversion on $\tau_3$, after the start of its execution. Similarly, later when $\tau_1$ is released, it again preempts $\tau_3$ since $\tau_4$ is still executing its NPR. Such repeated priority inversions after the start of the execution leads to a deadline miss on $\tau_3$. On the other hand, Figure 5.4 shows that scheduling these jobs using the lazy preemption approach avoids deadline misses in this scenario.

Eager preemption approach is most likely to be ineffective for tasksets with:

1. Higher priority tasks having short inter-arrival times leading to potentially repeated preemptions of medium priority tasks.

2. Medium priority tasks having comparatively large execution times in relation to their deadlines with many preemption points.

3. Lower priority tasks having large NPRs.

Tasksets with tasks having the above characteristics are potentially infeasible under EPA since the higher priority tasks may repeatedly preempt the medium priority tasks, that may then be blocked by the large NPRs of the lower priority tasks. This potentially causes deadline misses because of the \textit{lower priority interference} due to the \textit{priority inversions} that occur after the start of the task executions.
Chapter 5. Preemptive Behaviors in Limited Preemptive Scheduling

5.3 Evaluation of Preemptive Behavior under Global LP Scheduling

In this chapter, we investigate how the decision of choosing a scheduling paradigm, viz. EDF and FPS, and the preemption approach, viz. EPA and LPA, affect the number of preemptions at run-time. The number of run-time preemptions influence the preemption overheads in the schedule, which in turn influence, among others, amenability to energy aware scheduling, schedulability of lower criticality tasks in mixed criticality systems and resource contention. Remember that we consider tasksets in which the optimal fixed preemption points are already known. Therefore, the runtime overheads depend on whether or not a preemption actually occurs at these points. Put differently, since preemptions are possible only at these optimal preemption points, the system performance will depend on the actual number of preemptions instead.

Note that the performance of different combinations of schedulers and the approaches to preemption differ based on the underlying hardware and the specific application (task parameters, cache access patterns etc). As an exhaustive study on real cases that allows us to sufficiently generalize our results is resource and time demanding, we had to restrict the current experiments to the use of synthetic tasksets. However, our results form the basis for further work that will consider specific combinations of hardware platforms and applications.

5.3.1 Experimental Setup

The experimental methodology is similar to the one adopted by Buttazzo [31]. In order to perform the experiments, due to the very limited availability of simulators implementing limited preemptive scheduling under eager and lazy preemption approaches, we developed a simulator that takes as input the task parameters and generate the different schedules for a user defined time duration. The task parameters were generated using well accepted techniques and is
described in the following: We used the UUnifast-discard algorithm [73] to generate individual task utilizations that were varied between $U_{\text{min}}$ and $U_{\text{max}}$. For the case of FPS, we adopted the deadline-monotonic priority assignment– note that we are interested in the number of preemptions and not deadline misses. The scheduling algorithms, whose preemption behaviors we want to compare are in fact incomparable with respect to schedulability [15]. Therefore, to investigate the preemptive behavior, building on the speed-up bounds [47] and schedulability experiments, e.g., [3], that indicate high schedulability for tasksets that utilize up to 50% of the platform under both preemptive EDF and FPS, we set, $U_{\text{max}} = \frac{m}{2}$. Note that a fully preemptive schedule can be obtained using a limited preemptive scheduler by enabling preemptions after every unit of execution, therefore, if a taskset is preemptively schedulable, it is also LP schedulable. However, in one set of the experiments, we consider tasksets with utilization up to 100% of the processing platform in order to investigate the preemptive behavior for high utilization tasksets. The task periods were randomly generated between $T_{\text{min}} = 5$ and $T_{\text{max}} = 500$ (this represents tasks with periods $5 - 500\text{ms}$ as found in many typical real-time systems), and execution times were computed using the generated utilizations. We assumed deadlines to be equal to periods; note that this assumption does not affect the generality of the results since we consider the preemptive behavior and not schedulability. The largest NPR of each task was generated as a percentage of its execution time, with the ceiling function applied to obtain integer values– in our experiments, this was set to 10% (i.e., each task has no more than 9 preemption points). Note that we also vary the NPR lengths in one of the experiments. We counted the number of preemptions generated for one hundred tasksets under each of the following paradigms:

1. G-P-FPS
2. G-P-EDF
3. G-LP-FPS with eager preemptions (EPA-FPS)
4. G-LP-FPS with lazy preemptions (LPA-FPS)

5. G-LP-EDF with eager preemptions (EPA-EDF)

6. G-LP-EDF with lazy preemptions (LPA-EDF)

We simulated the respective schedules for a duration of 10000 time units.

**Weighted Metric:** In order to understand how the number of preemptions vary with a second parameter, e.g., number of tasks, in addition to utilization, we adopted a weighted measure similar to weighted schedulability [2]. We weighed the number of preemptions $N_i$, for a taskset $\Gamma_i$ with respect to a parameter $p$ over a simulation run for $\Delta$ time units, with the taskset utilization $U_i$ as follows:

$$W_p = \frac{\sum_{\forall \Gamma_i} U_i N_i}{\sum_{\forall \Gamma_i} U_i}$$

Here, the parameter $p$ could be, e.g., number of processors. We investigated how the number of preemptions in the actual schedule varies with:

1. Total utilization
2. Number of processors
3. Number of tasks
4. Length of the NPR.

In the following sections, we report our findings that show that limited preemptive scheduling on multiprocessors may exhibit counter-intuitive preemptive behaviors when compared to uniprocessor systems.
5.3.2 Results for Varying Total Utilization

In this set of experiments, we investigated the preemptive behavior of the scheduling algorithms for increasing utilizations. We considered a 4 processor platform and generated tasksets with 25 tasks and utilizations ranging from 1 to 4. We calculated the average number of preemptions per 100 time units, after simulating the schedule for a duration of 10000 time units—the results are reported in Figure 5.5. We observe that G-LP-EDF with EPA generates the maximum number of preemptions, and is followed by G-LP-FPS with eager preemptions. Perhaps surprisingly, G-P-EDF and G-P-FPS generate fewer preemptions than their limited preemptive counterparts with eager

![Figure 5.5: Average number of preemptions per 100 time units under varying utilizations.](image)

...
5.3 Evaluation of Preemptive Behavior under Global LP Scheduling

preemptions. Moreover, we observed that the uniprocessor behavior of EDF with respect to generating fewer number of preemptions than FPS [31] generalizes to the multiprocessor case; G-P-EDF generated less preemptions than G-P-FPS. The least number of preemptions is generated by G-LP-FPS with LPA. This approach generated slightly fewer number of preemptions even when compared to G-LP-EDF with LPA (the scaling of the graph makes it less visible—see an enlarged version in Figure 9.1 given in the appendix).

However, the above experiment is restricted in the sense that only a single processing platform with 4 processors was considered. In the following, we conduct experiments with varying number of processors after adopting the weighted metric described above.

5.3.3 Results for Varying Number of Processors

We generated tasksets with 25 tasks having utilizations ranging from 1 to $U_{max} = m/2$ for $m = 4$ to $m = 20$ in steps of 2. The results of the experiments are reported in Figure 5.6.

We can see that the number of preemptions, in general, decreases with increasing number of processors (since more processors for the same number of tasks imply a reduced need for preemption). The interesting observation here is that G-LP-EDF with EPA generates the most number of preemptions followed by G-LP-FPS with EPA. The fully preemptive variants of EDF and FPS generated fewer preemptions than their limited preemptive counterparts with EPA. The least number of preemptions are generated by G-LP-EDF and G-LP-FPS; both under LPA. Here again, even though G-LP-FPS with LPA generates slightly fewer number of preemptions compared to G-LP-EDF with LPA, the use of the weighted metric described above makes it less visible from the graph (see enlarged version of the above graph in Figure 9.2 given in the Appendix).
5.3.4 Results for Varying Number of Tasks

We varied the number of tasks per taskset from \( n = 6 \) to \( n = 26 \) in steps of 2 and counted the number of preemptions for tasksets with utilizations between 1 and \( U_{\max} = m/2 \). The results are reported in Figure 5.7. We observed a similar trend observed by Buttazzo [31]. The fully preemptive variant of EDF generated fewer preemptions than G-P-FPS. Moreover, the number of preemptions increased with increasing number of tasks. We expect that increasing the number of tasks further will lead to a decrease in the number of preemptions because individual task execution times will decrease to keep the utilization constant as noted by Buttazzo [31]—the decreasing trend is
5.3 Evaluation of Preemptive Behavior under Global LP Scheduling

observable when number of tasks change from 22 to 26.

![Graph showing weighted number of preemptions under varying number of tasks.](image)

Figure 5.7: Weighted number of preemptions under varying number of tasks.

We can see trends that are similar to the one observed in the previous experiments reported in this chapter: G-LP-EDF with EPA generated most preemptions while G-LP-FPS (together with G-LP-EDF) with LPA generated the least. Here again G-LP-FPS with LPA generated slightly fewer number of preemptions than G-LP-EDF with EPA (not visible due to scaling issues).
5.3.5 Results for Varying Lengths of Non-preemptive Regions

Lastly, we wanted to investigate if the preemption behavior would be any different had we chosen a different size for the non-preemptive regions. Therefore, we performed experiments counting the number of preemptions with varying NPRs. We considered a 4 processor platform and counted the number of preemptions generated when the NPR lengths changed from 5% to 100% of the task WCETs (no. of tasks per taskset=25). The experimental results are reported in Figure 5.8. The graph indicates that when using EPA (under EDF or FPS), the number of preemptions increases if the length of the NPRs increase without decreasing the number of preemptions points. This is observed by the increased number of preemptions when the NPR lengths increase from 35% to 40% (the number of preemptions remains unchanged at 2) and from 50% to 80% (when the number of preemptions remains unchanged at 1). Once past 80%, most of the tasks become non-preemptive since we apply the ceiling function, and hence there is a decrease in the number of preemptions. Similar trends are also observed in the case of EPA although less pronounced.

In all the cases, LPA outperformed all the other variants of the scheduling algorithms. Moreover, G-LP-EDF with EPA continued to have the highest number of preemptions for shorter NPR lengths, but showed similar performance as G-LP-FPS with EPA for larger NPR lengths (from around 45% as seen from Figure 5.8). Notably, for NPR lengths larger than 20%, EPA (under both EDF and FPS) generated fewer preemptions than the fully preemptive counterparts. An enlarged version of the above graph available in Figure 9.3 in the Appendix illustrates that for shorter NPR lengths, G-LP-FPS with LPA generates slightly fewer preemptions than G-LP-EDF with LPA. On the other hand, for larger NPR lengths, G-LP-FPS generates slightly more number of preemptions. However, we would like to clarify that this is observed only for NPR lengths larger than 50% (i.e., for tasks with a single preemption point). If there are more preemption points, then clearly FPS outperforms EDF. Moreover, note that very large NPR
5.3 Evaluation of Preemptive Behavior under Global LP Scheduling

5.3.5 Results for Varying Lengths of Non-preemptive Regions

Lastly, we wanted to investigate if the preemption behavior would be any different had we chosen a different size for the non-preemptive regions. Therefore, we performed experiments counting the number of preemptions with varying NPRs. We considered a 4 processor platform and counted the number of preemptions generated when the NPR lengths changed from 5% to 100% of the task WCETs (no. of tasks per taskset = 25). The experimental results are reported in Figure 5.8. The graph indicates that when using EPA (under EDF or FPS), the number of preemptions increases if the length of the NPRs increase without decreasing the number of preemptions points. This is observed by the increased number of preemptions when the NPR lengths increase from 35% to 40% (the number of preemptions remains unchanged at 2) and from 50% to 80% (when the number of preemptions remains unchanged at 1). Once past 80%, most of the tasks become non-preemptive since we apply the ceiling function, and hence there is a decrease in the number of preemptions. Similar trends are also observed in the case of EPA although less pronounced.

In all the cases, LPA outperformed all the other variants of the scheduling algorithms. Moreover, G-LP-EDF with EPA continued to have the highest number of preemptions for shorter NPR lengths, but showed similar performance as G-LP-FPS with EPA for larger NPR lengths (from around 45% as seen from Figure 5.8). Notably, for NPR lengths larger than 20%, EPA (under both EDF and FPS) generated fewer preemptions than the fully preemptive counterparts. An enlarged version of the above graph available in Figure 9.3 in the Appendix illustrates that for shorter NPR lengths, G-LP-FPS with LPA generates slightly fewer preemptions than G-LP-EDF with LPA. On the other hand, for larger NPR lengths, G-LP-FPS generates slightly more number of preemptions. However, we would like to clarify that this is observed only for NPR lengths larger than 50% (i.e., for tasks with a single preemption point). If there are more preemption points, then clearly FPS outperforms EDF. Moreover, note that very large NPR lengths may imply unschedulability due to the “long task problem” [5].

5.3.6 Observations

In the set of four experiments presented above, we have exercised a wide range of parameters to understand the run-time preemptive behavior of global scheduling algorithms. In the following, we enumerate the four main observations that can be made from the experiments.

Observation 5.3.1. Limited preemptive scheduling on multiprocessors does not necessarily reduce the number of preemptions; in fact with an eager preemption approach, the number of preemptions could be larger than in the case of fully preemptive scheduling, as well as global LP
The observation is interesting since it indicates that the scheduling dynamics on multiprocessor platform does not generalize from uniprocessors and hence design decisions, such as the decision to limit preemptions, need to be carefully taken. Not doing so, may result in unexpected consequences as illustrated by the increased number of preemptions under LP scheduling with EPA compared to fully preemptive schemes.

**Observation 5.3.2.** The preemption behavior of G-P-EDF observed by Buttazzo [31] generalizes to the case of multiprocessors; G-P-EDF generates fewer preemptions than G-P-FPS.

This observation indicates that when choosing between fully preemptive EDF and FPS, EDF is more suitable with respect to the minimizing the run-time number of preemptions. However, this does not generalize to limited preemptive scheduling as observed in the following.

**Observation 5.3.3.** Limited preemptively scheduling real-time tasks using an EDF paradigm on multiprocessors generates more preemptions than G-LP-FPS.

The above observation indicates that it might be better to adopt FPS instead of EDF in limited preemptive scheduling of real-time tasks (even though under fully preemptive scheduling EDF fared better). Moreover, as noted in the following, adopting the lazy preemption approach under FPS enables full exploitation of the benefits of using limited preemptive scheduling with respect to minimizing the run-time overheads when compared to the other combinations.

**Observation 5.3.4.** The observed number of preemptions in G-LP-FPS with lazy preemption is the least among all the considered algorithms; and hence is ideal with respect to minimizing preemption related overheads.
While acknowledging the benefits offered by the studies on a real implementation, we point out that experiments using synthetic tasksets, on the other hand, allows us to perform a significantly larger number of experiments, varying a wide range of parameters such as utilization, periods, number of tasks NPR lengths and number of processors (and forms the basis for our future work on a real hardware). The exhaustiveness of the considered parameters viz. total utilization, number of processors, number of tasks and non-preemptive regions, strengthens the confidence in our observations, allowing us to generalize our observations.

5.4 Analysis of the Experimental Results

In this section, we discuss the (counter-intuitive) experimental results in detail and identify reasons behind the observed behaviors. This can act as guidelines and inputs for real-time system designers to further optimize the performance of the system.

5.4.1 Preemptive Scheduling vs. Limited Preemptive Scheduling with Eager Preemptions

As seen from the evaluations presented in Section 5.3, particularly observation 5.3.1, global LP scheduling with eager preemptions incurs more preemptions than global fully preemptive scheduling. This is because more preemptions are required to resolve the priority inversions occurring due to the eager preemption of the lower priority executing task (not necessarily the lowest) that first completes executing its NPR. This is detailed in the following.

When using the eager preemption approach, if the first executing lower priority task that becomes preemptable is preempted by a higher priority job, what essentially happens is that the higher priority task transfers the priority inversion to the preempted task if it is not the lowest priority one (since there are lower priority tasks still executing on other processors). The preempted task, which is in the ready queue,
may preempt another lower priority task that first completes its NPR (again not necessarily the lowest priority one) thereby transferring priority inversion. This could potentially continue like a domino effect until no more priority inversion exists in the system. In order to resolve priority inversion for each task, there is a need for preemption, consequently increasing the number of preemptions. This is clarified in the following using a simple example.

![Diagram showing preemptions under Fully Preemptive Scheduling.]

**Figure 5.9: Preemptions under Fully Preemptive Scheduling.**

**Example 15.** Consider the scenario presented in Figure 5.9 in which 3 tasks $\tau_1$, $\tau_2$ and $\tau_3$, with priorities in the order $\tau_1 > \tau_2 > \tau_3$, are executing on 3 processors. Suppose a higher priority task $\tau_0$ is released. If the scheduler used is a fully preemptive scheduler, $\tau_0$ will preempt $\tau_3$ resulting in only a single preemption, as illustrated in Figure 5.9. On the other hand, under LP scheduling with eager preemptions, $\tau_0$ will preempt the first task that becomes preemptable, in this case $\tau_1$ (as seen from Figure 5.10). Note that $\tau_1$, has the highest priority among the executing ones. Consequently, there is a priority inversion on $\tau_1$ since the other processors are executing lower priority
tasks. The task \( \tau_1 \) will wait for one of the lower priority tasks to become preemptable. In our scenario, \( \tau_2 \) is the next task that becomes preemptable first. Consequently, \( \tau_2 \) will be preempted by \( \tau_1 \). However, the priority inversion persists because \( \tau_2 \), which was preempted, has a higher priority than \( \tau_3 \) that is still executing on \( P3 \), and hence there is one more preemption. Note that the number of preemptions in this case is 3 instead of 1 under fully preemptive scheduling.

The reduced number of preemptions under fully preemptive scheduling, with respect to EPA, is because of the fact that the highest priority task will preempt the lowest priority executing task, without causing the domino effect described previously. The priority inversions after the start of task executions has a direct impact on the lower priority interference consequently affecting schedulability as noted by us in [79].
5.4.2 Lazy Preemption Approach vs. Eager Preemption Approach

According to observation 5.3.3, the observed number of preemptions in the schedule can be significantly smaller under global LP scheduling with lazy preemptions, when compared to eager preemptions approach and fully preemptive schemes. This is because of the absence of priority inversions occurring after the start of the execution of the tasks. We explain this using the following example.

**Example 16.** Consider the same scenarios described in Figure 5.9 and 5.10. Under global LP scheduling with lazy preemption, the scheduler will preempt the lowest priority job \( \tau_3 \) and hence the domino effect described earlier will not occur as seen from Figure 5.11. Moreover, global LP scheduling with lazy preemption performs better than fully preemptive scheduling since the upper-bound on the number of preemptions in a task is determined by the number of preemption points instead of the number of higher priority task releases (that can be significantly larger).
The consequences of such a domino effect under EPA on the total number of observed preemptions can be severe on platforms with large number of processors since in the worst case \( m \) such priority inversions need to be resolved per high priority release which can be potentially very large especially in many-core systems.

### 5.4.3 Global Limited Preemptive FPS vs. Global Limited Preemptive EDF

In general, preemptive EDF generates fewer number of preemptions than preemptive FPS [31]. This is because, many of the preemptions necessitated by the fixed priority task assignment, do not occur under EDF. In the case of EDF the individual job priorities are assigned according to the earliest absolute deadlines. Consequently, preemptions required by task releases, that may have a higher priority under an FPS scheme, towards the end of the execution of tasks, that may have a lower priority under an FPS scheme, are avoided under EDF because of the deadline based priority ordering. On the other hand, in this scenario under FPS, a preemption occurs. Similarly, it is easily seen that uniprocessor limited preemptive EDF generates fewer preemptions than limited preemptive FPS. As seen from observation 5.3.2, similar to the uniprocessor case, G-P-EDF performs better in reducing the actual number of preemptions at run-time when compared to G-P-FPS.

However, under limited preemptive scheduling on multiprocessors, G-LP-EDF may generate more preemptions than G-LP-FPS as noted in observation 5.3.3.

The reason is that, EDF priority ordering generates more priority inversions consequently “forcing” eager preemptions. For example, under G-LP-FPS with LPA, higher priority tasks released during the execution of the final NPR of the lowest priority task wait for it to complete. This does not happen under EDF since at least one of the executing jobs may have a larger absolute deadline and hence can be preempted. We clarify the reason for the poor performance of G-LP-EDF using the following example.
Example 17. In this example, consider three tasks $\tau_2$, $\tau_3$, and $\tau_4$ that are currently executing on 3 processors and another task $\tau_1$ that is released during their execution as illustrated in Figure 5.12. Assume that $\tau_4$ has started executing its final NPR and the tasks have time periods $T_1 < T_2 < T_3 < T_4$. Under G-LP-EDF using lazy preemptions, when $\tau_1$ is released, $\tau_2$ is the task with the latest deadline and hence has the lowest priority; therefore $\tau_1$ preempts $\tau_2$. On the other hand under G-LP-FPS with lazy preemptions (as shown in Figure 5.13), assuming rate monotonic priority ordering, $\tau_4$ has the lowest priority and hence the scheduler waits for the final NPR of $\tau_4$ to complete instead of preempting $\tau_2$ (and hence requiring no preemption). When considering EPA, under G-LP-EDF with EPA, $\tau_1$ will preempt $\tau_3$ since it is the first preemption point available. Now since $\tau_3$ has an earlier absolute deadline than $\tau_2$, $\tau_3$ will preempt $\tau_2$ at the next preemption point of $\tau_2$. On the other hand, under G-LP-FPS with EPA, $\tau_1$ preempts $\tau_3$, but $\tau_3$ does not preempt $\tau_2$ due to its fixed (higher) priority when compared to $\tau_3$.

Therefore, for applications in which run-time preemptions are
directly or indirectly harmful, such as in the case of safety-critical system or energy constrained systems, it is best to use G-LP-FPS since it generates fewer number of preemptions than G-LP-EDF at runtime.

5.5 Discussions

In this section, we discuss the implications of choosing specific preemption approaches (EPA vs. LPA) on real-time systems design. Note that the performance of different choices of schedulers and the approaches to preemption differs based on the specific hardware and application (task parameters, cache access patterns etc), and hence we restrict the current discussion to a general case.

5.5.1 Preemption Overhead Accounting

As seen from our experiments, EPA performs worse than fully preemptive scheduling and LPA in terms of the number of preemptions.
However, EPA is still better when compared to fully preemptive schemes since the preemption points can be placed at favorable locations in the tasks’ code and/or the associated overheads can be accounted efficiently. On the other hand, under fully preemptive schemes, since the preemptions can happen arbitrarily, pessimistic approximations must be typically made. When accounting for preemption overheads, under both EPA and LPA, the effects of all the preemption points need to be factored in. This is because a preemption can occur at any of the identified preemption points at run-time. For example, Ward, Thekkilakattil and Anderson [53] presented a generic preemption overhead accounting scheme that “distributes” the overhead between the preempting and the preempted tasks. This method assumed the worst case preemption related overheads at every preemption point. However, when improving the estimation of preemption overheads at the preemption points by considering period relationships, the assumption that tasks with larger periods cannot preempt the ones with shorter periods does not hold any more (as seen from example 17). Hence, accurate preemption overhead estimation and accounting strategies may differ between EPA and LPA.

5.5.2 Energy Consumption

As observed by Bastoni et al. [2] arbitrary preemptions can be very costly (larger than 1 ms in terms of time when there is no high contention and ill-defined if there is contention). Consequently, under fully preemptive scheduling (EDF or FPS), the energy consumption resulting from these overheads can be significantly high [76]. The increased run-time overheads under fully preemptive scheduling also reduce the effectiveness of dynamic power management techniques [85] since the available run-time slack in the schedule is negligible in the worst case. On the other hand, under both the variants of global limited preemptive scheduling (and especially LPA) since the preemption points can be decided beforehand, they can be placed at locations where the temporal overheads (that influence the associated energy consumption) are acceptable. However, since the run-time
number of preemptions are higher under EPA when compared to LPA, the run-time overheads are also higher, and hence EPA is less amenable to energy savings using dynamic power management schemes.

### 5.5.3 Contention for Resources

Preemptions can result in high contention for resources such as caches and buses. Recall that preemption and migration delays can be ill-defined if there is high contention for shared caches [2]. In this context, overheads due to contention depend on the preemption points, e.g., preemptions at locations where cached data is less, when compared to locations where the cached data is more, typically implies less contention since there is only lesser data to be reloaded. It also depends on the number of preemptions; large number of preemptions potentially requires frequent cache and bus accesses, increasing contention in the system. Fully preemptive scheduling performs poorly if the preemptions happen at locations that significantly increases contention. EPA is also expected to have a degraded performance compared to LPA since the run-time number of preemptions are significantly high (even though they can be controlled offline by efficient preemption point placement strategies), as a result of which contention for resources is also potentially high. Therefore, LPA is the most suited in the context of minimizing contention in the system since it has the least number of run-time preemptions.

### 5.5.4 Mixed Criticality Scheduling

The foundation of mixed-criticality systems is based on the premises that different tasks have different levels of assurance on their WCET [86][87]. Higher criticality tasks which have/require high levels of assurances typically have over-approximated WCETs to account for e.g., preemption related overheads. The schedulability of lower criticality tasks depend on the actual execution time of higher criticality tasks [87]. There is a clear link between run-time preemptions on higher criticality tasks and their actual execution times.
Fewer run-time preemptions imply lower actual execution time, compared to its WCET, and hence lesser probability that the system switches to a higher criticality level. In this context, clearly, LPA is expected to perform well as it generates fewer run-time preemptions, implying reduced probability of switching to higher criticality levels consequently guaranteeing good service to lower criticality tasks. Our evaluations highlight the importance of the approach to preemptions on the probability of a mixed-criticality system switching to higher criticality levels.

The trade-offs between priority assignment, preemption point placement, preemption approach (EPA vs. LPA), energy aware and mixed criticality scheduling present interesting open problems.

5.6 Chapter Summary

In this chapter, we empirically investigated the preemptive behavior of G-LP-EDF and G-LP-FPS under eager and lazy preemption approaches, in addition to G-P-FPS and G-P-EDF, varying a wide range of parameters. Our investigations reveal a number of interesting observations with respect to the observed number of preemptions under the different paradigms. In particular:

1. We show that limited preemptive scheduling does not necessarily reduce the number of preemptions under global scheduling; together with EPA it may generate more preemptions than fully preemptive scheduling.

2. We show that the well known observation regarding the preemptive behavior of EDF on uniprocessors generalizes to multiprocessors; G-P-EDF generates fewer preemptions than G-P-FPS.

3. We also show that the reduction in preemptions observed with EDF on uni- and multiprocessors, however, does not generalize to global limited preemptive scheduling; G-LP-EDF suffers from more preemptions than G-LP-FPS.
4. Our experiments show that G-LP-FPS with LPA suffers from the least number of preemptions.

Our observations can be applied to a wide range of contexts like energy aware and mixed-criticality scheduling. For example, in the case of mixed-criticality systems, a lazy approach to preemption seems to be beneficial since it reduces the probability of the system switching to a higher criticality level due to WCET overruns resulting from increased preemption related overheads.

Note: This chapter is based on the paper An Empirical Investigation of Eager and Lazy Preemption Approaches in Global Limited Preemptive Scheduling, Abhilash Thekkilakattil, Kaiqian Zhu, Yonggao Nie, Radu Dobrin and Sasikumar Punnekkat, The 21st International Conference on Reliable Software Technologies (Ada-Europe), Pisa, Italy, June, 2016.
A point of great importance would be first to know what is the capacity of the earth?

The preemptive scheduling paradigm is known to strictly dominate the non-preemptive scheduling paradigm with respect to feasibility on uniprocessors. On the other hand, preemptively scheduling real-time tasks on uniprocessors, unlike non-preemptive scheduling, may lead to unschedulability due to, e.g., preemption related overheads. The limited preemptive scheduling paradigm, which is a generalization of preemptive and non-preemptive paradigms, has, however, the potential to reduce the preemption related overheads while enabling high processor utilization.

In this chapter, we focus on the characterization of the effects of

1This quote from Nikola Tesla's article On Light and Other High Frequency Phenomena is used here to highlight the need to reason about performance boundaries of, in this context, scheduling algorithms, e.g., using resource augmentation.
Chapter 6

Resource Augmentation for Uniprocessor Limited Preemptive Scheduling

A point of great importance would be first to know what is the capacity of the earth?\(^1\)

The preemptive scheduling paradigm is known to strictly dominate the non-preemptive scheduling paradigm with respect to feasibility on uniprocessors. On the other hand, preemptively scheduling real-time tasks on uniprocessors, unlike non-preemptive scheduling, may lead to unschedulability due to, *e.g.*, preemption related overheads. The limited preemptive scheduling paradigm, which is a generalization of preemptive and non-preemptive paradigms, has, however, the potential to reduce the preemption related overheads while enabling high processor utilization.

In this chapter, we focus on the characterization of the effects of

\(^1\)This quote from Nikola Tesla’s article *On Light and Other High Frequency Phenomena* is used here to highlight the need to reason about performance boundaries of, in this context, scheduling algorithms, *e.g.*, using resource augmentation.
increasing the computational resources on the limited preemptive feasibility of real-time tasks in order to quantify the sub-optimality of limited preemptive scheduling on uniprocessors. Specifically, we first derive the required processor speed-up bound that guarantees limited preemptive feasibility of any uniprocessor feasible taskset. Secondly, we demonstrate the applicability of the results in the context of controlling preemption related overheads while minimizing the required processor speed-up. In particular, we identify the preemptive behavior that minimizes preemption-related overheads, as well as derive the optimal processor speed associated with it. Finally, we examine the consequences of having more processors on limited preemptive feasibility and derive the bound on the number of processors that guarantees a specified limited preemptive behavior for any uniprocessor feasible real-time taskset.

The contribution essentially bridges the preemptive and non-preemptive real-time scheduling paradigms by providing significant theoretical results building on the limited preemptive scheduling paradigm, as well as provides analytical inputs to developers in order to perform various trade-offs, e.g., code refactoring, to control the preemptive behavior of real-time tasks.

### 6.1 Models and Notations

In this section, we introduce the notations used in the rest of the chapter whilst describing the task model, scheduling model, and the execution time model.

#### 6.1.1 Task and Processor model

We consider a set of sporadic real-time tasks \( \Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\} \) executing on a uniprocessor platform. Each \( \tau_i \) is characterized by a minimum inter-arrival time \( T_i \), a worst case execution time \( C_i^S \) at processor speed \( S \), and a relative deadline \( D_i \). We assume that tasks’ worst case execution times are equal to their worst case execution
requirements on a speed $S = 1$ processor. Let the length of the longest critical section of a task $\tau_i$, on a processor of speed $S$, be denoted by $CS_i^S$. We assume that the tasks are indexed according to the increasing order of their deadlines, which means that $D_{\min} = D_1$. We assume that every task $\tau_i$ has $m_i$ optimal preemption points [14][88] within its execution, where the $m_i^{th}$ point denotes the end of the task execution. Let $q_{i,j}^S$, $j = 1...m_i$ denote the length of the execution of $\tau_i$ from its start up to the $j^{th}$ optimal preemption point on a processor at speed $S$. In order to focus on the theoretical consequence of resource augmentation on the preemptive behavior of the taskset, and for the sake of clarity of presentation, we assume negligible preemption related overheads at these optimal preemption points. However such an assumption does not affect the generality of our results because the preemption related overheads can be accounted during the placement of the preemption points, e.g., as done by [50].

Let $\beta_i^S$ denote the blocking tolerance of $\tau_i$, which is the largest time for which $\tau_i$ can be blocked without causing any deadline miss [50]. Also, let $B_i^S \leq \beta_i^S$ denote the largest time for which $\tau_i$ is effectively blocked at run-time. LCM denotes the Least Common Multiple of the time periods of all the tasks in the set. The utilization $U_i^S$ of a task $\tau_i$ executing on a processor at speed $S$ is defined as $U_i^S = \frac{C_i^S}{T_i}$ and the utilization of the entire taskset is given by $U^S = \sum_{i=1}^{n} U_i^S$. The demand bound function of a task $\tau_i$, on a processor of speed $S$, during a time interval $[0, t]$ is given by Baruah [43],

$$DBF_i^S(t) = \max \left( 0, 1 + \left[ \frac{t - D_i}{T_i} \right] \right) C_i^S$$

For example, $DBF_i^1(t)$ denotes the cumulative processor time requested by $\tau_i$ during a time interval $[0, t]$ on a processor of speed $S = 1$. Additionally, the density of task $\tau_i$ is given by $\delta_i = \frac{C_i^1}{D_i}$, and the total density of the taskset by $\delta_{\text{tot}} = \sum_{\forall \tau_i} \delta_i$. In the following, we formally define a specified limited preemption length.
Definition 7. A specified limited preemption length of a task \( \tau_i \) is defined as the maximum specified length of the non-preemptive regions of \( \tau_i \).

For example, a specified limited preemption length may guarantee a specified upper-bound on the preemption related cost on \( \tau_i \) that guarantees schedulability. We denote the specified limited preemption length of a task \( \tau_i \) at speed \( S \) by \( L_i^S \). A specified limited preemption length can also be denoted by \( L_i \) in case it does not change with the processor speed.

Definition 8. A limited preemption requirement on a task \( \tau_i \) is defined as the requirement that the task \( \tau_i \) executes non-preemptively for a duration given by the specified limited preemption length.

A limited preemption requirement on any task \( \tau_i \) is said to be feasible if \( \tau_i \) can execute non-preemptively for the specified limited preemption length, consequently guaranteeing the feasibility of the specified limited preemptive behavior for \( \tau_i \).

Definition 9. The feasibility of a specified limited preemptive behavior of a taskset is defined as the existence of a real-time schedule that guarantees the non-preemptive execution of every task for the specified limited preemption length, while ensuring the absence of deadline misses in the schedule.

The feasibility of the specified limited preemptive behavior of the taskset can be guaranteed by speeding up the processor to control the length of the non-preemptive regions.

6.1.2 Uniprocessor LP-EDF Scheduling Model

We assume the Earliest Deadline First (EDF) scheduling paradigm. We assume that, whenever a higher priority job is released during the execution of a lower priority job of \( \tau_i \), instead of immediately preempting the job of \( \tau_i \), the scheduler blocks the higher priority job for \( Q_i^S \) time units on a processor of speed \( S \). Alternately, the tasks can
be composed of several non-preemptable chunks of code, whose maximum length is $Q^S_i$. Here, $Q^S_i$ is the length of the largest non-preemptive region of $\tau_i$ derived from the task attributes [9] [89]. Consequently, the maximum number of times the task $\tau_i$ can be preempted when a processor of speed $S$ is used, is given by $\left\lceil \frac{C^S_i}{Q^S_i} \right\rceil - 1$. If $L^S_i > Q^S_i$, then the system is not schedulable since the specified limited preemption length is greater than the bound on the maximum possible limited preemption length.

We assume a work conserving scheduler, i.e., the scheduler does not idle the processor when there are active tasks awaiting the processor. We leverage on the optimality of EDF under preemptive [19] and non-preemptive [4] paradigms to study the processor speed-up required to guarantee the feasibility of a required limited preemptive behavior of real-time tasks.

### 6.1.3 Execution Time Model

In this chapter we focus mainly on the theoretical consequences of resource augmentation, specifically processor speed-up, on the preemptive behavior of real-time tasks. We adopt the execution time model proposed by Marinoni and Buttazzo [90]. In this model, the execution time of each task consists of two parts— one that is processor speed dependent and the other that is processor speed independent. Let $\phi_i$ denote the fraction of execution time of $\tau_i$ that scales linearly with the processor speed. Consequently, the fraction $(1 - \phi_i)$ of the execution time of $\tau_i$ does not scale with the processor frequency.

To ease the readability, and without loss of generality, we assume that the taskset is initially executing on a processor of speed $S = 1$. We assume that, if $C^1_i$ is the execution time at speed $S = 1$, the task execution time of $\tau_i$ scales as follows:

$$C^S_i = \frac{\phi_i C^1_i}{S} + (1 - \phi_i)C^1_i$$
Such a model also allows us to use processor speed-up factors and processor speeds interchangeably. Changing the processor speed from $S = 1$ to $S = a$, is equivalent to speeding up the processor by a factor of ‘$a$’. Finally, we define

$$\phi = \min_{\forall \tau_i \in \Gamma} (\phi_i)$$

Therefore, in any time interval $t$, at least $\phi \sum_i^n DBF_i^1(t)$ units of execution scales with the processor speed.

### 6.2 Recapitulation of Feasibility Analysis of Uniprocessor Real-time Systems

The limited preemptive scheduling model proposed by Baruah [9] can be seen as generalizations of non-preemptive and preemptive scheduling models as they can simulate a preemptive behavior ranging from non-preemptive to fully preemptive. If $Q_i^S$ is set equal to 0 for all $\tau_i$, the system simulates a fully preemptive model, while if $Q_i^S$ is set equal to $C_i^S$, the system simulates a fully non-preemptive model [91]. In our approach we build on Baruah’s [9] model to study the feasibility of preemptive, non-preemptive, and limited preemptive scheduling of real-time tasks, when the amount of available resources change.

Let us now recall some previously published theoretical results presented by Jeffay [4] (in Section 4 of their paper) and Bertogna et al. [50] (in Section IV of their paper). Due to sustainability of the EDF scheduling scheme[92], these theorems can be generalized to a processor of speed $S$, ($S \geq 1$). A real-time taskset is feasible if the cumulative processor time requested by the set of tasks during any time interval does not exceed the size of that time interval [43]. The following theorems presented in a revised uniform format, determines the feasibility of uniprocessor real-time scheduling.

**Theorem 6.2.1.** (from [50]) A taskset is feasible on a speed $S$ processor, if and only if, $\forall i \in [1, n]$, $\beta_i^S \geq 0$
Chapter 6. Resource Augmentation for Uniprocessor Limited Preemptive Scheduling

Such a model also allows us to use processor speed-up factors and processor speeds interchangeably. Changing the processor speed from $S = 1$ to $S = a$, is equivalent to speeding up the processor by a factor of $a$. Finally, we define

$$\phi = \min \forall \tau \in \Gamma (\phi_i)$$

Therefore, in any time interval $t$, at least $\phi \sum_{i=1}^{n} DBF_j(t) \geq \text{units of execution}$ scales with the processor speed.

6.2 Recapitulation of Feasibility Analysis of Uniprocessor Real-time Systems

The limited preemptive scheduling model proposed by Baruah [9] can be seen as generalizations of non-preemptive and preemptive scheduling models as they can simulate a preemptive behavior ranging from non-preemptive to fully preemptive. If $Q_{S_i}$ is set equal to $0$ for all $\tau$, the system simulates a fully preemptive model, while if $Q_{S_i}$ is set equal to $C_{S_i}$, the system simulates a fully non-preemptive model [91].

In our approach we build on Baruah’s [9] model to study the feasibility of preemptive, non-preemptive, and limited preemptive scheduling of real-time tasks, when the amount of available resources change. Let us now recall some previously published theoretical results presented by Jeffay [4] (in Section 4 of their paper) and Bertogna et al. [50] (in Section IV of their paper). Due to sustainability of the EDF scheduling scheme [92], these theorems can be generalized to a processor of speed $S$ ($S \geq 1$). A real-time taskset is feasible if the cumulative processor time requested by the set of tasks during any time interval does not exceed the size of that time interval [43]. The following theorems presented in a revised uniform format, determines the feasibility of uniprocessor real-time scheduling.

Theorem 6.2.1. (from [50]) A taskset is feasible on a speed $S$ processor, if and only if,

$$\beta_{S_i} \geq 0 \quad \forall i \in [1,n]$$

where,

$$\beta_{S_i} = \min_{D_i \leq t < D_{i+1}} \left( t - \sum_{j=1}^{n} DBF_j^S(t) \right)$$

(6.1)

$$t = kT_j + D_j, \forall k \in N, j \in [1, n]$$

In the above theorem, $D_{n+1}$ is set as

$$D_{n+1} = \min (LCM, P)$$

Where,

$$P = \max \left\{ D_1, D_2, \ldots, D_n, \sum_{i=1}^{n} (T_i - D_i)U_i^S \right\} / 1 - U^S$$

When the $\beta_{S_i} = 0, \forall i \in [1,n]$, the taskset is feasible only under a fully preemptive scheduling scheme.

The above theorem can be used to determine the feasibility of limited preemptive scheduling on a processor at speed $S$, and is stated by the following theorem.

**Theorem 6.2.2.** (from [50]) A taskset is feasible under limited preemptive scheduling on a speed $S$ processor if, $\forall i \in [1, n]$,

$$B_{S_i} \leq \beta_{S_i}$$

where the blocking tolerance $\beta_{S_i}$ is given by equation 6.1 and $B_{S_i}$ is the largest blocking actually experienced by $\tau_i$ due to the limited preemptions on a processor of speed $S$.

The bound $Q_{S_k}^S$ on the length of the non-preemptive region of a task $\tau_k$ on a processor of speed $S$ is given by the following theorem.

**Theorem 6.2.3.** (from [50]) A taskset is feasible under limited preemptive scheduling on a speed $S$ processor if, $\forall k \in [1, n]$,

$$Q_{S_k}^S = \min_{1 \leq i < k} \beta_{S_i}^S$$
The task can execute entirely non-preemptively if $Q_k^S$ is greater than or equal its execution time $C_k^S$. Hence, we can use the above theorem to state the non-preemptive feasibility of the taskset, \textit{i.e.}, whether it is possible to find a non-preemptive schedule, as follows:

**Theorem 6.2.4.** (from [4]) A taskset is feasible under non-preemptive scheduling on a speed $S$ processor if, $\forall k \in [1, n],$

$$C_k^S \leq Q_k^S$$

### 6.3 Speed Augmentation for Limited Preemptive Scheduling

In this section, we examine the consequences of having a faster processor on the limited preemptive scheduling of real-time tasks. In modern processors, due to effects of \textit{e.g.}, the memory wall [93], the entire task execution times may not scale linearly with the processor speed. We consider the generalized execution time model proposed by Marinoni and Buttazzo [90] (that is more realistic) in which only a part of the WCETs scale linearly with the processor speed. First we show, in the following example, that in general it is not practical to use faster processors to achieve limited preemptive EDF feasibility. We do this by constructing a taskset for which the use processor speed-up to achieve limited preemptive feasibility is not practical.

**Example 18.** Consider two sporadic tasks $A$ and $B$, having execution times $C_A^1 = 5$ and $C_B^1 = 10$, $\phi_A = 0.4$ and $\phi_B = 0.4$, and deadlines $T_A = D_A = 8$ and $T_B = D_B = 30$. This means that it is not possible to speed up 3 units of execution of task $A$ and 6 units of execution of task $B$. Assume that the limited preemption requirement on task $B$ is 6. Consider the scenario, as shown in Figure 6.1, when a job of task $B$ is released at time instant $t$ and has immediately started its execution—at time $t+1$ it has finished only 1 unit of execution. If a job of $A$ is released at time $t+1$, clearly $A$ has a higher priority than $B$. When $A$ tries to preempt $B$, $B$ immediately starts executing non-preemptively. If task
The task can execute entirely non-preemptively if $Q_S k$ is greater than or equal to its execution time $C_S k$. Hence, we can use the above theorem to state the non-preemptive feasibility of the taskset, i.e., whether it is possible to find a non-preemptive schedule, as follows:

**Theorem 6.2.4.** *(from [4] )* A taskset is feasible under non-preemptive scheduling on a speed $S$ processor if, $\forall k \in [1, n]$, $C_S k \leq Q_S k$.

### 6.3 Speed Augmentation for Limited Preemptive Scheduling

In this section, we examine the consequences of having a faster processor on the limited preemptive scheduling of real-time tasks. In modern processors, due to effects of, for example, the memory wall [93], the entire task execution times may not scale linearly with the processor speed. We consider the generalized execution time model proposed by Marinoni and Buttazzo [90] (that is more realistic) in which only a part of the WCETs scale linearly with the processor speed. First we show, in the following example, that in general it is not practical to use faster processors to achieve limited preemptive EDF feasibility. We do this by constructing a taskset for which the use processor speed-up to achieve limited preemptive feasibility is not practical.

**Example 18.** Consider two sporadic tasks $A$ and $B$, having execution times $C_1 A = 5$ and $C_1 B = 10$, $\phi A = 0.4$ and $\phi B = 0.4$, and deadlines $T A = D A = 8$ and $T B = D B = 30$. This means that it is not possible to speed up $3$ units of execution of task $A$ and $6$ units of execution of task $B$.

Consider the scenario, as shown in Figure 6.1, when a job of task $B$ is released at time instant $t$ and has immediately started its execution— at time $t + 1$ it has finished only $1$ unit of execution. If a job of $A$ is released at time $t + 1$, clearly $A$ has a higher priority than $B$. When $A$ tries to preempt $B$, $B$ immediately starts executing non-preemptively. If task $B$ immediately starts executing the region of code that is independent of the processor speed, clearly the job of task $A$ will miss its deadline. In this case, no amount of speeding up the processor helps because, no matter what the processor speed is, $3$ units of execution of task $A$ and $6$ units of execution of task $B$ cannot execute faster, and hence the non-preemptive execution of task $B$ for $6$ units will lead to a deadline miss on task $A$ at time $t + 9$.

The example shows the difficulty of obtaining a speed-up bound for limiting preemptions. However, if the limited preemption requirement of a task is less than a certain fraction of the shortest deadline (whose exact value is presented later in this section), we can obtain the resource augmentation bound even if only a part of the execution time scales linearly with the processor speed. We then present resource augmentation bounds for the specific case in which it is possible to speed-up the entire task WCET.

**Theorem 6.3.1.** The processor speed $S_i$ that guarantees the feasibility of a limited preemption requirement $L_i$ for any task $\tau_i \in \Gamma$ is given by,

\[
S_i = \max_{D_1 \leq t < D_i} \left\{ \frac{\phi \sum_{j=1}^{n} DBF_{j}^{1}(t)}{t - L_i - (1 - \phi) \sum_{j=1}^{n} DBF_{j}^{1}(t)} \right\}
\]
Proof. The maximum length of the non-preemptive region for $\tau_i$ at speed 1 is given by [9],

$$Q_i^1 = \min_{D_1 \leq t < D_i} \left\{ t - \sum_{j=1}^{n} DBF_j^1(t) \right\}, \forall t, D_1 \leq t < D_i$$

Our aim is to find the processor speed $S_i$ that guarantees the feasibility of a limited preemption requirement $L_i$. Suppose,

$$L_i > Q_i^1 = t - \sum_{j=1}^{n} DBF_j^1(t)$$

We know that, of the total demand bound in any interval $t$, at least $\phi$ percentage scales with the processor speed. Thus, we can speed-up this part of the task executions to guarantee the limited preemptive execution of $\tau_i$ for $L_i$ units, i.e., $\forall t, D_1 \leq t < D_i$,

$$L_i \leq t - \left\{ \frac{\phi \sum_{j=1}^{n} DBF_j^1(t)}{S_i} + (1 - \phi) \sum_{j=1}^{n} DBF_j^1(t) \right\}$$

Hence, $\forall t, D_1 \leq t < D_i$,

$$S_i L_i \leq S_i t - \left\{ \phi \sum_{j=1}^{n} DBF_j^1(t) + S_i(1 - \phi) \sum_{j=1}^{n} DBF_j^1(t) \right\}$$

Solving for $S_i$, we get, $\forall t, D_1 \leq t < D_i$,

$$S_i \geq \left\{ \frac{\phi \sum_{j=1}^{n} DBF_j^1(t)}{t - L_i - (1 - \phi) \sum_{j=1}^{n} DBF_j^1(t)} \right\}$$

i.e.,

$$S_i = \max_{D_1 \leq t < D_i} \left\{ \frac{\phi \sum_{j=1}^{n} DBF_j^1(t)}{t - L_i - (1 - \phi) \sum_{j=1}^{n} DBF_j^1(t)} \right\}$$
6.3 Speed Augmentation for Limited Preemptive Scheduling

Consequently, we find the upper-bound on the required processor speed that guarantees a specified limited preemption requirement $L_i$ for any task $\tau_i \in \Gamma$.

**Lemma 6.3.1.** The upper-bound on the minimum processor speed $S_i$ that guarantees the feasibility of a limited preemption requirement $L_i$ for any task $\tau_i \in \Gamma$, during a time interval $t$ is given by,

$$S_i \leq \frac{y}{y - \frac{1}{\phi}}$$

where, $y = \frac{t}{L_i}, \forall t \in [D_i, D_i)$.

**Proof.** We know from theorem 6.3.1 that,

$$S_i = \max_{D_1 \leq t < D_i} \left\{ \frac{\phi \sum_{j=1}^{n} DBF_j^1(t)}{t - L_i - (1 - \phi) \sum_{j=1}^{n} DBF_j^1(t)} \right\}$$

Since we have assumed that the taskset is feasible, the upper-bound on the value of $\sum_{j=1}^{n} DBF_j^1(t)$ is $t$. Hence,

$$S_i \leq \left\{ \frac{\phi t}{t - L_i - (1 - \phi) t} \right\} \Rightarrow S_i \leq \frac{t}{\phi t - L_i}$$

Finally, substituting $y = \frac{t}{L_i}$,

$$S_i \leq \frac{\phi y}{\phi y - 1}$$

$$\Rightarrow S_i \leq \frac{y}{y - \frac{1}{\phi}}$$

In order to derive the actual value of the upper-bound on the required processor speed that guarantees the feasibility of a limited preemption requirement $L_i$, for any $\tau_i \in \Gamma$, during any time interval $t$ we consider the following two cases:
Chapter 6. Resource Augmentation for Uniprocessor Limited Preemptive Scheduling

Case 1: \( y \geq \left( 1 + \frac{1}{\phi} \right) \)

Case 2: \( 0 < y < \left( 1 + \frac{1}{\phi} \right) \)

In the following lemma, we bound the speed-up required for case 1, i.e., when \( y \geq \left( 1 + \frac{1}{\phi} \right) \).

Lemma 6.3.2. The upper-bound on the minimum processor speed \( S_i \) that guarantees the feasibility of a limited preemption requirement \( L_i \) for any task \( \tau_i \in \Gamma \), such that \( y \geq \left( 1 + \frac{1}{\phi} \right) \), is given by

\[
S_i \leq \left( 1 + \frac{1}{\phi} \right)
\]

where \( y = \frac{t}{L_i} \) and \( t \in [D_1, D_i) \)

Proof. Evaluating the limit of the equation in lemma 6.3.1 at \( y = \left( 1 + \frac{1}{\phi} \right) \), we get,

\[
S_i = \left( 1 + \frac{1}{\phi} \right)
\]

Evaluating the limit using l’Hopital’s rule, as \( y \) tends to infinity (\( \infty \)), we get,

\[
S_i = 1
\]

Therefore, for any value of \( y \in \left[ \left( 1 + \frac{1}{\phi} \right), \infty \right] \),

\[
S_i \leq \left( 1 + \frac{1}{\phi} \right)
\]

In the case 2 above, in general, it is not practical to use processor speed-up to guarantee a specified limited preemptive behavior because, in the worst case, the non-preemptive region does not scale with the processor speed (refer Example 18). However, if \( \phi = 1 \), it is possible to bound the required speed-up. This is derived in the following subsection.
6.3 Speed Augmentation for Limited Preemptive Scheduling

6.3.1 Speed-up Bound under Restricted Execution Time Model

In this subsection, for completeness, we consider the special case in which the entire task execution time scales linearly with processor speed.

When the entire task execution time scales with the processor speed, i.e., \( \phi = 1 \), we can restate Lemma 6.3.2 as follows:

**Lemma 6.3.3.** The upper-bound on the minimum processor speed \( S_i \) that guarantees the feasibility of a limited preemption requirement \( L_i \) for any task \( \tau_i \in \Gamma \), such that \( y \geq 2 \) and \( \phi = 1 \), is given by

\[
S_i \leq 2
\]

Furthermore, under the assumption that \( \phi = 1 \), Case 2 presented in the previous section can be split into two sub-cases as follows:

**Case 2.a:** \( 1 \leq y < 2 \)

**Case 2.b:** \( 0 < y < 1 \)

In the following lemma, we derive the speed-up bound for Case 2.a described above:

**Lemma 6.3.4.** The upper-bound on the minimum speed \( S_i \) that guarantees the feasibility of a specified limited preemption requirement \( L_i \) for any task \( \tau_i \), such that \( 1 \leq y < 2 \) and \( \phi = 1 \), is given by

\[
S_i \leq 2
\]

where \( y = \frac{t}{L_i} \) and \( \forall t \in [D_1, D_i) \).

**Proof.** On a unit speed processor \( t \) clock ticks are available in any time interval of length \( t \). In the worst case, the processor is fully occupied during \( t \), and hence the limited preemption requirement \( L_i \) cannot be feasibly executed. Let us assume an increase in the processor speed by a factor of 2. This implies that within an interval of time \( t \), there are in
effect $t' = 2t$ clock ticks. In this case the limited preemption requirement $L_i$ can be successfully executed without causing any deadline miss since $2t \geq L_i + t$ (remember that we are considering the case $t \geq L_i$). Therefore, when $1 \leq y < 2$, the tighter upper-bound is given by 2.

**Lemma 6.3.5.** The upper-bound on the minimum speed $S_i$ that guarantees the feasibility of a limited preemption requirement $L_i$ for any task $\tau_i$, such that $0 < y < 1$ and $\phi = 1$, is given by

$$S_i \leq \frac{2L_i}{t}$$

where $y = \frac{t}{L_i}$ and $\forall t \in [D_1, D_i)$.

**Proof.** On using a processor that is $S = \frac{L_i}{t}$ times faster, the number of clock ticks in the time interval $t$ increases from $t$ to $t' = t \times \frac{L_i}{t} = L_i$. Consequently, we can execute the original demand of length no greater than $t$, and a part $L_i - t$ of the limited preemption requirement $L_i$ at speed $S = \frac{L_i}{t}$. The remaining limited preemption requirement that cannot be executed is $L_i' = t$. On a processor of speed $S = \frac{L_i}{t}$, since $t < L_i$, we effectively have $\frac{t'}{t_i} = \frac{L_i}{t} > 1$. Using Lemma 6.3.3 and Lemma 6.3.4, the speed-up required denoted by $S_i'$ is $S_i' \leq 2$. Remember that we had already increased the processor speed by $\frac{L_i}{t}$, therefore, the tighter upper-bound $S_i$ is:

$$S_i \leq \frac{2L_i}{t}$$

In the following, we present a theorem that unifies the results in Lemma 6.3.3, Lemma 6.3.4 and Lemma 6.3.5 to obtain an integrated result on how the limited preemptivity changes with respect to processor speed.
6.3 Speed Augmentation for Limited Preemptive Scheduling

**Theorem 6.3.2.** The upper-bound on the minimum processor speed $S_i$ that guarantees the feasibility of a limited preemption requirement $L_i$ for any task $\tau_i \in \Gamma$ when $\phi = 1$ is given by, $\forall t > 0$, 

$$S_i \leq 2 \max \left( 1, \frac{L_i}{t} \right)$$

where, $y = \frac{t}{L_i}$.

**Proof.** When $\phi = 1$, the entire execution time of every task scales linearly with the processor speed. We know that $L_i$ is bounded by the maximum of the execution times of the tasks in the taskset, at speed $S = 1$ (i.e., its fully non-preemptive execution). Similarly, since $t$ is lower-bounded by the shortest deadline, we obtain $\frac{t}{L_i} > 0$.

When $y \geq 2$, according to Lemma 6.3.3, we obtain:

$$S_i \leq 2 \max \left( 1, \frac{1}{y} \right) = 2 \times 1 = 2$$

Similarly, when $1 \leq y < 2$, according to Lemma 6.3.4, we obtain

$$S_i \leq 2 \max \left( 1, \frac{1}{y} \right) = 2 \times 1 = 2$$

Finally, when $0 < y < 1$ the speed-up required, according to Lemma 6.3.5, is

$$S_i \leq 2 \max \left( 1, \frac{1}{y} \right) = \frac{2L_i}{t}$$

Therefore, it follows from lemmas 6.3.3, 6.3.4, and 6.3.5 that the speed-up required is bounded as follows:

$$S_i \leq 2 \max \left( 1, \frac{L_i}{t} \right)$$

The above theorem is valid for any time interval. However, in the following, we show that the largest speed-up is obtained at the shortest relative deadline.
Corollary 6.3.1. The upper-bound on the minimum processor speed $S_i$ that guarantees the feasibility of a limited preemption requirement $L_i$ for any task $\tau_i \in \Gamma$, when $\phi = 1$, is given by

$$S_i \leq 2 \max \left( 1, \frac{L_i}{D_{min}} \right)$$

The proof is intuitive as the value of $t$ for which the blocking from $L_i$ is maximum, is the smallest value of $t$ given by the shortest relative deadline $t = D_{min}$ (remember that $D_{min} = D_1$). It is when $t = D_{min}$ that the value $\frac{L_i}{t}$ is maximized. We have thus derived the upper-bound on the processor speed that guarantees the feasibility of a specified limited preemptive behavior for any task $\tau_i \in \Gamma$. The above result can be extended to derive the bound on the speed-up that guarantees non-preemptive execution of the entire task set $\Gamma$ for a duration of at least $L_{max}$, where $L_{max} = \max_{\forall \tau_i} L_i$.

Corollary 6.3.2. The upper-bound on the minimum processor speed $S$ that guarantees the feasibility of the specified limited preemption requirement $L_{max}$ for any taskset $\Gamma$, when $\phi = 1$, is given by

$$S \leq 2 \max \left( 1, \frac{L_{max}}{D_{min}} \right)$$

where $L_{max} = \max_{\forall \tau_i} (L_i)$.

Therefore, the speed-up that guarantees the non-preemptive feasibility of the entire taskset is given by the following.

Corollary 6.3.3. The upper-bound on the minimum processor speed $S$ that guarantees the non-preemptive EDF feasibility of any taskset $\Gamma$, when $\phi = 1$, is given by

$$S \leq 2 \max \left( 1, \frac{C_{max}}{D_{min}} \right)$$

where $C_{max} = \max_{\forall \tau_i} (C_i^1)$.
The executions of non-preemptive chunks of the tasks are independent of each other ([9]). Consequently, the processor speed-up bound that guarantees non-preemptive feasibility of the task with largest execution time will also guarantee the non-preemptive execution of the entire taskset.

**Sub-optimality:** The sub-optimality of limited preemptive (non-preemptive) scheduling when compared to an optimal uniprocessor preemptive scheduling scheme, with respect to any taskset $\Gamma$, can be quantified in terms of processor speed-up bound given in Corollary 6.3.2 (Corollary 6.3.3).

![Figure 6.2: Speed vs. feasibility.](image)

**Resource availability vs. limited preemptive feasibility:** We illustrate the change in preemptive behavior with respect to processor speed, under the assumption that the entire WCETs scales linearly with the processor speed, in Figure 6.3. We refer to it as the *feasibility bucket* that shows how feasibility changes with respect to processor speed. Note that the Figure 6.3 is an attempt at visually representing the results presented in this chapter in an intuitive manner that reflects the dominance of preemptive real-time scheduling over limited preemptive scheduling with specified NPRs, as well as non-preemptive scheduling.
128 Chapter 6. Resource Augmentation for Uniprocessor Limited Preemptive Scheduling

Figure 6.3: Speed vs. LP feasibility.

The base of the bucket in Figure 6.2 represents the set of all uniprocessor feasible real-time tasks on a processor of speed $S = 1$. Obviously, on increasing the processor speed to $S = x$, more tasksets become uniprocessor feasible. Consequently, the original set of tasks that was feasible at speed $S = 1$, becomes a subset of the tasksets feasible at speed $S = x$. At this point, we additionally consider the limited preemptive EDF feasibility of all uniprocessor feasible real-time tasks. Figure 6.3 illustrates how the limited preemptive feasibility changes from fully preemptive uniprocessor feasibility at speed $S = 1$ to the fully non-preemptive feasibility at speed $S = 2 \max \left(1, \frac{C_{\text{max}}}{D_{\text{min}}} \right)$.

6.4 Processor Augmentation for Limited Preemptive Scheduling

This chapter quantitatively compares limited preemptive scheduling with fully preemptive scheduling using the notion of resource augmentation. In the previous sections, we considered the use of processor speed-up to quantify the sub-optimality of limited
preemptive scheduling. Instead, in this section, we examine the consequences of having more number of processors on the limited preemptive feasibility (such as done by Lam and To [94] for global preemptive EDF). Specifically, we derive the processor augmentation bound that is defined as the upper-bound on the minimum number of processors on which any uniprocessor feasible taskset is guaranteed the specified limited preemptive behavior. The use of more number of processors is particularly interesting because of the widespread availability of multi-core systems that can be leveraged upon to limited preemptively schedule hard real-time tasks, while scheduling the soft and non real-time tasks in the background.

We first show that, in the worst case, the number of processors required to guarantee a specified limited preemptive behavior is equal to the number of tasks in the taskset. We then derive the processor augmentation bounds for the specific case in which the specified limited preemptive length is no larger than half the shortest deadline, and show that the upper-bound on the number of extra processor required is 3. In the following, let us consider the general case in which no restriction is placed on the specified length of the limited preemptive region.

**Lemma 6.4.1.** The minimum number of processors on which a uniprocessor feasible taskset \( \Gamma \) is guaranteed any specified limited preemptive behavior is \( n \), which is the number of tasks in \( \Gamma \).

**Proof.** We provide proof by constructing a taskset for which \( n \) processors are required to guarantee limited preemptive feasibility—remember that no restrictions exists on the values of the deadlines and specified limited preemption lengths.

Consider a taskset \( \Gamma \) with each \( \tau_i, i = 2, 3, \ldots, n \), having a specified limited preemption length \( L_i \) such that \( L_i > D_{i-1} \). In this case it is easily seen that if any two tasks are scheduled on the same processor, it is impossible to guarantee the limited preemptive execution of one of the tasks for a duration equal to the corresponding specified limited preemption length. This is because if a task \( \tau_j \) with the shorter relative
deadline is released when another task $\tau_k$ is executing, and if $\tau_k$ blocks $\tau_j$ for a duration $L_k$ then $\tau_j$ will miss its deadline since $L_k > D_j$. 

The above result shows that it is not possible to guarantee specified limited preemptive behaviors on fewer processors than the number of tasks. Therefore, in general, the increased processing capacity provided by multicore platforms cannot be leveraged to control preemptive behavior of real-time tasks using limited preemptive scheduling [9], unless each core is assigned a single task. However, the above result is derived for the worst case in which no assumptions are made about the taskset. If the largest specified limited preemption length is no more than half the shortest deadline, a tighter bound can be obtained. In the following subsection, we derive a density based test for limited preemptive scheduling, and then use this test to derive a tighter processor augmentation bound for limited preemptive feasibility.

6.4.1 Density Based Test for Uniprocessor Limited Preemptive Scheduling

There exists utilization/density based tests for schedulability and feasibility of preemptive real-time tasks under various assumptions [40]. However, to our knowledge, no utilization or density based test exists for limited- and non-preemptive scheduling even under restrictive assumptions. Speeding-up the processor by a constant factor is equivalent to ensuring a bound on the processor utilization (density, in case deadlines can be less than periods).

Yao et al.’s [91] observation that there exists no least upper-bound on the processor utilization below which the non-preemptive feasibility can be guaranteed, relies on the fact that it is possible to construct tasksets with arbitrarily low utilization that are infeasible under non-preemptive scheduling. We observe that the unschedulability primarily arises because of the fact that at least one task has an execution time greater than the shortest deadline. Consequently, such a condition can be seen as a necessary unschedulability test for
non-preemptive scheduling, in particular non-preemptive EDF scheduling, of sporadic real-time tasks.

**Observation 6.4.1.** A sporadic real-time taskset $\Gamma$ is infeasible under non-preemptive EDF if,

$$\exists \tau_i \in \Gamma: C_i \geq D_{\text{min}}$$

In the following, we first derive a density based test for limited preemptive EDF feasibility of sporadic real-time tasks. The test when instantiated in the context of non-preemptive EDF provides us with a sufficient density based test for non-preemptive EDF feasibility of sporadic real-time tasks.

**Lemma 6.4.2.** A sporadic real-time taskset $\Gamma$ is feasible under limited preemptive EDF, such that every task can execute non-preemptively for at most $L_{\text{max}}$ units, if,

$$\delta_{\text{tot}} \leq 1 - \frac{L_{\text{max}}}{D_{\text{min}}}$$

where $L_{\text{max}} = \max_{\forall \tau_i \in \Gamma} (L^1_i)$.

**Proof.** A taskset $\Gamma$ is limited preemptive EDF feasible if during any time interval of length $t$, the sum of total demand bound and the largest limited preemptive region of the tasks in the taskset is less than or equal to $t$. It is known that,

$$\sum_{\forall \tau_i \in \Gamma} DBF_i(t) \leq \delta_{\text{tot}} \times t$$

Therefore, a sufficient condition to guarantee limited preemptive EDF feasibility is given by, $\forall t \geq D_{\text{min}}$

$$\delta_{\text{tot}} \times t + L_{\text{max}} \leq t \Rightarrow \delta_{\text{tot}} \leq 1 - \frac{L_{\text{max}}}{t}$$
The value of $t$ that maximizes $\frac{L_{\text{max}}}{t}$ is $t = D_{\text{min}}$, and hence, the taskset is limited preemptive EDF feasible if:

$$\delta_{\text{tot}} \leq 1 - \frac{L_{\text{max}}}{D_{\text{min}}}$$

Instantiating the above test in the context of a fully non-preemptive EDF scheduler, we get the following test for non-preemptive EDF feasibility.

**Corollary 6.4.1.** A sporadic real-time taskset $\Gamma$ is feasible under non-preemptive EDF if,

$$\delta_{\text{tot}} \leq 1 - \frac{C_{\text{max}}}{D_{\text{min}}}$$

where $C_{\text{max}} = \max_{\forall \tau_i \in \Gamma} (C^i_1)$.

The tests presented above generalizes to a utilization based test when the deadlines of the tasks are equal to their time periods. This density based test is interesting since it runs in a time polynomial in the number of tasks, when compared to the exact demand bound based tests by Jeffay et al. [4] and Baruah [9] that runs in pseudo-polynomial time. The polynomial complexity of the density based test comes at the cost of necessity i.e., the test presented above is only a sufficient condition for schedulability. The density based test is especially applicable to the liquid task model presented by Abdelzaher et al. [66] in which the shortest deadline is orders of magnitude greater than the largest execution time.

### 6.4.2 Processor Augmentation Bound Derivation

We now show that it is enough to use 3 processors to guarantee limited preemptivity of a uniprocessor feasible taskset $\Gamma$ for which $\frac{D_{\text{min}}}{L_{\text{max}}} \geq 2$.

**Lemma 6.4.3.** The number of processors on which a uniprocessor feasible taskset $\Gamma$ is guaranteed limited preemptive feasibility, such that $\frac{D_{\text{min}}}{L_{\text{max}}} \geq 2$, is upper-bounded by 3.
The value of \( t \) that maximizes \( L_{\text{max}} \) is \( t = D_{\text{min}} \), and hence, the taskset is limited preemptive EDF feasible if:

\[
\delta_{\text{tot}} \leq 1 - \frac{L_{\text{max}}}{D_{\text{min}}}
\]

Instantiating the above test in the context of a fully non-preemptive EDF scheduler, we get the following test for non-preemptive EDF feasibility.

**Corollary 6.4.1.** A sporadic real-time taskset \( \Gamma \) is feasible under non-preemptive EDF if,

\[
\delta_{\text{tot}} \leq 1 - \frac{C_{\text{max}}}{D_{\text{min}}}
\]

where \( C_{\text{max}} = \max_{\forall \tau_i \in \Gamma} (C_1) \).

The tests presented above generalize to a utilization based test when the deadlines of the tasks are equal to their time periods. This density based test is interesting since it runs in a time polynomial in the number of tasks, when compared to the exact demand bound based tests by Jeffay et al. [4] and Baruah [9] that runs in pseudo-polynomial time. The polynomial complexity of the density based test comes at the cost of necessity i.e., the test presented above is only a sufficient condition for schedulability. The density based test is especially applicable to the liquid task model presented by Abdelzaher et al. [66] in which the shortest deadline is orders of magnitude greater than the largest execution time.

### 6.4.2 Processor Augmentation Bound Derivation

We now show that it is enough to use 3 processors to guarantee limited preemptivity of a uniprocessor feasible taskset \( \Gamma \) for which \( D_{\text{min}} \frac{L_{\text{max}}}{L_{\text{max}}} \geq 2 \).

**Lemma 6.4.3.** The number of processors on which a uniprocessor feasible taskset \( \Gamma \) is guaranteed limited preemptive feasibility, such that \( D_{\text{min}} \frac{L_{\text{max}}}{L_{\text{max}}} \geq 2 \), is upper-bounded by 3.

**Proof.** Substituting \( D_{\text{min}} \frac{L_{\text{max}}}{L_{\text{max}}} \geq 2 \) in Lemma 6.4.2, we get that if the total density of the taskset is no greater than 50%, the task set is LP-EDF feasible on a uniprocessor.

Therefore, if we partition \( \Gamma \) into subsets such that the utilization of each subset does not exceed 50%, then we can guarantee the limited preemptive feasibility of each subset on \( m \) unit speed processors, where \( m \) is equal to the number of such subsets.

In the worst case, in order to partition \( \Gamma \) with total utilization \( U_{\text{tot}} \leq 1 \) into subset of tasks, each with total utilization \( \leq \frac{1}{2} \), we need at most 3 processors.

The use of more number of processors to achieve predictability can be potentially interesting in systems where the slack, after scheduling the hard real-time tasks, are used to schedule soft real-time or non real-time tasks, e.g., using servers. The hard real-time tasks can be partitioned upon the multiple processors/cores to achieve predictability, while the soft- and non- real-time tasks can execute upon servers while maximizing the service to them using known schemes [95].

### 6.5 Discussion

In this section, we discuss the resource augmentation bounds derived in the chapter in different (but related) contexts, as well as elaborate on some details:

- **Enabling limited preemptive feasibility**: Feasibility guarantees cannot be provided under limited preemptive scheduling if the length of the largest non-preemptive region is greater than the shortest deadline in the taskset. However, as pointed out by [5], solutions are available to overcome this problem, e.g., by using code-refactoring or by changing design parameters. Code re-factoring can be performed to reduce the execution time of real-time tasks, e.g., by the use of more efficient code, that effectively amounts to using a faster processor. Similarly, scaling all deadlines/time periods by the same factor is similar to
speeding up the processor \(^2\). Additionally, parallel algorithms can be used to speed-up task executions, and hence, a specified limited preemptive behavior can be guaranteed by using one or more of the above techniques.

**Contribution:** The speed-up bound presented in this chapter quantifies the extent to which code-refactoring must be performed to reduce task execution times, or the extent to which task parameters must be adjusted, in order to address the unschedulability arising out of tasks with very large execution times. Similarly, the speed-up bound also quantifies the requirement on the amount of parallelizable code \(^97\) when parallel algorithms are used to guarantee limited preemptivity.

**Quantifying the accuracy of timing analysis tools:** Most timing analysis tools overestimate the Worst Case Execution Time (WCET) in order to provide safe bounds, consequently enabling hard real-time guarantees. However, overestimating WCETs cause significant loss of system utilization. One of the reasons behind this overestimation is the fact that it is very difficult to accurately account for preemption related overheads, especially on fully preemptive systems. On the other hand, the worst case preemption behavior that maximizes the associated overheads occurs very rarely in practice— nevertheless the system should be built to handle the worst case to provide hard guarantees. Preemption related overheads depend on the number of preemptions, as well as the points at which these preemptions occur. Hence, limiting preemptions to specified points in code improve WCET predictions since preemption overheads can be more accurately accounted during timing analysis.

\(^2\)Note that design parameters such as deadlines and time periods in many systems are negotiable not only in many soft real-time applications, but also in many hard real-time applications (please refer to \([96]\) for more details).
Contribution: The accuracy of timing analysis tools can be quantified in terms of the feasibility of limiting preemptions to specified preemption points in the code. We plan to further investigate this in a future work by considering the preemption costs in the analysis.

6.6 Chapter Summary

This chapter essentially bridges uniprocessor preemptive and non-preemptive real-time scheduling paradigms by providing significant theoretical results building on the limited preemptive scheduling approach. We investigated the sub-optimality of limited preemptive scheduling with respect to a uniprocessor optimal scheduling algorithm, like the preemptive EDF, using the widely accepted notion of resource augmentation. For this purpose, we investigated how extra resources affect the preemptive behavior of real-time tasks, and derived bounds on the 1) required processor speed-up and 2) required number of processors, that guarantees a specified limited preemptive behavior.

1. In this chapter, we considered a more realistic execution time model proposed by [90] in which only a fraction $\phi$ of the task execution time scales linearly with the processor speed and derived the speed-up bounds required to guarantee a specified limited preemptive behavior.

- If only a fraction $\phi$ of the task execution times scales with processor speed, the speed-up bound $S$ that guarantees limited preemptive feasibility is given by $S \leq \left(1 + \frac{1}{\phi}\right)$ in many cases.

- We showed that, if the entire execution time of the tasks scales with processor speed, the speed-up bound $S$ that guarantees non-preemptive execution of all tasks for a
duration no greater than $L_{\text{max}}$ is given by

$$S \leq 2 \max \left( 1, \frac{L_{\text{max}}}{D_{\text{min}}} \right)$$

2. We derived the bound on the number of processors required to guarantee a specified limited preemptive behavior of \textit{uniprocessor feasible} real-time tasks, allowing us to consider a different dimension with respect to the feasibility of limiting preemptions.

- In general, the number of processors is shown to be upper-bounded by the number of tasks in the taskset.

- In the specific case when the largest length of the specified limited preemptive regions in the taskset is not larger than half the shortest deadline, the number of processors is shown to be upper-bounded by 3.

The derived bounds when instantiated in the context of fully non-preemptive EDF allows us to quantify the sub-optimality of non-preemptive scheduling. Finally, we use the derived speed-up bounds to calculate the minimum processor speed-up required that guarantees a specified limited preemptive behavior, which in turn minimizes preemption related overheads in the schedule. The derived bounds quantify the extent to which source code must be re-factored or deadlines must be relaxed in order to address any infeasibility while limited preemptively scheduling real-time tasks.

**Note:** This chapter is based on the following papers:

Chapter 7

Resource Augmentation for Multiprocessor Limited Preemptive Scheduling

Few of the results obtained for a single processor generalize directly to the multiple processor case; bringing in additional processors adds a new dimension to the scheduling problem.

In the previous chapter, we derived resource augmentation bounds for uniprocessor limited preemptive scheduling. In this chapter, we consider the case of multiprocessor scheduling and derive associated speed-up factors required when limiting preemptions under the assumption of the floating NPR paradigm. Resource augmentation has been previously used to derive speed-up factors that guarantee Global Preemptive EDF (G-P-EDF) schedulability of all feasible real-time tasksets [69]. Phillips et al. in their paper [69], showed that when given processors that are twice as fast, G-P-EDF gives optimal performance.

1This quote is from C.L. Liu's paper [98], and is used here to highlight the interesting property that the resource augmentation bounds derived in this chapter in fact generalize from the single processor case.
Chapter 7

Resource Augmentation for Multiprocessor Limited Preemptive Scheduling

Few of the results obtained for a single processor generalize directly to the multiple processor case; bringing in additional processors adds a new dimension to the scheduling problem.¹

In the previous chapter, we derived resource augmentation bounds for uniprocessor limited preemptive scheduling. In this chapter, we consider the case of multiprocessor scheduling and derive associated speed-up factors required when limiting preemptions under the assumption of the floating NPR paradigm. Resource augmentation has been previously used to derive speed-up factors that guarantee Global Preemptive EDF (G-P-EDF) schedulability of all feasible real-time tasksets [69]. Phillips et al. in their paper [69], showed that when given processors that are twice as fast, G-P-EDF gives optimal performance

¹This quote is from C L Liu’s paper [98], and is used here to highlight the interesting property that the resource augmentation bounds derived in this chapter in fact generalizes from the single processor case.
Chapter 7. Resource Augmentation for Multiprocessor Limited Preemptive Scheduling

with respect to meeting deadlines. This work has been recognized as one of the seminal results in the field of multiprocessor real-time scheduling \(^2\). In this chapter, we consider limiting preemptions in global EDF, and study the interplay between Global Limited Preemptive EDF (G-LP-EDF) feasibility and processor speed. This also allows us to quantify the sub-optimality of Global Non Preemptive EDF (G-NP-EDF) in terms of the minimum speed-up required to guarantee G-NP-EDF feasibility of all feasible tasksets. The results derived in this chapter complement our previous results on uniprocessors, and provide a unified result on the sub-optimality of non-preemptive EDF on both uniprocessor and multiprocessor platforms.

7.1 Models and Notations

In this section, we introduce the notations used in this chapter, including the task and processor model and the scheduling model.

7.1.1 Task and Processor Model

We consider a set of \(n\) sporadic real-time tasks \(\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}\) executing on \(m\) identical processors. Each \(\tau_i\) is characterized by a minimum inter-arrival time \(T_i\), a worst case execution requirement \(C_i\), and a relative deadline \(D_i \leq T_i\). The tasks are ordered according to their increasing deadlines, i.e., \(D_i \leq D_{i+1}, 1 \leq i < n\), and \(D_{\text{min}}\) is used to denote \(D_1\). Similarly, the largest execution time is denoted by \(C_{\text{max}} = \max \{C_i : \forall \tau_i \in \Gamma\}\). Without loss of generality, we assume that the default speed of all processors is \(S = 1\). In common with [26] [23] [22] [21] [69] [44], we make the simplifying assumption that task execution times scale linearly with the processor speed in order to focus on the theoretical consequences of processor speed-up on the preemptive behavior of real-time tasks under global EDF. In other words we

\(^2\)See http://sites.ieee.org/tcrt/education/seminal-papers/
assume that when a processor that is two times faster is used, the worst case execution time becomes $\frac{C_i}{2}$, $\forall \tau_i \in \Gamma$. The model also allows us to use the terms ‘processor speed-up factor’ and ‘processor speed’ interchangeably. Changing the processor speed from $S = 1$ to $S = a$, is equivalent to speeding up the processor by a factor ‘$a$’.

We assume that every task $\tau_i$ can execute non-preemptively for a duration given by $L = \max_{\forall \tau_i \in \Gamma} (L_i)$. Each of these tasks in $\Gamma$ generates a sequence of jobs $J$ where a job in $J$ is represented by $J_k$. The density of a task $\tau_i$ is defined as $\delta_i = \frac{C_i}{D_i}$ and the maximum density is defined as

$$\delta_{\text{max}} = \max_{\forall \tau_i \in \Gamma} \left\{ \frac{C_i}{D_i} \right\}$$

We also define $\hat{\delta}_{\text{max}}$ as:

$$\hat{\delta}_{\text{max}} = \max_{\forall \tau_i \in \Gamma} \left\{ \frac{C_i}{D_i - L} \right\}$$

### 7.1.2 G-LP-EDF with Lazy Preemptions Model

We assume a deadline based scheduler: in any time interval $[t_a, t_f)$, first $m$ jobs having the earliest deadlines are assigned to the $m$ processors, with ties broken arbitrarily. Whenever a higher priority job $J_i$ is released and all $m$ processors are busy, with at least one processor executing a lower priority job, all the lower priority jobs begin executing non-preemptively for at most $L$ time units. After at most $L$ time units, the scheduler reschedules the entire set of tasks. In other words, $J_i$ preempts the lowest priority executing job after getting blocked for at most $L$ time units, or is allocated to the first processor that becomes idle. Since after a preemption the task can resume either on the same processor or on a different processor, for convenience, whenever we refer to a preemption we mean preemptions and/or migrations.
7.1.3 Definitions

The *demand* of a sequence of jobs $J$ over a time interval of length $t$ is defined as the cumulative execution time of all the jobs in $J$ scheduled in that interval. The *minimum demand* of a given sequence of jobs generated over an interval of length $t$ is defined as the minimum amount of execution that the sequence of jobs could require within $t$ in order to meet all its deadlines. This concept has been extended to sporadic task systems, where a task $\tau_i$’s *maxmin* demand over an interval of length $t$ is defined as the largest minimum demand by any sequence of jobs that could be legally generated by $\tau_i$ in $t$ [44].

Baruah *et al.* [44] introduced the *Forced Forward Demand Bound Function* (FF-DBF) that generalized the above concepts on a set of speed-$\sigma$ processors. The FF-DBF of any task $\tau_i$ over a time interval of length $t$ on an identical multiprocessing platform where all processors execute at speed $S = 1$ is defined as [44]:

$$
\text{FF-DBF}(\tau_i, t, \sigma) = q_i C_i + \begin{cases} 
C_i & \text{if } r_i \geq D_i \\
C_i - (D_i - r_i)\sigma & \text{if } D_i > r_i \geq D_i - \frac{C_i}{\sigma} \\
0 & \text{otherwise}
\end{cases}
$$

where, $\sigma$ is a positive real-number and,

$$
q_i \overset{\text{def}}{=} \left\lfloor \frac{t}{T_i} \right\rfloor
$$

and

$$
r_i \overset{\text{def}}{=} t \mod T_i
$$

The FF-DBF of the taskset, denoted by FF-DBF($\Gamma$, $t$, $\sigma$) is given by,

$$
\text{FF-DBF}(\Gamma, t, \sigma) = \sum_{\forall \tau_i} \text{FF-DBF}(\tau_i, t, \sigma)
$$

Consequently, the FF-DBF($\tau_i$, $t$, $\sigma$) can be seen as the *maxmin* demand of $\tau_i$ over an interval of length $t$, where the execution outside the interval occurs on a speed-$\sigma$ processor.
7.2 Global Limited Preemptive EDF Feasibility vs. Processor Speed

In this thesis, one of our aims is to study how the preemptive behavior of real-time tasks change with processor speed under global EDF based scheduling on a multiprocessing platform, which also allows us to quantify the sub-optimality of G-NP-EDF scheduling in terms of bounds on the required speed-up that guarantees G-NP-EDF feasibility. In the previous chapter, we introduced the concept of the feasibility bucket that illustrated how preemptive behavior of uniprocessor feasible real-time tasksets change with the processor speed. In Figure 6.2, the base of the bucket represented the set of real-time tasksets feasible on \( m \) identical processors \( (m \geq 1) \) of speed \( S = 1 \). On increasing the speed to \( S = x \) more tasksets become feasible, and the set of tasksets that were originally feasible at speed \( S = 1 \) becomes a subset of the set of tasksets feasible at speed \( S = x \).

Remember that in Figure 6.3 we illustrated our previous results, the feasibility bucket for uniprocessor Limited Preemptive EDF (LP-EDF) feasibility of real-time tasks [99]. The base of the bucket represented the uniprocessor feasible tasksets on a processor of speed \( S = 1 \), and on increasing the processor speed to \( S = 2 \max \left( 1, \frac{L}{D_{\text{min}}} \right) \), these tasksets become LP-EDF feasible on a uniprocessor. On further increasing the processor speed to \( S = 2 \max \left( 1, \frac{C_{\text{max}}}{D_{\text{min}}} \right) \), they become NP-EDF feasible on a uniprocessor.

In this chapter, we extend the uniprocessor LP-EDF feasibility bucket presented in Figure 6.3 to global EDF—this is presented in Figure 7.1. The dotted bucket inside represents the uniprocessor feasibility bucket, and is included to illustrate how the uniprocessor EDF results extend to multiprocessor global EDF. We refer to this figure as the feasibility bucket for G-LP-EDF scheduling of real-time tasks. The base of the outer bucket denotes the set of all tasksets feasible on \( m \)-processors of speed \( S = 1 \), and are henceforth referred to as \( m \)-processor feasible tasksets. When the speed of all \( m \) processors is
increased to \((2 - \frac{1}{m})\), all \(m\)-processor feasible tasksets are guaranteed G-P-EDF feasibility (as shown in [69]). We show that if the speed of all \(m\) processors is further increased to \(2 \max \left(1, \frac{L}{D_{\min}} \right) \left(2 - \frac{1}{m} \right)\), all \(m\)-processor feasible tasksets are guaranteed G-LP-EDF feasibility such that every task executes non-preemptively for a duration no more than \(L\). We also show that on increasing the speed of all \(m\) processors to \(2 \max \left(1, \frac{C_{\text{max}}}{D_{\min}} \right) \left(2 - \frac{1}{m} \right)\), which corresponds to the height of the outer bucket, all \(m\)-processor feasible tasksets are guaranteed G-NP-EDF feasibility (on \(m\)-processors).

We now recall the following result from Baruah et al. [44], which is a sufficient condition for global preemptive EDF feasibility.

**Theorem 7.2.1.** A taskset \(\Gamma\) is G-P-EDF feasible if \(\exists \sigma : \sigma \geq \delta_{\text{max}}\) such
that $\forall t \geq 0$,

$$FF-DBF(\Gamma, t, \sigma) \leq (m - (m - 1)\sigma)t$$

We assume that the taskset is initially G-P-EDF feasible and then use the above result by Baruah et al. [44] to derive the following bound on the required speed-up that guarantees G-LP-EDF feasibility. We first assume that the condition in Theorem 3.2.1 is not satisfied, and then calculate the processor speed-up required to satisfy the condition in Theorem 3.2.1.

**Lemma 7.2.1.** The speed $S$ that guarantees G-LP-EDF feasibility of any G-P-EDF feasible taskset $\Gamma$, such that every $\tau_i \in \Gamma$ can execute non-preemptively for a duration of at most $L$, is given by

$$S \leq \left(\frac{x}{x - 1}\right)$$

where, $x = \frac{L}{T}$, $\forall t > 0$.

**Proof.** According to Theorem 3.2.1, if a taskset is G-LP-EDF feasible,

$$\exists \sigma : \sigma \geq \hat{\delta}_{max}$$

such that $\forall t \geq 0$ and $L = \max_{\forall \tau_i \in \Gamma} (L_i)$,

$$FF-DBF(\Gamma, t, \sigma) \leq (m - (m - 1)\sigma)(t - L)$$

Suppose this is not true, i.e., $\forall \sigma, \forall t \geq 0$,

$$FF-DBF(\Gamma, t, \sigma) > (m - (m - 1)\sigma)(t - L)$$

Since the taskset is G-P-EDF feasible, according to Theorem 7.2.1,

$$\exists \sigma : \sigma \geq \delta_{max}, \forall t \geq 0$$

$$FF-DBF(\Gamma, t, \sigma) \leq (m - (m - 1)\sigma)t$$
To achieve G-LP-EDF feasibility we can speed-up all \( m \) processors such that, for the value of \( \sigma \) for which the \( \Gamma \) is G-P-EDF feasible, the following holds true:

\[
\forall t \geq 0 : \frac{\text{FF-DBF}(\Gamma, t, \sigma)}{S} \leq (m - (m - 1)\sigma)(t - L)
\]

\[
\Rightarrow \text{FF-DBF}(\Gamma, t, \sigma) \leq S \left((m - (m - 1)\sigma)(t - L)\right)
\]

This means that the following condition should be satisfied (since the taskset is both G-LP-EDF and G-P-EDF feasible).

\[
S \left[m - (m - 1)\sigma\right](t - L) \leq \left[m - (m - 1)\sigma\right]t \Rightarrow S \leq \frac{t}{t - L}
\]

Substituting for \( x = \frac{t}{L} \), we get,

\[
S \leq \frac{x}{x - 1}
\]

Proof. Evaluating the limit of the equation in lemma 7.2.2 at \( x = 2 \), we get

\[
S = \left(2 - \frac{1}{m}\right)\left(\frac{x}{x - 1}\right)
\]

Evaluating the limit using l'Hopital's rule as \( x \) tends to infinity (\( \infty \)), we get

\[
S = \left(2 - \frac{1}{m}\right)
\]

Therefore, for any value of \( x \in [2, \infty) \),

\[
S \leq \left(2 - \frac{1}{m}\right)\left(\frac{x}{x - 1}\right)
\]

We assumed that the taskset is \( m \)-processor feasible, and hence the maximum execution requirement that has to be completely scheduled in any interval of length \( t \) is at most \( t \), over all processors. Informally speaking, the above lemma indicates that it is sufficient to speed-up all the processors such that the execution requirement of length \( t \) finishes executing in no greater than \( t - L \) time units to guarantee G-LP-EDF feasibility (so that lower priority jobs can utilize this slack of length \( L \) to execute non-preemptively for at most \( L \) units). Note that, we assume the floating NPR scheduling mechanism in which the re-scheduling decision is deferred by \( L \) time units every time a higher priority task is released and requires a preemption.

**Lemma 7.2.2.** The speed \( S \) that guarantees G-LP-EDF feasibility of an \( m \)-processor feasible taskset \( \Gamma \), such that every \( \tau_i \in \Gamma \) can execute non-preemptively for a duration of at most \( L \), is given by

\[
S \leq \left(2 - \frac{1}{m}\right)\left(\frac{x}{x - 1}\right)
\]

where, \( x = \frac{t}{L} \), \( \forall t > 0 \).
7.2 Global Limited Preemptive EDF Feasibility vs. Processor Speed

Proof. According to Phillips et al. [69], the speed-up required to guarantee G-P-EDF feasibility of all m-processor feasible tasksets \( \Gamma \) is upper-bounded by \( (2 - \frac{1}{m}) \). Lemma 7.2.1 gives the processor speed-up bound that guarantees G-LP-EDF feasibility of a G-P-EDF feasible taskset. Therefore, the speed-up bound that guarantees G-LP-EDF feasibility of all m-processor feasible tasksets is obtained by multiplying the respective bounds.

We now derive the speed-up bound when \( \frac{t}{L} \) takes different values. Specifically, we consider the following cases:

Case 1: \( \frac{t}{L} \geq 2 \)

Case 2: \( 1 \leq \frac{t}{L} < 2 \)

Case 3: \( 0 < \frac{t}{L} < 1 \)

Lemma 7.2.3. The speed \( S \) that guarantees G-LP-EDF feasibility of an m-processor feasible taskset \( \Gamma \), such that every \( \tau_i \in \Gamma \) can execute limited preemptively for a duration of at most \( L \), where \( \frac{t}{L} \geq 2 \) and \( t > 0 \), is given by

\[
S \leq \left( 2 - \frac{1}{m} \right) \times 2
\]

Proof. Evaluating the limit of the equation in lemma 7.2.2 at \( x = 2 \), we get

\[
S = \left( 2 - \frac{1}{m} \right) \times 2
\]

Evaluating the limit using l’Hopital’s rule as \( x \) tends to infinity (\( \infty \)), we get

\[
S = \left( 2 - \frac{1}{m} \right)
\]

Therefore, for any value of \( x \in [2, \infty] \),

\[
S \leq \left( 2 - \frac{1}{m} \right) \times 2
\]
The speed $S$ that guarantees G-LP-EDF feasibility of an $m$-processor feasible task set $\Gamma$, such that every $\tau_i \in \Gamma$ can execute limited preemptively for a duration of at most $L$, where $1 \leq \frac{t}{L} < 2$ and $t > 0$, is given by

$$S \leq \left( 2 - \frac{1}{m} \right) \times 2$$

Proof. According to our assumption the task set is $m$-processor feasible. Therefore, in any time interval of length $t$, the maximum execution requirement that has to be completely scheduled is at most $t$, over all the processors. To guarantee limited preemptive feasibility, we need to ensure that the execution requirement $t$ completes in $t - L$ time units on every processor (so that lower priority jobs can utilize this slack of length $L$ to execute non-preemptively for at most $L$ units).

We know that, $1 \leq \frac{t}{L_i} < 2$ and thus we have $t \geq L$ and $t < 2L$. In the worst case, all $m$-processors are fully occupied during the interval of length $t$. Therefore it is not possible to feasibly execute the non-preemptive region of length $L$. Let us assume an increase in the processor speed by a factor of 2. This implies that within an interval of length $t$, there are in effect $t' = 2t$ clock ticks. It is clear that $2t \geq L + t$ since $t \geq L$. Thus, any lower priority job can execute non-preemptively on any processor for a duration $L$ within $t$ without causing deadline misses. Therefore, the speed-up required for this case is exactly upper-bounded by 2.

According to Phillips et al. [69] the upper-bound on the speed-up that guarantees G-P-EDF feasibility of the set of all $m$-processor feasible task sets is $(2 - \frac{1}{m})$. Therefore, to guarantee G-LP-EDF feasibility, when $1 \leq \frac{t}{L_i} < 2$ and $t > 0$, the value of $S$ should be

$$S \leq \left( 2 - \frac{1}{m} \right) \times 2$$

Lemma 7.2.5. The speed $S$ that guarantees G-LP-EDF feasibility of an $m$-processor feasible task set $\Gamma$, such that every $\tau_i \in \Gamma$ can execute
7.2 Global Limited Preemptive EDF Feasibility vs. Processor Speed

limited preemptively for a duration of at most $L$, where $0 < \frac{L}{t} < 1$ and $t > 0$, is given by

$$S \leq \left( 2 - \frac{1}{m} \right) \times \frac{2L}{t}$$

Proof. On increasing the processor speed to $S = \frac{L}{t}$, the number of clock ticks in any time interval $t$ increases from $t$ to $t' = t \times \frac{L}{t} = L$. We can now execute the original execution requirement of length $t$ and $L - t$ units of the non-preemptive region $L$, using the $L$ clock ticks in the time interval $t$ at speed $S = \frac{L}{t}$. Let the remaining length of the non-preemptive region that cannot be executed without a deadline miss in the interval $t$, be denoted by $L' = t$. We know that $t < L$, thus, in effect we get $\frac{L'}{t} = \frac{L}{t} > 1$.

Using lemmas 7.2.3, and 7.2.4, the upper-bound on the speed (denoted by $S'$), that guarantees G-LP-EDF feasibility is

$$S' \leq \left( 2 - \frac{1}{m} \right) \times 2.$$ 

Since we had already increased the processor speed by $\frac{L}{t}$, the upper-bound on the actual speed $S$ is:

$$S \leq \left( 2 - \frac{1}{m} \right) \times \frac{2L}{t}$$

$\square$

We obtain the following theorem by combining the speed-up bounds for the three cases presented above.

Theorem 7.2.2. The speed $S$ that guarantees the G-LP-EDF feasibility of an $m$-processor feasible taskset $\Gamma$, such that every $\tau_i \in \Gamma$ can execute limited preemptively for a duration $L$ in any time interval $t > 0$, is given by

$$S \leq 2 \max \left( 1, \frac{L}{t} \right) \left( 2 - \frac{1}{m} \right)$$

Proof. In the general case, $L$ is bounded by the maximum of the execution times of the tasks in the taskset (i.e., for their fully non-preemptive execution), and $t$ by the shortest deadline.
Consequently, \( \forall \tau_i, \frac{1}{L} > 0 \). It follows from Lemmas 7.2.3 7.2.4 7.2.5 that the speed-up required to guarantee G-LP-EDF feasibility in the general case is

\[
S \leq 2 \max \left( 1, \frac{L}{t} \right) \left( 2 - \frac{1}{m} \right)
\]

When \( \frac{L}{t} \geq 2 \), we obtain

\[
S \leq \left( 2 - \frac{1}{m} \right) \times \frac{4}{L} = \left( 2 - \frac{1}{m} \right) \times \frac{4}{2} = \left( 2 - \frac{1}{m} \right) \times 2
\]

Similarly, when \( 1 \leq \frac{L}{t} < 2 \), we obtain

\[
S \leq \left( 2 - \frac{1}{m} \right) \times \frac{4}{L} = \left( 2 - \frac{1}{m} \right) \times \frac{4}{1} = \left( 2 - \frac{1}{m} \right) \times 2
\]

Finally when \( 0 < \frac{L}{t} < 1 \), we have

\[
S \leq \left( 2 - \frac{1}{m} \right) \times \frac{2L}{t}
\]

Therefore, for any \( \frac{L}{t} > 0 \), the speed-up required that guarantees G-LP-EDF feasibility is

\[
S \leq 2 \max \left( 1, \frac{L}{D_{min}} \right) \left( 2 - \frac{1}{m} \right)
\]

**Corollary 7.2.1.** The speed \( S \) that guarantees the G-LP-EDF feasibility of an \( m \)-processor feasible taskset \( \Gamma \), such that every \( \tau_i \in \Gamma \) can execute limited preemptively for a duration \( L \), is given by

\[
S \leq 2 \max \left( 1, \frac{L}{D_{min}} \right) \left( 2 - \frac{1}{m} \right)
\]

This is straightforward, as the value of \( t \) that maximizes \( \left( 2 - \frac{1}{m} \right) \times \frac{4L}{t} \) is the smallest value of \( t \) at which the condition in Theorem 3.2.1 should be evaluated, and is given by \( t = D_{1} (D_{min}) [44] \). We obtain the sub-optimality of G-NP-EDF by substituting \( L = C_{max} \), and is formally presented in the following.
Corollary 7.2.2. The speed $S$ that guarantees G-NP-EDF feasibility of an $m$-processor feasible taskset $\Gamma$ is given by

$$S \leq 2 \max \left( 1, \frac{C_{\text{max}}}{D_{\text{min}}} \right) \left( 2 - \frac{1}{m} \right)$$

We hence obtain the resource augmentation bound, specifically an upper-bound on the required processor speed-up, that guarantees G-LP-EDF feasibility.

7.3 Chapter Summary

In this chapter, we investigate how G-LP-EDF feasibility changes with respect to processor speed, and use this to quantify the sub-optimality of G-NP-EDF. Specifically, we show that the speed-up required to guarantee the G-LP-EDF feasibility of a set of $m$-processor feasible tasksets is given by,

$$S \leq 2 \max \left( 1, \frac{L}{D_{\text{min}}} \right) \left( 2 - \frac{1}{m} \right)$$

Remarkably, our results on uniprocessors extend directly to the multiprocessor case. This allows us to present a unified result on how the feasibility of limiting preemptions under EDF based scheduling changes with processor speed on uniprocessor and multiprocessor platforms, enabling us to present a unified result on the sub-optimality of non-preemptive EDF on both the platforms.

Note: This chapter is based on the paper *The Global Limited Preemptive Earliest Deadline First Feasibility of Sporadic Real-time Tasks*, Abhilash Thekkilakattil, Sanjoy Baruah, Radu Dobrin and Sasikumar Punnekkat, The 26th Euromicro Conference on Real-time Systems, IEEE, Madrid, Spain, July, 2014
Chapter 8

Preemption Control using Resource Augmentation

For over a decade prophets have voiced the contention that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers in such a manner as to permit cooperative solution.

Many real-time systems consist of data intensive real-time tasks that are cooperatively scheduled, where each task is composed of many non-preemptable chunks of code [27]. If the size of these non-preemptable chunks are significantly large, it may cause large blocking leading to unschedulability. Similarly, many methods [14] have been proposed to place preemption points in the task code such that the preemption overheads are minimized. However, if the duration between any two optimal preemption points is significantly high, it may lead to deadline misses due to blocking. Thiele [100] presented

1This quote from Gene Amdahl’s famous paper [97] published in 1967 on which Amdahl’s law is based on shows that the cynicism towards the possibility of having faster computers may not be justifiable.
Chapter 8

Preemption Control using Resource Augmentation

For over a decade prophets have voiced the contention that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers in such a manner as to permit cooperative solution.¹

Many real-time systems consist of data intensive real-time tasks that are cooperatively scheduled, where each task is composed of many non-preemptable chunks of code [27]. If the size of these non-preemptable chunks are significantly large, it may cause large blocking leading to unschedulability. Similarly, many methods [14] have been proposed to place preemption points in the task code such that the preemption overheads are minimized. However, if the duration between any two optimal preemption points is significantly high, it may lead to deadline misses due to blocking. Thiele [100] presented

¹This quote from Gene Amdahl’s famous paper [97] published in 1967 on which Amdahl’s law is based on shows that the cynicism towards the possibility of having faster computers may not be justifiable.
Chapter 8. Preemption Control using Resource Augmentation

the possibility of speeding up the processor in order to achieve a schedulable system. Equivalently, processor speed-up can be used to control the execution times of the non-preemptable chunks and the processor demand to guarantee that tasks can execute non-preemptively for a specified duration. Thereby, the number of preemptions can be reduced and/or the preemption points can be placed at optimal locations with respect to the preemption costs. In this section, we focus on deriving the minimum processor speed-up that guarantees the specified preemption behavior, thereby minimizing preemption overheads.

Note that, in this chapter we focus on uniprocessor systems. However, the results can be trivially extended to the multiprocessor case.

8.1 Models and Notations

In this section, we describe the notations used in the rest of the chapter whilst describing the task model and processor model.

8.1.1 Task and Processor model

We consider a set of sporadic real-time tasks denoted by \( \Gamma = \{ \tau_1, \tau_2, \ldots, \tau_n \} \) executing on a uniprocessor platform. Each task \( \tau_i \) has a minimum inter-arrival time \( T_i \), a worst case execution time \( C^S_i \) at processor speed \( S \), and a relative deadline \( D_i \). Without loss of generality, we assume that tasks’ worst case execution times are equal to their worst case execution requirements on a speed \( S = 1 \) processor. We denote the length of the longest critical section of a task \( \tau_i \), on a processor of speed \( S \), by \( CS^S_i \). The tasks are assumed to be indexed according to the increasing order of their deadlines, which means that \( D_{\min} = D_1 \). We assume that every task \( \tau_i \) has \( m_i \) optimal preemption points [14] within its execution, where the \( m_i^{th} \) point denotes the end of the task execution. Let \( q_{i,j}^S \), \( j = 1...m_i \) denote the length of the execution of \( \tau_i \) from its start up to the \( j^{th} \) optimal preemption point on
Chapter 8. Preemption Control using Resource Augmentation

the possibility of speeding up the processor in order to achieve a schedulable system. Equivalently, processor speed-up can be used to control the execution times of the non-preemptable chunks and the processor demand to guarantee that tasks can execute non-preemptively for a specified duration. Thereby, the number of preemptions can be reduced and/or the preemption points can be placed at optimal locations with respect to the preemption costs. In this section, we focus on deriving the minimum processor speed-up that guarantees the specified preemption behavior, thereby minimizing preemption overheads.

Note that, in this chapter we focus on uniprocessor systems. However, the results can be trivially extended to the multiprocessor case.

8.1 Models and Notations

In this section, we describe the notations used in the rest of the chapter whilst describing the task model and processor model.

8.1.1 Task and Processor model

We consider a set of sporadic real-time tasks denoted by $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$ executing on a uniprocessor platform. Each task $\tau_i$ has a minimum inter-arrival time $T_i$, a worst case execution time $C_{S_i}$ at processor speed $S$, and a relative deadline $D_i$. Without loss of generality, we assume that tasks’ worst case execution times are equal to their worst case execution requirements on a speed $S = 1$ processor.

We denote the length of the longest critical section of a task $\tau_i$, on a processor of speed $S$, by $CS_{S_i}$. The tasks are assumed to be indexed according to the increasing order of their deadlines, which means that $D_{min} = D_1$. We assume that every task $\tau_i$ has $m_i$ optimal preemption points [14] within its execution, where the $m_i$th point denotes the end of the task execution. Let $q_{S_i,j}$, $j = 1, ..., m_i$ denote the length of the execution of $\tau_i$ from its start up to the $j$th optimal preemption point on a processor at speed $S$.

Task attributes

Set of task level limited preemption requirements

Sensitivity analysis

Preemption reduction constraints

Step 1

Step 2

Figure 8.1: Methodology overview

8.2 Methodology

While in the previous chapters we derived the upper-bound on the required processor speed-up that guarantees the feasibility of a user specified limited preemptive behavior, in this section, we apply this bound to enable trade-offs between processor speed and preemption overheads.

Definition 10. The **minimum processor speed** $S_{min}$ that guarantees the feasibility of a specified limited preemptive behavior is defined as $S_{min} = \min(S)$, where $S \in$ the set of available processor speeds such that, $\forall \tau_i$ in $\Gamma$,

$$Q_i^{S_{min}} \geq L_i^{S_{min}}$$
Here, $L_{i}^{S_{min}}$ is the specified limited preemption length for $\tau_{i}$ that guarantees the feasibility of a specified limited preemptive behavior per $\tau_{i}$. We can then calculate the speed-up required to guarantee the feasibility of the limited preemption requirement $L_{i}^{S}$, which will in turn guarantee specified bounds on the preemption related costs.

**Methodology Overview:** Our method is composed of 2 steps as shown in Figure 8.1:

**Step 1:** Specifying task level limited preemption requirements to (a) reduce the number of preemptions. (b) enable preemptions at optimal preemption points. (c) enable critical sections execution within non-preemptive regions.

**Step 2:** Perform sensitivity analysis using the task parameters and the specified limited preemption requirements to derive the minimum processor speed that guarantees the desired limited preemptive behavior.

In the following sub-sections we describe each of the steps in detail, followed by some evaluation results.

**8.2.1 Specification of Task-level Limited Preemption Requirements**

We can derive task level limited preemption requirements to

(a) reduce the number of preemptions.

(b) enable preemptions at optimal preemption points.

(c) enable critical sections execution within non-preemptive regions.

In the following, we explain in detail how we derive the associated task level limited preemption requirements.
8.2.1 Specification of Task-level Limited Preemption Requirements

(a) Reducing the number of preemptions

If the schedulability of a taskset is guaranteed considering the upper-bound on the preemption related overheads, it is indeed schedulable considering the exact overheads. The preemption related overheads can be upper-bounded by the product of the upper-bound on the number of preemptions and the upper-bound on the penalty associated with a single preemption. The upper-bound on the number of times a task \( \tau_i \), characterized by a non-preemptive region of maximum length \( Q^S_i \), can be preempted while executing on a speed \( S \) processor is given by [9] and [91] as \( \left\lceil \frac{C^S_i}{Q^S_i} \right\rceil - 1 \). Therefore, the limited preemption length \( L^S_i \), \( \forall \tau_i \in \Gamma \), on a speed \( S \) processor, that guarantees at most \( p_i \) preemptions on \( \tau_i \), can be specified as:

\[
L^S_i \geq \frac{C^S_i}{p_i + 1} \Rightarrow L^S_i = \left\lceil \frac{C^S_i}{p_i + 1} \right\rceil
\]

(8.1)

It is evident that, on a speed 1 processor, if \( Q^1_i < L^1_i \), where \( L^1_i \) is calculated according to equation 8.1, \( \tau_i \) can be guaranteed to incur no more than \( p_i \) preemptions. Hence, we have to find a processor speed \( S \) which ensures that:

\[
Q^S_i \geq L^S_i = \left\lceil \frac{C^S_i}{p_i + 1} \right\rceil
\]

(b) Enabling preemptions at optimal preemption points

The possibility of enforcing preemptions only at optimal preemption points [14] depends on the maximum length of the non-preemptive region on a processor of a given speed \( S \). Remember that \( q^S_{i,j} \) denotes the length of execution of \( \tau_i \) up to its \( j^{th} \) optimal preemption point on a speed \( S \) processor. Hence, the limited preemption requirement for a task \( \tau_i \) can be specified as the largest interval between any two consecutive optimal preemption points of \( \tau_i \) when it executes on a speed \( S \) processor:

\[
L^S_i = \max_{1 \leq j < m} \left( q^S_{i,j+1} - q^S_{i,j}, q^S_{i,1} \right)
\]

(8.2)
Consequently, our goal is to find the processor speed $S$ that satisfies:

$$Q_i^S \geq L_i^S = \max_{1 \leq j < m} (q_{i,j+1}^S - q_{i,j}^S, q_{i,1}^S)$$

An illustrative example that shows how the non-preemption

![Figure 8.2: Limited preemption requirement to guarantee preemptions only at optimal preemption points](image)

requirement is derived considering the optimal preemption points is given in Figure 8.2. In this figure, the *actual length of the largest non-preemptive region* is assumed to be calculated using the result by [9], on a processor of speed 1. If a high priority task is released immediately after $op_1$, it is impossible to defer the preemption to $op_2$ unless the maximum length of the non-preemptive region $Q_1^1$ is $op_2 - op_1$. Therefore, the non-preemption requirement $L_i^1$, in this case is given by, $op_2 - op_1$.

(c) **Executing critical sections within non-preemptive regions**

If the maximum length of the non-preemptive region $Q_i^1$ of a task $\tau_i$ is shorter than its largest critical section $CS_i^1$, on a speed 1 processor, resource sharing protocols are required. This is because it, in this case, it is impossible to guarantee the non-preemptive execution of the critical sections without jeopardizing schedulability. On the other hand, this issue can be solved by using a faster processor. The processor speed that guarantees the
non-preemptive execution of critical sections, under a limited preemptive scheduling paradigm, is given by the speed $S$ that satisfies the relation:

$$Q_i^S \geq L_i^S = CS_i^S$$  \hspace{1cm} (8.3)
guarantee the requirements of reducing the number of preemptions as well as retain the possibility of preemption placement at optimal preemption points, while guaranteeing the execution of critical sections entirely within non-preemptive regions.

8.2.2 Sensitivity Analysis for Preemption Control

If the length of the largest non-preemptive region is less than the specified limited preemption length on a speed 1 processor, i.e., $Q^1_i < L^1_i$, it means that $\tau_i$ cannot execute non-preemptively for the specified duration. Therefore, we need to use a faster processor of speed $S$ such that $Q^S_i \geq L^S_i$. In most situations, changing the processor speed may also change the specified limited preemption lengths to satisfy the desired preemption related cost control requirements, as well as the maximum possible lengths of the limited preemptive regions of the tasks. The lowest processor speed that guarantees the specified limited preemption requirements lies in the interval $[S_{low} = 1, S_{high}]$, where $S_{high}$ corresponds to the bounds derived in the previous chapters. For example, if the entire execution time of the tasks scale linearly with the processor speed,

$$S_{high} = 2 \max \left(1, \frac{L_{max}}{D_{min}}\right)$$

where $L_{max} = \max_{\forall \tau_i \in \Gamma} (L^1_i)$.

We can perform a sensitivity analysis on the speeds between 1 and $S_{high}$ in order to calculate the minimum processor speed $S_{min}$ which guarantees that every task $\tau_i$ can exhibit the specified limited preemptive behavior, i.e., $Q^S_{min} \geq L^S_{min}$.

The length of the maximum non-preemptive regions increase with decrease in the demand bound as shown by [9]. Therefore, it can be easily shown that the maximum length of the non-preemptive regions increases monotonically with the processor speed (even if only a part of the WCET scales linearly with processor speed). Hence the correctness
Chapter 8. Preemption Control using Resource Augmentation

guarantee the requirements of reducing the number of preemptions as well as retain the possibility of preemption placement at optimal preemption points, while guaranteeing the execution of critical sections entirely within non-preemptive regions.

8.2.2 Sensitivity Analysis for Preemption Control

If the length of the largest non-preemptive region is less than the specified limited preemption length on a speed processor, i.e., \( Q_1 i < L_1 i \), it means that \( \tau_i \) cannot execute non-preemptively for the specified duration. Therefore, we need to use a faster processor of speed \( S \) such that \( Q_S i \geq L_S i \). In most situations, changing the processor speed may also change the specified limited preemption lengths to satisfy the desired preemption related cost control requirements, as well as the maximum possible lengths of the limited preemptive regions of the tasks. The lowest processor speed that guarantees the specified limited preemption requirements lies in the interval \([S_{low}, S_{high}]\), where \( S_{high} \) corresponds to the bounds derived in the previous chapters. For example, if the entire execution time of the tasks scale linearly with the processor speed, \( S_{high} = 2 \max(1, L_{max} D_{min}) \) where \( L_{max} = \max(\forall \tau_i \in \Gamma(L_1 i)) \).

We can perform a sensitivity analysis on the speeds between 1 and \( S_{high} \) in order to calculate the minimum processor speed \( S_{min} \) which guarantees that every task \( \tau_i \) can exhibit the specified limited preemptive behavior, i.e., \( Q_{S_{min}} i \geq L_{S_{min}} i \). The length of the maximum non-preemptive regions increase with decrease in the demand bound as shown by [9]. Therefore, it can be easily shown that the maximum length of the non-preemptive regions increases monotonically with the processor speed (even if only a part of the WCET scales linearly with processor speed). Hence the correctness and optimality of our method is given by the correctness of the binary search.

8.2.3 Example

We illustrate our method using a simple example. Consider the taskset given in table 8.1 executing on a processor of speed 1. We assume an execution time model where the task executions scale linearly with the processor speed. In other words, if the processor speed is increased to 2 from a default speed of 1, the tasks execute twice as fast.

<table>
<thead>
<tr>
<th>Task ( \tau_i )</th>
<th>( C_i )</th>
<th>( D_i )</th>
<th>( T_i )</th>
<th>( Q_i )</th>
<th>Max. no. of preemptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \tau_1 )</td>
<td>2</td>
<td>5</td>
<td>50</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>( \tau_2 )</td>
<td>50</td>
<td>230</td>
<td>230</td>
<td>3</td>
<td>16</td>
</tr>
<tr>
<td>( \tau_3 )</td>
<td>70</td>
<td>360</td>
<td>370</td>
<td>3</td>
<td>23</td>
</tr>
<tr>
<td>( \tau_4 )</td>
<td>60</td>
<td>900</td>
<td>900</td>
<td>3</td>
<td>19</td>
</tr>
<tr>
<td>( \tau_5 )</td>
<td>80</td>
<td>990</td>
<td>1000</td>
<td>3</td>
<td>26</td>
</tr>
</tbody>
</table>

Table 8.1: Example taskset (speed=1)

According to the method proposed by [9], the maximum length of the floating NPRs per task at speed \( S = 1 \) is given in table 8.1. At speed \( S = 1 \), there are at most 16 preemptions on task \( \tau_2 \), 23 preemptions on \( \tau_3 \), 19 preemptions on \( \tau_4 \) and 26 preemptions on \( \tau_5 \). Let us assume that more than 3 preemptions on \( \tau_4 \) will lead to a deadline miss in the schedule. Note that the maximum length of the floating NPR of \( \tau_4 \) is given by \( Q_4 = 3 \), and can only guarantee that \( \tau_4 \) is preempted no more than 19 times. We perform a sensitivity analysis, as described in the previous section, to find the lowest processor speed that guarantees that task \( \tau_4 \) is preempted no more than three times. In this case, our algorithm gives an output of \( S_{opt} = 3.4 \). The length of the non-preemptive region of each task and the number of preemptions at this speed are enumerated in table 8.2 where we can see that \( \tau_4 \) is preempted no more than 3 times. In order to show that our derived speed is the lowest one which can guarantee the desired
Chapter 8. Preemption Control using Resource Augmentation

### Table 8.2: The preemptive behavior of the taskset at speed $S_{opt}$

<table>
<thead>
<tr>
<th>Task</th>
<th>Optimal Speed ($S_{opt}$=3.4)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$C_i^{S_{opt}}$</td>
</tr>
<tr>
<td>$\tau_1$</td>
<td>0.588235</td>
</tr>
<tr>
<td>$\tau_2$</td>
<td>14.705882</td>
</tr>
<tr>
<td>$\tau_3$</td>
<td>20.588234</td>
</tr>
<tr>
<td>$\tau_4$</td>
<td><strong>17.647058</strong></td>
</tr>
<tr>
<td>$\tau_5$</td>
<td>23.529411</td>
</tr>
</tbody>
</table>

We evaluated the theoretical results by testing the sensitivity analysis for a set of 1000 tasksets generated using the UUniFast algorithm [25]. Each taskset had 3 to 8 tasks, and the task periods ranged from 8 to 30 with the LCM of the periods no greater than 1500. In order to gain insights into the speed-up required for tasksets with the long task problem, we appropriately modified the task parameters of a few of these tasksets. We calculated the required minimum speed-up that guarantees a fully non-preemptive schedule under the assumption that the entire WCETs scale linearly with the processor speed. We are interested in the speed-up that guarantees a fully non-preemptive schedule because it corresponds to the maximum speed-up required to guarantee any limited preemptive behavior, \( i.e., \) if the task can execute fully non-preemptively, it can execute non-preemptively for a lesser duration.

8.3 Evaluation

We plotted the average and maximum speed-ups required to guarantee a fully non-preemptive schedule for different utilization ranges (presented in Figures 8.4 and 8.5). We observed that, in general, the required average and maximum speed-up factors increases with utilization. Additionally, the average required speed-up was found to be well below \( 2 \) as seen from Figure 8.4. However, from Figure 8.5, we observed that the maximum speed-up required was the highest for utilizations between \( 0.5 \) and \( 0.6 \), and is greater than \( 16 \). On closer examination, we found that the corresponding taskset "suffered" from the long task problem, referred to by [5], in which at least one task has an execution time greater than the shortest deadline. It is possible to construct similar tasksets (that have very high required speed-up factors) in all utilization ranges, \( i.e., \) it is possible to construct tasksets having any utilization that require arbitrarily large speed-ups to guarantee non-preemptive feasibility.
8.3 Evaluation

We evaluated the theoretical results by testing the sensitivity analysis for a set of 1000 tasksets generated using the UUniFast algorithm [25]. Each taskset had 3 to 8 tasks, and the task periods ranged from 8 to 30 with the LCM of the periods no greater than 1500. In order to gain insights into the speed-up required for tasksets with the long task problem, we appropriately modified the task parameters of a few of these tasksets. We calculated the required minimum speed-up that guarantees a fully non-preemptive schedule under the assumption that the entire WCETs scale linearly with the processor speed. We are interested in the speed-up that guarantees a fully non-preemptive schedule because it corresponds to the maximum speed-up required to guarantee any limited preemptive behavior, i.e., if the task can execute fully non-preemptively, it can execute non-preemptively for a lesser duration.

Figure 8.4: Required average speed-up that guarantees non-preemptive feasibility.

ranges (presented in Figures 8.4 and 8.5). We observed that, in general, the required average and maximum speed-up factors increases with utilization. Additionally, the average required speed-up was found to be well below 2 as seen from Figure 8.4.

However, from Figure 8.5, we observed that the maximum speed-up required was the highest for utilizations between 0.5 and 0.6, and is greater than 16. On closer examination, we found that the corresponding taskset “suffered” from the long task problem, referred to by [5], in which at least one task has an execution time greater than the shortest deadline. It is possible to construct similar tasksets (that have very high required speed-up factors) in all utilization ranges, i.e., it is possible to construct tasksets having any utilization that require arbitrarily large speed-ups to guarantee non-preemptive feasibility. Abdelzaher et al. [66] have identified a large class of real-time tasks, called liquid tasks, where the shortest deadline is much greater than the largest computation time in the taskset. Our evaluations indicate that using faster processors to guarantee specified limited preemptive
behaviors can be particularly feasible for liquid tasks since the speed-up required may not be significantly large.

### 8.4 Chapter Summary

In this chapter, we presented a methodology to control the preemptive behavior of real-time tasks with greater granularity using the possibility of processor speed-up. We first derive the length of the NPRs that satisfy the given constrains, such as specified bounds on the number of preemptions, and then perform a sensitivity analysis on the processor speeds to derive the minimum processor speed that guarantees that the specified constraints are satisfied. Note that, even though we have used processor speed-up, other methods such as increasing periods and deadlines can also be adopted in a similar manner to achieve the desired preemptive behavior.

**Note:** This chapter is based on the following papers:

Chapter 9

Conclusions and Future Work

So many people today—and even professional scientists—seem to me like someone who has seen thousands of trees but has never seen a forest. A knowledge of the historical and philosophical background gives that kind of independence from prejudices of his generation from which most scientists are suffering. This independence created by philosophical insight is, in my opinion, the mark of distinction between a mere artisan or specialist and a real seeker after truth.

The use of performance enhancing features such as caches and pipelines in modern processors have introduced new challenges in real-time scheduling, particularly with respect to fully preemptive real-time scheduling. On one hand, preemptions enable very high utilization of the processing platform, while on the other, the overheads that they introduce often offset the efficiency gains. Non-preemptive

1 Albert Einstein to Robert A. Thornton in a letter dated 07 Dec 1944 available from The Albert Einstein Archives at The Hebrew University of Jerusalem (Archival Call Number: 61-574), as quoted in [101] [102]
Chapter 9

Conclusions and Future Work

So many people today— and even professional scientists— seem to me like someone who has seen thousands of trees but has never seen a forest. A knowledge of the historical and philosophical background gives that kind of independence from prejudices of his generation from which most scientists are suffering. This independence created by philosophical insight is— in my opinion— the mark of distinction between a mere artisan or specialist and a real seeker after truth.

The use of performance enhancing features such as caches and pipelines in modern processors have introduced new challenges in real-time scheduling, particularly with respect to fully preemptive real-time scheduling. On one hand, preemptions enable very high utilization of the processing platform, while on the other, the overheads that they introduce often offset the efficiency gains. Non-preemptive

---

1Albert Einstein to Robert A. Thornton in a letter dated 07 Dec 1944 available from The Albert Einstein Archives at The Hebrew University of Jerusalem (Archival Call Number: 61-574), as quoted in [101] [102]
scheduling can become very inefficient in meeting deadlines even at very small utilizations due to blocking e.g., if a task has an execution time greater than the shortest deadline. One of the techniques that is widely used to control preemption related overheads and blocking is to limit preemptions in the schedule.

In this thesis, we investigated limited preemptive scheduling of real-time tasks on uniprocessor and multiprocessor systems. The contributions presented in this thesis enables a formal analysis of limited preemptively scheduled real-time tasks on uniprocessor and multiprocessor platforms.

9.1 Summary of the Contributions

In this section, we summarize how the two goals presented in Chapter 1 have been addressed by the different contributions. Note that the two goals represent two sides of the problem. Specifically, the first goal addresses the “how” aspect in the context of multiprocessors, while the second goal addresses the trade-offs involved while guaranteeing a specified preemptive behavior to achieve better schedulability.

G1: Extend limited preemptive scheduling theory to the context of multiprocessors.

(a) We derived a schedulability test that determines the schedulability of a set of real-time tasks under Global Limited Preemptive EDF (G-LP-EDF). This test was derived under the assumption of the lazy preemption approach in which the preempting task waits for the lowest priority task to complete executing its non-preemptive region. Moreover, the test was derived assuming the well known floating non-preemptive region scheduling model in which the non-preemptive regions are floating in the task code and is triggered by the activation of a higher priority task.
9.1 Summary of the Contributions

To our knowledge, this is the first such test for G-LP-EDF scheduling of real-time tasks under the floating NPR model.

(b) We also derived a schedulability test that determines the schedulability of a set of real-time tasks under Global Limited Preemptive FPS (G-LP-FPS). The test was derived under the assumption of the *eager preemption approach* in which the preempting task preempts the first executing lower priority task that completes its NPR (not necessarily the lowest priority task). We assumed that the tasks have a set of fixed preemption points in the task code and that preemptions can occur only at these points. Our experiments comparing taskset schedulability under G-LP-FPS with eager preemptions and G-LP-FPS with lazy preemptions using synthetic task sets showed that, in general, the eager preemption approach outperforms the lazy approach.

To our knowledge, this is the first such test for G-LP-FPS with eager preemptions under the fixed preemption points model.

(c) We then investigated the preemptive behavior of G-LP-EDF and G-LP-FPS under both eager and lazy approaches using simulations with synthetic tasksets. We showed that limited preemptive scheduling on multiprocessor platforms may not necessarily reduce the actual number of preemptions; together with an eager approach global limited preemptive scheduling actually generates more preemptions than fully preemptive scheduling.

G2: **Quantify the sub-optimality of limited preemptive scheduling that guarantees a specified limited preemptive behavior.**

(a) We investigated the sub-optimality of limited preemptive and non-preemptive scheduling on uniprocessor and
multiprocessor platforms, thereby bridging both preemptive and non-preemptive scheduling paradigms.

For any uniprocessor feasible taskset, we showed that:

- If only a fraction $\phi$ of the task execution times scales with processor speed, the upper-bound on the speed-up factor $S$ that guarantees limited preemptive feasibility on uniprocessors is given by $S \leq \left(1 + \frac{1}{\phi}\right)$ in some specified cases.

- If the entire execution time of the tasks scales linearly with the processor speed, the speed-up bound $S$ that guarantees non-preemptive execution of all tasks for a duration no greater than $L_{\text{max}}$ on a uniprocessor is given by

  $$S \leq 2 \max \left(1, \frac{L_{\text{max}}}{D_{\text{min}}} \right)$$

- The number of processors required to guarantee a specified limited preemptive behavior was shown to be upper-bounded by the number of tasks in the taskset.

- In the specific case in which the largest length of the specified limited preemptive regions in the taskset is no more than half the shortest deadline, the number of processors required to guarantee a specified limited preemptive behavior was shown to be upper-bounded by 3.

(b) For any multiprocessor feasible taskset, we showed that:

- The speed-up required to guarantee the G-LP-EDF feasibility of a set of m-processor feasible tasksets for the floating NPR model, such that all tasks can execute non-preemptively for a duration $L$, is given by,

  $$S \leq 2 \max \left(1, \frac{L}{D_{\text{min}}} \right) \left(2 - \frac{1}{m} \right)$$
Remarkably, our speed-up results on uniprocessors extend directly to the multiprocessor case.

(c) Lastly, we leverage on the speed-up bounds and present a sensitivity analysis based method that calculates the exact processor speed-up required for any given set of real-time tasks, such that the tasks can execute limited preemptively for the specified duration.

Even though we considered specific combinations of the scheduling algorithm, \( \text{viz.} \), EDF and FPS, approach to preemption, \( \text{viz.} \), EPA and LPA, and mechanisms for implementing limited preemptive scheduling, \( \text{viz.} \), floating NPRs and fixed preemption points, for deriving the schedulability tests and speed-up factors, the techniques used are general enough to be applied to all of the combinations.

9.2 Conclusions

In chapter 3, we investigated floating non-preemptive region scheduling on multiprocessors using global EDF. We proposed one form of floating NPR scheduling in which the re-scheduling decision is deferred by a duration \( L \), every time a high priority task is released, and derived a schedulability analysis. Such a floating NPR approach can be useful in the context of a best effort strategy to reduce preemption related overheads. It may also be useful in enabling self-suspensions while waiting for resources in real-time systems with shared resources (by exploiting the slack \( L \)).

In chapter 4, we investigated fixed preemption point scheduling on multiprocessors using global FPS. We showed that, for large NPRs, an eager preemption approach is beneficial when compared to a lazy preemption approach in improving schedulability. This suggests that, when sharing resources with large critical sections, \( \text{e.g.,} \) GPUs, the eager preemption approach might be beneficial since it enables faster processor access for tasks with higher priority.
Our investigations in chapter 5 complete the picture by making several interesting observations about the preemptive behavior under eager and lazy approaches, considering preemptive and limited preemptive EDF and FPS. The counter-intuitive suggestions imply that system designers need to be very careful when implementing limited preemptive scheduling on multiprocessors.

While the evaluations in Chapter 4 examined one performance parameter of limited preemptive scheduling with respect to fully preemptive scheduling, specifically average schedulability (which typically quantifies the average performance), the 6th and the 7th chapter examines another, namely speed-up factors (which quantifies the worst case performance). Note that the speed-up factors derived for G-LP-EDF assumed a floating NPR scheduler. It is very likely that under fixed preemption points, the speed-up factors are going to be even higher. Therefore, the upper-bound on speed-up factors derived for G-LP-EDF under floating NPR in Chapter 7 may actually be a lower bound in the general context of global limited preemptive scheduling (considering different mechanisms such as floating NPRs and fixed preemption points).

Finally, in Chapter 8, we presented a methodology that exploits the possibility of having faster processors to achieve a fine-grained control of preemption related overheads. Since the prospects of further increasing processor speeds is widely debated, for the sceptics, we would like to point out that our methodology can be instantiated in the context of other task parameters, e.g., relaxing deadlines and time periods [96], to achieve preemption control. Relaxing deadlines and time periods could be an even better strategy considering the fact that the entire execution time of the tasks may not scale linearly with the processor speed due to the effects of memory wall [93]. Note that we have demonstrated the infeasibility of using processor speed-up for preemption control under such an execution time model in Chapter 6. While in this context, as a final note, we would like to point out that the scepticism towards the possibility of having faster processors has always existed (as indicated by the opening sentence in Gene Amdahl’s
paper [97] from 1967), and the computers have (always) become faster.

In this thesis, we have considered different approaches to limited preemptive scheduling on multiprocessors. The results presented in this thesis facilitates the analysis of limited preemptively scheduled real-time tasks on uniprocessor and multiprocessor platforms.

9.3 Future Work

In this section, we discuss how the work presented in this thesis can be extended.

9.3.1 Minimizing Preemption Related Overheads by exploiting Time Triggered Scheduling

There exists a possibility of exploiting the determinism guaranteed by time triggered scheduling to minimize and efficiently account for preemption related overheads. The methodology may roughly follow the technique proposed by Dobrin et al. [103]. Firstly, the (deterministic) schedule that guarantees high schedulability while minimizing preemption related overheads is computed offline. This offline schedule can then be translated back to periodic real-time task attributes using the methodology proposed by Dobrin et al. [103]. Note that such an approach is different from the technique proposed by Dobrin and Fohler [104] to reduce preemptions in standard FPS. In [104], a simple re-ordering of the task executions takes place to eliminate preemptions, instead of carefully enabling preemptions at optimal locations.

The advantage of the proposed method is that the complexity of schedule generation can be handled offline, while at runtime, since the tasks are executed according to standard FPS, EDF or even table driven scheduling, the associated flexibility can be exploited. The possibility to obtain the best-of-both offline and online scheduling makes this interesting future work.
9.3.2 Preemption Overhead Accounting in Multiprocessor Systems

Accounting for preemption overheads under global limited preemptive EDF and FPS is an interesting area for future work. Current schedulability analysis techniques assume that the preemption overheads are added to the execution time of the NPRs. That is, a task-centric overhead accounting technique in which the preemption overhead is accounted for in the WCET of the preempted task, as opposed to preemption-centric overhead accounting in which the overheads are added to the WCET of the preempted task. In one of the publications related to this thesis [53], we proposed a preemption overhead accounting technique that combines task-centric and preemption-centric overhead accounting technique by accounting for a part of the overhead in the preempting tasks’ and the rest in the preempted tasks’ WCET. The schedulability was maximized by using a linear programming technique that optimizes the distribution of the overheads while maximizing schedulability. However, the basic technique introduced in [53] considered minimization of the total taskset utilization instead of individual task response times.

One possibility of extending the work presented in this thesis would involve investigating techniques to account for preemption overheads while minimizing task response times.

9.3.3 Probabilistic Schedulability Analysis for Mixed Criticality Systems with Fixed Preemption Points

When tasks consist of fixed preemption points, it may as well be possible to calculate the distribution of the preemption related overheads at each of these points (especially cache related preemption and migration delays). This enables a probabilistic worst case response time analysis of limited preemptively scheduled real-time tasks. The advantage of such an analysis becomes apparent in the context of mixed-criticality systems [87] where higher criticality tasks can be given stronger schedulability guarantees, while the lower criticality
tasks can be given weaker schedulability guarantees by assuming pessimistic and optimistic threshold probabilities respectively with respect to the overheads when placing preemption points. For example, in the case of high criticality tasks, preemption point placement and schedulability analysis can be performed considering worst case preemption overhead (even though the associated probability of occurrence might be negligible). On the other hand, for low criticality tasks, preemption point placement and schedulability analysis can be performed by considering a better than worst case overhead (e.g., the preemption overhead with the largest probability of occurrence), potentially improving efficiency.

In this context, developing methodologies to place preemption points considering the probabilities of the overheads and providing appropriate probabilistic guarantees to each task in the taskset, depending on its criticality, is an interesting area for future work.

### 9.3.4 Preemption Thresholds for Preemption Points to Improve Global Limited Preemptive Scheduling

Davis et al. [15] showed that eager and lazy approaches are in fact incomparable with respect to schedulability. Specifically, there are tasksets that eager preemption approach can schedule that the lazy approach cannot, and vice versa. Tasksets that are unschedulable under eager preemptions typically contain medium priority tasks with many preemption points and low priority tasks with large NPRs. On the other hand, tasksets unschedulable under lazy approach contains high priority tasks with short deadlines relative to their execution time (laxity). Consequently, in order to leverage on the best-of-both, there is a need to enable eager preemptions at some specified fixed preemption points in the task code, while disabling preemptions at a few other specified points. Bril et al. [105] presented an interesting solution that integrates preemption threshold scheduling with limited preemptive scheduling using fixed preemption points. This solution, may have a significant impact on the schedulability of real-time tasks scheduled using global limited preemptive FPS and EDF scheduling algorithms.
Investigating techniques to improve schedulability under global limited preemptive EDF and FPS, by combining the best-of-both eager and lazy approaches using approaches similar to Bril et al.’s [105] forms interesting future work.

9.3.5 A Practical Approach to Quantifying the Sub-optimality of Scheduling Algorithms

Two of the main criticisms that we received for our works on the sub-optimality of non-preemptive and limited preemptive scheduling [99] [32] are that, firstly, linear speed-up of execution time is usually not possible for most systems due to the effects of memory wall [93], and secondly, speed-up factor is not a useful metric because of the limitations of achieving speed-up without prohibitive increase in energy requirements and heat dissipation. In [68] we addressed the first of these concerns by relaxing the execution time model using the more realistic execution time model proposed by Marinoni and Buttazzo [90]. In that the execution time of a task is divided into two parts, one that is speed dependent and the other that is speed independent. This is similar to the parallelizable and non-parallelizable code in the Amdhal’s law [97].

Therefore, investigating the possibility of using the amount of code that must be parallelizable to guarantee a desired behavior in the schedule, as a metric to quantify the performance of scheduling algorithms is another potential future work.
Chapter 9. Conclusions and Future Work

Investigating techniques to improve schedulability under global limited preemptive EDF and FPS, by combining the best-of-both eager and lazy approaches using approaches similar to Bril et al.’s [105] forms interesting future work.

9.3.5 A Practical Approach to Quantifying the Sub-optimality of Scheduling Algorithms

Two of the main criticisms that we received for our works on the sub-optimality of non-preemptive and limited preemptive scheduling [99] [32] are that, firstly, linear speed-up of execution time is usually not possible for most systems due to the effects of memory wall [93], and secondly, speed-up factor is not a useful metric because of the limitations of achieving speed-up without prohibitive increase in energy requirements and heat dissipation. In [68] we addressed the first of these concerns by relaxing the execution time model using the more realistic execution time model proposed by Marinoni and Buttazzo [90]. In that the execution time of a task is divided into two parts, one that is speed dependent and the other that is speed independent. This is similar to the parallelizable and non-parallelizable code in the Amdhal’s law [97].

Therefore, investigating the possibility of using the amount of code that must be parallelizable to guarantee a desired behavior in the schedule, as a metric to quantify the performance of scheduling algorithms is another potential future work.

Bibliography


[37] Hilary Holz, Anne Applin, Bruria Haberman, Donald Joyce, Helen Purchase, and Catherine Reed. Research methods in computing: What are they, and how should we teach them? In *Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education*, 2006.


[77] Christoph Borchert, Horst Schirmeier, and Olaf Spinczyk. Return-address protection in c/c++ code by dependability


[95] Hennadiy Leontyev and James Anderson. A hierarchical multiprocessor bandwidth reservation scheme with timing


Appendix A

This appendix provides a closer look at the graphs given in Figures 5.5, 5.6 and 5.8. Even though the results in graph 5.7 showed a similar trend, the difference is not very visible and hence we omit it.

Figure 9.1: Number of preemptions under varying utilization.
Appendix A

This appendix provides a closer look at the graphs given in Figures 5.5, 5.6 and 5.8. Even though the results in graph 5.7 showed a similar trend, the difference is not very visible and hence we omit it.

Figure 9.1: Number of preemptions under varying utilization.
The Figure 9.1, which is an enlarged version of the graph in Figure 5.5, illustrates that G-LP-FPS with LPA generates fewer preemptions compared to G-LP-EDF with LPA as the task utilizations increase. This indicates that for tasksets with large utilization, global FPS based LP schedulers are most suitable for reducing the number of preemptions at runtime.

![Figure 9.1](image)

**Figure 9.1: Weighted number of preemptions under varying number of processors.**

The graph in Figure 9.2 is an enlarged version of the graph in Figure 5.6, and shows that G-LP-FPS with LPA generates fewer preemptions compared to G-LP-EDF with LPA. Note that when the number of processors increase, keeping the number of tasks constant, both G-LP-FPS and G-LP-EDF tends to show similar performance. However, as is clearly seen, for tasksets with large number of tasks

![Figure 9.2](image)

**Figure 9.2: Weighted number of preemptions under varying number of processors.**

The graph in Figure 9.3 is an enlarged version of the graph 5.8, that illustrates that for shorter NPR lengths, G-LP-FPS with LPA generates fewer preemptions than G-LP-EDF with LPA, while for larger NPR lengths, G-LP-FPS generates more preemptions. This is because, as we increase the percentage of WCET that forms an NPR from 75% to 100%, the tasks with small periods (and consequently small WCETs) tends to be fully non-preemptive because of the ceiling function. This eliminates many preemptions occurring due to the scenario described in Section 5.4.3 (see Example 17). Normally under G-LP-FPS with LPA, higher priority tasks released during the execution of the final
relative to the number of processors, G-LP-FPS is the most suitable for reducing the number of preemptions at runtime.

![Graph](Figure 9.1.png)

Figure 9.1: Enlarged version of the graph in Figure 5.5, illustrating that G-LP-FPS with LPA generates fewer preemptions compared to G-LP-EDF with LPA as task utilizations increase. This indicates that for tasksets with large utilization, global FPS-based LP schedulers are most suitable for reducing the number of preemptions at runtime.

![Graph](Figure 9.2.png)

Figure 9.2: Weighted number of preemptions under varying number of processors.

![Graph](Figure 9.3.png)

Figure 9.3: Weighted number of preemptions under varying NPR lengths.

The graph in Figure 9.3 is an enlarged version of the graph 5.8, that illustrates that for shorter NPR lengths, G-LP-FPS with LPA generates fewer preemptions than G-LP-EDF with LPA, while for larger NPR lengths, G-LP-FPS generates more preemptions. This is because, as we increase the percentage of WCET that forms an NPR from 75% to 100%, the tasks with small periods (and consequently small WCETs) tends to be fully non-preemptive because of the ceiling function. This eliminates many preemptions occurring due to the scenario described in Section 5.4.3 (see Example 17). Normally under G-LP-FPS with LPA, higher priority tasks released during the execution of the final
NPR of lowest priority task wait for it to complete its NPRs. On the other hand, under G-LP-EDF, since jobs that may have a higher priority under FPS, because of their short periods, may have larger absolute deadlines and hence can as well be preempted. When tasks with short periods become non-preemptive, these tasks can no longer be preempted in the situations described in Example 17. Hence, G-LP-EDF tends to incur fewer preemptions than G-LP-FPS (similar to their preemptive variants).
Populärvetenskaplig svensk sammanfattning

Datorsystem används idag i ett stort antal sammanhang som vi dagligen kommer i kontakt med, t.ex., låsningsfria bromsar (ABS) i en bil. En stor del av dessa datasystem kallas för realtidssystem, där systemet måste utföra en mängd uppgifter inom en fördefinierad tid. Schemaläggningsalgoritmer och analys används för att bestämma när dessa uppgifter utförs påen viss processor, för att, i slutändan, garantera deras slutförande före en fördefinierad tid, s.k. deadline. Anvndningen av modern hårdvara, såsom flerkärniga (multicore) processorer, introducerar nya utmaningar för att säkerställa aktualitet beräkningarna. En särskild utmaning är oförräntligbarheten som introduceras av såkallade preemptions, dvs. processbyten då en process (ett task) avbryts av ett annat task som har ett högre prioritet. Dessa avbrott tar tid att hantera och därmed medför en extra försening för andra task, som kan i värsta fall missa sina deadlines, som kan i sin tur ventyra mnniskoliv och miljön.

Denna avhandling föreslår nya tekniker och analyser för att hantera preemptions i säkerhetskritiska realtidssystem som använder multicore hårdvara. Vi introducerar nya analyser som avgör om alla task i de ovannämnda systemen klarar sina deadlines när de är schemalagda enligt befintliga prioritetsbaserade algoritmer. Avhandlingen kvantifierar också schemaläggningsbarheten av system som körs med så kallad limited preemptive schemaläggning, där task för avbryta
varandra ett fördefinierad max antal gånger. Vi använder, i detta syfte, möjligheterna som erbjuds av modern hårdvara med avseende påökningen av processorfrekvensen (hastigheten), såatt vissa task kan exekveras snabbare. Detta medför dock extra kostnader med tanke på den extra energin som krävs för att snabba upp processorerna. Denna avhandling introducerar också metoder för att variera processorhastigheten i syfte att begränsa antalet preemptions mellan task, för en optimal kostnad.
Popular Science Summary in English

Today, computer systems exist in a number of devices that we use on a daily basis e.g., antilock braking system (ABS) in a car. A large amount of these computer systems are so called real-time systems, where the system has to perform a variety of tasks before specified deadlines. A scheduling algorithm is typically used to guarantee that the computations meet their respective deadlines. However, the use of modern computing hardware, such as multicore processors, brings new challenges to the scheduling algorithms towards ensuring the timeliness of the computations. One particular challenge of interest is the unpredictability created by preemptions, i.e., the interruption of the execution of a task by another one with a higher importance/priority. Such preemptions typically take time from the system to administrate, and consequently lead to delays in the task executions, potentially causing deadline misses that in the worst case may endanger human lives and the environment.

This thesis proposes new scheduling techniques and associated analysis to limit preemptions to enable the use of modern multicore processors in safety critical real-time systems. In particular, we propose new analyses to determine whether or not deadlines will be missed under the existing and commonly used priority based scheduling approaches. The thesis also quantifies the worst case performance of limited preemptive scheduling in terms of the processor
speed-up required to achieve optimality in terms of schedulability. Modern computer systems offer the designer the possibility to speed up the processor to achieve a faster execution of certain tasks. This, however, comes with a cost in terms of increased energy consumption. This thesis also proposes techniques to modify task parameters or speed-up the processors to guarantee a specified bound on the number of preemptions to minimize associated costs.
Modern computer systems offer the designer the possibility to speed up the processor to achieve a faster execution of certain tasks. This, however, comes with a cost in terms of increased energy consumption. This thesis also proposes techniques to modify task parameters or speed-up the processors to guarantee a specified bound on the number of preemptions to minimize associated costs.