Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be “used for any purpose other than private study, scholarship, or research.” If a user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of “fair use” that user may be liable for copyright infringement.

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation.
The Van Houten library has removed some of the personal information and all signatures from the approval page and biographical sketches of theses and dissertations in order to protect the identity of NJIT graduates and faculty.
ABSTRACT

NEW DATA STRUCTURES, MODELS, AND ALGORITHMS FOR REAL-TIME RESOURCE MANAGEMENT

by

Xinfa Hu

Real-time resource management is the core and critical task in real-time systems. This dissertation explores new data structures, models, and algorithms for real-time resource management.

At first, novel data structures, i.e., a class of Testing Interval Trees (TITs), are proposed to help build efficient scheduling modules in real-time systems. With a general data structure, i.e., the TIT* tree, the average costs of the schedulability tests in a wide variety of real-time systems can be reduced. With the Testing Interval Tree for Vacancy analysis (TIT-V), the complexities of the schedulability tests in a class of parallel/distributed real-time systems can be effectively reduced from $O(m^2n\log n)$ to $O(m\log n + m\log m)$, where $m$ is the number of processors and $n$ is the number of tasks. Similarly, with the Testing Interval Tree for Release time and Laxity analysis (TIT-RL), the complexity of the online admission control in a uni-processor based real-time system can be reduced from $O(n^2)$ to $O(n\log n)$, where $n$ is the number of tasks. The TIT-RL tree can also be applied to a class of parallel/distributed real-time systems. Therefore, the TIT trees are effective approaches to efficient real-time scheduling modules.

Secondly, a new utility accrual model, i.e., UAM+, is established for the resource management in real-time distributed systems. UAM+ is constructed based on the timeliness of computation and communication. Most importantly, the interplay between
computation and communication is captured and characterized in the model. Under UAM+, resource managers are guided towards maximizing system-wide utility by exploring the interplay between computation and communication. This is in sharp contrast to traditional approaches that attempt to meet the timing constraints on computation and communication separately. To validate the effectiveness of UAM+, a resource allocation algorithm called IAUASA is developed. Simulation results reveal that IAUASA is far superior to two other resource allocation algorithms that are developed according to traditional utility accrual model and traditional idea. Furthermore, an online algorithm called IDRSA is also developed under UAM+, and a Dynamic Deadline Adjustment (DDA) technique is incorporated into IDRSA algorithm to explore the interplay between computation and communication. The simulation results show that the performance of IDRSA is very promising, especially when the interplay between computation and communication is tight. Therefore, the new utility accrual model provides a more effective approach to the resource allocation in distributed real-time systems.

Thirdly, a general task model, which adapts the concept of calculus curve from the network calculus domain, is established for those embedded real-time systems with random event/task arrivals. Under this model, a prediction technique based on history window and calculus curves is established, and it provides the foundation for dynamic voltage-frequency scaling in those embedded real-time systems. Based on this prediction technique, novel energy-efficient algorithms that can dynamically adjust the operating voltage-frequency according to the predicted workload are developed. These algorithms aim to reduce energy consumption while meeting hard deadlines. They can
accommodate and well adapt to the variation between the predicted and the actual arrivals of tasks as well as the variation between the predicted and the actual execution times of tasks. Simulation results validate the effectiveness of these algorithms in energy saving.
NEW DATA STRUCTURES, MODELS, AND ALGORITHMS FOR REAL-TIME RESOURCE MANAGEMENT

by
Xinfa Hu

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Department of Computer Science

May 2010
Copyright © 2010 by Xinfa Hu

ALL RIGHTS RESERVED
NEW DATA STRUCTURES, MODELS, AND ALGORITHMS
FOR REAL-TIME RESOURCE MANAGEMENT

Xinfa Hu

Dr. Joseph Leung, Dissertation Advisor
Distinguished Professor of Computer Science, NJIT

Dr. Michael A. Baltrush, Committee Member
Associate Professor and Chairperson of Computer Science, NJIT

Dr. Vincent Oria, Committee Member
Associate Professor of Computer Science, NJIT

Dr. Edwin Hou, Committee Member
Associate Professor of Electrical and Computer Engineering, NJIT

Dr. Jie Hu, Committee Member
Assistant Professor of Electrical and Computer Engineering, NJIT
BIOGRAPHICAL SKETCH

Author: Xinfa Hu  
Degree: Doctor of Philosophy  
Date: May 2010  

Undergraduate and Graduate Education:  

- Doctor of Philosophy in Computer Science, New Jersey Institute of Technology, Newark, NJ, 2010  
- Master of Engineering in Computer Science, Xi’an Jiaotong University, Xi’an, P. R. China, 2000  
- Bachelor of Engineering in Computer Science, Xi’an Jiaotong University, Xi’an, P. R. China, 1994  

Major: Computer Science

Technical Reports and Publications:

Xinfa Hu and Joseph Leung,  
“Integrating communication cost into the utility accrual model for the resource allocation in distributed real-time systems,”  

Xinfa Hu and Joseph Leung,  
“Testing interval trees for real-time scheduling systems,”  

Xinfa Hu, Guoliang Xing, and Joseph Leung,  
“Exploring the interplay between computation and communication in distributed real-time scheduling,” submitted to IEEE Transactions on Computers, October 2009.

Xinfa Hu and Guoliang Xing,  
This work is dedicated to
my wife, Fan Zhou,
and to
my parents, Songbing Hu and Baoxiu Li,
all of whom have dedicated so much of their lives,
and themselves,
to me.
Many thanks to my advisor, Professor Joseph Leung, for all his help throughout my PhD study.

Special thanks to Professor Michael A. Baltrush, Professor Vincent Oria, Professor Edwin Hou, and Professor Jie Hu for serving as members of the Committee.

I am very grateful to the Computer Science Department of NJIT for consistent financial support throughout my PhD study.

Special thanks to my friend, Dr. Guoliang Xing at the Michigan State University, for his valuable and constructive advice for my papers.

Special thanks to Professor Michael A. Baltrush, for helping me improve the presentation of my dissertation.

I wish to express my sincere gratitude to Professor Vincent Oria for his helpful advice, friendship, and encouragement throughout my PhD study.

I appreciate the advice and encouragement from Professor Andrew Sohn and Professor Alexander Thomasian. Without the encouragement from them and Professor Vincent Oria, I would never be able to accomplish my PhD study here at NJIT.

Many thanks to Professor Grace Wang for correcting my English errors.

And finally, I would like to acknowledge my debt to my wife, Fan Zhou. Without her consistent help, support, and encouragement, I would never have reached this milestone.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Background</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Objectives</td>
<td>3</td>
</tr>
<tr>
<td>2 TESTING INTERVAL TREES FOR REAL-TIME SCHEDULING SYSTEMS ...</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Definition and Properties of the TIT tree</td>
<td>7</td>
</tr>
<tr>
<td>2.2 TIT* Tree and Its Applications to Real-Time Scheduling Systems</td>
<td>9</td>
</tr>
<tr>
<td>2.3 TIT-V Tree and Its Applications to Real-Time Scheduling Systems</td>
<td>12</td>
</tr>
<tr>
<td>2.3.1 Definition and Properties of the TIT-V Tree</td>
<td>14</td>
</tr>
<tr>
<td>2.3.2 Operation on TIT-V Tree and Its Complexity</td>
<td>15</td>
</tr>
<tr>
<td>2.3.3 Using TIT-V Tree to Construct Feasibility Test for DDRAA</td>
<td>17</td>
</tr>
<tr>
<td>2.3.4 Using TIT-V Tree to Construct Feasibility Test for a Generic Resource Allocation Algorithm</td>
<td>20</td>
</tr>
<tr>
<td>2.4 TIT-RL Tree and Its Applications to Real-Time Scheduling Systems</td>
<td>21</td>
</tr>
<tr>
<td>2.4.1 Definition and Properties of the TIT-RL Tree</td>
<td>23</td>
</tr>
<tr>
<td>2.4.2 Operations on TIT-RL Tree and Their Complexities</td>
<td>27</td>
</tr>
<tr>
<td>2.4.3 Using TIT-RL Tree to Construct ACA</td>
<td>27</td>
</tr>
<tr>
<td>3 NEW UTILITY ACCRUAL MODEL FOR RESOURCE ALLOCATION IN ASYNCHRONOUS REAL-TIME DISTRIBUTED SYSTEMS</td>
<td>32</td>
</tr>
<tr>
<td>3.1 System Model</td>
<td>35</td>
</tr>
<tr>
<td>3.2 Task, Message, and Scheduling Models</td>
<td>36</td>
</tr>
<tr>
<td>3.3 The New Utility Accrual Model</td>
<td>38</td>
</tr>
<tr>
<td>3.3.1 Utility Function</td>
<td>38</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS
(Continued)

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.2 Utility Accrual Criteria</td>
<td>40</td>
</tr>
<tr>
<td>3.4 Interplay-Aware Utility Accrual Scheduling Algorithm</td>
<td>42</td>
</tr>
<tr>
<td>3.4.1 The Algorithm</td>
<td>42</td>
</tr>
<tr>
<td>3.4.2 Complexity Analysis</td>
<td>47</td>
</tr>
<tr>
<td>3.4.3 An Example</td>
<td>48</td>
</tr>
<tr>
<td>3.5 Simulation Analysis</td>
<td>51</td>
</tr>
<tr>
<td>3.5.1 Simulation Settings</td>
<td>52</td>
</tr>
<tr>
<td>3.5.2 Simulation Results</td>
<td>54</td>
</tr>
<tr>
<td>4 EXPLORING THE INTERPLAY BETWEEN COMPUTATION AND COMMUNICATION IN DISTRIBUTED REAL-TIME SCHEDULING</td>
<td>60</td>
</tr>
<tr>
<td>4.1 System Model</td>
<td>61</td>
</tr>
<tr>
<td>4.2 Scheduling Element Model</td>
<td>61</td>
</tr>
<tr>
<td>4.3 Utility Function</td>
<td>63</td>
</tr>
<tr>
<td>4.4 Dynamic Deadline Adjustment</td>
<td>66</td>
</tr>
<tr>
<td>4.5 Interplay-aware Distributed Resource Allocation Algorithm</td>
<td>70</td>
</tr>
<tr>
<td>4.5.1 Two-level Scheduling Framework</td>
<td>70</td>
</tr>
<tr>
<td>4.5.2 The Algorithm</td>
<td>71</td>
</tr>
<tr>
<td>4.5.3 Complexity Analysis</td>
<td>76</td>
</tr>
<tr>
<td>4.6 Simulation Analysis</td>
<td>78</td>
</tr>
<tr>
<td>4.6.1 Simulation Settings</td>
<td>79</td>
</tr>
<tr>
<td>4.6.2 Simulation Results</td>
<td>82</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS

## 5 CALCULUS CURVE BASED ONLINE REAL-TIME DYNAMIC VOLTAGE- FREQUENCY SCALING

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Calculus Curves</td>
<td>91</td>
</tr>
<tr>
<td>5.1.1 Arrival Curve</td>
<td>91</td>
</tr>
<tr>
<td>5.1.2 Service Curve</td>
<td>92</td>
</tr>
<tr>
<td>5.2 System and Task Model</td>
<td>93</td>
</tr>
<tr>
<td>5.3 Schedulability/Feasibility Analysis</td>
<td>93</td>
</tr>
<tr>
<td>5.3.1 Schedulability/Feasibility Analysis According to Preemptive Earliest Deadline First Policy</td>
<td>94</td>
</tr>
<tr>
<td>5.3.2 Schedulability/Feasibility Analysis According to Preemptive Fixed Priority Policy</td>
<td>96</td>
</tr>
<tr>
<td>5.4 Online Real-Time DVS Algorithms</td>
<td>97</td>
</tr>
<tr>
<td>5.4.1 History Window Based Prediction</td>
<td>97</td>
</tr>
<tr>
<td>5.4.2 Prediction-Enabled EDF Based Online Real-Time DVS Algorithm</td>
<td>102</td>
</tr>
<tr>
<td>5.4.3 Prediction-Enabled Fixed Priority Based Online Real-Time DVS Algorithm</td>
<td>106</td>
</tr>
<tr>
<td>5.4.4 Further Discussion on the Algorithms</td>
<td>110</td>
</tr>
<tr>
<td>5.5 Simulation Analysis</td>
<td>111</td>
</tr>
<tr>
<td>5.5.1 Simulation Settings</td>
<td>112</td>
</tr>
<tr>
<td>5.5.2 Simulation Results</td>
<td>113</td>
</tr>
</tbody>
</table>

## 6 CONCLUSION

## 7 FUTURE WORK
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPENDIX  THE ADJUST OPERATION ON TIT-V TREE (FOR CASE 4)</td>
<td>123</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>128</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Complexity Analysis</td>
<td>47</td>
</tr>
<tr>
<td>3.2</td>
<td>Parameters for the Task Graph in Figure 3.1</td>
<td>50</td>
</tr>
<tr>
<td>3.3</td>
<td>Simulation Settings(1)</td>
<td>53</td>
</tr>
<tr>
<td>3.4</td>
<td>Simulation Settings(2)</td>
<td>53</td>
</tr>
<tr>
<td>4.1</td>
<td>Complexity Analysis</td>
<td>77</td>
</tr>
<tr>
<td>4.2</td>
<td>Simulation Settings(1)</td>
<td>80</td>
</tr>
<tr>
<td>4.3</td>
<td>Simulation Settings(2)</td>
<td>81</td>
</tr>
<tr>
<td>4.4</td>
<td>Simulation Settings(3)</td>
<td>81</td>
</tr>
<tr>
<td>5.1</td>
<td>Simulation Settings</td>
<td>113</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The TIT tree</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>A TIT tree</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>The TIT* tree</td>
<td>9</td>
</tr>
<tr>
<td>2.4</td>
<td>Schedulability test by using TIT* tree</td>
<td>10</td>
</tr>
<tr>
<td>2.5</td>
<td>Deadline-driven heuristic resource allocation algorithm</td>
<td>13</td>
</tr>
<tr>
<td>2.6</td>
<td>The feasibility test algorithm</td>
<td>13</td>
</tr>
<tr>
<td>2.7</td>
<td>The TIT-V tree</td>
<td>14</td>
</tr>
<tr>
<td>2.8</td>
<td>Four cases</td>
<td>15</td>
</tr>
<tr>
<td>2.9</td>
<td>The TIT-V tree based feasibility test algorithm</td>
<td>17</td>
</tr>
<tr>
<td>2.10</td>
<td>Compute available vacancy</td>
<td>18</td>
</tr>
<tr>
<td>2.11</td>
<td>(a) and (b) compute $AvailableVacancy$ and (c) TIT-V tree after inserting $T'_6$</td>
<td>19</td>
</tr>
<tr>
<td>2.12</td>
<td>The generic resource allocation algorithm</td>
<td>21</td>
</tr>
<tr>
<td>2.13</td>
<td>Online admission control algorithm</td>
<td>23</td>
</tr>
<tr>
<td>2.14</td>
<td>The TIT-RL tree</td>
<td>26</td>
</tr>
<tr>
<td>2.15</td>
<td>TIT-RL tree based online admission control algorithm</td>
<td>29</td>
</tr>
<tr>
<td>2.16</td>
<td>Feasibility test</td>
<td>30</td>
</tr>
<tr>
<td>3.1</td>
<td>Task graph with precedence relationships</td>
<td>37</td>
</tr>
<tr>
<td>3.2</td>
<td>Utility functions</td>
<td>39</td>
</tr>
<tr>
<td>3.3</td>
<td>IAUASA scheduling algorithm</td>
<td>44</td>
</tr>
<tr>
<td>3.4</td>
<td>IAUASA scheduling</td>
<td>50</td>
</tr>
<tr>
<td>3.5</td>
<td>Utility ratios achieved vary with the increment of data volume</td>
<td>54</td>
</tr>
</tbody>
</table>
### LIST OF FIGURES
(Continued)

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.6</td>
<td>Utility ratios achieved vary with the increment of the workload of computation</td>
<td>55</td>
</tr>
<tr>
<td>3.7</td>
<td>Utility ratios achieved vary with the increment of the number of processors</td>
<td>55</td>
</tr>
<tr>
<td>3.8</td>
<td>Utility ratios achieved vary with the increment of channel speed</td>
<td>56</td>
</tr>
<tr>
<td>3.9</td>
<td>Utility ratios achieved vary with the increment of system utility</td>
<td>57</td>
</tr>
<tr>
<td>4.1</td>
<td>Task graphs</td>
<td>62</td>
</tr>
<tr>
<td>4.2</td>
<td>Utility function of scheduling element $E_{ij}$</td>
<td>64</td>
</tr>
<tr>
<td>4.3</td>
<td>Two-level scheduling framework</td>
<td>70</td>
</tr>
<tr>
<td>4.4</td>
<td>Interplay-aware distributed resource scheduling algorithm</td>
<td>74</td>
</tr>
<tr>
<td>4.5</td>
<td>Simplified message sequence chart for the normal case</td>
<td>75</td>
</tr>
<tr>
<td>4.6</td>
<td>Utility ratios achieved vary with the increment of the workload of computation</td>
<td>82</td>
</tr>
<tr>
<td>4.7</td>
<td>Utility ratios achieved vary with the increment of data volume</td>
<td>82</td>
</tr>
<tr>
<td>4.8</td>
<td>Utility ratios achieved vary with the decrement of the number of processors</td>
<td>83</td>
</tr>
<tr>
<td>4.9</td>
<td>Utility ratios achieved vary with the increment of channel speed</td>
<td>83</td>
</tr>
<tr>
<td>4.10</td>
<td>Utility ratios achieved vary with the increment of interplay factor $\alpha$</td>
<td>84</td>
</tr>
<tr>
<td>5.1</td>
<td>Schedulability analysis according to preemptive EDF policy</td>
<td>95</td>
</tr>
<tr>
<td>5.2</td>
<td>History window based prediction for EDF policy</td>
<td>99</td>
</tr>
<tr>
<td>5.3</td>
<td>History window based prediction for Fixed Priority policy</td>
<td>100</td>
</tr>
<tr>
<td>5.4</td>
<td>Power-aware prediction-enabled EDF algorithm</td>
<td>103</td>
</tr>
<tr>
<td>5.5</td>
<td>Frequency analysis</td>
<td>104</td>
</tr>
<tr>
<td>5.6</td>
<td>Schedulability analysis</td>
<td>106</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------------------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>5.7</td>
<td>Power-aware prediction-enabled fixed priority algorithm</td>
<td>107</td>
</tr>
<tr>
<td>5.8</td>
<td>Deadline miss with infinite levels of frequencies</td>
<td>114</td>
</tr>
<tr>
<td>5.9</td>
<td>Energy consumption and energy saving with infinite levels of frequencies</td>
<td>114</td>
</tr>
<tr>
<td>5.10</td>
<td>Energy savings with infinite levels of frequencies under different history window widths</td>
<td>115</td>
</tr>
<tr>
<td>5.11</td>
<td>Energy consumption and energy savings with limited levels of frequencies</td>
<td>116</td>
</tr>
<tr>
<td>5.12</td>
<td>Energy consumption and energy savings with varying execution/computation time</td>
<td>117</td>
</tr>
</tbody>
</table>
Real-time computer systems have wide applications in many fields in the real world, such as digital control, signal processing, medical diagnosis and monitoring, telecommunication, industrial automation, military command and control, and multimedia. Unlike general purpose computer systems, the tasks to be performed by these real-time computer systems have timing constraints, and the services provided by real-time computer systems must be delivered in a timely way. Whether the tasks could be accomplished within the specified timing constraints and the services could be provided in a timely way depend on whether the resources in the systems could be managed efficiently and the requests of resources could be always satisfied sufficiently timely. This makes real-time resource management the core and critical task in almost all real-time computer systems.

1.1 Background

Over the past few years, real-time resource management has been extensively studied in various flavors. While a lot of problems could be dealt with by employing existing techniques, many important problems are in need of exploration. Among them, how to find appropriate data structures for building efficient real-time resource management did not receive too much attention in the past. However, solutions to this problem are of great importance in real-time systems in the sense that well-designed data structures not only make resource management efficient (thus improve system performance in reducing
complexity), but also make more resources available for applications (thus improve system performance in meeting timing constraints).

The other important problem is the model for the resource management in asynchronous real-time distributed systems, which are emerging in many domains, including defense, telecommunication and industrial automation, for the purpose of strategic mission management. These systems are distinguished in the sense that they must be able to accommodate significant run-time uncertainties that are inherent in their application environment and system resource states. This violates the static, deterministic, synchronous premises on which most classical/conventional real-time computing concepts, theorems, and techniques are founded. Hence, how to establish an appropriate model for resource management in these systems is a core task. Resource management in asynchronous real-time distributed systems has been explored for years. Up to now, lots of work has been conducted under Jensen’s utility accrual models. These models are constructed based on the timeliness of computation or communication. Resource management under these models is limited due to the fact that they are inadequate for capturing the interplay between computation and communication, which are two main factors in asynchronous real-time distributed systems. Solutions to this problem will establish the foundations for more effective resource management in asynchronous real-time distributed systems.

Another important problem is concerned with the efficient power/energy management in those embedded real-time systems with random event/task arrivals. Most past and current work on power-efficient real-time resource management is based on classical/conventional task models, i.e., periodic, aperiodic and sporadic task models.
These models, however, are incapable of accommodating random task arrivals. A more general task model is needed to capture the characteristics of random task arrival, and corresponding foundations are needed for building power-efficient resource management in those embedded real-time systems.

1.2 Objectives

This dissertation explores the following techniques for the resource management in real-time systems: (1) new data structures, (2) new model and algorithms for asynchronous real-time distributed systems, and (3) new model, technique, and algorithms for embedded real-time systems.

The first objective is to establish new data structures for building efficient real-time resource management. Some new data structures are established and applied to the resource management of several classes of real-time systems. These new data structures not only help to construct efficient resource management, but also save processing resource and significantly improve system performance.

The second objective is to establish new utility accrual model for the resource management in asynchronous real-time distributed systems. The new model overcomes the inadequacy of existing utility accrual models and can fully capture the interplay between computation and communication, which are the two main factors in asynchronous real-time distributed systems. New resource allocation algorithms under the new model are developed. Extensive simulations show the excellence of these algorithms. The results validate the effectiveness of the new model for resource management in asynchronous real-time distributed systems.
The third objective is to establish new task model and foundations for power-efficient resource management in embedded real-time systems. Conventional task models are inadequate for accommodating the random (including burst) arrivals of tasks. The new general model adapts the concept of calculus curve from network calculus domain and uses calculus curves to characterize random event/task arrivals and system processing capacity. History window based prediction technique is established under the general task model. The prediction technique provides the foundation for online real-time Dynamic Voltage-frequency Scaling (DVS). Two online DVS algorithms are developed based on the prediction technique. Extensive simulations are conducted. Both algorithms exhibit excellent performance in energy saving.
CHAPTER 2
TESTING INTERVAL TREES FOR REAL-TIME SCHEDULING SYSTEMS

In real-time systems, the efficiency of the resource Scheduling Module (SM) is of critical importance [1, 2, 3]. An efficient SM not only implies the overhead of the SM is low but also makes it possible to obtain better decisions on resource allocation without loss of system performance. Better decisions usually are more time-consuming and can be obtained only at the cost of system performance. Due to the stringent timing constraints and the high cost of analyzing and computing the optimal resource allocation decisions, some online real-time scheduling systems have to sacrifice the optimality of their decisions for the speed with which the decision can be computed [5, 6].

The efficiency of a real-time SM depends not only on how efficient the underlying algorithms employed in the SM are but also on how efficiently these algorithms are implemented. On one hand, a good algorithm with poor implementation may still be unacceptable in practice. On the other hand appropriate implementation of the algorithm can further improve the efficiency of the SM. In the past, how to apply some novel and effective data structures to the SMs so as to improve their efficiency did not receive much attention. The author believes that by introducing effective data structures, the efficiency of many real-time SMs could be improved, which in turn will help to improve the performance of the system. This is of great importance in the domain of real-time systems. The author is motivated to find novel and effective data structures to help construct efficient SMs. Because feasibility analysis (or schedulability analysis) is the critical part of a SM, The author will focus on how to find novel and effective data structures for conducting efficient feasibility test. It is easy to see that the main task of the
feasibility analysis is actually to check whether a group of intervals (corresponding to the execution of tasks) could be arranged without conflicts between them. Hence, the author first introduces the Testing Interval Tree (TIT), a balanced binary tree that is constructed based on intervals, and use it as the basic data structure. The author then extends this data structure for different uses. The first extension of TIT tree is the TIT* tree, which does not rely on any specifics of the underlying scheduling/testing algorithm, and is a general data structure that can be applied to a wide variety of real-time scheduling systems to reduce the average cost of the schedulability test. The second extension of TIT tree is the Testing Interval Tree for Vacancy analysis (TIT-V), which is used to conduct vacancy (unoccupied intervals) analysis in some parallel/distributed real-time systems; whenever a task/message is to be added to the task/message set, the schedulability test computes the available vacancy for that task/message according to the current TIT-V tree. Lastly, the TIT tree is extended to the Testing Interval Tree for Release time and Laxity analysis (TIT-RL), which is used to conduct the admission control in a uni-processor based real-time service system; whenever a request arrives, the admission control component checks whether the requested service could be feasibly provided according to the current TIT-RL tree. Because the TIT trees can effectively reduce the cost of the corresponding feasibility/schedulability tests, they provide an effective approach to constructing efficient SMs.
2.1 Definition and Properties of the TIT Tree

Before proceeding to the discussion of the TIT tree, the author defines a simple task model, under which a task \( T \) is characterized by a triple \((r, d, e)\), where \( r \), \( e \) and \( d \) are the release time, the absolute deadline and the execution time of \( T \), respectively.

Figure 2.1 The TIT tree.

The TIT tree (Figure 2.1) is based on intervals and used for interval analysis. Its properties can be summarized as follows.

1. It is a balanced binary tree.
2. There are two types of nodes on it, i.e., the leaf nodes which represent tasks and the non-leaf nodes which represent intervals.
3. Every leaf node is characterized by a triple, which defines a valid interval for a task to execute. For example, \((r_{x1}, d_{x1}, e_{x1})\) defines a valid interval \((r_{x1}, d_{x1})\) for \( T_{x1} \) with start point \( r_{x1} \) and end point \( d_{x1} \).
4. Every non-leaf node defines an interval. For example, \((Start, End)\) defines an interval with start point \( Start \) and end point \( End \).
5. The interval of a non-leaf node covers those of its children. For example, \((Start_1, End_1)\) covers \((Start_{x1}, End_{x1})\) and \((Start_{x2}, End_{x2})\), and \((Start_2, End_2)\) covers \((r_{x1}, d_{x1})\) and \((r_{x2}, d_{x2})\), where \( r_{x1} \) and \( d_{x1} \) are the release time and absolute deadline of task \( T_{x1} \), respectively, and \( r_{x2} \) and \( d_{x2} \) are the release time and absolute deadline of task \( T_{x2} \), respectively.
6. The leaf nodes are placed in ascending order of their release times, and if more than one node has identical release time, they are placed in ascending order of their deadlines.
7. For any non-leaf node, the interval of its left child is smaller than that of its right child, compared first on start point and then on end point if needed. For example, for \((Start_i, End_i)\), either \((Start_i < Start_k)\) or \((Start_i = Start_k) \) and \((End_i \leq End_k))\) holds.
There are two basic operations on the TIT tree, i.e., \textit{Insert} and \textit{Delete/Remove}.

\textit{Insert} is invoked to insert a new task into the tree. It is accomplished in two steps. At the first step, it starts from the root of the TIT tree and searches down the tree to find an appropriate location where the new task should be placed. This step will identify a non-leaf node, and the new task should be inserted as its child. At the second step, the new task is put at the location that is identified in the first step. If the identified non-leaf node has only one child, \textit{Insert} only needs to insert the new task as the left or right child of that node; otherwise, the identified node is split into two nodes, and the intervals of the two nodes are reset accordingly. Figure 2.2 illustrates a TIT tree. Suppose that a new task \( N (12, 17) \) is to be inserted into the tree, node \( (11, 16) \) will be split into two nodes (say \( O1 \) and \( O2 \)); one of the nodes (say \( O1 \)) and \( F (13, 16) \) will become the left and right children of the other node (i.e., \( O2 \)), respectively. \( E (11, 13) \) and \( N (12, 17) \) will become the left and right children of \( O1 \), respectively. The intervals of \( O1 \) and \( O2 \) are both set to \( (11, 17) \) so as to cover the intervals of their children. If the split causes the TIT tree to lose balance, \textit{rotation} is needed to rebalance the tree. Throughout this chapter and the Appendix, the rotation operation is similar to that with an AVL tree \cite{7}. \textit{Insert} also includes a procedure to update the intervals of the nodes on the path starting from the parent of the new task to the root of the tree.

\textit{Delete/Remove} operation is invoked to delete a leaf node from the TIT tree. For this operation, two cases may exist. In the first case, it only needs to delete the leaf node, and no other operations are involved. In the second case, the removal of the leaf node causes the TIT tree to lose balance, and \textit{rotation(s)} is needed to rebalance the tree. Similar to \textit{Insert}, \textit{Delete} also includes a procedure to update related intervals.

It is easy to see that for a TIT tree containing \( n \) leaves, the height of the tree is bounded by \( O(\log n) \). For both \textit{Insert} and \textit{Delete}, their complexities are bounded by the height of the tree, i.e., \( O(\log n) \).
2.2 TIT* Tree and Its Applications to Real-Time Scheduling Systems

Schedulability tests are usually performed by calling the underlying scheduling algorithm to preprocess the whole task/message set. (In this section, the author uses tasks to describe TIT* tree, and illustrates how to apply TIT* tree to the schedulability test of tasks. The basic principles also apply to the schedulability test in message scheduling.)

The main problem with this approach is that whenever a task is added to the task set, to test the schedulability of the new task set, the system needs to process the whole task set. The overhead of the test will be very high if the task set constantly contains a large number of tasks (this is very likely in an online dynamic environment, where new tasks are constantly added to the system). This overhead, however, may be reduced due to the fact that the joining of the new task may influence only a limited number of tasks, not the whole task set. Test on the whole task set is needed only in the worst case.

![Figure 2.3 The TIT* tree.](image)

The TIT* tree proposed here fully realizes this fact. Whenever the system performs the test, it only needs to test the schedulability of the tasks that correspond to a subtree of a TIT* tree, which corresponds to the whole task set. The TIT* tree (Figure
2.3) is an extension of the TIT tree, and inherits all the properties of TIT tree including the following property.

(1) Every non-leaf node contains two pointers. One pointer points to the first task that is bounded by the interval of this node, and the other pointer points to the last task that is bounded by the interval of this node.

To see how to apply the TIT* tree to the schedulability test, let’s look at the example in Figure 2.4. For simplicity, the “first task” and “last task” pointers of all non-leaf nodes are omitted except those of the node with interval (11, 18). Suppose that currently there are seven unfinished tasks in the scheduling queue (i.e., A, B, C, D, E, F, and G) and another task H (15, 20) arrives, the schedulability test is performed in two steps.

**Step 1:** Find the set of tasks that may conflict with task H. This is accomplished by a checking procedure that starts from the root. At each node, it checks to see whether the interval of this node overlaps with that of task H. If the two intervals overlap, it checks the children of this node. This procedure repeats until it reaches a leaf node or a non-leaf node that satisfies: (1) both of its children overlap with task H, or (2) one of its child overlaps with task H and its children overlap with each other. In the case that even the root does not overlap with H, the schedulability test is not needed at all. (No task currently in the system conflicts with H.) For the above example, the checking procedure ends at the node with interval (11, 18).

**Step 2:** Once it identifies the node and hence the corresponding set of tasks, the schedulability test is conducted against this set of tasks plus H. In the above example, a schedulability test on tasks E, F, G and H is performed.

![Figure 2.4 Schedulability test by using TIT* tree.](image)
The average cost of the TIT* tree based schedulability test is analyzed as follows. Suppose that there are $n$ tasks currently on a TIT* tree, and the underlying task scheduling algorithm is preemptive Earliest Deadline First (EDF). The average cost of the schedulability test is computed as follows. (The average cost includes two parts, i.e., the cost of search and that of EDF to process the specified task set. For a TIT* tree containing $n$ tasks, its height is bounded by $(\log n+2)$. At height $i$, the number of nodes on a TIT* tree is at most $2^i$. The search will take $(i+1)$ steps, and the cost of EDF will be $O\left(\frac{n}{2^i \log \frac{n}{2^i}}\right)$.)

$$\text{Average Cost} = O\left(\frac{\sum_{i=0}^{\log(n+2)} 2^i ((i+1) + \frac{n}{2^i \log \frac{n}{2^i}})}{\sum_{i=0}^{\log(n+2)} 2^i}\right) = O((\log n)^2)$$

By comparison, the average cost of the schedulability test without the TIT* tree will be $O(n \log n)$.

It is easy to see that the advantage of TIT* tree lies in that it helps to reduce the number of tasks to be tested, and thus reduce the average cost of the schedulability test. Additionally, the advantage of TIT* tree does not rely on any specifics of the underlying scheduling algorithm, and this makes it a general data structure and applicable to a wide variety of scheduling systems with different scheduling policies. For example, the underlying scheduling algorithm could be the preemptive or non-preemptive version of Highest Priority First, Least Slack Time First, Highest Utility/Benefit First, or some other similar algorithm (the average cost of the schedulability test is still $O((\log n)^2)$). Further study reveals that the TIT* tree is applicable to those schedulability tests that need to
process the whole task set whenever a task is to be added to the task set, no matter whether the test is conducted online or offline, and whether the underlying scheduling algorithm is preemptive or non-preemptive.

2.3 TIT-V Tree and Its Applications to Real-Time Scheduling Systems

Consider a parallel/distributed real-time system containing $m$ processors. There are $n$ independent tasks to be dispatched to these processors. Suppose every task has a release time, an absolute deadline and the workload to be finished by it. Every task can be replicated, and the workload of the task can be partitioned and distributed to these replicas. Replicas are dispatched to processors (but more than one replica of the same task can not be dispatched to the same processor). Tasks/replicas are preemptively scheduled according to their deadlines on every processor. The objective is to find a mapping of tasks/replicas to processors such that the *deadline-satisfied ratio* (the ratio of the number of tasks whose deadlines are met to the total number of tasks) is maximized.

Because this problem is NP-hard, only heuristic/approximation algorithms can be employed in the real world. A simple heuristic approach is to first sort the tasks in ascending order of deadline and then test the feasibility of tasks one by one in that order. On every processor, tasks are also processed according to their deadlines. It turns out that this heuristic can be well applied to real system to solve the aforementioned and similar problems. For example, in [8], a best-effort algorithm called *DPR* is constructed according to this heuristic to maximize the deadline-satisfied ratio in a distributed real-time system, and another algorithm based on similar heuristic is also constructed to
achieve the same goal. The highest level framework of this heuristic is listed in Figure 2.5, which is similar to the highest level framework in [8].

**Figure 2.5** Deadline-driven heuristic resource allocation algorithm.

**DDRAA( T )**

<table>
<thead>
<tr>
<th>Input:</th>
<th>T = {T1, T2, ..., Tn}; /* the task set to be processed */</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>resource allocation result;</td>
</tr>
</tbody>
</table>

1. Sort tasks T1, T2, ..., Tn in ascending order of deadline;
2. For T = T1' to Tn' do /* T1' .. Tn' are in ascending order of deadline */
3. Determine_Replicas_Processors(T); /* determine the number of replicas and the processors for task T */

**Determine_Replicas_Processors (T )**

<table>
<thead>
<tr>
<th>Input:</th>
<th>T; /* the task to be processed */</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>determine the number of replicas and the processors for T if enough resource is available for it, otherwise do not allocate any resource for it;</td>
</tr>
</tbody>
</table>

Variables:
- PT = ∅; /* the set of processors that have NO replica of T */
- P; /* the set of all processors */
- PR = ∅; /* the set of processors that HAVE replicas of T */
- MinResponse; /* the minimum response time */
- PID; /* ID of the processor that has the minimum response time */

1. MinResponse = ∞;
2. PT = P - PR;
3. If (PT == ∅) return FAILURE;
4. For each processor q ∈ PT do
5. 
6. ResponseTime = EDF_AnalyzeResponse(q, T, |PR|+1); /* |PR|+1 is the current number of processors that will get a replica of T, and the workload of the replica that is to be tested on q is the total workload of T divided by (|PR|+1) */
7. If (ResponseTime < MinResponse)
8. 
9. MinResponse = ResponseTime;
10. PID = q;
11. PT = PT - {PID};
12. PR = PR + {PID};
13. If (MinResponse > T.d) goto step 1;
14. For each processor q ∈ PR - {PID} do
15. 
16. If (EDF_AnalyzeResponse(q, T, |PR| > T.d) goto step 1;
17. Return SUCCESS;

**Figure 2.6** The feasibility test algorithm.
The framework of the feasibility test (Determine_Replicas_Processors()) is listed in Figure 2.6. It is similar to the feasibility test contained in [8]. The subroutine EDF_AnalyzeResponse() contained in Determine_Replicas_Processors() uses Earliest Deadline First (EDF) rule to perform response time analysis (because tasks on every processor are processed according to EDF rule). It is easy to see that the complexity of this feasibility test is $O(m^2 n \log n)$, given $n$ independent tasks and $m$ processors. (In the worst case, a task $T$ may have $m$ replicas. To decide one replica, the test tries every processor that has no replica of $T$. The test takes $O(n \log n)$ time on every processor. Hence the total cost is $O(m^2 n \log n).$) In the following subsection, the TIT-V tree is introduced to construct more efficient feasibility tests.

2.3.1 Definition and Properties of the TIT-V Tree

In a TIT-V tree, a vacancy is an interval that is not occupied by any task. Every vacancy has a left-endpoint and a right-endpoint. The TIT-V tree (Figure 2.7) is used for vacancy analysis. Its properties can be summarized as follows.

![Figure 2.7 The TIT-V tree.](image)

(1) A TIT-V tree is an extension of the TIT tree.
(2) A node in a TIT-V tree is characterized by a 5-tuple $(S_i, E_i, vs_i, ve_i, v_i)$ (Figure 2.7), where $S_i$ and $E_i$ are the start and end points of interval $(S_i, E_i)$, $vs_i$ and $ve_i$
are the left-most and right-most points of the vacancies contained in \((S_i, E_i)\), and \(v_i\) is the total length of the vacancies contained in \((v_{S_i}, v_{E_i})\) (please note that there may be more than one vacancy within \((v_{S_i}, v_{E_i})\), and they are separated by some intervals that are occupied by tasks).

(3) For a non-leaf node, the interval of its left child is smaller than that of its right child, compared on start point. For example, for node \((S_k, E_k, v_{S_k}, v_{E_k}, v_k)\), the interval of its left child \(((S_a, E_a))\) is smaller than the interval of its right child \(((S_b, E_b))\), i.e., \((S_a < S_b)\).

(4) Given a non-leaf node in a TIT-V tree, the interval defined by its left child never overlaps with that by its right child, and the end point of its left child is equal to the start point of its right child. For example, in Figure 2.7, \((E_a = S_b)\) holds.

(5) For a non-leaf node, its parameters are decided according to those of its child/children. For example, in Figure 2.7, for node \((S_k, E_k, v_{S_k}, v_{E_k}, v_k)\), the following holds: 
\[ v_{S_k} = Min\{v_{S_a}, v_{S_b}\} = v_{S_a}, \quad v_{E_k} = Max\{v_{E_a}, v_{E_b}\} = v_{E_b}, \quad v_k = (v_a + v_b), \quad S_k = Min\{S_a, S_b\} = S_a \quad \text{and} \quad E_k = Max\{E_a, E_b\} = E_b. \]

### 2.3.2 Operation on TIT-V Tree and Its Complexity

![Image: Four cases.](image)

**Figure 2.8** Four cases.

The main operation on the TIT-V tree is *Adjust*. It is invoked when a task (say \(T=(r, d, e)\)) is to be inserted into a TIT-V tree (say \(Titv\)). \(Titv\) needs to be adjusted because some vacancies of it may be occupied by \(T\). The main work contained in *Adjust* is to find the left-most point of vacancy \(P_1\) (Figure 2.7) and the right-most point of vacancy \(P_2\), such that \((r \leq P_1 \leq P_2 \leq d)\), and the total length of the vacancies within interval \((P_1, P_2)\) is equal to \(e\). Once \(P_1\) and \(P_2\) are identified, all the vacancies within \((P_1, P_2)\) will be occupied by \(T\). *Titv* needs to be adjusted according to the remaining vacancies and those vacancies, created due to \(T\). To be more specific, four cases may exist (Figure 2.8).
Case 1: \( T = (t_7, t_8, e) \) and \((t_7, t_8)\) does not overlap with the interval defined by \( T_{itv} \) (i.e., \((S_0, E_0)\)). So, a new vacancy (i.e., \((E_0, t_7)\)) needs to be appended to the right side of \( T_{itv} \). Besides, a leaf node created according to \( T \) also needs to be appended to the right side of the tree.

Case 2: \( T = (t_1, t_3, e) \) and it can finish before \( S_0 \). A new leaf node needs to be created and appended to the left side of the tree. Please note that if \( T = (t_1, t_6, e) \), another vacancy \((E_0, t_6)\) needs to be appended to the right side of \( T_{itv} \).

Case 3: \( T = (t_2, t_3, e) \) and it can not finish before \( S_0 \) (i.e., part of the vacancies contained in \((S_0, E_0)\) will be occupied by \( T \)). the system needs to find the right-most point that will be occupied by \( T \) and adjust the tree accordingly (because all the vacancies between \( t_2 \) and that right-most point will be occupied by \( T \)). Similar to case 2, if \( T = (t_2, t_6, e) \) and it can finish before \( E_0 \), another vacancy \((E_0, t_6)\) needs to be appended to the right side of \( T_{itv} \).

Case 4: \( T = (t_4, t_5, e) \) and \( T \) will occupy some vacancies contained in \((S_0, E_0)\). This is the most complicated case. The system needs to find the left-most point and the right-most point that will be occupied by \( T \) and adjust the tree accordingly (because the vacancies between that left-most point and that right-most point will be occupied by \( T \)). Similar to case 2 and case 3, if \( T = (t_4, t_6, e) \) and it can finish before \( E_0 \), another vacancy \((E_0, t_6)\) needs to be appended to the right side of the TIT-V tree. (Please refer to the Appendix for more details about the process on this case. For the other cases, their processes can be easily constructed by employing subroutines in the Appendix.)

Because the complexity of every operation contained in Adjust is bounded by the height of the TIT-V tree, the complexity of Adjust is bounded by the height of the tree.
Given a TIT-V tree containing \( n \) leaves, the height of the tree is bounded by \( O(\log n) \).

Hence, the complexity of \textit{Adjust} is \( O(\log n) \).

2.3.3 Using TIT-V Tree to Construct Feasibility Test for DDRAA

\begin{verbatim}
Determine_Replicas_Processors (T)
Input: T; /*the task to be processed*/
Output: determine the number of replicas and the processors for T if enough resource is available for it, otherwise do not allocate any resource for it;
1: Max = 0;
2: For \( p = 1 \) to \( m \) do /* test T on processor 1 to processor n*/
3: Result[p], AvailableVacancy = Compute_Vacancy (p, T); /*compute the available vacancies on processor p*/
4: Result [p], NodeID = p; /*record the processor ID*/
5: If (Max < Result [p], AvailableVacancy)
6: Max = Result [p], AvailableVacancy;
7: Node = p; /*record the processor that has the maximum available vacancies*/
8: If (Max \( \geq \) T, e)
9: Dispatch T to Node; /*one node is enough*/
10: Return;
11: Sort Result [ ] in descending order of AvailableVacancy;
12: Num = 0;
13: Sum = 0;
14: For \( i = 1 \) to \( m \) do
15: Sum = Sum + Result [i]. AvailableVacancy;
16: If (Sum \( \geq \) T, e)
17: Num = i;
18: Break for loop;
19: If (Sum < T, e)
20: Return; /*no resource is allocated for T*/
21: Else
22: Sum = 0;
23: For \( i = 1 \) to (Num - 1) do
24: Make a replica of T, and dispatch it to node Result [i]. NodeID ;
25: The processing time of this replica is Result [i]. AvailableVacancy ;
26: Sum = Sum + Result [i]. AvailableVacancy ;
27: Make a replica of T, and dispatch it to node Result [Num]. NodeID ;
28: The processing time of this replica is set to (T, e - Sum) ;
29: Return;

Figure 2.9 The TIT-V tree based feasibility test algorithm.
\end{verbatim}

Now, the TIT-V tree is employed to reconstruct the feasibility test for DDRAA (listed in Figure 2.5). The pseudo code of the TIT-V tree based feasibility test is listed in Figure 2.9 and Figure 2.10.

\textit{Determine_Replicas_Processors}(T) (Figure 2.9) is used to determine the number of replicas of \( T \) and the processors to which these replicas can be feasibly dispatched.
Compute_Vacancy(p, T) (Figure 2.10) is used to compute the total length of the available vacancies for T on processor p.

<table>
<thead>
<tr>
<th>Compute_Vacancy (p, T)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> p; /<em>the processor ID</em>/</td>
</tr>
<tr>
<td>T; /*the task to be tested */</td>
</tr>
<tr>
<td><strong>Output:</strong> AvailableVacancy; /<em>the available vacancies within the interval (T, r, T, d) on processor p</em>/</td>
</tr>
</tbody>
</table>

/*TITv is the TIT-V tree constructed according to the tasks on p*/
| Case 1: (TITv is EMPTY) or (TITv. Root. S ≥ T. d) or (TITv. Root. E ≤ T. r) |
| Case 2: (other cases) |
| If (TITv. Root. S > t. r) |
| If (TITv. Root. E < t. d) |
| AvailableVacancy= (TITv. Root. S - T. r) + TITv. Root. v + (T. d - TITv. Root. E); |
| Else |
| Travel down the tree, compute the total length of vacancies within (TITv.Root.S, T.d), and record it in Vacancy; |
| AvailableVacancy= (TITv. Root. S - T. r) + Vacancy; |
| Else |
| If (TITv. Root. E < t. d) |
| Travel down the tree, compute the total length of the vacancies that lie at the left side of T. r (these vacancies can not be occupied by T), and record it in Uncovered_v; |
| AvailableVacancy= (T. d - TITv. Root. E) + (TITv. Root. v - Uncovered_v); |
| Else |
| Travel down the tree, compute the total length of the vacancies that lie at the left side of T. r (these vacancies can not be occupied by T), and record it in Uncovered_v; |
| Travel down the tree, compute the total length of vacancies within (TITv.Root.S, T.d), and record it in Vacancy; |
| AvailableVacancy= Vacancy - Uncovered_v; |
| Return (AvailableVacancy); |

**Figure 2.10** Compute available vacancy.

It is easy to see the complexity of Compute_Vacancy() is bounded by the height of the TIT-V tree, i.e., O(logn). Hence, the for loop (Figure 2.9) from step 2 to step 7 runs in O(mlogn). The sorting in step 11 can be done in O(mlogm). Because the Adjust operation on a TIT-V tree can be finished in O(logn) time, the complexity of steps 23–28 is O(mlogn). (In the worst case, every processor gets a replica of T, the corresponding TIT-V tree is adjusted, and there are at most m processors.)
Thus the complexity of \textit{Determine\_Replicas\_Processors()} is $O(m\log n + m \log m)$.

Compared to $O(m^2 \log n)$, this is a big improvement.

Figure 2.11(a) and Figure 2.11(b) show the computations of available vacancies for task $T_6=(11, 19, 6)$ and task $T'_6=(3, 19, 5)$ based on a given TIT-V tree. (This tree is constructed by inserting tasks $T_1=(0, 10, 2), T_2=(5, 13, 2), T_3=(14, 16, 1), T_4=(10, 17, 2)$ and $T_5=(6, 18, 4)$ into an empty TIT-V tree one by one.). As is shown the total length of the available vacancies for $T_6$ is 5 time units while that for $T'_6$ is 7 time units. Figure 2.11(c) is the adjusted TIT-V tree after inserting task $T'_6$.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure2_11}
\caption{(a) AvailableVacancy= 5 \hspace{1cm} (b) AvailableVacancy= 7 \hspace{1cm} (c) TIT-V tree after inserting $T'_6$}
\end{figure}

**Theorem 2.1** Under DDRAA, a replica $T= (r, d, e)$ can be feasibly scheduled on a processor $p$ if and only if the total length of the available vacancies returned by Compute\_Vacancy($p, T$) is equal to or larger than $e$.

**Proof:** $\leftarrow$ If $T$ is schedulable under preemptive EDF on processor $p$ (and no task misses its deadline), this certainly implies that there are enough vacancies within $(r, d)$ for accommodating $T$. Because Compute\_Vacancy($p, T$) always computes the total length of the available vacancies within $(r, d)$, the AvailableVacancy returned by Compute\_Vacancy($p, T$) will be equal to or larger than $e$. 

\begin{itemize}
  \item \textit{Theorem 2.1} Under DDRAA, a replica $T= (r, d, e)$ can be feasibly scheduled on a processor $p$ if and only if the total length of the available vacancies returned by Compute\_Vacancy($p, T$) is equal to or larger than $e$.
  \item \textit{Proof:} $\leftarrow$ If $T$ is schedulable under preemptive EDF on processor $p$ (and no task misses its deadline), this certainly implies that there are enough vacancies within $(r, d)$ for accommodating $T$. Because Compute\_Vacancy($p, T$) always computes the total length of the available vacancies within $(r, d)$, the AvailableVacancy returned by Compute\_Vacancy($p, T$) will be equal to or larger than $e$.
\end{itemize}
(1) Before the process on replica $T$, all tasks (or replicas) on processor $p$ are schedulable under preemptive EDF. The process on $T$ will have no influence on those tasks because tasks are processed in ascending order of their deadlines. Hence those tasks will still be schedulable, and they will occupy the same intervals even if $T$ is dispatched to processor $p$. (2) $\text{Compute\_Vacancy}(p,T)$ always computes the total length of the available vacancies within $(r, d)$. If the total length returned by it is equal to or larger than $e$, this implies that enough vacancies can be found for $T$. Obviously, it is safe to conclude that $T$ will be schedulable under preemptive EDF.

**Theorem 2.2** With TIT-V tree, the complexity of $\text{Compute\_Vacancy}()$ is $O(\log n)$, and the complexity of $\text{Determine\_Replicas\_Processors}()$ is $O(m\log n + m\log m)$, given $n$ tasks and $m$ processors.

**Proof:** This can be proved by previous complexity analysis.

2.3.4 Using TIT-V Tree to Construct Feasibility Test for a Generic Resource Allocation Algorithm

Further study shows that the TIT-V tree can be applied to a class of real-time scheduling systems. Figure 2.12 is the framework of a generic resource allocation algorithm. It is similar to the frameworks in [8, 9, 10, 11, 12, 13, 14]. This algorithm can be instantiated to achieve different objectives, e.g., maximizing deadline-satisfied ratio [8], maximizing utility/benefit [9, 10, 11, 12, 13, 14] (in this case, every task is associated with a utility value), maximizing deadline-satisfied ratio of the tasks with high priorities (in this case, every task is associated with a priority), etc. Accordingly, a scheduling rule is applied to every processor. To maximize deadline-satisfied ratio, EDF is applied; to maximize utility, a utility based discipline such as DASA [15] is applied; to maximize the deadline-satisfied ratio of the tasks with high priorities, the highest priority first rule is applied.

The sorting in GRAA (see below) will sort tasks according to the objective. For example, if the objective is to maximize utility, tasks are sorted in non-increasing order of utility value; if the objective is to maximize the deadline-satisfied ratio of the tasks with
high priorities, tasks are sorted in non-increasing order of priority, etc. GRAA uses the
same `Determine_Replicas_Processors()` as that in Figure 2.9, which in turn uses the same
`Compute_Vacancy()` as that in Figure 2.10.

<table>
<thead>
<tr>
<th>GRAA(Γ)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> Γ = {T₁, T₂, ..., Tₙ}; <em>the task set to be processed</em>/</td>
</tr>
<tr>
<td><strong>Output:</strong> resource allocation result;</td>
</tr>
<tr>
<td>1. Sort tasks T₁, T₂, ..., Tₙ according to the objective;</td>
</tr>
<tr>
<td>2. For T = T₁ to Tₙ do /* process tasks in sorted order */</td>
</tr>
<tr>
<td>3. Determine_Replicas_Processors(T);</td>
</tr>
</tbody>
</table>

Figure 2.12 The generic resource allocation algorithm.

**Theorem 2.3** Under an instantiated GRAA, a replica T = (r, d, e) can be feasibly scheduled on a processor p if and only if the total length of the available vacancies returned by `Compute_Vacancy(p, T)` is equal to or larger than e.

**Proof:** The proof is similar to that of Theorem 2.1 except that tasks are now processed according to the objective of the instantiated GRAA. □

**Theorem 2.4** Under an instantiated GRAA, the complexity of `Compute_Vacancy()` is \(O(\log n)\), and the complexity of `Determine_Replicas_Processors()` is \(O(m\log n + m\log m)\), given n tasks and m processors.

**Proof:** Because the instantiated GRAA uses the same `Compute_Vacancy()` and the same `Determine_Replicas_Processors()` as those used in DDRAA, Theorem 2.4 holds. □

### 2.4 TIT-RL Tree and Its Applications to Real-Time Scheduling Systems

This section studies TIT-RL tree and its application to the online admission control in a real-time system. Previous work on how to use novel data structures to improve the efficiency of online admission control can be found in [16]. There, an augmented red-black tree [7] is used for a real-time service system.

Consider an open system that is designed to provide online real-time services for customers. Customers send requests to the system and specify the types of the services
and the time intervals within which the services are needed. This system can be viewed as a model extracted from some applications such as online media service, call admission and other service [16, 17, 18, 19, 20, 21]. The system will enforce admission control over the requests. The policy of the admission control is simple: if a requested service can be feasibly provided, the request is admitted, and a corresponding task will be created to provide the specified service within the specified interval, otherwise, it is rejected. Suppose tasks are executed non-preemptively, and the system aims to (1) minimize the \textit{max-flow} (i.e., the maximum response time) [21] and (2) maximize the number of accepted requests. Because this is an online system, and it has no idea about the future requests, it employs some heuristics to process the requests. To achieve the first objective, the system always processes a task (created due to a request) at the earliest available time (but never earlier than its release time). The point behind this heuristic is that the online First In First Out (FIFO) discipline is optimal in minimizing max-flow for single processor [21]. To achieve the second objective, it tries to accept every request whenever possible since the system has no idea about the future requests.

Accordingly, the online admission control algorithm (ACA) can be constructed as Figure 2.13. ACA is used to check whether a new task $T(r, d, e)$ (created due to a new request) can be safely accepted ($T$ can be finished within interval $(r, d)$, and no accepted tasks miss their deadlines), given that there are $n$ accepted tasks, including those that have already been released and those that haven’t been released.

Admitted tasks will be put at the appropriate positions in the task queue. Whenever a task completes, the task scheduler always picks the next task from the head of the queue for execution.
ACA \((T, \Gamma)\)

**Input:** 
\(T\); /*the task to be tested*/

\(\Gamma = \{T_1, T_2, \ldots, T_n\}; /*the set of the admitted tasks*/

**Output:** accept or reject \(T\);

1. \(k = \text{Position} \ (T, \Gamma)\); /*find the appropriate position of \(T\) according to its release time*/
2. Check the feasibility of putting \(T\) at the \(k\)th position;
3. If (FEASIBLE)
   4. Insert \(T\) into the task queue at the \(k\)th position;
   5. Return FEASIBLE; /*\(T\) is accepted*/
5. Else
6. For \(i = (k+1) \) to \(n\) do
7. Check the feasibility of putting \(T\) at the \(i\)th position;
8. If (FEASIBLE)
9. Insert \(T\) into the task queue at the \(i\)th position;
10. Return FEASIBLE; /*\(T\) is accepted*/
11. Return INFEASIBLE; /*\(T\) is rejected*/

**Figure 2.13** Online admission control algorithm.

It is easy to see, the complexity of ACA is \(O(n^2)\). (Step 1 will take \(O(\log n)\) time by using binary search; step 2 will take \(O(n)\) time because the system needs to check all those tasks that are ordered after \(T\); step 7 will be executed \((n-k-1)\) times in the worst case; hence the complexity of steps 7 and 8 will be \(O(n^2)\).)

In the next subsection, the TIT-RL tree is introduced to reduce the complexity of ACA.

### 2.4.1 Definition and Properties of the TIT-RL Tree

The TIT-RL tree (Figure 2.14) is an extension of the TIT tree, and it is used for release time and laxity analysis. A TIT-RL tree has all the properties of a TIT tree except the following.

1. A non-leaf node in the TIT-RL tree is characterized by a triple \((\text{Start}, \text{End}, \text{LR})\) and a 4-tuple \((s_{\text{start}}, \text{unoccupied}, s_{\text{end}}, ll)\). \text{Start} and \text{End} are the start and end points of interval \((\text{Start}, \text{End})\), and \text{LR} (Last Release time) is the release time of the task that is last released within \((\text{Start}, \text{End})\). \text{s_{start}} and \text{s_{end}} identify the start and end points of current schedule within \((\text{Start}, \text{End})\). \text{unoccupied} is the total unoccupied time units within \((s_{\text{start}}, s_{\text{end}})\) (please note that this interval is contained in \((\text{Start}, \text{End})\) and is not necessarily equal...
to interval \((\text{Start}, \text{End})\), and \(ll\) (largest laxity) is the largest laxity of the schedule within \((\text{s}\_\text{start}, \text{s}\_\text{end})\). The largest laxity of a schedule within \((\text{s}\_\text{start}, \text{s}\_\text{end})\) is defined as the maximum number of time units that the schedule can be pushed backwards without causing any task to lose its deadline. This implies that a task with that much of processing time can be safely inserted at \(\text{s}\_\text{start}\) without causing any task to miss its deadline.

(2) The definition of a leaf node is similar to that of a non-leaf node except that the triple \((\text{Start}, \text{End}, \text{LR})\) is replaced with a 4-tuple \((r, d, e, \text{LR})\) (where \(r\), \(d\) and \(e\) are the release time, absolute deadline and execution time of a task \(T\), respectively). Please note that the \(\text{LR}\) in a leaf node is always set to the \(r\) of this node. Although it is not useful for a leaf node, it will facilitate the operations on the TIT-RL tree.

For a leaf node, its parameters are decided as follows.

\[
\begin{align*}
\text{LR} &= \text{s}\_\text{start} = r; \\
\text{unoccupied} &= 0; \\
\text{s}\_\text{end} &= (r+e); \\
ll &= [d-(r+e)];
\end{align*}
\]

(In the following discussion, for a leaf node, its \(r\) corresponds to the \text{Start}, and its \(d\) corresponds to the \text{End}.)

For a non-leaf node, its parameters are determined according to those of its child (children). Given a non-leaf node \(\text{Parent}\) having two children \(\text{Node1}\) and \(\text{Node2}\), its parameters are determined as follows.

\[
\begin{align*}
\text{Parent}\_\text{Start} &= \text{Min}\{\text{Node1}\_\text{Start}, \text{Node2}\_\text{Start}\} \quad \text{---(A1)} \\
\text{Parent}\_\text{End} &= \text{Max}\{\text{Node1}\_\text{End}, \text{Node2}\_\text{End}\} \quad \text{---(A2)} \\
\text{Parent}\_\text{LR} &= \text{Max}\{\text{Node1}\_\text{LR}, \text{Node2}\_\text{LR}\} \quad \text{---(A3)} \\
\text{Parent}\_\text{s}\_\text{start} &= \text{Min}\{\text{Node1}\_\text{s}\_\text{start}, \text{Node2}\_\text{s}\_\text{start}\} \quad \text{---(A4)} \\
\end{align*}
\]

For the \(\text{s}\_\text{end}\), \text{unoccupied} and \(ll\) of \(\text{Parent}\), they depend on the relationship between interval \((\text{Node1}\_\text{s}\_\text{start}, \text{Node1}\_\text{s}\_\text{end})\) and interval \((\text{Node2}\_\text{s}\_\text{start}, \text{Node2}\_\text{s}\_\text{end})\). To be more specific, four cases exist.

**Case 1:** \((\text{Node1}\_\text{s}\_\text{end} \leq \text{Node2}\_\text{s}\_\text{start})\). They are obtained according to (A5.1), (A6.1) and (A7.1), respectively.

\[
\begin{align*}
\text{Parent}\_\text{s}\_\text{end} &= \text{Node2}\_\text{s}\_\text{end} \quad \text{---(A5.1)} \\
\text{Parent}\_\text{unoccupied} &= (\text{Node1}\_\text{unoccupied} + \text{Node2}\_\text{unoccupied} + \\
& \quad \text{Node2}\_\text{s}\_\text{start} - \text{Node1}\_\text{s}\_\text{end}) \quad \text{---(A6.1)} \\
\text{Parent}\_ll &= \text{Min}\{\text{Node1}\_ll, (\text{Node1}\_\text{unoccupied} + \text{Node2}\_ll + \\
& \quad \text{Node2}\_\text{s}\_\text{start} - \text{Node1}\_\text{s}\_\text{end})\} \quad \text{---(A7.1)}
\end{align*}
\]
Case 2: \((\text{Node2.s.end} \leq \text{Node1.s.start})\). They are obtained according to (A5.2), (A6.2) and (A7.2), respectively.

\[
\begin{align*}
\text{Parent.s.end} &= \text{Node1.s.end} \quad \text{(A5.2)} \\
\text{Parent.unoccupied} &= (\text{Node1.unoccupied} + \text{Node2.unoccupied} + \\
& \quad \text{Node1.s.start-Node2.s.end}) \quad \text{(A6.2)} \\
\text{Parent.\text{ll}} &= \min\{\text{Node2.ll}, (\text{Node2.unoccupied} + \text{Node1.ll} + \\
& \quad (\text{Node1.s.start-Node2.s.end}))\} \quad \text{(A7.2)}
\end{align*}
\]

Case 3: \((\text{Node2.s.end} > \text{Node1.s.start} > \text{Node2.s.start})\). In this case, the overlap part of the two intervals needs to be taken into account, and they are obtained according to (A5.3), (A6.3) and (A7.3), respectively.

\[
\begin{align*}
\text{Parent.unoccupied} &= (\max\{(\text{Node1.unoccupied} + \text{Node1.s.start} - \text{Node2.s.end}), 0\} + \text{Node2.unoccupied}) \quad \text{(A6.3)} \\
\text{Parent.\text{ll}} &= \min\{\text{Node2.ll}, (\text{Node2.unoccupied} + \text{Node1.\text{ll}} - \\
& \quad (\text{Node2.s.end}-\text{Node1.s.start}))\} \quad \text{(A7.3)} \\
\text{if } ((\text{Node2.s.end-Node1.s.start}) < \text{Node1.unoccupied}) \\
\quad \text{Parent.s.end} &= \text{Node1.s.end} \quad \text{(A5.3)} \\
\text{else} \quad \text{Parent.s.end} &= (\text{Node1.s.end} + \text{Node2.s.end} - \\
& \quad \text{Node1.s_start-Node1.unoccupied}) \quad \text{(A5.3)}
\end{align*}
\]

Case 4: \((\text{Node1.s.end} > \text{Node2.s.start} > \text{Node1.s.start})\). Similar to Case 3, the overlap part of the two intervals needs to be taken into account, and they are obtained according to (A5.4), (A6.4) and (A7.4), respectively.

\[
\begin{align*}
\text{Parent.unoccupied} &= (\max\{(\text{Node2.unoccupied} + \text{Node2.s.start} - \text{Node1.s.end}), 0\} + \text{Node1.unoccupied}) \quad \text{(A6.4)} \\
\text{Parent.\text{ll}} &= \min\{\text{Node1.ll}, (\text{Node1.unoccupied} + \text{Node2.\text{ll}} - \\
& \quad (\text{Node1.s_end}-\text{Node2.s_start}))\} \quad \text{(A7.4)} \\
\text{if } ((\text{Node1.s_end-Node2.s_start}) < \text{Node2.unoccupied}) \\
\quad \text{Parent.s.end} &= \text{Node2.s.end} \quad \text{(A5.4)} \\
\text{else} \quad \text{Parent.s.end} &= (\text{Node2.s_end} + \text{Node1.s_end} - \\
& \quad \text{Node2.s_start-Node2.unoccupied}) \quad \text{(A5.4)}
\end{align*}
\]
Figure 2.14 The TIT-RL tree.
2.4.2 Operations on TIT-RL Tree and Their Complexities

The basic operations on the TIT-RL tree include *Insert* and *Delete/Remove*.

*Insert* is invoked to insert a new task. This operation is similar to the *Insert* operation discussed in Section 2.1 except that the parameters of nodes need to be adjusted according to the definition of TIT-RL tree. The adjustment of parameters is conducted according to what is discussed in Section 2.4.1.

*Delete/Remove* is invoked to delete a leaf node from a TIT-RL tree. This operation is similar to the *Delete/Remove* described in Section 2.1 except that the parameters of related nodes need to be adjusted according to the definition of TIT-RL tree after the removal of the leaf node. The basic idea involved in the adjustment is similar to what is discussed in Section 2.4.1.

It is easy to see that the complexities of both *Insert* and *Delete/Remove* are \( O(\log n) \), given a TIT-RL tree containing \( n \) tasks.

2.4.3 Using TIT-RL Tree to Construct ACA

Now, the TIT-RL tree is employed to reconstruct the ACA algorithm (Figure 2.13). The pseudo code of the TIT-RL tree based algorithm is listed in Figure 2.15. The basic idea of the new algorithm is the same as that contained in Figure 2.13. In Figure 2.15, ACA first checks some simple cases (steps 2-7). More complicated cases are processed by steps 8-32. Basically, it first finds the appropriate position for a new task \( T \) (step 9) and then checks whether it can be safely inserted into that position (steps 12-27). The checking procedure starts from *Temp* (this is the task before which the new task is to be inserted) and goes up the tree. If any node indicates deadline miss (i.e., the updated largest laxity of the node is less than zero), ACA stops current checking procedure and attempts to
insert the new task before the next task (step 17). This invokes a new checking procedure. If $T$ can not be inserted into any position, it is rejected (steps 14 and 29). Otherwise, it is inserted before $First$ (step 23) or inserted at the end of the task queue (step 31). See step 23 and step 31, when the new node is inserted in the queue, its parameters may be adjusted if needed. The adjustment is used to make the updated tree conform to the definition of TIT-RL tree. However, it never changes the actual executions of tasks, nor does it have any impact on the admission of future tasks.

Figure 2.16 shows how the test is conducted, given a TIT-RL tree and a new task (6, 10, 1). Please note that ACA updates the parameters of some nodes during the test. Whether the test succeeds or not, those parameters that are changed need to be restored. This procedure can be avoided by using two copies of parameters. One copy is used only for test, and its values are copied from the other one. The copy operation is needed only for those nodes whose parameters are changed in the test. During the test, the parameters of every related node are first copied and then changed.

**Definition 2.1** (Safe Acceptance) A task $T = (r, d, e)$ can be safely accepted if a suitable position (on the TIT-RL tree) can be found for $T$, and it can be inserted there without causing any task (including $T$ itself) to miss its deadline.

**Theorem 2.5** A new task $T = (r, d, e)$ can be safely accepted by the system if and only if ACA returns TRUE when it processes the corresponding TIT-RL tree.

**Proof:** ← (1) That $T$ is schedulable implies that a position, which is the earliest suitable position according to current system status, is available for $T$. (2) ACA always tries to find the earliest suitable position for $T$. Hence, ACA will be able to find that position, successfully insert $T$ there and return TRUE.

→ (1) Before the test, all existing tasks are schedulable. (2) When ACA conducts the test, it always tries to find the earliest suitable position for the new task such that the new task can be safely inserted there (i.e., it does not cause any existing task to miss its deadline, and there is enough vacancy to
accommodate it). ACA returns TRUE implies that such a position is available for $T$. Hence it can be safely accepted.

\[\text{ACA (Titrl, T)}\]

Input: Titrl; /*the TIT-RL tree that contains all accepted tasks */
T; /*the new task to be tested*/
Output: TRUE/FALSE; /*T is admitted/rejected */

1. Create a new node $\text{NewNode}$ according to $T$;
2. Case 1: ($T. d < \text{Titrl. root. Start}$)
   3. Insert $\text{NewNode}$ into the front of the queue;
   4. Return TRUE;
3. Case 2: ($T. r > \text{Titrl. root. End}$)
   4. Insert $\text{NewNode}$ into the end of the queue;
   5. Return TRUE;
4. Case 3: (Other cases)
   5. Search down the tree, and find the first leaf node $\text{First}$ such that
      ($\text{First. r} > T. r$) or (($\text{First. r} == T. r$) and ($\text{First. d} > T. d$));
   6. If (NOT FOUND)
      7. Goto step 28;
   8. Temp = $\text{First}$;
   9. If (($\text{Temp-prev. end} + T. e) > T. d$) /*the new task can not be safely accepted*/
      10. Return FALSE;
   11. Push in $T. e$ time units before $\text{Temp}$, and adjust its parameters;
   12. If (Temp. ll < 0) /*implies deadline miss*/
      13. $\text{First} = \text{First-next}$; /*attempt to insert the new task before the next task in the task queue*/
      14. If (First == NULL) /*implies the new task can not be inserted before ANY task in the task queue*/
      15. Goto step 28;
   16. Else Goto step 12;
   17. Temp = Temp-parent; /*go upward the tree*/
   18. If (Temp == NULL) /*implies the test succeeds*/
      19. Adjust the parameters of $\text{NewNode}$, and insert $\text{NewNode}$ before $\text{First}$;
      20. Return TRUE;
   21. Else
      22. Adjust the parameters of $\text{Temp}$;
      23. Goto step 16;
   24. If (($\text{Titrl. root. s_end} + T. e) > T. d$) /*the new task can not be safely accepted*/
      25. Return FALSE;
   26. Else
      27. Adjust the parameters of $\text{NewNode}$, and insert $\text{NewNode}$ into the end of the queue;
      28. Return TRUE;

Figure 2.15 TIT-RL tree based online admission control algorithm.
Figure 2.16 Feasibility test.
**Theorem 2.6**  Given $n$ existing tasks in the system, the complexity of ACA is $O(n \log n)$.

**Proof:** It is easy to see from Figure 2.15, the running time of one checking procedure in ACA is bounded by the height of the tree, i.e., $O(\log n)$. In the worst case, the checking procedure will be invoked at most $n$ times. Hence the complexity of ACA is $O(n \log n)$. 

The TIT-RL tree based ACA algorithm can also be applied to some parallel/distributed scheduling systems that are designed to achieve the same objectives as the service system described before. This can be easily accomplished by using the TIT-RL tree based ACA as a building block on every processor.
CHAPTER 3

NEW UTILITY ACCRUAL MODEL FOR RESOURCE ALLOCATION IN ASYNCHRONOUS REAL-TIME DISTRIBUTED SYSTEMS

In Distributed Real-Time Systems (DRTSs), communication cost is no longer negligible. Whether activities can be completed in time depends on whether the computations and the communications involved in them can be completed in a timely way. Hence communication, in terms of meeting timing constraint, is as important a factor as computation in DRTSs. Furthermore, the timeliness of computation relies on that of communication, and vice versa. This property requires that the resource allocation in DRTSs fully realize the interplay between computation and communication.

In the literature of resource scheduling for distributed real-time systems, a lot of work was devoted to the issues of minimizing response time [23, 24], load balancing that seeks to distribute the workload over nodes in a balanced way [25, 26], load sharing that tries to transfer workload from overloaded nodes to under-loaded nodes [27, 28, 29, 30] and maximizing the probability of meeting task deadlines [31]. Meanwhile, some work concentrated on minimizing the execution time of computation, or minimizing the communication cost, or both [32, 33, 34, 35].

In recent years, the utility/benefit related models have been intensively studied and applied to many DRTSs.

In [36, 37], a model called Q-RAM (QoS-based Resource Allocation Model) is proposed. Utility under Q-RAM is determined based on the Quality of Service (QoS) along multiple QoS dimensions (e.g., timeliness, reliability, security, and data quality). The QoS along every dimension depends on the amount of resource(s), the larger the
amount of resource, the higher the utility. For every application, a utility function is defined. Resources are apportioned among applications in a way such that the system utility is maximized. Applications are then set up according to the apportionment.

Similarly, in [38, 39], a utility model is proposed for adaptive resource management in dynamic distributed real-time systems. This model is further studied in [40, 41]. Utility under this model is defined as a function of extrinsic attributes and service attributes (or QoS levels). Resource allocation under this model is to find some settings of extrinsic and service attributes such that the system utility is maximized. Applications are then set up according to these settings.

The Jensen's Utility Accrual Models (UAM) [42, 43] takes a different approach for resource scheduling. Firstly, UAM focuses on timeliness, which is the main concern in almost all real-time systems. Accordingly, utility under UAM is defined as a function of the completion time of a task. For example, a utility function under UAM may be defined as a function of the completion time of a computation (task) [15] or a communication (task) [44]. Secondly, resource allocation under UAM is to find a schedule through scheduling simulation analysis such that the system utility is maximized. Extensive research has been conducted under UAM. For example, in [9, 10, 11, 12, 13, 14, 15, 42, 43, 44, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59], various techniques and algorithms are investigated under UAM. It was shown that UAM is very effective for resource allocation in soft real-time systems [48, 49, 50], especially under overload situations, which are usually a primary concern in most real-time systems.

To accommodate the dependency relationship between tasks, an extended UAM called Joint Utility Accrual Model (JUAM) is proposed in [45]. Under JUAM, the joint
utility of a task is defined as a function of the completion-time utility and progressive utility of some other tasks. The completion-time utility and progressive utility of a task depend on its completion time and progress.

In DRTSs, the timeliness of activities is inherently determined by the interplay between computations and communications. Nevertheless, the utility functions under UAM are mostly constructed based on computation or communication, and the interplay between computation and communication is not reflected in utility functions. Consequently, the interplay between computation and communication is not effectively and fully explored by resource scheduling under UAM. As resource scheduling model is the key component for ensuring system timeliness, it must capture and characterize the interplay between computation and communication. Motivated by this key observation, the author proposes a new utility accrual model called UAM⁺, which is constructed based on the timeliness of computation and communication. A utility function under UAM⁺ is defined as a function of the completion times of a computation and a communication, and the interplay between the computation and communication is also characterized in the function. Accordingly, resource managers under UAM⁺ are guided to perform resource allocation by exploring the interplay between computation and communication. The author also develops a resource allocation algorithm called IAUASA (see Section 3.4) to validate the effectiveness of the UAM⁺ model. Note that the interplay relationship is different from the joint dependency relationship under JUAM [45]. Firstly, the joint utility of a dependent task (say a communication task) is a function of the progressive utility and completion-time utility of a depended task (say a computation task), while utility under UAM⁺ is determined based on the completion times of the computation and
the communication. Secondly, under JUAM, the completion-time utility and progressive utility of the depended task depend on its completion time and progress and do not depend on the dependent task, while under UAM\(^+\), utility can not be determined solely based on the completion time of a computation or a communication because utility is a function of the completion times of the computation and communication.

### 3.1 System Model

Assume a distributed real-time system that contains \( n \) (homogeneous or heterogeneous) processors. These processors are interconnected by a network. There is a (logical) channel/connection from every processor to each of the other processors. On every processor, the tasks are preemptively scheduled according to their priorities, i.e., highest priority first. On every channel, the messages are processed according to their tag numbers. A message with tag number \( K \) must wait until the message with tag number \((K-1)\) is processed. When a message is transmitted over a channel, the end-to-end communication cost of it is directly proportional to the volume of the data in the message. Unless mentioned otherwise, it is assumed that it will take one unit of time to transmit one unit of data. Note that UAM\(^+\) does not rely on any specifics of the task scheduling policy, the message scheduling policy, the processor, and the underlying network. As is shown in Section 3.4, UAM\(^+\) only provides guidelines for resource managers by specifying the constraints on communication and computation and characterizing the interplay between them. The problem of how to explore the interplay and how to allocate specific resources to meet the constraints is addressed by resource managers, and is outside the scope of the model. Accordingly, a relatively simple system model outlined
above is assumed; this allows the author to focus on the evaluation of the UAM\textsuperscript{+} model rather than the discussion of the details of a complicated system. The author notes that such a methodology is commonly adopted in the literature. For example, in [2] (and the references therein), when the schedulability of a group of tasks is studied, only the execution of these tasks are counted; no switching overhead, contention on resources, or other overheads are assumed. These assumptions allow the schedulability test to be studied without being involved in the lengthy discussion of other specifics of the system. Similarly, when conducting dynamic voltage-frequency scaling, people assume that the energy expense and time overhead of voltage-frequency switching is negligible [60, 61, 62]. This enables them to concentrate on their models and algorithms.

### 3.2 Task, Message, and Scheduling Models

Suppose groups of tasks and their precedence relationships are characterized by Directed Acyclic Graphs (DAGs). Nodes and edges in a DAG (Figure 3.1) represent tasks and the precedence relationships among tasks. (Throughout this chapter, node and task are used interchangeably.) Furthermore, it is assumed the precedence relationship is established only due to data dependence (i.e., the successor has to wait for the completion of its predecessors only because it needs the data from its predecessors). Data is sent from a predecessor to a successor through message transmission. The weight associated with each edge in the graph represents the data volume that will be transmitted from the corresponding predecessor to the corresponding successor. For example, there will be $v_{34}$ units of data to be sent from $T_3$ to $T_4$. 
A task $T_i$ is characterized by a triple $(r_i, p_i, d_i)$, where $r_i, p_i$ and $d_i$ are the release time, processing time and relative deadline of $T_i$, respectively. A task is released only after it has received all required data from its predecessors. For those tasks that have no predecessors, their release times are set to the time when the task graph is released. Similarly, a message $M_{ij}$ (corresponds to $v_{ij}$) is characterized by a triple $(mr_{ij}, v_{ij}, md_{ij})$, where $mr_{ij}$ is the time when the data is ready, and $v_{ij}$ and $md_{ij}$ are the data volume and relative deadline of $M_{ij}$, respectively. For a message $M_{ij}$, its release time is decided according to $mr_{ij} = (r_i + f_i)$, where $r_i$ and $f_i$ are the release time and relative finishing time of $T_i$, respectively. For a task $T_i$ having $k$ predecessors, its release time is decided according to $r_i = \max\{mr'_{ji} + f'_{ji}\}$, where $1 \leq j \leq k$, $mr'_{ji}$ and $f'_{ji}$ are the release time and relative finishing time of $M'_{ji}$, respectively. If $T_i$ and $T'_j$ are dispatched to the same processor, the communication cost is zero, i.e., $f'_{ji} = 0$.

![Figure 3.1 Task graph with precedence relationships.](image)

A scheduling element is defined as the combination of the computation (the task) and communication (the message) along a directed edge (excluding the successor task) in the DAG. If a node in the graph has no successor, the corresponding scheduling element contains no communication. A 5-tuple is used to characterize a scheduling element $E_{ij}$.
(corresponds to $T_i \rightarrow T_j$): $(r_{ij}, COMP_{Dij}, COMM_{Dij}, comp_{ij}, comm_{ij})$, where $r_{ij}$ is the release time of task $T_i$ (i.e., $r_{ij} = r_i$), $COMP_{Dij}$ and $COMM_{Dij}$ are the relative deadlines of the computation and communication of $E_{ij}$, respectively (i.e., $COMP_{Dij} = d_i$ and $COMM_{Dij} = md_{ij}$), $comp_{ij}$ is the processing time of the computation of $E_{ij}$ (i.e., $comp_{ij} = p_i$), and $comm_{ij}$ is the data volume that needs to be transmitted by $E_{ij}$ (i.e., $comm_{ij} = v_{ij}$).

For example, in Figure 3.1, $E_{23} = (r_2, d_2, md_{23}, p_2, v_{23})$. For $E_i = \{E_{i,1}, E_{i,2}, \ldots, E_{i,\pi}\}$ (i.e., $E_i$ is the set of scheduling elements that originate from the same node $T_i$ in the DAG), all the scheduling elements in it have the same release time, processing time of computation and relative deadline of computation but may have different relative deadlines of communications and data volume. In addition, all scheduling elements will have the same completion time of computation, which is decided by the completion time of task $T_i$.

### 3.3 The New Utility Accrual Model

#### 3.3.1 Utility Function

Assume a simple utility function under UAM. Figure 3.2(a) is the utility function of a task $T_i$. $COMP_i$ is the timing constraint (for achieving positive utility) on $T_i$. Throughout this chapter, $COMP_i$ is assumed to be equal to the deadline of $T_i$. (It must be pointed out that the timing constraint on a task is not necessary equal to its deadline. In a soft real-time system, a computation may miss its deadline but still obtain some positive utility [43].) As is shown in Figure 3.2(a), $T_i$ makes contribution to the system only if it could complete no later than $COMP_i$. 

Now, suppose $T_i$ needs to send a message $M_{ij}$ to another task $T_j$. The deadline of $M_{ij}$ is $COMM_{D_{ij}}$. Under UAM$^+$, a scheduling element $E_{ij}$ will be defined, and the utility function for it will be defined as in Figure 3.2(b). As is shown in Figure 3.2(b), the utility function $(u_{ij}(\text{comp, comm}))$ of $E_{ij}$ is defined as a function of the completion time (the difference between the time when a computation/communication is released and the time when it is finished) of the computation and that of the communication. $COMP_{ij}$ and $COMM_{ij}$ are the timing constraints on computation and communication for achieving positive utility. Note that $COMP_{ij}$ is different from $COMP_{D_{ij}}$. The latter marks the deadline of the computation of $E_{ij}$ while the former marks the latest time point by which the computation of $E_{ij}$ should complete so as to achieve positive utility. Similarly, $COMM_{ij}$ is different from $COMM_{D_{ij}}$. (It must be pointed out that if $COMP_i$ is not equal to the deadline of $T_i$, $COMP_{D_{ij}}$ in Figure 3.2(b) should be replaced with $COMP_i$.) The introduction of $COMP_{ij}$ and $COMM_{ij}$ will make it natural to construct more complicated

Figure 3.2 Utility functions.
utility functions in soft real-time systems. (More complicated utility functions can be defined according to system level analysis [43].)

From Figure 3.2(b), two features of the utility function are observed under UAM+: (1) the utility that can be achieved relies not only on the completion time of the computation but also on that of the communication, and (2) the interplay between computation and communication has critical influence on determining the timeliness of computation and that of communication (and thus the utility obtained) (for example, a short completion time of computation will make a long completion time of communication acceptable (without loss of utility), and vice versa). Given a point \((cp_{ij}, cm_{ij})\) (where \(cp_{ij}\) and \(cm_{ij}\) are the completion time of the computation and that of the communication, respectively), if it is bounded in the shaded region (i.e., it satisfies \((cp_{ij} + cm_{ij}) \leq (COMP_Dij + COMM_Dij)\)), \(E_{ij}\) will contribute positive utility \((U_{ij})\) to the system. This provides a framework for resource managers to optimize resource allocation by exploring the interplay between computation and communication. By contrast, the resource managers under UAM will check whether \((cp_{ij} \leq COMP_Dij)\) and \((cm_{ij} \leq COMM_Dij)\) are met or not. If either of them can not be met, no utility can be obtained even if \((cp_{ij} + cm_{ij})\) is far less than \((COMP_Dij + COMM_Dij)\).

Because the construction of the utility function is an engineering approach [43], the author will not dwell on this topic in this dissertation.

3.3.2 Utility Accrual Criteria

Given a task graph containing a group of tasks \(T= \{T_1, T_2, \ldots, T_n\}\) and a processor set \(P= \{P_1, P_2, \ldots, P_m\}\) that is connected by a network, the author is interested in the goal that the resource managers should try to achieve and how to achieve the goal. The ability of a
model to provide unified criteria for resource allocation is not only central to but also critical for distributed real-time systems.

Like UAM, timing constraints under UAM+ are characterized in utility functions, and the goal for resource allocation is to maximize system-wide utility. Under UAM+, this problem can be formally expressed as follows. (Suppose the utility function of an element $E_{ij}$ is defined as that in Figure 3.2(b).)

Find a mapping $M: T \rightarrow P$ s.t.

$$Utility = \max \left\{ \sum_{i=1}^{n} \sum_{j=1}^{m} (U_{ij} \times X_{ij}) \right\}$$

where,

$$X_{ij} = \begin{cases} 
1 & \text{if } \{(cp_{ij} \leq COMP_{ij}) \land (cm_{ij} \leq COMM_{ij}) \land \left(COMP_{ij} \times \frac{cm_{ij}}{cp_{ij}} + cm_{ij} \right) \leq COMM_{ij} \land \left(U_{ij} \geq 0\right)} \\
0 & \text{otherwise;}
\end{cases}$$

$cp_{ij}$: the completion time of the computation of $E_{ij}$;
$cm_{ij}$: the completion time of the communication of $E_{ij}$;

Unlike UAM, UAM+ is constructed based on the timeliness of computation and communication. The interplay between computation and communication is also reflected in the utility function. This requires resource managers under UAM+ treat computation and communication as a whole, try to explore the interplay between them, and optimize resource allocation along two dimensions, i.e., computation and communication.

By contrast, a utility function under UAM is defined based on the timeliness of a computation or a communication, and the interplay between computation and communication is not reflected in the utility function. As a result, resource managers under UAM strive for meeting the timing constraints on computation and communication.
separately. The following example will further illustrate this issue. Consider a simple scenario, where there are two tasks (computations) $T_i$ and $T_j$, and $T_i$ needs to send a message $M_{ij}$ to $T_j$. Suppose $T_i$ can be finished very quickly but $M_{ij}$ will miss its deadline according to current system status; however, $T_i$ and $M_{ij}$ as a whole is still acceptable and will not cause any utility loss from the system level view. This scenario will typically fail the feasibility test under UAM.

3.4 Interplay-Aware Utility Accrual Scheduling Algorithm

To analyze, evaluate and validate the effectiveness of the UAM$^+$ model, this section presents a heuristic resource allocation algorithm IAUASA (Interplay-Aware Utility Accrual Scheduling Algorithm). IAUASA is constructed under UAM$^+$ and aims to maximize system-wide utility. Because the optimization problem of mapping tasks to processors is NP-hard, IAUASA attempts to find some suboptimal solutions through a heuristic approach. The algorithm is listed in Figure 3.3. To help describe the algorithm, the example in Figure 3.1 will be referred to throughout this section. The parameters for Figure 3.1 are listed in Table 3.2.

3.4.1 The Algorithm

Before proceeding to the detailed discussion on the algorithm, we first introduce an invalid node. A node is said to be invalid if the scheduling element set that is constructed based on it is currently identified as the best candidate set, but some elements in the set can not be feasibly scheduled. Thus the invalid flag is used to indicate that this node should not be selected immediately after this round; otherwise the same set as last will be constructed.
The frameworks of algorithm IAUASA and its subroutines are listed in Figure 3.3.

Given a task graph DAG, IAUASA will repeatedly process the remaining part of the task graph until every node is processed. The process is conducted according to two cases.

**Case 1:** A node (BestNode), whose current predecessor element set has the largest total utility, can be found (steps 2-8, IAUASA()). For example, in Figure 3.1, node T4 will be selected in the first round because its predecessor element set \{E_{01}, E_{02}, E_{13}, E_{23}, E_{24}, E_{34}, E_{44}\} currently has the largest total utility among all predecessor element sets. (For E_{44}, T4 is the predecessor of itself.) The schedulability of this element set is then checked (step 9, IAUASA()). If the schedulability test is successful, IAUASA will process related tasks and messages according to the schedulability test result (steps 12-20, IAUASA()). For every related task, IAUASA marks it as processed, dispatches it to the processor determined during the schedulability test, and assigns a priority to it according to the order it is processed on that processor during the schedulability test. For every related message, IAUASA dispatches it to the channel determined during the schedulability test, and assigns a tag number to it according to the order it is processed on that channel during the schedulability test. (Note that a message will not exist until the corresponding task creates it.)
**Algorithm: IAUASA (DAG,)**

**Input:** DAG;  
**Output:** scheduling result;  
1. Repeat  
2. MaxUtility = 0;  
3. For (every unprocessed and valid node Ti) do  
4. Utility = ComputeSupportingUtility(Ti, ElementSet);  
5. If (Utility > MaxUtility)  
6. BestNode = Ti;  
7. MaxUtility = Utility;  
8. Until (every node in DAG is processed);  
9. Mark BestNode as invalid node;  
10. Clear all invalid flags;  
11. For (every element ei in ElementSet)  
12. If (task Ti is not marked as processed)  
13. Mark Ti as processed;  
14. Assign a priority to Ti;  
15. Dispatch Ti to corresponding processor;  
16. Assign a tag number to message Mj;  
17. Dispatch Mj to corresponding channel;  
18. Until (every node in DAG is processed);  

**Subroutine: ComputeSupportingUtility (Ti, ElementSet)**

**Input:** Ti, ElementSet;  
**Output:** ElementSet, TempU;  
*the immediate predecessors of a task are linked by a list*/  
1. TempU = 0;  
2. Ti ← the first predecessor of Ti;  
3. While (Ti ≠ NULL)  
4. Add element ei to ElementSet;  
5. TempU = TempU + Ui;  
6. If (Ti is neither processed nor temporarily processed)  
7. Mark Ti as temporarily processed;  
8. TempU = ComputeSupportingUtility (Ti, ElementSet);  
9. Ti ← the next predecessor of Ti;  
10. Return <ElementSet, TempU>.

**Subroutine: SchedulabilityTest (ElementSet, BestNode)**

**Input:** ElementSet, BestNode;  
**Output:** FEASIBLE/INFEASIBLE;  
1. While (ElementSet ≠ NULL)  
2. Take element ei from ElementSet if all the predecessors of Ti are processed or temporarily processed;  
3. If (FindProcessor (Ti) ≠ UNDEFINED)  
4. Remove every ei from ElementSet;  
5. Else  
6. Return <INFEASIBLE>;  
7. Find suitable processor for task BestNode */  
8. If (FindProcessor (BestNode) ≠ UNDEFINED)  
9. Return <FEASIBLE>;  
10. Else  
11. Return <INFEASIBLE>.

**Subroutine: ComputeSupportedUtility (Ti)**

**Input:** Ti;  
**Output:** TempU;  
*the immediate successors of a task are linked by a list*  
1. TempU = 0;  
2. Ti ← the first successor of Ti;  
3. While (Ti ≠ NULL)  
4. TempU = TempU + Ui;  
5. If (Ti is not temporarily processed)  
6. Mark Ti as temporarily processed;  
7. TempU = TempU + ComputeSupportedUtility (Ti);  
8. Ti ← the next successor of Ti;  

**Subroutine: DetermineProcessor (Ti)**

**Input:** Ti;  
**Output:** MinProcessor;  
1. MinProcessor = UNDEFINED;  
2. For (every processor P) do  
3. TempCompComm = COMPLD + COMM_DI;  
4. If (TempCompComm < MinCompComm)  
5. MinCompComm = TempCompComm;  
6. MinProcessor = P;  
7. Return <MinProcessor>.

---

**Figure 3.3** IAUASA scheduling algorithm.
Case 2: A suitable node that meets the criteria of Case 1 can not be found. In this case, IAUASA tries to find an unprocessed, invalid but ready node (BestNode) such that all of its predecessors have been processed, and its successor element set currently has the largest total utility (steps 22-27, IAUASA()). The rationale behind this idea is that because BestNode currently supports the largest utility, IAUASA attempts to schedule it with the hope to achieve the largest potential utility because all utility supported by BestNode is unachievable without processing of it. Once BestNode is found, IAUASA tries to find a suitable processor for it. IAUASA then marks BestNode as processed, dispatches it to that processor, and assigns a priority to it. Additionally, IAUASA also assigns a tag number to every related message, and dispatches it to corresponding channel.

Given a node BestNode and its predecessor element set ElementSet, subroutine SchedulabilityTest() is used to find suitable channels and processors for related messages and tasks. Basically, SchedulabilityTest() first picks a task $T_i$ (step 2, SchedulabilityTest()), and then tries to find a processor for it. If such a processor is found, SchedulabilityTest() removes all predecessor elements of $T_i$ from ElementSet (step 5, SchedulabilityTest()). This procedure repeats until all elements in ElementSet are checked. SchedulabilityTest() then returns FEASIBLE, which indicates test success. If, however, during the test, any element can not be successfully processed, SchedulabilityTest() terminates, and returns INFEASIBLE, which indicates test failure.

Given a task $T_i$, subroutine ComputeSupportingUtility() is used to compute the total utility of the predecessor elements of $T_i$. For example, in Figure 3.1, the predecessor
elements of $T_4$ are $E_{01}$, $E_{02}$, $E_{13}$, $E_{23}$, $E_{24}$, $E_{34}$ and $E_{44}$, and the collective utility of its predecessor elements is 240 units according to Table 3.2.

Given a task $T_i$, subroutine $\text{ComputeSupportedUtility}()$ is used to compute the total utility of the successor elements of $T_i$. For example, in Figure 3.1, the successor elements of $T_2$ are $E_{23}$, $E_{24}$, $E_{34}$ and $E_{44}$, and the collective utility of its successor elements is 187 units according to Table 3.2.

Given a node $\text{BestNode}$, subroutine $\text{DetermineProcessor}()$ is used to find a suitable processor for $\text{BestNode}$ such that it will result in the minimum utility loss if $\text{BestNode}$ is dispatched to it. Because $\text{BestNode}$ is an invalid node, this implies that whichever processor it is dispatched to, at least one element (say $E_{x, \text{BestNode}}$) will lose its utility. Hence, $\text{DetermineProcessor}()$ tries to find a suitable processor so as to minimize the utility loss.

Given a node $T_i$, subroutine $\text{FindProcessor}()$ is used to find a suitable processor for $T_i$ such that all the communications between $T_i$ and its predecessors can be finished in a timely way (that is, for any predecessor $T_j$ of $T_i$, $(cp_{ji}, cm_{ji})$ is bounded in the valid region defined by utility function $u_{ji}$, where $cp_{ji}$ is the completion time of $T_j$ and $cm_{ji}$ is the completion time of the communication between $T_j$ and $T_i$), and $T_i$ completes earlier on this processor than on any of other processors. If such processor does not exist, UNDEFINED is returned by $\text{FindProcessor}()$.

See subroutine $\text{DetermineProcessor}()$. $\text{CommCompletionTime}(M_{ji})$ computes the completion time of message $M_{ji}$ on the channel from $P_j$ to $P$, where $P_j$ is the processor to which task $T_j$ is dispatched. Because messages on every channel are processed according to their tag numbers that are determined according to the order in which the messages are
dispatched to the channel, $CommCompletionTime(M_{ji})$ can obtain the absolute finishing time of $M_{ji}$ by simply adding the end-to-end communication cost of $M_{ji}$ to the absolute finishing time of the last message on the channel. The completion time of $M_{ji}$ is then obtained by subtracting its release time from its absolute finishing time.

See subroutine $FindProcessor()$. $CompCompletionTime(T_i, P)$ is used to compute the completion time of task $T_i$ on processor $P$. Specifically, $CompCompletionTime()$ computes the absolute finishing time of $T_i$ by simulating a preemptive priority scheduler to process all the tasks on processor $P$. The completion time of $T_i$ is then obtained by subtracting its release time from its absolute finishing time.

### 3.4.2 Complexity Analysis

Given $m$ processors and a task graph containing $n$ nodes and $l$ edges, the complexities of $IAUASA$ and its subroutines are listed in Table 3.1.

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$IAUASA()$</td>
<td>$O(mn^2(n \log n + l))$</td>
</tr>
<tr>
<td>$ComputeSupportingUtility()$</td>
<td>$O(l)$</td>
</tr>
<tr>
<td>$ComputeSupportedUtility()$</td>
<td>$O(l)$</td>
</tr>
<tr>
<td>$SchedulabilityTest()$</td>
<td>$O(lm(n \log n + l))$</td>
</tr>
<tr>
<td>$DetermineProcessor()$</td>
<td>$O(mn)$</td>
</tr>
<tr>
<td>$FindProcessor()$</td>
<td>$O(m(n \log n + l))$</td>
</tr>
<tr>
<td>$CompCompletionTime()$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>$CommCompletionTime()$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

The complexity of $CompCompletionTime()$ is $O(n \log n)$ because the process on $n$ tasks according to preemptive priority policy can be done in $O(n \log n)$ time.

The complexity of $CommCompletionTime()$ is $O(1)$ because it can be done within constant number of steps.
The complexity of \textit{DetermineProcessor()} is $O(mn)$ because there are $m$ processors, there are at most $n$ immediate predecessor elements for a given node or task, and the complexity of \textit{CommCompletionTime()} is $O(1)$.

The complexity of \textit{FindProcessor()} is $O(mn \log n)$ because there are $m$ processors, there are at most $n$ immediate predecessor elements for a given node/task, the complexity of \textit{CompCompletionTime()} is $O(n \log n)$, and the complexity of \textit{CommCompletionTime()} is $O(1)$.

The complexity of \textit{ComputeSupportingUtility()} is $O(l)$ because there are at most $l$ predecessor elements for a given node/task.

The complexity of \textit{ComputeSupportedUtility()} is $O(l)$ because there are at most $l$ successor elements for a given node/task.

The complexity of \textit{SchedulabilityTest()} is $O(lmn \log n)$ because there are at most $l$ elements in $\text{ElementSet}$, and the complexity of \textit{FindProcessor()} is $O(mn \log n)$.

The complexity of \textit{IAUASA()} is $O(lmn^3 \log n)$ because the Repeat-Until loop can be repeated at most $n^2$ times, and the complexity of \textit{SchedulabilityTest()} is $O(lmn \log n)$.

\subsection*{3.4.3 An Example}

Given three processors and the task graph in Figure 3.1 with parameter settings in Table 3.2, the scheduling result produced by \textit{IAUASA} is shown in Figure 3.4. Note that the utility associated with element $e_{55}$ is lost because its computation can not be finished by time 8. The whole process is conducted as follows.

At the first round, node $T_4$ is selected since its predecessor element set \{\textit{E01, E02, E13, E23, E24, E34, E44}\} currently has the largest total utility. The schedulability of this set is then checked. At first, task $T_0$ is picked because it has no unprocessed predecessor. $T_0$
can be dispatched to processor $P_0$, and it can be completed at time 3. After $T_0$ is processed, $T_1$ and $T_2$ can be picked (because their predecessor $T_0$ is processed). Suppose that $T_1$ is picked first, and it is dispatched to processor $P_0$. (In this way, the communication cost between $T_1$ and $T_0$ could be avoided, and $T_1$ has the same completion time on $P_0$ as it has on $P_1$ or $P_2$.) For element $E_{01}$, the completion times of the computation and communication of it are 3 and 0. It is easy to see that point (3, 0) is bounded in the valid region of utility function $U_{01}$. Next, $T_2$ can be picked. This task can be dispatched to either $P_1$ or $P_2$ because it will have identical completion time on $P_1$ and $P_2$ and its completion time on $P_1$ or $P_2$ will be less than that on $P_0$ due to $T_1$. Suppose that $T_2$ is dispatched to $P_1$. The communication cost between $T_2$ and $T_0$ will be 2 time units according to Table 3.2. Hence, $T_2$ is released at time (3+2)=5. For element $E_{02}$, the completion times of the computation and communication of it are 3 and 2. It is easy to see that point (3, 2) is bounded in the valid region of utility function $U_{02}$. After $T_2$ is processed, $T_3$ can be picked, and it is dispatched to $P_1$ to avoid the communication cost between $T_2$ and it. Because the communication cost between $T_1$ and $T_3$ is 2, $T_3$ is released at time 9. Next, $T_4$ is dispatched to $P_1$ to avoid the communication cost between $T_3$ and it. It is easy to check that the completion times of the computation and communication of every element ($E_{13}$, $E_{23}$, $E_{24}$, $E_{34}$ and $E_{44}$) is bounded in the valid region of the corresponding utility function.

At the second round, node $T_6$ is selected since the total utility of its predecessor element set $\{E_{06}, E_{66}\}$ is larger than that of $T_5$'s predecessor element set $\{E_{05}, E_{55}\}$. $T_6$ is dispatched to processor $P_2$. 
At the third round, node $T_5$ is selected, and the schedulability of element set $\{E_{05}, E_{55}\}$ is checked. Unfortunately, $E_{55}$ can not be successfully processed due to $T_5$.

At the last round, node $T_5$ is selected, and the schedulability of element set $\{E_{05}\}$ is checked (note that $E_{55}$ is not in the element set). $T_5$ is dispatched to processor $P_0$ because its completion time on $P_0$ is the smallest.

<table>
<thead>
<tr>
<th>Table 3.2 Parameters for the Task Graph in Figure 3.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

(In Table 3.2, every 7-tuple $(U_{ij}, COMM_{ij}, COMM_{Dij}, COMP_{ij}, COMP_{Dij}, comp_{ij}, comm_{ij})$ defines the parameters for a scheduling element $E_{ij}$. For example, 7-tuple $(1, 8, 2, 8, 6, 3, 2)$ in Table 3.2 defines the parameters for $E_{01}$ with $U_{01}=1$, $COMM_{01}=8$, $COMM_{D01}=2$, $COMP_{01}=8$, $COMP_{D01}=6$, $comp_{01}=3$ and $comm_{01}=2$.)

Figure 3.4 IAUASA scheduling.
3.5 Simulation Analysis

To see how well resource allocation can be achieved under UAM+, extensive simulations were conducted with IAUASA, and its performance is compared with that of two other resource allocation algorithms, i.e., DASA_variant and COMM.

DASA_variant is developed according to UAM model, and is a variant of DASA [15]. DASA is constructed under UAM and has been widely used for resource allocation in distributed real-time systems [9, 10, 11, 12, 13, 14]. DASA_variant works in a similar way to IAUASA except: (1) whenever feasible, it seeks to allocate resources to the task set that currently has the highest collective utility density, and (2) it is concerned about meeting deadlines when processing communications. The abovementioned task set is constructed by first selecting an unprocessed task and then recursively adding all its direct and indirect predecessors to the set. The collective utility density is defined as the ratio of the total utility of the tasks in the set to the total processing time of them. The goal of DASA_variant is also to maximize system-wide utility. It is worthy of mention that in [15], when DASA processes tasks (phases), it first computes collective utility densities based on every task and then processes tasks according to the collective utility densities associated with them. It never recomputes utility densities later on. By contrast, DASA_variant will dynamically recompute collective utility densities based on unprocessed tasks, and the utility associated with those processed tasks will not be included in later computation of collective utility densities. This is similar to how IAUASA computes collective utility.

COMM is developed based on traditional idea, which attempts to optimize resource allocation in distributed environments through minimizing communication cost.
Because this approach is widely adopted in both distributed systems [4] and real-time systems [2], the author is interested in whether IAUASA is preferable when compared with COMM. COMM works in a similar way to IAUASA except that it seeks to minimize system-wide communication cost whenever feasible. Specifically, when it processes a task graph, it repeatedly selects the task set that currently contains the highest collective communication cost, and tries to find a processor such that this set of tasks can be successfully scheduled on it. This process is repeated until all tasks in the graph are processed. Like DASA_variant, COMM treats computation and communication separately, and aims to meet their timing constraints.

The complexities of DASA_variant and COMM are in the same order as that of IAUASA. The simulations are conducted along five dimensions, namely, data volume (or load of communication), (workload of) computation, number of processors, channel speed, and system utility.

3.5.1 Simulation Settings
The simulations are classified into two groups. One group consists of 100 tasks. The task graph is taken from the STG (Standard Task Graph) lib of [63]. It is generated by samepred [63] with random seed 6. The method is described in [64]. The other group consists of 88 tasks. The corresponding task graph is also taken from the STG lib of [63]. This task graph is built from a real-world robot control application.

The corresponding simulation settings for these groups are listed in Table 3.3 and Table 3.4. Settings in Table 3.3 are used for the simulations along computation, data volume, number of processors, and system utility. Settings in Table 3.4 are used for the simulations along channel speed.
Table 3.3 Simulation Settings(1)

<table>
<thead>
<tr>
<th></th>
<th>Group-1</th>
<th>Group-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of tasks</td>
<td>100</td>
<td>88</td>
</tr>
<tr>
<td>Task graph</td>
<td><code>samepred</code></td>
<td><code>robot control</code></td>
</tr>
<tr>
<td>Channel speed</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>COMM_Dij</code></td>
<td>uniformly</td>
<td>uniformly</td>
</tr>
<tr>
<td></td>
<td>distributed</td>
<td>distributed</td>
</tr>
<tr>
<td></td>
<td>between [200, 300]</td>
<td></td>
</tr>
<tr>
<td><code>COMP_Dij</code></td>
<td>uniformly</td>
<td>uniformly</td>
</tr>
<tr>
<td></td>
<td>distributed</td>
<td>distributed</td>
</tr>
<tr>
<td></td>
<td>between [200, 300]</td>
<td></td>
</tr>
<tr>
<td><code>COMM_{ij}</code></td>
<td><code>(COMM_Dij + COMP_Dij)</code></td>
<td></td>
</tr>
<tr>
<td><code>COMP_{ij}</code></td>
<td><code>(COMM_Dij + COMP_Dij)</code></td>
<td></td>
</tr>
<tr>
<td><code>comp_{ij}</code></td>
<td>(1) initially generated uniformly from [1, 100];</td>
<td>(1) initially generated uniformly from [1, 100];</td>
</tr>
<tr>
<td></td>
<td>(2) varies from (Initial Value+0) to (Initial Value+100), with step length 10;</td>
<td>(2) varies from (Initial Value+0) to (Initial Value+100), with step length 10;</td>
</tr>
<tr>
<td><code>comm_{it}</code></td>
<td>(100+v), where v is uniformly distributed between [200, 300];</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(Initial Value+0) to (Initial Value+100), with step length 10;</td>
<td></td>
</tr>
<tr>
<td><code>U_{ij}</code></td>
<td>uniformly</td>
<td>uniformly</td>
</tr>
<tr>
<td></td>
<td>distributed</td>
<td>distributed</td>
</tr>
<tr>
<td></td>
<td>between [1, 100];</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(2) varies from 10 to 2;</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.4 Simulation Settings(2)

<table>
<thead>
<tr>
<th></th>
<th>Group-1</th>
<th>Group-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of tasks</td>
<td>100</td>
<td>88</td>
</tr>
<tr>
<td>Task graph</td>
<td><code>samepred</code></td>
<td><code>robot control</code></td>
</tr>
<tr>
<td>Channel speed</td>
<td>varies from 1.0, 1.1, 1.2, ..., until 2.0;</td>
<td></td>
</tr>
<tr>
<td><code>COMM_Dij</code></td>
<td>uniformly</td>
<td>uniformly</td>
</tr>
<tr>
<td></td>
<td>distributed</td>
<td>distributed</td>
</tr>
<tr>
<td></td>
<td>between [200, 300];</td>
<td></td>
</tr>
<tr>
<td><code>COMP_Dij</code></td>
<td>uniformly</td>
<td>uniformly</td>
</tr>
<tr>
<td></td>
<td>distributed</td>
<td>distributed</td>
</tr>
<tr>
<td></td>
<td>between [200, 300];</td>
<td></td>
</tr>
<tr>
<td><code>COMM_{ij}</code></td>
<td><code>(COMM_Dij + COMP_Dij)</code></td>
<td></td>
</tr>
<tr>
<td><code>COMP_{ij}</code></td>
<td><code>(COMM_Dij + COMP_Dij)</code></td>
<td></td>
</tr>
<tr>
<td><code>comp_{ij}</code></td>
<td>uniformly</td>
<td>uniformly</td>
</tr>
<tr>
<td></td>
<td>distributed</td>
<td>distributed</td>
</tr>
<tr>
<td></td>
<td>between [1, 100];</td>
<td></td>
</tr>
<tr>
<td><code>comm_{it}</code></td>
<td>(100+v), where v is uniformly distributed between [200, 300];</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(Initial Value+0) to (Initial Value+100), with step length 10;</td>
<td></td>
</tr>
<tr>
<td><code>U_{ij}</code></td>
<td>uniformly</td>
<td>uniformly</td>
</tr>
<tr>
<td></td>
<td>distributed</td>
<td>distributed</td>
</tr>
<tr>
<td></td>
<td>between [1, 100];</td>
<td></td>
</tr>
</tbody>
</table>

Because DASA\_variant allocates resources based on utility functions defined under UAM, to facilitate comparison and analysis, it is assumed that if a task in a DAG has k outgoing edges, it contains k virtual independent subtasks, which correspond to the computations of the k scheduling elements. These virtual subtasks have the same release
time, processing time and relative deadline. The utility defined (under UAM\(^+\)) along an edge is the utility defined (under UAM) for the corresponding subtask, and the utility inputted to $\textit{DASA\_variant}$ is of the same amount as the utility inputted to $\textit{IAUASA}$ though they have different meanings. For example, subtask $T_{01}$ (Figure 3.1) is associated with $U_{01}$ (see Table 3.2). To construct the utility function for $T_{01}$, only $U_{01}$, $\textit{COMP\_D}_{01}$ and $\textit{comp}_{01}$ of the corresponding 7-tuple in Table 3.2 are needed. In addition, it is assumed that for a given task $T_i$, if there is an edge entering it, the corresponding predecessor (of $T_i$) will be the predecessor of all its virtual subtasks.

### 3.5.2 Simulation Results

![Utility Ratio Graphs](image)

**Figure 3.5** Utility ratios achieved vary with the increase of data volume.

Figure 3.5 shows that the \textit{utility ratios} (defined as the ratio between the utility obtained and the utility available) achieved by $\textit{IAUASA}$, $\textit{DASA\_variant}$ and $\textit{COMM}$ decrease with the increase of data volume. For $\textit{DASA\_variant}$ and $\textit{COMM}$, the increasing data volume causes more and more communications to miss their deadlines, thus resulting in the loss of utility. For $\textit{IAUASA}$, the increasing data volume causes more and more scheduling elements to be unable to complete in a timely way.
Figure 3.6 Utility ratios achieved vary with the increase of the workload of computation.

Figure 3.7 Utility ratios achieved vary with the increase of the number of processors.

Figure 3.6 shows that the utility ratios achieved by the three algorithms also decrease with the increase of the workload of computation. For DASA_variant and COMM, the increasing workload of computation causes more and more computations to be unable to meet their timing constraints (for achieving utility), thus resulting in the loss of utility. For IAUASA, the increasing workload of computation causes more and more scheduling elements to be unable to complete in a timely way.
From Figure 3.7(b), it is easy to see that with the increase of the number of processors, the utility obtained by three algorithms increases. The reason is straightforward: for DASA_variant and COMM, more processors imply that more computations can meet their timing constraints, and for IAUASA, more processors allow more scheduling elements to be finished in a timely way. In Figure 3.7(a), three algorithms exhibit similar behavior: there is almost no utility increment even if the number of processors is increased. This is because there are very few parallel tasks/scheduling elements in robot control, and hence the parallel resources (i.e., processors) can not be fulfilled. Therefore, the utility ratios achieved by three algorithms do not increase with the increase in the number of processors.

![Graphs](a) robot control (b) samepred

**Figure 3.8** Utility ratios achieved vary with the increase of channel speed.

From Figure 3.8, it is easy to see that with the increase in channel speed, the utility obtained by three algorithms increases. The reason is that for DASA_variant and COMM, the increasing channel speed allows more and more communications to finish before their deadlines, and for IAUASA, the increasing channel speed makes more and more scheduling elements finish in a timely way. Also, as shown in both Figure 3.8(a)
and Figure 3.8(b), with the increase of channel speed, both IAUASA and DASA_variant eventually obtained all available utility while COMM only achieves this in the simulation with robot control. The main reasons are as follows. (1) Unlike that in robot control, the task graph in samepred contains lots of parallel tasks, which implies more competition on resources; thus tasks should be scheduled in an appropriate way so as to achieve the maximum utility ratio. (2) COMM conducts resource allocation according to communication cost, and at any time, it always tries to allocate resources for the task set that currently contains the largest collective communication cost; this heuristic eventually causes the unschedulability of some tasks and hence the loss of some utility in samepred.

![Graphs showing utility ratios](image)

**Figure 3.9** Utility ratios achieved vary with the increase of system utility.

In Figure 3.9, both Figure 3.9(a) and Figure 3.9(b) indicate that the utility ratios achieved by COMM experienced a small increment. The main reason is that with the increase of system utility, the ratio of the collective utility of the task sets selected by COMM to the whole system utility increased a small amount. In Figure 3.9(b), the utility ratio achieved by IAUASA always stabilizes at a high level while that by DASA_variant experienced a decrease. The reason is that with the increase of system utility, the amount
of unachievable utility also increases. Although DASA_variant strived for keeping the obtained utility ratio from decreasing by adjusting resource allocation, its ability is limited because it can not explore the interplay between computation and communication. By contrast, IAUASA is able to explore the interplay between computation and communication, and accordingly can adjust the resource allocation so as to keep the achieved utility ratio stabilized at a high level. In Figure 3.9(a), even the utility ratio obtained by IAUASA experienced a decrease. This is because the task graph of robot control is almost a chain with very few parallel tasks/scheduling elements (thus very few choices). This eventually limited IAUASA's ability to adjust resource allocation.

Figure 3.5 and Figure 3.9 show that the difference between the utility ratio achieved by IAUASA and those by DASA_variant and COMM in robot control is not as great as it is in samepred. The main reason is that the task graph in robot control is almost a chain, with very few branches. This results in very limited parallel tasks/scheduling elements and very few choices, and forces three algorithms to proceed nearly along the same path.

Figure 3.5 and Figure 3.9 show that the utility ratios achieved by IAUASA are always much higher than those by DASA_variant and COMM. The reason is that while DASA_variant and COMM strive for meeting the timing constraints on computation and communication separately, IAUASA processes computation and communication as a whole, and explores the interplay between them. Consequently, a communication that misses its deadline under DASA_variant or COMM may be acceptable under IAUASA if the corresponding computation completes early enough. Similarly, a computation that can not meet its timing constraint under DASA_variant or COMM may still be acceptable
under IAUASA if the corresponding communication completes early enough. As a result, a communication or computation that results in utility loss under DASA_variant or COMM may not necessarily cause utility loss under IAUASA. In Figure 3.8, like IAUASA, DASA_variant eventually obtained all system utility. This is due to the high channel speed, which enables every communication to finish before its deadline. This is the only situation under which DASA_variant may be comparable with IAUASA. For COMM, it can not even achieve all system utility under this situation (Figure 3.8(b)). This indicates that UAM, which is constructed based on the timeliness of computation or communication, is inadequate for capturing the interplay between computation and communication from the system level view. This eventually causes resource allocation under UAM to be unable to approach the optimal point as close as possible. Likewise, the traditional idea to optimize resource allocation by minimizing communication cost is also unable to approach the optimal point as close as possible because of its inability to explore the interplay between computation and communication.

Figure 3.5 to Figure 3.9 also show an important feature of IAUASA: the more choices (implies more parallelism among tasks/scheduling elements), the more IAUASA outperforms DASA_variant and COMM. This indicates the advantage of IAUASA in the resource allocation in parallel and distributed environments. This result conforms to the author's prediction and clearly demonstrates the motivation of proposing the UAM+ model, i.e., in distributed real-time systems, the interplay between computation and communication has critical influence on system timeliness. Optimizing resource allocation in a distributed real-time environment along one dimension (i.e., computation or communication) is thus inadequate for achieving system-wide objective.
CHAPTER 4
EXPLORING THE INTERPLAY BETWEEN COMPUTATION AND COMMUNICATION IN DISTRIBUTED REAL-TIME SCHEDULING

In Chapter 3, an extended utility accrual model called UAM+ was proposed and both computation and communication are integrated into the model. More importantly, the interplay between computation and communication is also captured in the model. Utility obtained under UAM+ depends on the completion times of computation and communication and the interplay between them. Similar to [42, 43], under UAM+, resource is allocated through task/message scheduling, and feasibility analysis is conducted through schedulability simulation analysis. Furthermore, resource managers under UAM+ are guided to conduct resource allocation by exploring the interplay between computation and communication rather than separately processing them. It is shown that the UAM+ is very effective for resource allocation in DRTSs [65].

This chapter furthers the study on UAM+ by exploring the online resource allocation under UAM+. In this chapter, the author proposes a class of General Utility Functions (GUFs) under the UAM+ model to fully capture and characterize the interplay between computation and communication in DRTSs. Accordingly, a technique called Dynamic Deadline Adjustment (DDA) is proposed to fully explore the interplay and help resource managers proceed towards utility accrual. An online algorithm called IDRSA, which integrates DDA technique, is then developed to perform resource scheduling for DRTSs. IDRSA adopts a two-level scheduling framework to decompose resource scheduling into subprocesses and distribute them to processing nodes so as to reduce the cost of resource scheduling through parallel processing. In addition, IDRSA incorporates
the Testing Interval Tree* (TIT*) (proposed in Chapter 2) to effectively reduce the costs of the schedulability tests for tasks and messages.

4.1 System Model
Assume a DRTS, where there are $m$ processing nodes connected by a network. Tasks are dispatched to nodes, and messages are transmitted over the network. One of the nodes works as a coordinator and the others are subordinates. Every subordinate has a local scheduler to process the tasks dispatched to it, and tasks are scheduled according to the preemptive Earliest Deadline First (EDF) rule [2]. There is a logical channel (or connection) connecting every pair of nodes. For every channel, there is a message scheduler to process the messages on it, and messages are scheduled according to non-preemptive EDF rule. In addition, it is assumed that there is a control channel connecting every pair of nodes. The control channels are dedicated to the transmission of control information. By exchanging control information, the coordinator and subordinates cooperate to perform resource scheduling. Furthermore, it is assumed that control channels provide guaranteed service and the cost of transmitting a control message from a source node to a destination node is bounded by a constant $C_c$.

4.2 Scheduling Element Model
Assume that groups of real-time tasks arrive at the system randomly. For every group of tasks, there are precedence relationships among them due to data dependences. Tasks and precedence relationships are depicted by Directed Acyclic Graphs (DAGs). In Figure 4.1,
DAG1 arrives at \( t_1 \), and DAG\( i \) arrives at \( t_i \). There is precedence relationship between task \( T_{1,0} \) and task \( T_{1,3} \) because \( T_{1,3} \) needs some data from \( T_{1,0} \) to start its execution.

![Task graphs](image)

**Figure 4.1** Task graphs.

A task \( T_i \) is denoted by a triple \((r_i, p_i, d_i)\), where \( r_i \), \( p_i \), and \( d_i \) are the release time, processing time, and relative deadline of \( T_i \), respectively. A task is released only after it has received all required data from its predecessor(s). Similarly, a message \( M_{ij} \) (sent to \( T_j \) by \( T_i \)) is denoted by a triple \((mr_{ij}, comm_{ij}, md_{ij})\), where \( mr_{ij} \) is the time when the data is ready, and \( comm_{ij} \) and \( md_{ij} \) are the data volume and relative deadline of \( M_{ij} \), respectively. For a message \( M_{ij} \), its release time is determined according to \( mr_{ij}=(r_i+f_i) \), where \( r_i \) and \( f_i \) are the release time and relative finishing time of \( T_i \), respectively. For a task \( T_i \) having \( k \) predecessors, its release time is determined according to \( r_i=\max\{(mr_{il}+f_{li})\} \), where \( 1 \leq l \leq k \), and \( mr_{li} \) and \( f_{li} \) are the release time and relative finishing time of \( M_{li} \), respectively. If \( T_i \) and \( T_j \) are dispatched to the same processor, the communication cost is zero, i.e., \( f_{ij} = 0 \).

A *scheduling element* is defined as the combination of the computation (i.e., the task) and the communication (i.e., the message) along a directed edge. A 5-tuple is used to denote a scheduling element \( E_{ij} \) (corresponds to \( T_i \rightarrow T_j \)): \((r_{ij}, COMP\_D_{ij}, COMM\_D_{ij},...\)
comp\textsubscript{ij}, \textit{comm}\textsubscript{ij} ), where \( r\textsubscript{ij} \) is the release time of task \( T\textsubscript{i} \) (i.e., \( r\textsubscript{ij} = r\textsubscript{i} \)), \( \text{COMP}_D\textsubscript{ij} \) and \( \text{COMM}_D\textsubscript{ij} \) are the deadlines of the computation and communication of \( E\textsubscript{ij} \), respectively (i.e., \( \text{COMP}_D\textsubscript{ij} = d\textsubscript{i} \) and \( \text{COMM}_D\textsubscript{ij} = m\textsubscript{dij} \)), \( \text{comp}_\textsubscript{ij} \) is the computation time of \( E\textsubscript{ij} \) (i.e., \( \text{comp}_\textsubscript{ij} = p\textsubscript{i} \)), and \( \text{comm}_\textsubscript{ij} \) is the data volume that needs to be transmitted by \( E\textsubscript{ij} \). For \( E\textsubscript{i} = \{E\textsubscript{i,1}, E\textsubscript{i,2}, \ldots, E\textsubscript{i,nk}\} \) (i.e., \( E\textsubscript{i} \) is the set of scheduling elements that originate from the same task \( T\textsubscript{i} \) in a DAG), all scheduling elements in \( E\textsubscript{i} \) have identical release times, computation times, and deadlines of computations (because their computations are the same, i.e., task \( T\textsubscript{i} \)), but they may have different data volume and deadlines of communications. In addition, all scheduling elements will have identical completion times of computations, which are determined by the completion of task \( T\textsubscript{i} \).

### 4.3 Utility Function

Because the interplay between computation and communication is the key factor in determining the timeliness of activities in DRTSs, resource scheduling must be interplay-aware. In this chapter, the author proposes a class of General Utility Functions (GUFs) under UAM\textsuperscript{+} to capture and characterize the interplay. These GUFs will provide guidelines for optimizing resource allocation by exploring the interplay between computation and communication.

The utility function of a scheduling element \( E\textsubscript{ij} \) is depicted in Figure 4.2. \( \text{COMP}_\textsubscript{ij} \) and \( \text{COMM}_\textsubscript{ij} \) are the timing constraints on computation and communication for achieving positive utility. Note that \( \text{COMP}_\textsubscript{ij} \) is different from \( \text{COMP}_D\textsubscript{ij} \) in that the latter marks the deadline of the computation of \( E\textsubscript{ij} \) while the former marks the latest time point by which the computation of \( E\textsubscript{ij} \) should finish so as to achieve utility. \( \text{COMP}_\textsubscript{ij} \) may be less or
greater than $COMP_D_{ij}$. Similarly, $COMM_{ij}$ is different from $COMM_D_{ij}$. As shown in Figure 4.2, if the completion times of computation and communication are bounded within the shaded region, uniform utility $U_{ij}$ can be achieved; otherwise, no utility can be obtained. To be more specific, there are four cases contained in this figure.

$$f_{ij}(comp, comm): A_{ij} \times comp + B_{ij} \times comm = C_{ij}$$

**Figure 4.2** Utility function of scheduling element $E_{ij}$.

**Case 1:** if the communication completes no later than $comm_2$, $U_{ij}$ units of utility can be achieved if only the computation finishes no later than $COMP_{ij}$.

**Case 2:** if the computation completes no later than $comp_2$, $U_{ij}$ units of utility can be achieved if only the communication finishes no later than $COMM_{ij}$.

**Case 3:** If the computation completes after $comp_2$ (say $comp_1$) and the communication completes after $comm_2$ (say $comm_1$), $comp_1$ and $comm_1$ must meet the constraint defined by $f_{ij}(comp, comm)$, i.e., $A_{ij} \times comp_1 + B_{ij} \times comm_1 \leq C_{ij}$, so as to make contribution to the system.
Case 4: The completion times of computation and communication are not bounded within the shaded region, and no utility can be obtained.

The existence of \( f_{ij}(\text{comp}, \text{comm}) \) sets a constraint on the combined completion times of computation and communication, and it is used to characterize the interplay between computation and communication. On one hand, the interplay between computation and communication reflects the fact that computation and communication function together to determine the timeliness of activities in DRTSs, and the completion of one side may force some constraint(s) on that of the other side; on the other hand, the interplay provides a space for exploring more flexible solutions, which makes it possible to adjust resource allocation for computation and communication as a whole based on the loads of computation and communication and currently available resources for computation and communication, rather than seeking to meet the constraints on computation and communication separately. This is of critical importance for DRTSs, where due to the interplay between computation and communication, resource scheduling must be interplay-aware and resource optimization should be performed by exploring a two-dimensional space, i.e., computation and communication.

Combined with utility function, the previous definition of \( E_{ij} \) can be extended to

\[
(r_{ij}, \text{COMP}_D_{ij}, \text{COMM}_D_{ij}, \text{comp}_{ij}, \text{comm}_{ij}, \text{COMP}_{ij}, \text{COMM}_{ij}, U_{ij}, f_{ij}(\text{comp}, \text{comm})).
\]

Given a task graph containing \( n \) tasks \( T_1, \ldots, T_n \), and for every scheduling element \( E_{ij} \), its utility function is defined as Figure 4.2, the optimization goal of resource scheduling is to maximize system utility. This can be formally expressed as follows.

Find a mapping \( M: \text{Set of tasks} \rightarrow \text{Set of processors} \) s.t.

\[
\text{Utility} = \max \{ \sum_{i=1}^{n} \sum_{j=1}^{n} (U_{ij} \times X_{ij}) \}
\]
where,
\[
X_{ij} = \begin{cases} 
1 & \text{if } \{(cp_{ij} \leq COMP_{ij}) \wedge (cm_{ij} \leq COMM_{ij}) \\
& \wedge (A_y \times cp_{ij} + B_y \times cm_{ij} \leq C_y) \\
& \wedge (U_y \geq 0)\}; \\
0 & \text{otherwise}; 
\end{cases}
\]

cp_{ij}: the completion time of the computation of \(E_{ij}\);

\(cm_{ij}\): the completion time of the communication of \(E_{ij}\);

Because the derivation and the construction of the utility functions are application-specific and are subject to a system-wide engineering process [43], the author will not dwell into this topic in this dissertation.

### 4.4 Dynamic Deadline Adjustment

Dynamic Deadline Adjustment (DDA) under UAM* is critical in the sense that resource scheduling in a DRTS must take the interplay between computation and communication into account, and computation and communication separately meeting their timing constraints is inadequate for system utility accrual. In Figure 4.2, suppose the computation of \(E_{ij}\) completes at time \(comp_1\), and the communication of \(E_{ij}\) completes at some time after \(comm_1\) but before \(COMM_{ij}\). It is easy to see that both computation and communication meet their timing constraints, but the obtained utility is zero. The reason is due to the interplay between computation and communication, neither the computation nor the communication can individually determine the timeliness of a scheduling element. Thus, to determine the final deadlines of computation and communication, the interplay between them needs to be taken into account. When the computation and communication of \(E_{ij}\) are dispatched to a processor and a channel, simply assigning \(COMP_{D_{ij}}\) and \(COMM_{D_{ij}}\) to their deadlines is inadequate for utility accrual. Their
deadlines should be adjusted in a way towards utility accrual. The DDA technique observes the following two rules.

**Rule 1:** DDA should be performed towards utility accrual.

**Rule 2:** The deadline adjustment of a task/message should not adversely influence those existing tasks/messages on a node/channel. This rule should be observed; otherwise the adjustment will invalidate previous process on tasks/messages.

Consider a scheduling element \( E_{ij} = (r_{ij}, COMP_{Dij}, COMM_{Dij}, comp_{ij}, comm_{ij}, COMP_{ij}, COMM_{ij}, U_{ij}, f_{ij}(comp, comm) ) \). If it could be successfully scheduled, there exists at least one scheme \( s \) satisfying the following conditions (I), (II), and (III). (In the following, \( resp_{-comp}^s \) is the response time of the computation of \( E_{ij} \), and \( resp_{-comm}^s \) is the response time of the communication of \( E_{ij} \).)

\[
(resp_{-comp}^s \leq COMP_{ij}) \land (resp_{-comm}^s \leq COMM_{ij}) \tag{I}
\]

\[
(A_{ij} \times resp_{-comp}^s + B_{ij} \times resp_{-comm}^s) \leq C_{ij} \tag{II}
\]

No scheduling elements processed before are adversely influenced \( \tag{III} \)

In this case, the deadlines of the computation and communication of \( E_{ij} \) are adjusted according to (A1) and (A2) (see below), and they are then dispatched to corresponding processor and channel. The rationales behind this idea are as follows.

(1) Intuitively, a larger adjustment slot \( \Delta_{ij} \) implies that processor and that channel as a whole are the least loaded. It is desirable to distribute some load to them from a balance point of view.
(2) There will be a larger space for adjusting the deadlines of the computation and communication of \(E_{ij}\), which makes it possible to leave more capacity to those elements that will be processed after \(E_{ij}\) so as to obtain as much utility as possible.

\[
deadline\_of\_computation = (resp\_comp_{ij} + \Delta_{ij}^k) \quad \text{(A1)}
\]

\[
deadline\_of\_communication = (resp\_comm_{ij} + \Delta_{ij}^k) \quad \text{(A2)}
\]

\[
\Delta_{ij}^k = \max\{\Delta_{ij}^s\} \quad (1 \leq s \leq (m-1), \text{ and suppose there are (m-1) schemes})
\]

\[
\Delta_{ij}^s = \frac{(C_{ij} - A_{ij} \times \text{resp}\_comp_{ij} - B_{ij} \times \text{resp}\_comm_{ij})}{(A_{ij} + B_{ij})}
\]

In the case that only condition (III) is unsatisfiable due to the communication of \(E_{ij}\), \(T_i\) and \(T_j\) should be dispatched to the same processor if feasible. In all other cases, \(E_{ij}\) will be put aside until the second time it is selected, and its utility is set to zero.

When \(E_{ij}\) is processed the second time, the deadlines of its computation and communication are adjusted according to the following cases.

**Case 1:** Condition (III) is unsatisfiable. In this case, the deadline of its computation and the release time and deadline of its communication are first adjusted according to (A3)–(A5) (see below), and the final deadlines of its computation and communication are then determined according to (A6) and (A7) (see below).

In the following, \(d_{\text{comp}}^s\) \((1 \leq s \leq (m-1))\) is the deadline of the computation of the last element adversely influenced by \(E_{ij}\), and \(\text{miss}^s\) is the missed time interval of the computation of that element. \(r_{\text{comm}}^s\) and \(d_{\text{comm}}^s\) are the release time and deadline of the
communication of the last element adversely influenced by $E_{ij}$, and $d^+_{comm}$ is obtained by applying a small adjustment to $d^+_{comm}$. $speed^s$ is the channel speed.

$$\text{deadline of computation}^s = d^+_{\text{comp}} + \text{miss}^s$$  \hspace{1cm} (A3)  
(if computation causes problem)

$$\text{release time of communication}^s = r^s_{\text{comm}}$$  \hspace{1cm} (A4)  
(if communication causes problem)

$$\text{deadline of communication}^s = \max\{d^+_{\text{comm}} + \frac{\text{comm}^s_{ij}}{speed^s}\}$$  \hspace{1cm} (A5)  
(if communication causes problem)

**Case 2:** All other cases. The deadlines of its computation and communication are adjusted according to (A6) and (A7) (see below). The rationales behind this idea are as follows.

1. Because all currently ready scheduling elements have zero utility, assigning smaller deadlines to one element will not cause utility loss of the other elements.

2. This will help to minimize the response times of those scheduling elements that violate conditions (I), (II) and (III).

In all other cases the deadline of computation and that of communication are adjusted according to (A6) and (A7).

$$\text{deadline of computation} = \text{resp}_{\text{comp}}^k$$  \hspace{1cm} (A6)

$$\text{deadline of communication} = \text{resp}_{\text{comm}}^k$$  \hspace{1cm} (A7)

$$\text{resp}_{\text{comp}}^k + \text{resp}_{\text{comm}}^k = \min\{(\text{resp}_{\text{comp}}^s + \text{resp}_{\text{comm}}^s)\} \quad (1 \leq s \leq (m-1))$$
4.5 Interplay-aware Distributed Resource Scheduling Algorithm

This section discusses a distributed resource scheduling algorithm, which integrates the DDA technique to explore the interplay between computation and communication. Because resource scheduling in DRTSs is inherently complicated, this algorithm adopts some effective approaches to reduce its complexity. These approaches include the two-level scheduling framework and the Testing Interval Tree* (TIT*). The two-level scheduling framework is adopted to decompose resource scheduling into subprocesses and perform resource allocation in parallel manner. The TIT* tree is adopted to reduce the cost of the schedulability tests contained in the algorithm. Because TIT* tree is discussed in Chapter 2, the description of it will not be repeated in this chapter. Before discussing the algorithm in detail, we first describe two-level scheduling framework.

4.5.1 Two-level Scheduling Framework

Under two-level scheduling framework, a distributed system contains a coordinator (or global manager) and some subordinates (or local managers). Although subordinates may apply some node-specific policies to local resource management, the global manager coordinates their actions and performs resource management from a system point of view.

Figure 4.3 Two-level scheduling framework.
Whenever a group of tasks arrive, the coordinator and subordinates work together to perform resource allocation for them. To be more specific, the coordinator will pick the scheduling elements one by one, dispatch them to subordinates to perform schedulability tests for computation and communication, collect and analyze the results obtained from subordinates, optimize resource allocation, and distribute elements to appropriate nodes and channels. Accordingly, subordinates will perform schedulability tests for computation and communication in parallel, return test results to the coordinator, and accommodate specified scheduling elements.

This two-level scheduling framework provides an effective approach for reducing the complexity of distributed real-time scheduling system. Resource scheduling under this framework is decomposed into subprocesses, which are distributed to and processed in parallel by subordinates. Hence, the complexity of resource scheduling is reduced through parallelism.

The roles of coordinator and subordinate are dynamically reconfigurable. For example, to avoid single point failure of the coordinator and improve the fault-tolerance of the two-level scheduling framework, every node is capable of working as coordinator when necessary; in case of the failure of current coordinator, an active node is selected as the new coordinator. Current coordinator and a subordinate may also switch roles when necessary.

4.5.2 The Algorithm

Before proceeding to the details of the algorithm, we assume that every subordinate maintains a task TIT* tree containing all unfinished tasks on it, and for every
communication channel connecting it and another subordinate, it also maintains a message TIT* tree containing all unfinished messages on that channel.

In Figure 4.4, the whole algorithm consists of two parts, i.e., GM and LM. GM resides on the coordinator, and LM resides on every subordinate. In Figure 4.4 (a), the coordinator processes scheduling elements according to their utility, and always picks a ready element (all of its predecessors have been processed) currently having the largest utility. By this way, the coordinator attempts to maximize system-wide utility. Once a suitable element, say $e_{xy}$, is identified, the coordinator dispatches it to subordinates to perform schedulability tests for the computation and communication of it. In Figure 4.4 (b), if the currently processed element has no predecessor, every subordinate needs to perform tests for both computation and communication; otherwise, one subordinate needs to perform the test for computation and the other subordinates only need to perform the test for communication because elements are processed according to their precedence relationships. For example, once the node and channel for an element $e_{xy}$ are determined, the node for another element $e_{yz}$ is accordingly predetermined. Thus, a test for the computation of $e_{yz}$ on other nodes is unnecessary. In Figure 4.4 (a) and Figure 4.4 (c), once a suitable node and a suitable channel are determined, the computation and communication of $e_{xy}$ are dispatched to them, and the corresponding TIT* trees are updated.

In Figure 4.4 (a), ChooseElement() is used to find a ready scheduling element currently having the largest utility. DispatchForTest ($e_{xy}$) is used to dispatch $e_{xy}$ to subordinates to perform schedulability tests. CollectFeedback () is used to collect test results from subordinates. AnalyzeOptimize() is used to analyze the results and optimize
resource allocation from a system's point of view. DDA technique is integrated into this part. If the completion times of computation and communication are bounded within the valid region (the shaded region, Figure 4.2) defined by the corresponding utility function, and the joining of current element has no adverse influence on other elements processed so far, the scheme with the maximal adjustment slot (see Section 4.4) is chosen; otherwise, the utility of $e_{xy}$ is set to zero, and it is put aside until the second time it is selected. After a global analysis, the coordinator will decide which node and which channel the computation and communication of $e_{xy}$ should be dispatched to. 

$DispatchForExecution()$ is used to send out the final decision.

In Figure 4.4 (b), the normal case (i.e., the element under test has predecessor(s)), if this subordinate is specified for performing the test for computation, it first invokes $ScheduabilityTestComp()$, and then invokes $ReplyFeedbackComp()$ to send out the test result to coordinator and other nodes; otherwise, it first calls $CollectFeedbackComp()$ to obtain the test result of computation, and then invokes $ScheduabilityTestComm()$ to perform the test for communication (on the channel connecting this node and the node specified for performing the test for computation). $ReplyFeedbackComm()$ is used to send the test result to coordinator. The test result simply contains the information about whether the computation/communication is schedulable on that node/channel, what is the response time and whether other tasks/messages are adversely influenced or not, and other information. In the special case (i.e., the element under test has no predecessor), every subordinate needs to perform the test for computation and the test for communication on every channel connecting this node and another node. $ReplyFeedback()$ is then invoked to send out the test result.
Figure 4.4 Interplay-aware distributed resource scheduling algorithm.
In Figure 4.4 (c), the normal case, if this subordinate is specified for accommodating the computation of $e_{xy}$, it invokes `InsertTask()` and `InsertComm()` to add the computation and communication of $e_{xy}$ to its task and message TIT* trees; otherwise, if this subordinate is specified for updating its message TIT* tree, it invokes `InsertComm()` to add the communication of $e_{xy}$ to its message TIT* tree. In the special case, similar actions are taken by specified subordinates.

![Simplified message sequence chart for the normal case.](image)

**Figure 4.5** Simplified message sequence chart for the normal case.

To help understand the algorithm, a simplified Message Sequence Chart (MSC) (Figure 4.5) is used to demonstrate the interactions among managers. Figure 4.5 is for the normal case. The MSC for the special case is similar to Figure 4.5 except that it contains phases 1, 2, and 4, and in phase 2 every subordinate sends a control message containing the test results of computation and communication to the coordinator.
4.5.2 Complexity Analysis

Because IDRSA is an online algorithm, it needs to consider the newly arrived DAG as well as those DAGs that have already been processed by IDRSA but have not finished (some tasks/messages of these DAGs have not completed, and they are still in the system). Suppose that $DAG_t$, which contains $N$ tasks and $E$ edges, arrives at the system at time $t$, and currently the maximum number of tasks on a node is bounded by $N_t$ and the maximum number of messages on a channel is bounded by $N_e$, the cost of processing $DAG_t$ by the algorithm and its subroutines is computed in Table 4.1.

Subroutine ChooseElement() is used to find the scheduling element currently having the largest utility. This can be done in $O(\log E)$ time because the number of currently ready elements is at most $E$.

Subroutines DispatchForTest(), DispatchForExecution(), ReplyFeedbackComp(), ReplyFeedbackComm(), and ReplyFeedback() are used to deliver control information. Hence the cost of each of them is in $O(Cc)$ time.

Subroutine SchedulabilityTestComp() is used to perform the schedulability test for computation. Its cost is in $O(\log^2(N_t+N))$ because there are at most $(N_t+N)$ tasks on a task TIT* tree, and the test will take $O(\log^2(N_t+N))$ time.

Subroutine SchedulabilityTestComm() is used to perform the schedulability test for communication. Its cost is in $O(\log^2(N_e+E))$ because there are at most $(N_e+E)$ messages on a message TIT* tree, and the test will take $O(\log^2(N_e+E))$ time.

Subroutine CollectFeedbackComp() is used to collect the test result of computation. Its cost depends on how fast the specified subordinate can finish the test.
Because the cost of \textit{SchedulabilityTestComp()} is in $O(\log^2(N_t+N))$, the cost of \textit{CollectFeedbackComp()} is in $O(\log^2(N_t+N)+Cc)$.

\textbf{Table 4.1} Complexity Analysis

<table>
<thead>
<tr>
<th>Subroutine/Algorithm</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{DispatchForTest()}</td>
<td>$O(Cc)$</td>
</tr>
<tr>
<td>\textit{DispatchForExecution()}</td>
<td>$O(\log E)$</td>
</tr>
<tr>
<td>\textit{ReplyFeedbackComp()}</td>
<td></td>
</tr>
<tr>
<td>\textit{ReplyFeedbackComm()}</td>
<td></td>
</tr>
<tr>
<td>\textit{ReplyFeedback()}</td>
<td></td>
</tr>
<tr>
<td>\textit{ChooseElement()}</td>
<td>$O(\log E)$</td>
</tr>
<tr>
<td>\textit{CollectFeedback()}</td>
<td>$O(\log^2(N_t+N)+m\times\log^2(N_e+E))$</td>
</tr>
<tr>
<td>\textit{AnalyzeOptimize()}</td>
<td>$O(\log m)$</td>
</tr>
<tr>
<td>\textit{SchedulabilityTestComp()}</td>
<td>$O(\log^2(N_t+N))$</td>
</tr>
<tr>
<td>\textit{SchedulabilityTestComm()}</td>
<td>$O(\log^2(N_e+E))$</td>
</tr>
<tr>
<td>\textit{CollectFeedbackComp()}</td>
<td>$O(\log^2(N_t+N))$</td>
</tr>
<tr>
<td>\textit{InsertTask()}</td>
<td>$O(\log(N_t+N))$</td>
</tr>
<tr>
<td>\textit{InsertComm()}</td>
<td>$O(\log(N_e+E))$</td>
</tr>
<tr>
<td>\textit{Algorithm IDRSA}</td>
<td>$O(E(\log^2(N_t+N)+m\times\log^2(N_e+E)))$</td>
</tr>
</tbody>
</table>

Subroutine \textit{CollectFeedback()} is used to collect test results. Its cost depends on how fast LMs can finish tests. Because the costs of \textit{SchedulabilityTestComp()}, \textit{SchedulabilityTestComm()}, \textit{CollectFeedbackComp()}, \textit{ReplyFeedbackComp()}, \textit{ReplyFeedbackComm()}, and \textit{ReplyFeedback()} are in $O(\log^2(N_t+N))$, $O(\log^2(N_e+E))$, $O(\log^2(N_t+N)+Cc)$, $O(Cc)$, $O(Cc)$, and $O(Cc)$, respectively, the cost of \textit{CollectFeedback()} is in $O(\log^2(N_t+N)+m\times\log^2(N_e+E))$. (Note that in the special case, every subordinate needs to perform the test for communication on $(m-2)$ channels connecting it to the other $(m-2)$ nodes.)

It is easy to see that the costs of \textit{InsertTask()} and \textit{InsertComm()} are in $O(\log (N_t+N))$ and $O(\log (N_e+E))$, respectively.
Subroutine `AnalyzeOptimize()` is used to find the best scheme among all available schemes. Its cost is in $O(\log m)$ because there are at most $(m-1)$ schemes received by the coordinator. (Note that in the special case, although every node needs to perform the test for communication on $(m-2)$ channels, it only needs to choose and send out the best result.)

The complexity of IDRSA is in $O(E(\log^2(N_t+N) + m \times \log^2(N_e+E)))$ because the cost of `CollectFeedback()` is in $O(\log^2(N_t+N) + m \times \log^2(N_e+E))$, which dominates the cost of IDRSA, and the Repeat-Until loop in GM will be executed at most $2E$ times.

### 4.6 Simulation Analysis

The simulations are designed to test how well IDRSA performs in the presence of overload of computation, overload of communication, or both, and tight interplay between computation and communication. Accordingly, the simulations are performed along five dimensions, i.e., (load of) computation, (load of) communication or data volume, (channel) speed, number of processors, and interplay factor. To evaluate the performance of IDRSA, another scheduling algorithm called `DASA_variant` (discussed in Chapter 3) is also included in these simulations. `DASA_variant` is a variant of DASA [15]. DASA is constructed under UAM and has been widely applied to the resource scheduling in distributed real-time systems [9, 10, 11, 12, 13, 14]. Like DASA, `DASA_variant` is constructed based on UAM. `DASA_variant` seeks to maximize system-wide utility by greedily picking and allocating resources for the task set currently having the highest collective utility density (defined as the ratio of the total utility of the tasks in the task set to the total processing time of them); this procedure repeats until all tasks are processed.
When performing resource scheduling, *DASA_var*iant processes computation and communication separately. For computation, *DASA_var*iant tries to meet its timing constraint for achieving utility, and for communication, *DASA_var*iant tries to meet its deadline. It is worthy of mention that the complexity of *DASA_var*iant is much higher than that of *IDRSA*.

### 4.6.1 Simulation Settings

The simulations are classified into two groups. One group consists of 100 tasks. The task graph is taken from the Standard Task Graph (STG) lib of [63], and it is generated by *samepred* [63] with random seed 6 according to the method described in [64]. The other group consists of 88 tasks. The corresponding task graph is also taken from the STG lib of [63] and it is built from a *robot control* application. Each group contains a series of simulations along the dimensions mentioned before.

To facilitate the performance analysis of the two algorithms in the presence of the interplay between computation and communication, the interplay factor $\alpha$ ($1 \leq \alpha < \infty$) is introduced; $\alpha$ is used to denote how tightly computation and communication are constrained together, the larger the $\alpha$, the tighter the constraint on the combined completion times of computation and communication. As shown in Table 4.2, Table 4.3, and Table 4.4, for those simulations along *computation, data volume, speed*, and *number of processors*, $\alpha$ is set to $5/4$ if condition (II) (see below) is satisfied; otherwise, $\alpha$ is set to a value such that condition (II) is satisfiable. This actually sets a loose constraint on the combined completion times of computation and communication. Thus the interplay between computation and communication will play a very limited role in these simulations. By contrast, for those simulations along *interplay factor*, much tighter
constraints are set by condition (I2) (see below). This implies that the interplay will play an important role in these simulations.

\[(\text{Max } \{\text{COMP}_{ij}, \text{COMM}_{ij}\} + \Delta_i) \leq (1/\alpha) \times (\text{COMP}_{ij} + \text{COMM}_{ij}) \leq 1 \times (\text{COMP}_{ij} + \text{COMM}_{ij}) (11)\]

(where $\Delta_i$ is an adjustment factor)

\[\text{comp}_{ij} \leq (1/\alpha) \times (\text{COMP}_{ij} + \text{COMM}_{ij}) \leq 1 \times (\text{COMP}_{ij} + \text{COMM}_{ij}) (12)\]

Table 4.2 Simulation Settings

<table>
<thead>
<tr>
<th>Group-1</th>
<th>Group-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of tasks: 100</td>
<td>Number of tasks: 88</td>
</tr>
<tr>
<td>Task graph: samepred</td>
<td>Task graph: robot control</td>
</tr>
<tr>
<td>Channel speed: 1.0;</td>
<td></td>
</tr>
<tr>
<td>$\text{COMM}<em>{ij} = \text{COMM}</em>{Dij}$; uniformly distributed between [200, 300];</td>
<td></td>
</tr>
<tr>
<td>$\text{COMP}<em>{ij} = \text{COMP}</em>{Dij}$; uniformly distributed between [200, 300];</td>
<td></td>
</tr>
<tr>
<td>$\text{comp}_{ij}$: (1) initially generated uniformly between [1, 100];</td>
<td></td>
</tr>
<tr>
<td>(2) varies from (Initial Value+0) to (Initial Value+100), with step length 10;</td>
<td></td>
</tr>
<tr>
<td>$\text{comm}_{ij}$: (1) initially generated uniformly between [200, 300];</td>
<td></td>
</tr>
<tr>
<td>(2) varies from (Initial Value+0) to (Initial Value+100), with step length 10;</td>
<td></td>
</tr>
<tr>
<td>$U_{ij}$; uniformly distributed between [1, 100];</td>
<td></td>
</tr>
<tr>
<td>Number of processors:</td>
<td></td>
</tr>
<tr>
<td>(1) initially 10;</td>
<td></td>
</tr>
<tr>
<td>(2) varies from 10 to 2;</td>
<td></td>
</tr>
<tr>
<td>Interplay factor:</td>
<td></td>
</tr>
<tr>
<td>(1) $A_{ij} = B_{ij}$</td>
<td></td>
</tr>
<tr>
<td>(2) $(C_{ij}/A_{ij}) = (1/\alpha) \times (\text{COMP}<em>{ij} + \text{COMM}</em>{ij})$</td>
<td></td>
</tr>
<tr>
<td>(3) $\alpha$ is set to 54/ if</td>
<td></td>
</tr>
<tr>
<td>$(\text{Max } {\text{COMP}<em>{ij}, \text{COMM}</em>{ij}} + \Delta_i) \leq 1/\alpha \times (\text{COMP}<em>{ij} + \text{COMM}</em>{ij}) \leq 1 \times (\text{COMP}<em>{ij} + \text{COMM}</em>{ij})$;</td>
<td></td>
</tr>
<tr>
<td>(where $\Delta_i$ is an adjustment factor and $\Delta_i &gt; 0$)</td>
<td></td>
</tr>
<tr>
<td>otherwise, $\alpha$ is set to a value such that the above condition is satisfied;</td>
<td></td>
</tr>
</tbody>
</table>
Because *DASA\_variant* allocates resources based on utility functions defined under UAM, to facilitate comparison and analysis, it is assumed that if a task in a DAG has $k$ outgoing edges, it contains $k$ virtual independent subtasks. The utility defined under

\textbf{Table 4.3 Simulation Settings(2)}

<table>
<thead>
<tr>
<th>Group-1</th>
<th>Group-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of tasks: 100</td>
<td>Number of tasks: 88</td>
</tr>
<tr>
<td>Task graph: samepred</td>
<td>Task graph: robot control</td>
</tr>
<tr>
<td>Channel speed: varies from 1.0, 1.1, 1.2, ..., until 2.0;</td>
<td></td>
</tr>
<tr>
<td>$COMM_{ij}=COMM_D_{ij}$ uniformly distributed between [200, 300];</td>
<td></td>
</tr>
<tr>
<td>$COMP_{ij}=COMP_D_{ij}$ uniformly distributed between [200, 300];</td>
<td></td>
</tr>
<tr>
<td>$comp_{ij}$ uniformly distributed between [1, 100];</td>
<td></td>
</tr>
<tr>
<td>$comm_{ij}(100+v)$, where $v$ is uniformly distributed between [200, 300];</td>
<td></td>
</tr>
<tr>
<td>$U_{ij}$ uniformly distributed between [1, 100];</td>
<td></td>
</tr>
<tr>
<td>Number of processors: 10;</td>
<td></td>
</tr>
<tr>
<td>Interplay factor:</td>
<td></td>
</tr>
<tr>
<td>(1) $A_{ij} = B_{ij}$</td>
<td></td>
</tr>
<tr>
<td>(2) $(C_{ij}/A_{ij}) = (1/\alpha) \times (COMP_{ij} + COMM_{ij})$</td>
<td></td>
</tr>
<tr>
<td>(3) $\alpha$ is set to 5/4 if</td>
<td></td>
</tr>
<tr>
<td>$\left(\text{Max}(COMP_{ij}, COMM_{ij})+\Delta_{ij}\right) \leq (1/\alpha) \times (COMP_{ij} + COMM_{ij}) \leq 1 \times (COMP_{ij} + COMM_{ij})$;</td>
<td></td>
</tr>
<tr>
<td>(where $\Delta_{ij}$ is an adjustment factor and $\Delta_{ij} &gt; 0$)</td>
<td></td>
</tr>
<tr>
<td>Otherwise, $\alpha$ is set to a value such that the above condition is satisfied;</td>
<td></td>
</tr>
</tbody>
</table>

\textbf{Table 4.4 Simulation Settings(3)}

<table>
<thead>
<tr>
<th>Group-1</th>
<th>Group-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of tasks: 100</td>
<td>Number of tasks: 88</td>
</tr>
<tr>
<td>Task graph: samepred</td>
<td>Task graph: robot control</td>
</tr>
<tr>
<td>Channel speed: 1.0;</td>
<td></td>
</tr>
<tr>
<td>$COMM_{ij}=COMM_D_{ij}$ uniformly distributed between [200, 300];</td>
<td></td>
</tr>
<tr>
<td>$COMP_{ij}=COMP_D_{ij}$ uniformly distributed between [200, 300];</td>
<td></td>
</tr>
<tr>
<td>$comp_{ij}$ uniformly distributed between [1, 100];</td>
<td></td>
</tr>
<tr>
<td>$comm_{ij}(100+v)$, where $v$ is uniformly distributed between [200, 300];</td>
<td></td>
</tr>
<tr>
<td>$U_{ij}$ uniformly distributed between [1, 100];</td>
<td></td>
</tr>
<tr>
<td>Number of processors: 10;</td>
<td></td>
</tr>
<tr>
<td>Interplay factor:</td>
<td></td>
</tr>
<tr>
<td>(1) $A_{ij} = B_{ij}$</td>
<td></td>
</tr>
<tr>
<td>(2) $(C_{ij}/A_{ij}) = (1/\alpha) \times (COMP_{ij} + COMM_{ij})$</td>
<td></td>
</tr>
<tr>
<td>(1/\alpha varies from 0.1 to 0.9, with step length 0.1.)</td>
<td></td>
</tr>
<tr>
<td>(3) $0.1 \leq (1/\alpha) \leq 0.9$ and $\text{comp}<em>{ij} \leq (1/\alpha) \times (COMP</em>{ij} + COMM_{ij}) \leq 1 \times (COMP_{ij} + COMM_{ij})$</td>
<td></td>
</tr>
</tbody>
</table>
UAM* along an edge is the utility defined under UAM for the corresponding subtask, and the utility input to DASA_variant is of the same amount as the utility input to IDRSA though they have different meanings. In addition, it is assumed that for a given task \( T_i \), if there is an edge entering it, the corresponding predecessor (of \( T_i \)) will be the predecessor of all its virtual subtasks.

4.6.2 Simulation Results

Figure 4.6 Utility ratios achieved vary with the increase of computation workload.

Figure 4.7 Utility ratios achieved vary with the increase of data volume.
As shown in Figure 4.6, the utility ratios (defined as the utility obtained versus the utility available) obtained by the two algorithms decrease with the increase of computation workload. For \textit{DASA\_variant}, the increasing computation workload makes more and more computations unable to complete within their constraints, and for \textit{IDRSA}, the increasing computation workload makes more and more scheduling elements unable to complete in a timely way.

\textbf{Figure 4.8} Utility ratios achieved vary with the decrease of the number of processors.

\textbf{Figure 4.9} Utility ratios achieved vary with the increase of channel speed.
Figure 4.10 Utility ratios achieved vary with the increase of interplay factor $\alpha$.

Figure 4.7(b) shows that the utility ratio obtained by IDRSA experienced a small decrease with the increase of data volume. This is because the increasing data volume (or load of communication) eventually makes some elements unable to complete in a timely way. Although IDRSA is interplay-aware and able to adjust resource allocation according to the loads of computation and communication, its ability to adjust is not unlimited. For example, to alleviate the problem of the increasing data volume, IDRSA may dispatch a pair of communicating tasks to the same processor, but the power of this approach is limited due to the fact that if too many computations are dispatched to one processor. This will eventually lead to overload of computation on this processor. As a result, some scheduling elements can not finish in time, and the corresponding utility is lost. For DASA_variant, it experienced an even smaller decrease of utility ratio. The reason is that its unawareness of the interplay between computation and communication results in low utility ratio before the increase of data volume, which implies that some utility has already been lost due to its unawareness of the interplay; hence the increasing data volume has very little impact on the utility it obtained. From Figure 4.7 (a), it is easy to
see that DASA\_variant exhibits good performance though it is unaware of the interplay between computation and communication. This is because the DAG of robot control is almost a chain, with very few parallel tasks, and DASA\_variant dispatched almost all computations to one processor. Obviously, in this case the increasing data volume has very little impact on the utility it obtained.

Figure 4.8(b) indicates that the utility ratios obtained by the two algorithms decrease with the decreasing number of processors. For DASA\_variant, fewer processors will make fewer computations complete within their timing constraints, and for IDRSA, fewer processors make fewer scheduling elements processed in a timely way. In Figure 4.8 (a), the utility ratios obtained by the two algorithms almost do not vary with the decreasing number of processors, and even DASA\_variant exhibits good performance. The reason is similar to what is mentioned before, i.e., the DAG of robot control is almost a chain, which implies that one can dispatch almost all computations to one processor (hence some processors are unoccupied). Obviously, removing those unoccupied processors results in no utility loss.

Figure 4.9 shows that the utility ratios achieved by the two algorithms increase with the increasing channel speed. For DASA\_variant, the increasing speed makes more and more communications able to complete before their deadlines, and for IDRSA, the increasing speed makes more and more scheduling elements processed in a timely way.

Figure 4.10 shows that the utility ratio obtained by DASA\_variant drops a lot with the increase of interplay factor, while that by IDRSA maintains at a high level though a small decrease is also seen. The reason is that the interplay between computation and communication becomes tighter and tighter with the increasing interplay factor, and
DASA\textsubscript{variant} loses more and more utility due to its unawareness of the interplay. By contrast, IDRSA fully realizes the interplay, and is able to adjust the allocation of processors and channels according to the interplay and the loads of computations and communications.

In Figure 4.6 and Figure 4.8 (b), compared to its counterpart, IDRSA performs very well when the load of computation is heavy or the processing capacity for computation is low. This is because IDRSA is an interplay-aware algorithm, and is able to adjust the allocation of processors and channels according to the loads of computations and the processing capacity for computations so as to meet the constraints on the combined completion times of computations and communications. These results suggest the excellence of IDRSA in the presence of heavy computation load or low processing capacity for computation.

From Figure 4.6 to Figure 4.10, it is easy to see that IDRSA performs much better than DASA\textsubscript{variant}. The reason is that DASA\textsubscript{variant} is constructed based on traditional UAM, and its unawareness of the interplay between computation and communication in a DRTS leads to the loss of a large amount of utility. IDRSA, however, fully realizes the interplay, and is able to flexibly adjust the allocation of processors and channels according to the interplay, the loads of computations and communications, and the available processing capacity for computations and communications. This indicates that due to the interplay between computation and communication in a DRTS, separately meeting the timing constraints on computation and communication is inadequate for utility accrual from a system's point of view.
Power/energy consumption is a critical issue in the system design of the battery-powered devices such as mobile, portable and embedded devices, as well as the desktop and server systems (because high power consumption produces high heat, which causes high temperature and eventually reduces system performance and reliability).

Over the past few years, the Dynamic Voltage-frequency Scaling (DVS) technique has been applied to many systems to reduce energy consumption by reducing the supply voltage and operating frequency at run time. The DVS technique is based on the fact that the energy dissipated per cycle with CMOS circuitry scales quadratically to the supply voltage \( (E \propto V^2) \), and over the range of allowed voltages the highest frequency at which the processor will run correctly drops approximately proportional to the voltage \( (f \propto V) \). (Hence the energy dissipated per cycle also scales quadratically to the frequency \( (E \propto f^2) \).)

DVS has been proven to be a powerful technique for reducing energy consumption, and thus has been extensively studied not only in general-purpose computing systems \([61, 66, 67, 68, 69]\) (and the references therein) but also in real-time systems, where the DVS technique is extended to reduce energy consumption while meeting timing constraints. In this aspect, extensive work has been done under the periodic task model (where every task is associated with a period and the task is invoked periodically) \([60, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83]\), or the sporadic task model (where every task is associated with a minimum interarrival time and the
interval between two consecutive invocations of a task is at least of that length) [84]. In addition, some work also studied the real-time DVS techniques under a more general task model, where tasks have arbitrary arrival times and arbitrary deadlines [62, 75, 85, 86]. In [62, 86], DVS algorithms are proposed to reduce the energy consumption of a set of tasks with arbitrary arrival times and arbitrary deadlines; but the proposed algorithms are static/offline. In [75], an online DVS algorithm called OLDVS is proposed; but the algorithm is mainly based on Worst Case Execution Time (WCET) analysis, and the basic idea behind the algorithm is to exploit the unfilled WCET slices. Energy saving with this approach is limited. Consider a simple scenario where there is only one task $T=(r, e, d)$ (where $r$, $e$ and $d$ are the release time, the WCET and the relative deadline of task $T$, respectively), and $d > e$. According to [75], $T$ will be executed under full speed even if a slower speed is fast enough for $T$ to complete no later than its deadline. In [85], an online algorithm called AVR is proposed; nevertheless, this algorithm relies on an assumption, i.e., the computed speed is always available, no matter how high it is. This assumption is impractical in real world because the highest speed of a specific processor is limited. Besides, the approaches proposed in [75] and [85] can only be applied to those real-time systems where tasks/jobs are scheduled according to Earliest Deadline First (EDF) rule [2].

In this dissertation, the author advances the research on online real-time DVS by applying new method and theory, i.e., Network Calculus curve [87], to hard real-time systems under a general task model, where events/tasks may arrive randomly, and no assumptions are made about their periodicity, minimum inter-arrival time and so on. This work is motivated by the following observations.
(1) Network calculus curves will make it possible to establish a more general task model, which can capture the characteristics of a wide spectrum of tasks, including burst arriving tasks and the traditional periodic and sporadic tasks, and characterize them in a general way. This will make it possible to study real-time DVS under a more general task model and investigate some general real-time DVS techniques and algorithms.

(2) While static/offline real-time DVS can be performed with all necessary information in hand, dynamic/online real-time DVS has to be conducted with very limited information, which makes online real-time DVS hard to tackle. The network calculus curves inherently have the ability to accommodate random/dynamic system features. This will make it possible to investigate online real-time DVS through an effective and rigorous approach.

(3) Real-time systems with random/dynamic characteristics are tough to design and verify. Network calculus curves help alleviate this difficulty, and they provide an effective and powerful approach for system design, validation, and verification, which is hard but of critical importance for real-time systems. The calculus curve approach will make it possible to formally analyze and verify the schedulability/feasibility of a random/dynamic real-time system. This will also make it possible to analyze and verify the feasibility of applying the new online real-time DVS technique and algorithms to those random/dynamic real-time systems.

To capture the characteristics of those events/tasks arriving randomly, the concept of *calculus curve* from network calculus domain is adapted, and *arrival calculus curves* are used to characterize the random arrivals of events/tasks. The arrival calculus curve makes it possible to establish a more general task model, where no periodicity and
minimum inter-arrival time are assumed. More importantly, this task model is able to accommodate burst arrivals of events/tasks. Similarly, service calculus curves are also used to characterize the random and dynamic processing capacity dedicated to events/tasks. Based on calculus curves, the author first proposes a history window based prediction technique, which is used to predict future computational requirement according to calculus curves and history records. The author then develops energy-efficient online real-time DVS algorithms, which incorporate the history window based prediction technique, and are capable of dynamically adjusting system operating voltage-frequency according to the predicted computational requirement. The author validates and verifies the feasibility and correctness of the new technique and algorithms in a formal way.

The new algorithms are constructed on EDF and fixed priority policies, and have the capability to predict the computational requirement due to the random arrivals of future events/tasks. This implies that the real-time DVS proposed in this dissertation is based not only on the existing computational requirement but also on that which may be requested in the future. This feature distinguishes the new algorithms from existing online real-time DVS algorithms. Predicting the future computational requirement is critically important in a dynamic random hard real-time environment. In such an environment, conducting DVS without predicting future computational requirement may lead to system failure even if feasibility analysis is well conducted at the system design stage.

The new algorithms are also able to accommodate and respond to not only the variation between the predicted and the actual event/task arrivals but also the variation
between the predicted and the actual execution times of tasks. This feature distinguishes the new algorithms from those static/offline real-time DVS algorithms that are based on static information.

5.1 Calculus Curves

In this section, the concept of calculus curves from network calculus domain is adapted to characterize the arrivals of events/tasks and the system processing capacity for events/tasks. Network calculus is a mathematical approach originally intended to model, analyze, and design networks. The foundation of network calculus is the mathematical Min-Plus and Max-Plus algebras, which are useful for constructing mathematical models of discrete event systems. In recent years, network calculus has been intensively studied for flow processing in a variety of areas such as network, multimedia, embedded systems and so on. One of the important features of the network calculus approach is that it facilitates system design, validation, and verification, and enables system design to be formally proved and verified. The network calculus approach has led to many important research outcomes that provide deep insights into communication networks, multimedia systems, and embedded systems.

In the network calculus domain, network calculus curves are used to characterize flows and the processing capacity of network nodes. In this dissertation, they are adapted to characterize random arriving events/tasks and the processing capacity of real-time systems.

5.1.1 Arrival Curve
The arrival curve is used to characterize the random arrivals of events/tasks in a hard real-time system. If a function \( R(t) \) \((t \geq 0)\) is used to denote the number of a class of events that may arrive at the system within \([0, t]\), the arrival curve for this class of events is defined as follows.

**Definition 5.1** Arrival Curve \( \alpha(t) \) is a wide-sense increasing function (i.e., \( \alpha(t) \leq \alpha(s) \) for all \( t \leq s \)). For \( \forall s \geq 0, t \geq 0 \) it satisfies: \( R(s+t) - R(s) \leq \alpha(t) \), and \( \alpha(t) = 0 \) for \( \forall t < 0 \).

According to this definition, \( \alpha(t) \) \((t \geq 0)\) is the upper bound of the number of a class of events that may arrive in any time interval of length \( t \), although the arrivals of events may be random (including burst arrivals).

### 5.1.2 Service Curve

The service curve is used to characterize the processing capacity (in terms of processor cycles) of a hard real-time system. If a function \( C(t) \) \((t \geq 0)\) is used to denote the number of cycles that a system can offer to the process of a class of events within \([0, t]\), the service curve for this class of events is defined as follows.

**Definition 5.2** Service Curve \( \beta(t) \) is a wide-sense increasing function (i.e., \( \beta(t) \leq \beta(s) \) for all \( t \leq s \)). For \( \forall s \geq 0, t \geq 0 \) it satisfies: \( C(s+t) - C(s) \geq \beta(t) \), and \( \beta(t) = 0 \) for \( \forall t < 0 \).

According to this definition, \( \beta(t) \) \((t \geq 0)\) is the lower bound of the number of cycles that the system can offer to the process of a class of events in any time interval of length \( t \).
5.2 System and Task Model

Consider a hard real-time system that is designed to process \( m \) classes of events. The total capacity for processing these events is characterized by a service curve \( \beta(t) \) (which is the minimum service curve that makes class 1 to class \( m \) schedulable), and the corresponding frequency is \( f_{max} \). Given an \( i \) (\( 1 \leq i \leq m \)), the arrivals of the events of class \( i \) are characterized by an arrival curve \( \alpha_i(t) \) (\( t \geq 0 \)). To facilitate later analysis, an \( \alpha_0(t) \) is defined as \( \alpha_0(t) = 0 \) for \( t \geq 0 \). Events of every class arrive at the system randomly. For every event of a class \( i \), a task \( T_i \) will be invoked and executed once. For every task \( T_i \), it is characterized by a triple \( (r_i, e_i, d_i) \), where \( r_i \) is the release time (this is set when an event arrives), \( e_i \) is the predicted WCET according to a benchmark processor with operating frequency \( f_s \), and \( d_i \) is the relative deadline. Multiple instances of a task may exist in the system concurrently.

5.3 Schedulability/Feasibility Analysis

In a hard real-time environment, all tasks must be finished no later than their deadlines; DVS in such environment must take the timing constraints into account, and guarantee that all deadlines are met. The schedulability analyses conducted in this section are used to find the minimum necessary voltage-frequency level for processing events, and are the foundations for the new online real-time DVS algorithms.

The schedulability analysis according to EDF policy is conducted in Section 5.3.1 and that according to fixed priority policy is conducted in Section 5.3.2
5.3.1 Schedulability/Feasibility Analysis According to Preemptive Earliest Deadline First Policy

With this policy, events from all classes are processed according to their deadlines (i.e., earliest deadline first). Tasks with earlier deadlines can preempt the executions of those with later deadlines. Tasks with identical deadlines will be processed in First Come First Serve (FCFS) fashion. Given $m$ classes of events with arrival curves $\alpha_i(t)$ to $\alpha_m(t)$ and the total processing capacity that is characterized by $\beta(t)$, the following Theorem 5.1 gives a necessary and sufficient condition for the schedulability test according to preemptive EDF policy.

**Theorem 5.1** \[\beta(t) \geq \sum_{j=1}^{m} (\alpha_j(t-d_j) \times e_j \times f_s) \quad (\forall t \geq 0) \quad \Leftrightarrow \quad \text{class 1 to class m are schedulable with EDF policy.}\]

**Proof:** $\Leftarrow$ By contradiction.

Suppose $\exists t' \quad \beta(t') < \sum_{j=1}^{m} (\alpha_j(t' \times d_j) \times e_j \times f_s) . \quad$ 

$\Rightarrow \exists t_1, \exists t_2 \quad ((t_2 - t_1) = t') \quad$, and $t_1$ is the start point of a busy period (no system idle during this period). Suppose class 1, \ldots, class $m$ are in increasing order of relative deadline, and their relative deadlines are $d_1, \ldots, d_m$.

**Case 1 (special case):** $\exists k \leq m, (t_1 + d_k) > t_2$. In this case, the following holds: 

$\alpha_j(t_2 - d_j - t_1) = 0 \quad$ for $(k \leq j \leq m)$.

Class 1 to class $(k - 1)$ are schedulable within $[t_1, t_2]$

$\Rightarrow \beta(t_2 - t_1) \geq \sum_{j=1}^{k-1} (\alpha_j(t_2 - d_j - t_1) \times e_j \times f_s) . \quad$

$\Rightarrow \beta(t_2 - t_1) \geq \sum_{j=1}^{k-1} (\alpha_j(t_2 - d_j - t_1) \times e_j \times f_s) + \sum_{j=k+1}^{m} 0 . \quad$

i.e., $\beta(t') \geq \sum_{j=1}^{m} (\alpha_j(t' \times d_j) \times e_j \times f_s) . \quad$

This contradicts the assumption.

**Case 2:** $(t_1 + d_m) < t_2.$
Class 1 to class $m$ are schedulable within $[t_1, t_2]$

\[ \Rightarrow \beta(t_2 - t_1) \geq \sum_{j=1}^{m} \left( \alpha_j \left( t_2 - d_j - t_i \right) \times e_j \times f_j \right) \]

i.e., \[ \beta(t') \geq \sum_{j=1}^{m} \left( \alpha_j \left( t' - d_j \right) \times e_j \times f_j \right) . \]

This contradicts the assumption.

\[ \Rightarrow \]

By contradiction.
Suppose that class 1 to class $m$ are unschedulable, $t'$ is the arrival time of the first task (say $T_i$) that misses its deadline, and it belongs to class $i$.

**Case 1:** $t'$ is the start point of a busy period (Figure 5.1). (Suppose that class 1, class 2, ..., class $m$ are in increasing order of relative deadline.) Because the events whose deadlines are later than that of $T_i$ will have no influence on $T_i$, the following holds:

\[ \sum_{j=1}^{i-1} \left( \alpha_j \left( t'+d_i - t'-d_j \right) \times e_j \times f_j \right) + \left( \alpha_i \left( t'+d_i - t'-d_i \right) \times e_i \times f_i \right) > \beta(t'+d_i - t') . \]

Let $t = d_i$, then

\[ \sum_{j=1}^{i-1} \left( \alpha_j \left( t - d_j \right) \times e_j \times f_j \right) + \left( \alpha_i \left( t - d_i \right) \times e_i \times f_i \right) > \beta(t) \]

This contradicts the given condition.

**Case 2:** $t'$ is not the start point of a busy period, but $t''$ is the nearest (from the left side of $t'$) start pint of a busy period (Figure 5.1). (Suppose that class 1, class 2, ..., class $m$ are in increasing order of relative deadline.) Because the events whose deadlines are later than that of $T_i$ will have no influence on $T_i$, the following holds:

\[ \sum_{j=1}^{i} \left( \alpha_j \left( t'+d_i - t''-d_j \right) \times e_j \times f_j \right) + \left( \alpha_i \left( t'+d_i - t''-d_i \right) \times e_i \times f_i \right) > \beta(t'+d_i - t'') \]

\[ (1 \leq k \leq m, \text{ and } k \neq i) . \]

Let $t = \left( t' + d_i - t'' \right)$, then the following holds:

\[ \sum_{j=1}^{i} \left( \alpha_j \left( t - d_j \right) \times e_j \times f_j \right) + \left( \alpha_i \left( t - d_i \right) \times e_i \times f_i \right) > \beta(t) . \]

This contradicts the given condition. 

\[ \square \]
Theorem 5.1 states that for all the events of class 1 to class \( m \) to be feasibly processed in time according to EDF policy, the processing capacity \( \beta(t) \) must satisfy the above condition, and that the processing capacity \( \beta(t) \) that satisfies the above condition is high enough for processing the events of class 1 to class \( m \).

5.3.2 Schedulability/Feasibility Analysis According to Preemptive Fixed Priority Policy

With this policy, a priority \( P_i \) (1\( \leq \)i\( \leq \)m) is assigned to every event of class \( i \). Throughout this chapter, it is assumed that for any \( P_i \), 1\( \leq \)P_i\( \leq \)m. For any pair of priorities \( (P_i, P_j) \), class \( i \) has higher priority than class \( j \) if \( (P_i < P_j) \). Events from all classes are processed according to their corresponding priorities (i.e., highest priority first), and tasks with higher priorities can preempt the executions of those tasks with lower priorities. Tasks with identical priorities will be executed in FCFS fashion. Given \( m \) classes of events with arrival curves \( \alpha_i(t) \) to \( \alpha_m(t) \) and the total processing capacity that is characterized by \( \beta(t) \), the following Theorem 5.2 gives a necessary and sufficient condition for the schedulability test according to preemptive fixed priority policy.

**Theorem 5.2**  
Let \( \beta'_i(t) = \max_{0 \leq u \leq t} \{ \beta(u) - \sum_{j=1}^{i-1} (\alpha_j(u) \times e_j \times f_s) \} \quad (1 \leq i \leq m); \)
\[ \beta''_j(t) \text{ satisfies: } \max_{u \geq 0} \{ \min_{u : \beta''_j(t+u) \geq \alpha_j(t) \times e_j \times f_s} \} \leq d_j \]

\[ \Rightarrow \text{class 1 to class j are schedulable with fixed priority policy}(1 \leq j \leq m). \]

**Proof:** \( \Leftarrow \) By contradiction.

Suppose \( \exists i, \exists t' \quad \beta'_i(t'+u) \geq (\alpha_i(t') \times e_i \times f_s) \Rightarrow u > d_i \), i.e.,
\[ \beta'_i(t'+d_i) < (\alpha_i(t') \times e_i \times f_s). \]

\[ \Rightarrow \exists t_1, \exists t_2 \quad ((t_2 - t_1) = t'), \text{ and } t_1 \text{ is the start point of a busy period (i.e., system is never idle during this period of time). Now, suppose that} \]
\[ ((\alpha_i(t_2 - t_1) - k - 1) \times e_i \times f_s) \leq \beta'_i(t_2 - t_1 + d_i) \leq ((\alpha_i(t_2 - t_1) - k) \times e_i \times f_s) \]
This implies that \( k \) tasks from class \( i \) will miss their deadlines in interval \([t_1, t_2]\). But this contradicts the given condition.

\( \Rightarrow \) By induction.
Step 1: Base case, \( k = 1 \). It can be proved by contradiction. Suppose that class 1 is unschedulable. \( \Rightarrow \exists t' (\beta_1'(t'+d_i) < \alpha_i(t') \times e_i \times f_s) \). But this contradicts with the given condition. Thus Theorem 5.2 holds for \( k = 1 \).
Step 2: Suppose that Theorem 5.2 holds for \( k = 1 \) to \( i \).
Step 3: Prove Theorem 5.2 holds for \( k = (i+1) \) by contradiction. Suppose that class 1 to class \( (i+1) \) are unschedulable. To be more specific, class \( (i+1) \) is unschedulable (because class 1 to class \( i \) are schedulable according to assumption and have higher priorities than class \( (i+1) \), they will not be influenced by class \( (i+1) \)).
\( \Rightarrow \exists t' (\beta_{i+1}'(t'+d_{i+1}) < \alpha_{i+1}(t') \times e_{i+1} \times f_s) \). But this contradicts the given condition. Thus Theorem 5.2 holds for \( k = (i+1) \).

Theorem 5.2 states that for all the events of class 1 to class \( m \) to be feasibly processed in time according to fixed priority policy, the processing capacity \( \beta(t) \) must satisfy the above condition, and that the processing capacity \( \beta(t) \) that satisfies the above condition is high enough for processing the events of class 1 to class \( m \).

### 5.4 Online Real-Time DVS Algorithms

#### 5.4.1 History Window Based Prediction

The history window based prediction technique is employed to predict the requirement of computation within a future time interval (i.e., the prediction interval, see Figure 5.2 and Figure 5.3) based on history records.

Let \( t_c \) be the current time. Suppose there are \( n \) existing tasks \( T'_1, T'_2, \ldots, T'_n \) in that order in the task queue. The absolute deadlines of \( T'_1, \ldots, T'_n \) are \( D'_1, \ldots, D'_n \), respectively. To predict the computational requirement, both the EDF based algorithm and the fixed priority based algorithm use a history window, which contains \( w \) recording
points \((t_1, \ldots, t_w)\) (see Figure 5.2 and Figure 5.3). The history window will slide with the advance of time, and the recording points will be updated accordingly. At every recording point \(t_q\) \((1 < q \leq w)\), the number of cycles that have been offered to every class is recorded. These records can be easily constructed if a counter is used for every class. The counter is used to record the number of cycles devoted to the process of the events of that class. Suppose that \(B^q_j\) is used to denote the number of cycles that were offered to class \(j\) within interval \([t_q, t_w)\). \(B^q_j\) can be easily obtained according to the information recorded at the recording points. To facilitate later analysis, a remaining computational requirement function \(\delta(t - t_c)\) is defined as follows.

**Definition 5.3** (remaining computational requirement function \(\delta(t - t_c)\)) Given a time point \(t_c\), the remaining computational requirement (due to a set of specified ready tasks) function at \(t_c\) is defined as \(\delta(t - t_c)\), which is the minimum computational requirement that the system should satisfy at time \(t\) \((t \geq t_c)\), so as to make every task (in the task set) finish no later than its deadline.

In addition, to facilitate later description, \(S(f, t)\) is used to denote the number of cycles a system can offer within an interval of length \(t\) under operating frequency \(f\).

The objective of the prediction is to find an upper bound as low as possible for the computational requirement that may be requested in a specified future time interval. The rationale behind this idea is that a lower upper bound implies less computational requirement, and hence a lower frequency is sufficient enough for completing the computations in a timely way.

**5.4.1.1 History Window Based Prediction for EDF Policy**
In Figure 5.2, suppose that tasks $T'_1, \ldots, T'_n$ are in non-decreasing order of deadline, and their deadlines are $D'_1, \ldots, D'_n$.

![Figure 5.2 History window based prediction for EDF policy.](image)

Consider a prediction interval $[t_c, D'_i]$ ($1 \leq i \leq n$). Under EDF policy, the computational requirement due to future arrivals of events from class $j$ ($1 \leq j \leq m$) that should be satisfied by a time point $t$ ($t_i \leq t \leq D'_i$) can be computed as

$$(\alpha_j (t - t_c - d_j) \times e_j \times f_s).$$

Given a recording point $t_k$ in the history window, it can also be computed as

$$(\alpha_j (t - t_k - d_j) \times e_j \times f_s - B_j^k).$$

The smallest one computed according to the history window can be obtained as

$$\min_{1 \leq k \leq (w-1)} (\alpha_j (t - t_k - d_j) \times e_j \times f_s - B_j^k).$$

Thus, under EDF policy, the total computational requirement that should be satisfied by time $t$ can be computed as

$$\left(\sum_{j=1}^{m} \min_{1 \leq k \leq (w-1)} (\alpha_j (t - t_k - d_j) \times e_j \times f_s), \min_{1 \leq k \leq (w-1)} (\alpha_j (t - t_k - d_j) \times e_j \times f_s - B_j^k) \right) + \delta_{T'_i}(t - t_i).$$
where $\delta_{T'}(t-t_c)$ is the remaining computational requirement function (at $t_c$) due to tasks $T'_1$, ..., $T'_i$. In particular, $\delta_{T'}(t-t_c) = \delta(t-t_c)$, which is the remaining computational requirement due to tasks $T'_1$, ..., $T'_n$.

Suppose the system is operating under frequency $f_i$. The following condition should be satisfied so as to meet the computational requirement:

$$S(f_i, (t-t_c)) \geq \left( \sum_{j=1}^{m} \min\{\alpha_j(t-t_c-d_j) \times e_j \times f_i\} \right) \min\{\alpha_j(t-t_c-d_j) \times e_j \times f_i - B_j^i\} + \delta_{T'}(t-t_c) \quad (t_c \leq t \leq D'_i).$$

### 5.4.1.2 History Window Based Prediction for Fixed Priority Policy

![Figure 5.3](image)

**Figure 5.3** History window based prediction for Fixed Priority policy.

In Figure 5.3, suppose that tasks $T'_1$, ..., $T'_n$ are in non-increasing order of priority, and their priorities are $P'_1 < ... < P'_n$, i.e., $T'_1$ has the highest priority and $T'_n$ has the lowest priority.

Consider a task $T'_i$ ($1 \leq i \leq n$). To make $T'_1$ to $T'_i$ finish no later than their deadlines under fixed priority policy, the computational requirement due to the future arrivals of
events from class \( j \) \((1 \leq j \leq (P'_i - 1))\) that should be satisfied by a time point \( t \) \((t_c \leq t \leq D'_i)\) can be computed as

\[
(\alpha_j (t - t_c) \times e_j \times f_j).
\]

Given a recording point \( t_k \) in the history window, it can also be computed as

\[
(\alpha_j (t - t_k) \times e_j \times f_j - B^k_j).
\]

The smallest one computed according to the history window can be obtained as

\[
\min_{1 \leq k \leq (w-1)} (\alpha_j (t - t_k) \times e_j \times f_j - B^k_j).
\]

The computational requirement due to the future arrivals of those tasks that have priorities not higher than \( T'_i \) that should be satisfied at a time point \( t \) \((t_c \leq t \leq D'_i)\) can be computed as

\[
\beta_{p'_i} (t - t_c),
\]

where \( \beta_{p'_i} (t - t_c) \) is the minimum service curve for class \( P'_i \) to class \( m \) to be schedulable.

Given a recording point \( t_k \), it can also be computed as

\[
(\beta_{p'_i} (t - t_k) - \sum_{j \in P'_i} B^k_j).
\]

The smallest one computed according to the history window can be obtained as

\[
\min_{1 \leq k \leq (w-1)} (\beta_{p'_i} (t - t_k) - \sum_{j \in P'_i} B^k_j).
\]

Thus, to make tasks \( T'_1 \) to \( T'_i \) finish no later than their deadlines under fixed priority policy, the total computational requirement that should be satisfied by time \( t \) can be computed as
( \sum_{j=1}^{P'_{i}-1} \min \{ (\alpha_j (t - t_c) \times e_j \times f_j), \min_{1 \leq k \leq (w-1)} (\alpha_j (t - t_k) \times e_j \times f_j - B_j^k) \} \\
+ \min \{ \beta'_{P'_{i}} (t - t_c), \min_{1 \leq k \leq (w-1)} (\beta'_{P'_{i}} (t - t_k) - \sum_{j=P'_{i}} B_j^k) \} + \delta_{T'_{i}} (t - t_c) \),

where \( \delta_{T'_{i}} (t - t_c) \) is the remaining computational requirement function (at \( t_c \)) due to tasks \( T'_{i}, \ldots, T'_{n} \). In particular, \( \delta_{T'_{n}} (t - t_c) = \delta (t - t_c) \), which is the remaining computational requirement due to tasks \( T'_{1}, \ldots, T'_{n} \).

Suppose the system is operating under frequency \( f_i \). The following condition should be satisfied so as to meet the computational requirement due to \( T'_{i} \) and the tasks that may arrive in the future:

\[
S(f_i, (t-t_c)) \geq ( \sum_{j=1}^{P'_{i}-1} \min \{ (\alpha_j (t - t_c) \times e_j \times f_j), \min_{1 \leq k \leq (w-1)} (\alpha_j (t - t_k) \times e_j \times f_j - B_j^k) \} \\
+ \min \{ \beta'_{P'_{i}} (t - t_c), \min_{1 \leq k \leq (w-1)} (\beta'_{P'_{i}} (t - t_k) - \sum_{j=P'_{i}} B_j^k) \} + \delta_{T'_{i}} (t - t_c) \) (t_c \leq t \leq D'_{i}).
\]

5.4.2 Prediction-Enabled EDF Based Online Real-Time DVS Algorithm

In Figure 5.4, \( PAEDF_P \) works in a similar way to \( EDF \) except that it employs a history window based prediction technique to predict computational requirement, and has the capability to adjust voltage-frequency level according to computational requirement so as to save energy. Basically, whenever \( PAEDF_P \) is invoked, it first constructs the remaining computational function \( \delta_{T'_{i}} (t - t_c) \) (1 \( \leq i \leq n \)) (step 2). This is accomplished by computing the computational requirement that must be satisfied by every deadline, starting from \( D'_{1} \) until \( D'_{n} \). \( \delta_{T'_{i}} (t - t_c), \ldots, \delta_{T'_{n}} (t - t_c) \) can be obtained by one round of scanning tasks \( T'_{1}, \ldots, T'_{n} \).
Frequency $f_i$ (1 ≤ i ≤ n) is derived according to the computational requirement due to the remaining computational requirement and the computational requirement that may arise in the future (steps 4-11). It is easy to see from Figure 5.4, frequency $f_i$ (1 ≤ i ≤ n) is computed as the smallest value that satisfies the following inequality:

$$S(f_i, (t - t_c)) \geq \sum_{j=1}^{m} \min\{\alpha_j(t - t_c - d_j) \times e_j \times f_i\} \times \min\{\alpha_j(t - t_c - d_j) \times e_j \times f_i - B_j^k\}$$

$$+ \delta_{T_x}(t - t_c) \quad (t_c \leq t \leq D_i'),$$

**Figure 5.4** Power-aware prediction-enabled EDF algorithm.
where \( B_j^k = (Window_j^\pi - Window_j^\eta) \), and \( Window_j^\pi \) and \( Window_j^\eta \) are the history records for class \( j \) at \( t_w \) and \( t_b \), respectively.

\( f_c \) is set to the maximum \( f_i (1 \leq i \leq n) \) (step 12). Steps 13-19 are used to update the history window. Note that between two consecutive frequency adjustment points, the processor cycles devoted to every class \( j \) \((1 \leq j \leq m)\) will be added to \( Window_j^\pi \). This operation is incorporated into step 21. After the history window is updated, current frequency is set to \( f_c \), and tasks are processed according to the EDF rule until \( T'_x \) completes (step 21). \( T'_x \) is the last task (in the sequence of \( T'_1, \ldots, T'_n \)) that the frequency computed with respect to its deadline \( D'_x \) is equal to \( f_c \). (Note that \( T'_1, \ldots, T'_n \) are in non-decreasing order of deadline.)

**Proposition 5.1** (for PAEDF_P policy) \( f_i \leq f_{\max} \) \((1 \leq i \leq n)\), where \( f_{\max} \) is the frequency corresponding to \( \beta(t) \).

**Proof:** In Figure 5.5, assuming that the frequency setting is \( f_{\max} \) before time \( t_c \). Because the remaining computational requirement at \( t_c \) is less than or equal to the theoretical remaining computational requirement at \( t_c \) (due to the fact that the number of the actual arrivals of tasks is always less than or equal to the theoretical upper bound number), the following holds:

\[
\delta_{\gamma_i} (t - t_c) \leq \delta^*(t - t_c) \quad (t_c \leq t),
\]

where \( \delta^*(t - t_c) \) is the theoretical remaining computational requirement at time \( t_c \).

**Figure 5.5** Frequency analysis.

Thus the following holds:

\[
\delta_{\gamma_i} (t - t_c) \leq \delta^*(t - t_c) \quad (t_c \leq t) \Rightarrow \delta_{\gamma_i} (t - t_c) \leq \delta^*(t - t_c) \quad (t_c \leq t \leq D_i').
\]

Because class 1 to class \( m \) are schedulable under \( f_{\max} \), the following condition is always satisfied:
the following inequality must hold:

\[ S(\alpha_{j}, (t - t_{c})) \geq \left( \sum_{j=1}^{m} (\alpha_{j}(t - t_{c} - d_{j}) \times e_{j} \times f_{i}) + \delta^{*}(t - t_{c}) \right) \quad (t_{c} \leq t \leq D'_{i}) \]

Because

\[ \sum_{j=1}^{m} \min\{(\alpha_{j}(t - t_{c} - d_{j}) \times e_{j} \times f_{i}), \min_{1 \leq k \leq (w-1)}(\alpha_{j}(t - t_{k} - d_{j}) \times e_{j} \times f_{i} - B_{k}^{i})\} \]

\[ \leq \sum_{j=1}^{m} (\alpha_{j}(t - t_{c} - d_{j}) \times e_{j} \times f_{i}) \quad (t_{c} \leq t \leq D'_{i}) , \]

the following inequality must hold:

\[ S(f_{\text{max}}, (t - t_{c})) \geq \left( \sum_{j=1}^{m} \min\{(\alpha_{j}(t - t_{c} - d_{j}) \times e_{j} \times f_{i}), \min_{1 \leq k \leq (w-1)}(\alpha_{j}(t - t_{k} - d_{j}) \times e_{j} \times f_{i} - B_{k}^{i})\} \right) \]

\[ + \delta_{T_{i}}^{*}(t - t_{c}) \quad (t_{c} \leq t \leq D'_{i}) . \]

Thus, a frequency \( f_{i} \) that is not higher than \( f_{\text{max}} \) can be found to satisfy the above inequality, i.e., \( f_{i} \) is less than or equal to \( f_{\text{max}} \) and it satisfies the following condition:

\[ S(f_{i}, (t - t_{c})) \geq \left( \sum_{j=1}^{m} \min\{(\alpha_{j}(t - t_{c} - d_{j}) \times e_{j} \times f_{i}), \min_{1 \leq k \leq (w-1)}(\alpha_{j}(t - t_{k} - d_{j}) \times e_{j} \times f_{i} - B_{k}^{i})\} \right) \]

\[ + \delta_{T_{i}}^{*}(t - t_{c}) \quad (t_{c} \leq t \leq D'_{i}) . \]

The following proves that under frequency \( f_{c} \), the remaining computational requirement at \( D'_{x} \) is less than or equal to the theoretical remaining computational requirement at \( D'_{x} \).

Because

\[ f_{c} = f_{\alpha} = \text{MAX}\{f_{1}, f_{2}, \ldots, f_{n}\} \]

and

\[ S(f_{x}, (t - t_{c})) \geq \left( \sum_{j=1}^{m} \min\{(\alpha_{j}(t - t_{c} - d_{j}) \times e_{j} \times f_{x}), \min_{1 \leq k \leq (w-1)}(\alpha_{j}(t - t_{k} - d_{j}) \times e_{j} \times f_{x} - B_{k}^{i})\} \right) \]

\[ + \delta_{T_{i}}^{*}(t - t_{c}) \quad (t_{c} \leq t \leq D'_{x}) , \]

this indicates that the frequency \( f_{c} \) (\( f_{c} \geq f_{0} \)) is high enough to satisfy the predicted (theoretical) computational requirement within \([t_{c}, D'_{x}]\) and the computational requirement due to tasks \( T'_{1}, \ldots, T'_{x} \). Thus, the remaining computational requirement at \( D'_{x} \) must be less than or equal to the theoretical remaining computational requirement, i.e., there is no unreasonable remaining computational requirement at \( D'_{x} \) under frequency \( f_{c} \). In particular, if \( x \) is equal to \( n \), the frequency \( f_{c} \) is high enough to satisfy the predicted theoretical computational requirement within \([t_{c}, D'_{n}]\) and the computational requirement due to tasks \( T'_{1}, \ldots, T'_{n} \), and there is no unreasonable remaining computational requirement at \( D'_{n} \) under frequency \( f_{c} \).

Now, the assumption made at the very beginning of the proof can be removed. \( \square \)

**Theorem 5.3** If class 1 to class \( m \) with arrival curve \( \alpha_{i}(t) \) to \( \alpha_{m}(t) \) are schedulable with service curve \( \beta(t) \) under EDF rule, then they are schedulable under PAEDF_\( P \).
Figure 5.6 Schedulability analysis.

Proof: In Figure 5.6, \( t_{c1} \) and \( t_{c2} \) are two consecutive frequency adjustment points, and the frequency setting at \( t_{c1} \) is \( f_{c1} \). For interval \([t_{c1}, t_{c2}]\), only the following needs to be proved: (1) frequency \( f_{c1} \) is high enough to satisfy the predicted theoretical computational requirement plus the remaining computational requirement at \( t_{c1} \) throughout \([t_{c1}, t_{c2}]\) (so, no deadline miss happens throughout \([t_{c1}, t_{c2}]\)), and (2) under frequency \( f_{c1} \), no unreasonable remaining computational requirement at \( t_{c2} \), i.e., the remaining computational requirement at time \( t_{c2} \) is less than or equal to the theoretical computational requirement at \( t_{c2} \) (so, no deadline miss is caused due to the frequency adjustment at \( t_{c1} \)). It is easy to see from the proof of Proposition 5.1, both (1) and (2) are true. Because this holds at every frequency adjustment point, Theorem 5.3 holds.

5.4.3 Prediction-Enabled Fixed Priority Based Online Real-Time DVS Algorithm

In Figure 5.7, \( \text{PAPRI}_P \) works in a similar way to fixed priority policy except that it uses the history window based prediction technique to predict the computational requirement, and is able to adjust the voltage-frequency level according to the computational requirement so as to save energy. Basically, whenever \( \text{PAPRI}_P \) is invoked, it first constructs the remaining computational function \( \delta_{T_i}(t - t_c) \) (1\( \leq i \leq n \)) (step 2). Similar to \( \text{PAEDF}_P \), \( \delta_{T_1}(t - t_c), \ldots, \delta_{T_n}(t - t_c) \) can be obtained by one round of scanning the tasks \( T'_{1}, \ldots, T'_{n} \). To compute \( f_i \), \( \text{PAPRI}_P \) first computes the computational requirement due to those future events that have higher priorities than \( T'_i \) (steps 5-10). It then computes the computational requirement due to those future events that have priorities not higher than \( T'_i \) (steps 11-16). Frequency \( f_i \) is then determined according to
Figure 5.7 Power-aware prediction-enabled fixed priority algorithm.

the total computational requirement (step 18). It is easy to see from Figure 5.6, frequency $f_i$ is computed as the smallest value that satisfies the following inequality:
\[
S(f_i, (t - t_c)) \geq \left( \sum_{j=1}^{P_i' - 1} \min\{\alpha_j(t - t_c) \times e_j \times f_s, \min_{1 \leq k \leq (w-1)} (\alpha_j(t - t_k) \times e_j \times f_s - B^k_j)\} + \min\{\beta_{P_i'}(t - t_c), \min_{1 \leq k \leq (w-1)} (\beta_{P_i'}(t - t_k) - \sum_{j=P_i'}^{m} B^k_j)\} + \delta_{T_{x}}(t - t_c) \right) (t_c \leq t \leq D')
\]

where \(B^k_j = (\text{Window}^w_j - \text{Window}^k_j)\), and \(\text{Window}^w_j\) and \(\text{Window}^k_j\) are the history records for class \(j\) at \(t_w\) and \(t_k\), respectively, and \(\beta_{P_i'}(t - t_c)\) is the minimum service curve for class \(P_i'\) to class \(m\) to be schedulable. Note that \(\beta_{P_i'}(u)\) can be constructed in the system design stage.

\(f_c\) is set to the maximum \(f_i\) (1 ≤ i ≤ n) (step 19). Steps 20-26 are used to update history window. Note that between two consecutive frequency adjustments, the processor cycles devoted to every class \(j\) (1 ≤ j ≤ m) will be added to \(\text{Window}^w_j\). This operation is incorporated into step 28. After the history window is updated, current frequency is set to \(f_c\), and the tasks are processed according to the highest priority first rule until \(T'_{x}\) completes (step 28). \(T'_{x}\) is the last task (in the sequence of \(T'_{1}, ..., T'_{n}\)) that the frequency computed with respect to its deadline \(D'_{x}\) is equal to \(f_c\). Note that \(T'_{1}, ..., T'_{n}\) are in non-increasing order of priority.

**Proposition 5.2** (for PAPRI\_P policy) \(f_i \leq f_{\max} \ (1 \leq i \leq n)\), where \(f_{\max}\) is the frequency corresponding to \(\beta(t)\).

**Proof:** In Figure 5.5, assuming that the frequency setting is \(f_{\max}\) before time \(t_c\). Because the remaining computational requirement at \(t_c\) is less than or equal to the theoretical remaining computational requirement at \(t_c\) (due to the fact that the number of the actual arrivals of tasks is always less than or equal to the theoretical upper bound number), the following holds:
\[
\delta_{T_{x}}(t - t_c) \leq \delta^*(t - t_c) \quad (t_c \leq t),
\]
where \(\delta^*(t - t_c)\) is the theoretical remaining computational requirement at time \(t_c\).
Thus the following must hold:
\[ \delta^*_{\tau_i}(t-t_c) - \delta^*(t-t_c) \quad (t_c \leq t) \Rightarrow \delta^*_{\tau_i}(t-t_c) - \delta^*(t-t_c) \quad (t_c \leq t \leq D'_i) \]

Because class 1 to class \( m \) are schedulable under \( f_{\text{max}} \), at any time point \( t \) \((t_c \leq t \leq D'_i)\), the following holds:
\[ S(f_{\text{max}},(t-t_c)) \geq \left( \sum_{j=1}^{P_i-1} \min\left\{ \alpha_j(t-t_c) \times e_j \times f_s \right\}, \min_{1 \leq k \leq (w-1)} (\alpha_j(t-t_k) \times e_j \times f_s - B^k_j) \right) \]

Because
\[ \sum_{j=1}^{P_i-1} \min\left\{ \alpha_j(t-t_c) \times e_j \times f_s \right\}, \min_{1 \leq k \leq (w-1)} (\alpha_j(t-t_k) \times e_j \times f_s - B^k_j) \]

and
\[ \min\left\{ \beta'_{p_i}(t-t_c), \min_{1 \leq k \leq (w-1)} (\beta'_{p_i}(t-t_k) - \sum_{j=P_i}^{m} B^k_j) \right\} \leq \beta'_{p_i}(t-t_c) \quad (t_c \leq t \leq D'_i) \]

the following inequality must hold:
\[ S(f_{\text{max}},(t-t_c)) \geq \left( \sum_{j=1}^{P_i-1} \min\left\{ \alpha_j(t-t_c) \times e_j \times f_s \right\}, \min_{1 \leq k \leq (w-1)} (\alpha_j(t-t_k) \times e_j \times f_s - B^k_j) \right) \]

\[ + \min\left\{ \beta'_{p_i}(t-t_c), \min_{1 \leq k \leq (w-1)} (\beta'_{p_i}(t-t_k) - \sum_{j=P_i}^{m} B^k_j) \right\} + \delta_{\tau_i}(t-t_c) \quad (t_c \leq t \leq D'_i) \]

Thus, a frequency \( f_is \) that is not higher than \( f_{\text{max}} \) can be found to satisfy the above inequality, i.e., \( f_is \) is less than or equal to \( f_{\text{max}} \) and it satisfies the following condition:
\[ S(f_i,(t-t_c)) \geq \left( \sum_{j=1}^{P_i-1} \min\left\{ \alpha_j(t-t_c) \times e_j \times f_s \right\}, \min_{1 \leq k \leq (w-1)} (\alpha_j(t-t_k) \times e_j \times f_s - B^k_j) \right) \]

\[ + \min\left\{ \beta'_{p_i}(t-t_c), \min_{1 \leq k \leq (w-1)} (\beta'_{p_i}(t-t_k) - \sum_{j=P_i}^{m} B^k_j) \right\} + \delta_{\tau_i}(t-t_c) \quad (t_c \leq t \leq D'_i) \]

The following proves that under frequency \( f_c \), the remaining computational requirement at \( D'_x \) is less than or equal to the theoretical remaining computational requirement at \( D'_x \).

Because
\[ f_c = f_x = \text{MAX}\{ f_1, f_2, \ldots, f_n \} \]

and
\[ S(f_x,(t-t_c)) \geq \left( \sum_{j=1}^{P_i-1} \min\left\{ \alpha_j(t-t_c) \times e_j \times f_s \right\}, \min_{1 \leq k \leq (w-1)} (\alpha_j(t-t_k) \times e_j \times f_s - B^k_j) \right) \]

\[ + \min\left\{ \beta'_{p_i}(t-t_c), \min_{1 \leq k \leq (w-1)} (\beta'_{p_i}(t-t_k) - \sum_{j=P_i}^{m} B^k_j) \right\} + \delta_{\tau_i}(t-t_c) \quad (t_c \leq t \leq D'_x) \]
this indicates that the frequency \( f_c \) is high enough to satisfy the predicted (theoretical) computational requirement within \([t_c, D']\) and the computational requirement due to tasks \( T'_1, ..., T'_x \). Thus, the remaining computational requirement at \( D' \) must be less than or equal to the theoretical remaining computational requirement at \( D' \), i.e., there is no unreasonable remaining computational requirement at \( D' \) under frequency \( f_c \). In particular, if \( x \) is equal to \( n \), the frequency \( f_c \) is high enough to satisfy the predicted theoretical computational requirement within \([t_c, D']\) and the computational requirement due to tasks \( T'_1, ..., T'_n \), and there is no unreasonable remaining computational requirement at \( D' \) under frequency \( f_c \).

Now, the assumption made at the very beginning of the proof can be removed.  

**Theorem 5.4** If class 1 to class \( m \) with arrival curve \( \alpha_i(t) \) to \( \alpha_m(t) \) are schedulable with service curve \( \beta(t) \) under fixed priority rule, then they are schedulable under PAPRI_P.

**Proof:** The proof can be accomplished in a way similar to Theorem 5.3. 

5.4.4 Further Discussion on the Algorithms

- **Complexity analysis.** It is easy to see from Figures 5.4 and 5.7 that the complexities of both PAPRI_P and PAEDF_P are \( O(wnm) \), given \( m \) classes of events, \( n \) tasks in the task queue, and a history window of width \( w \). Because \( w \) is usually a small constant, the complexities of both PAPRI_P and PAEDF_P are dominated by \( n \) and \( m \).

- **Online real-time DVS without prediction.** So far the author discussed the online real-time DVS based on history window based prediction. An interesting problem is whether the online real-time DVS could be conducted without prediction, i.e., frequency is determined solely based on existing computational requirement. Unfortunately, this naive idea does not work. In Section 5.5, two algorithms (PAPRI_NP and PAEDF_NP) that attempt to conduct DVS without prediction are constructed, and both of them failure. This indicates that in a
random hard real-time environment, online DVS without prediction may cause system fail. This is of great importance for conducting online DVS in hard real-time systems.

- **Accommodate and respond to variations.** From Figure 5.4 and Figure 5.7, it is easy to see that both $PAEDF_P$ and $PAPRI_P$ conduct frequency adjustment at a time point when a specified existing task actually completes, and the completion of this task depends on the actual execution times of other tasks and the actual arrival of events. If the actual execution times of those tasks are less than the predicted WCETs of them, the specified task will complete earlier than it is predicted. Similarly, if the actual number of arrived events is less than that of the predicted events, the specified task will also complete earlier than is predicted. This implies that the frequency adjustment contained in both $PAPRI_P$ and $PAEDF_P$ depends on the actual rather than the predicted execution times of tasks and the actual rather than the predicted arrival of events. Therefore, both $PAPRI_P$ and $PAEDF_P$ are able to accommodate and respond to the variation between the WCETs (predicted execution time) and the actual execution times, and the variation between the predicted arrivals of events and the actual arrivals of events.

### 5.5 Simulation Analysis

This section studies the effectiveness of the online real-time DVS algorithms by simulation. To facilitate the evaluation, it is assumed that (1) the time overhead and energy expense of voltage-frequency switching is negligible [61, 62], and (2) the time
overhead and energy expense of the algorithms are negligible. As a matter of fact, these assumptions are made, explicitly or implicitly, in almost all real-time DVS research except those that specifically address those issues. As mentioned before, although energy saving is the objective, meeting timing constraints is required in hard real-time environments. So, these algorithms are evaluated along two dimensions, i.e., energy consumption and deadline miss. It is necessary to check how well these algorithms perform in energy saving when compared to those algorithms that do not conduct DVS. To check how well these algorithms perform in meeting timing constraints, they are compared with two other algorithms (i.e., PAPRI_NP and PAEDF_NP), which are constructed in a similar way to PAPRI_P and PAEDF_P except that they conduct DVS without prediction.

5.5.1 Simulation Settings

The simulation contains 9 classes of events, and the settings for every class are listed in Table 5.1. The priorities assigned to class 1, class 2, ..., class 9 are \(P_1, P_2, ..., P_9\), and they satisfy: \(P_1 < P_2 < ... < P_9\), i.e., class 1 has the highest priority and class 9 has the lowest priority. See Table 5.1, when infinite levels of frequencies is assumed, the highest frequency is set to 90MHz, and the corresponding voltage is 5.2V. Under this assumption, frequency can be set to any value between the highest frequency and the lowest frequency (0MHz), and the voltage will be adjusted accordingly. When limited levels of frequencies is assumed, there are four optional frequencies, i.e., 90MHz, 54MHz, 36MHz and 18MHz, and the corresponding voltages are 5.2V, 3.3V, 2.2V and 1.0V. Hence, the frequency and voltage adjustment is limited. The width of the history
window is set to 5 except in Figure 5.10, where it is also set to 10 so as to study the energy savings under different widths of history window.

Table 5.1 Simulation Settings

<table>
<thead>
<tr>
<th>class</th>
<th>processing task</th>
<th>WCET (under benchmark frequency ( f_s = 1 \text{MHZ} ))</th>
<th>relative deadline</th>
<th>priority</th>
<th>( \alpha(t) ) (t ( \geq 0 )) (( \alpha(t)=0 ) for t&lt;0)</th>
<th>infinite levels of frequencies</th>
<th>Limited levels of frequencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( T_1 )</td>
<td>8ms</td>
<td>10ms</td>
<td>( P_1 )</td>
<td>250t+5</td>
<td>( 90 \times 10^6 ) t</td>
<td>( 90 \times 10^6 ) t</td>
</tr>
<tr>
<td>2</td>
<td>( T_2 )</td>
<td>8ms</td>
<td>10ms</td>
<td>( P_2 )</td>
<td>250t+5</td>
<td>( 54 \times 10^6 ) t</td>
<td>( 36 \times 10^6 ) t</td>
</tr>
<tr>
<td>3</td>
<td>( T_3 )</td>
<td>5ms</td>
<td>8ms</td>
<td>( P_3 )</td>
<td>400t+8</td>
<td>( 18 \times 10^6 ) t</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>( T_4 )</td>
<td>5ms</td>
<td>8ms</td>
<td>( P_4 )</td>
<td>400t+8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>( T_5 )</td>
<td>5ms</td>
<td>8ms</td>
<td>( P_5 )</td>
<td>400t+8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>( T_6 )</td>
<td>3ms</td>
<td>12ms</td>
<td>( P_6 )</td>
<td>500t+10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>( T_7 )</td>
<td>3ms</td>
<td>12ms</td>
<td>( P_7 )</td>
<td>500t+10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>( T_8 )</td>
<td>3ms</td>
<td>12ms</td>
<td>( P_8 )</td>
<td>500t+10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>( T_9 )</td>
<td>3ms</td>
<td>12ms</td>
<td>( P_9 )</td>
<td>500t+10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.5.2 Simulation Results

Figure 5.8 (a) and (b) are the simulation results under infinite levels of frequencies assumption. As it is shown that no task misses its deadline with both \( PAEDF_P \) and \( PAPRI_P \) while lots of tasks miss their deadlines with \( PAEDF_NP \) and \( PAPRI_NP \). The reason is that while \( PAEDF_P \) and \( PAPRI_P \) conduct frequency adjustment based on the computational requirement of existing tasks and that of the predicted future tasks, \( PAEDF_NP \) and \( PAPRI_NP \) conduct frequency adjustment solely based on the computational requirement of existing tasks. These results indicate that in a random hard real-time environment, conducting DVS without considering future computational requirement may lead to system failure (deadline miss). Please note that the setting of \( \beta(t) \) is high enough for all events to be feasibly processed in time. (This can be verified by the
pure EDF and pure PRIORITY algorithms. With both of them, no task misses its deadline.) Please also note that because there is no deadline miss with PAEDF_P, EDF, PAPRI_P and PRIORITY, the "number of tasks" with each of them is zero (Figure 5.8).

**Figure 5.8** Deadline miss with infinite levels of frequencies.

**Figure 5.9** Energy consumption and energy saving with infinite levels of frequencies.
Figure 5.9 (a) and (b) are the simulation results under infinite levels of frequencies assumption. It is easy to see that the energy consumptions under both $PAEDF_P$ and $PAPRI_P$ are much less than those under pure $EDF$ and pure $PRIORITY$ throughout the simulation interval. Compared to $EDF$, $PAEDF_P$ constantly saves 10% or more energy (see the bottom figure of Figure 5.9(a)). For $PAPRI_P$, it saves more than 61% energy when compared to its counterpart (see the bottom figure of Figure 5.9(b)).

![Figure 5.10](image)

(a) Energy saving by $PAEDF_P$ under different widths of history window
(b) Energy saving by $PAPRI_P$ under different widths of history window

**Figure 5.10** Energy savings with infinite levels of frequencies under different history window widths.

Figure 5.10 shows that both $PAEDF_P$ and $PAPRI_P$ save more energy under history window of width 5 than that under history window of width 10. This indicates that wider history window cause more energy saving. The reason is that a wider history window provides more points for prediction, and thus provides more opportunities for adjusting frequency to lower levels.

Figure 5.11 is the simulation result under limited levels of frequencies assumption. It is also shown that the energy consumptions under both $PAEDF_P$ and
$PAPRI_P$ are much lower than those under their corresponding counterparts. Compared to $EDF$ and $PRIORITY$, the energy constantly saved by $PAEDF_P$ and $PAPRI_P$ is above 2% and 55%, respectively.

![Energy Consumption and Energy Savings](image)

Figure 5.11 Energy consumption and energy savings with limited levels of frequencies.

Figure 5.9 and Figure 5.11 show that both $PAEDF_P$ and $PAPRI_P$ outperform their corresponding counterparts. The reason is that while $PAEDF_P$ and $PAPRI_P$ have the capability to dynamically adjust the operating frequency according to computational requirement, $EDF$ and $PRIORITY$ always work at the highest frequency. As a result, $PAEDF_P$ and $PAPRI_P$ finish the same computational work as their counterparts but at reduced energy consumption. This result holds even with limited levels of frequencies constraint (Figure 5.11).

Figure 5.9 and Figure 5.11 show that the energy saved by both $PAEDF_P$ and $PAPRI_P$ under limited levels of frequencies is less than that under infinite levels of
frequencies. The reason is that with limited levels of frequencies constraint, they can only choose from a limited set of frequencies. As a result, they can not always find the most suitable frequency (i.e., the computed frequency), and most of the time they have to pick a frequency that is close to but higher than the computed frequency, so as to avoid deadline misses. This limitation eventually results in less energy saving.

Figure 5.12  Energy consumption and energy savings with varying execution/computation time.

With the simulations in Figure 5.12 (a) and 5.12 (b), the actual execution times of tasks vary from 10% to 100% of their corresponding WCETs. These simulations are used to test how well PAEDF_P and PAPRI_P perform when the actual execution times of tasks are different from their WCETs. As shown in Figure 5.12 (a) and 5.12 (b), both PAEDF_P and PAPRI_P perform much better than their corresponding counterparts in
all simulations. Compared with pure EDF, PAEDF_P saves more than 15% of the energy, and PAPRI_P saves more than 61% of the energy compared with pure PRIORITY. It is also easy to see that PAEDF_P and PAPRI_P perform well even when the actual execution times of tasks are as low as only 10% of their corresponding WCETs. This indicates that both PAEDF_P and PAPRI_P can accommodate and well adapt to the variation between the predicted and the actual execution times of tasks.
CHAPTER 6
CONCLUSION

In this dissertation, new data structures, models, algorithms, and techniques for real-time resource management are explored. The main contributions of this dissertation are summarized as follows.

A class of TIT trees is constructed. The TIT* tree is a general data structure that can be applied to a wide variety of real-time scheduling systems to perform the schedulability test of tasks (or messages). It can effectively reduce the average costs of the schedulability tests. The TIT-V tree can be applied to the schedulability tests of a class of parallel/distributed real-time systems, and the complexity of the corresponding schedulability tests can be reduced from $O(m^2n\log n)$ to $O(m\log n + m\log m)$. The TIT-RL tree can be applied to the online admission control in a uni-processor based real-time system, and the complexity of the online admission control can be reduced from $O(n^2)$ to $O(n\log n)$. The TIT-RL tree can also be used as the building block for a class of parallel/distributed real-time systems. Compared to those non-TIT tree based scheduling modules, the TIT tree based ones are much more efficient. Therefore, the TIT trees are effective approaches to efficient real-time scheduling modules. More details about TIT trees can be found in [22].

A new utility accrual model called UAM^+ is established for the resource allocation in asynchronous real-time distributed systems. The model is constructed based on the timeliness of both computation and communication. Moreover the interplay between computation and communication is also captured and characterized in the model. A resource allocation algorithm called IAUASA is developed under UAM^+. The
performance of IAUASA is much superior to two other resource allocation algorithms that are developed according to conventional UAM and conventional idea. Therefore, UAM⁺ provides a more effective framework for resource managers to optimize resource allocation along two dimensions, i.e., computation and communication, rather than conventional one dimension, i.e., computation or communication, in distributed real-time systems. More details about UAM⁺ model can be found in [65].

An online distributed algorithm called IDRSA is developed under the UAM⁺ model to conduct resource allocation in a distributed real-time system. IDRSA integrates DDA technique to explore the interplay between computation and communication. Extensive simulations reveal the excellent performance of IDRSA, especially when the interplay between computation and communication is tight. This not only proves the excellence of IDRSA in the resource allocation in distributed real-time systems, but also further validates the effectiveness of the UAM⁺ model for the resource management in distributed real-time systems. More details about IDRSA can be found in [88].

Calculus curve based real-time DVS technique is established. This technique is able to accommodate random event/task arrivals. Novel real-time DVS algorithms based on the technique are developed. These algorithms are able to accommodate and respond to the variation between the predicted and the actual execution times of tasks as well as the variation between the predicted and the actual arrivals of events, and they are excellent in energy saving. Therefore, the calculus curve based real-time DVS technique is an effective approach to energy-efficient real-time resource management in random hard real-time environments. More details about this technique can be found in [89].
CHAPTER 7
FUTURE WORK

The preceding chapters demonstrate that the proposed data structures, models, algorithms, and techniques can benefit real-time systems. This chapter discusses some directions that further work may take in the future.

In Chapter 2, the TIT tree is studied. Because the TIT tree is a basic data structure, more extensions of it could be explored and applied to more real-time systems to improve their efficiency and performance. Some TIT trees may be specifically designed for some specific systems, others may be applicable to a number of systems. Because efficiency and performance are always of critical importance for real-time systems, how to find and construct more TIT trees and effectively apply them to more real-time systems in the real world is an interesting work and deserves further exploration.

In Chapter 3, UAM$^+$ model is studied. Because UAM$^+$ well captures and characterizes the interplay between computation and communication in distributed real-time systems, it provides an effective approach to constructing effective resource management in such systems. In the future, more effective resource allocation algorithms and techniques under UAM$^+$ can be developed and applied to different distributed real-time systems.

In Chapter 4, a two-level scheduling framework is discussed. It can effectively decompose resource scheduling into subprocesses and reduce system complexity through parallelism. In the future, the two-level scheduling framework can be further investigated to improve system scalability and fault-tolerance.
In Chapter 5, calculus curve based real-time DVS technique is studied. This technique is able to accommodate random event/task arrivals, and it has been successfully integrated into two real-time scheduling algorithms. In the future, it will be an interesting work to integrate this technique into more real-time algorithms to conduct energy-efficient resource management. This is of special significance for those embedded real-time systems that need to deal with random event/task arrivals. In addition, how to integrate leakage power optimization into the history window based prediction technique is also worthy of further investigation.
APPENDIX

THE ADJUST OPERATION ON TIT-V TREE (FOR CASE 4)

The Adjust algorithm (for the process on case 4) and its subroutines are illustrated in Figure A.1 to A.6. Note that the process described here does not include how to append a leaf node to the right side of a TIT-V tree, because this can be easily accomplished by inserting the node to the tree at the right-most position. Figure A.1 is the top level framework of the algorithm. Figure A.2 and A.3 are the frameworks for adjusting the left subtree and the right subtree of CommonParent, respectively. Figure A.4 illustrates how to merge and balance two subtrees $X$ and $F$ where the height of $X$ is less than that of $F$, i.e., $|X| < |F|$. In Figure A.4(a), the tree rooted from $X$ is the left subtree, and that rooted from $F$ is the right subtree. In the case that the tree rooted from $X$ is the right subtree, and that rooted from $F$ is the left subtree, the process is similar to A.4. Figure A.4(a) shows the two subtrees to be processed. At first, Adjust needs to find the left-most node $B$, such that $|X| = |B|$. Once $B$ is found, a new node $BX$ will be created (Figure A.4(b)). If this cause $A$ loses balance, a LL rotation is needed (the LL and LR rotations are similar to those with AVL tree [7], and LL rotation is applied to a node when the Left subtree of the Left child of that node cause unbalance and LR rotation is applied to a node when the Right subtree of the Left child of that node cause unbalance). Figure A.4(c) is the tree obtained after applying a LL rotation to $A$ in Figure A.4(b). If $C$ loses balance after the LL rotation, a LR rotation is needed. Figure A.4(d) is the tree obtained after applying a LR rotation to $C$ in Figure A.4(c). In the case that $A$ is balanced but $C$ lose balance in Figure A.4(b), a LL rotation is needed, and the resulting tree will be the same as that in Figure A.4(d).
Node1—the node whose interval of vacancy contains point P (see Figure 2.7).
Node2—the node whose interval of vacancy contains point P2 (see Figure 2.7).

CommonParent ← Find the nearest common parent of Node1 and Node2

Adjust the left subtree of CommonParent

Adjust the right subtree of CommonParent

Temp ← CommonParent

Tree rooted from Temp is balanced

Y

H1 ← height of the left subtree of Temp
H2 ← height of the right subtree of Temp

N

|H1 - H2| = 2

Rotation

Merge_and_Balance the two subtrees

Root of the resulting subtree replaces Temp

Temp ← root of the resulting subtree

Subtree rooted from Temp is a balanced subtree

Temp ← Temp parent

Temp is the root of the whole tree

Yes

End

Figure A.1 Adjust algorithm (for the process on case 4).
If $\text{Current}_1$ is the right child of its parent, or the left child of $\text{CommonParent}$, then:

$\text{Current}_1 \leftarrow \text{Current}_1$.parent

Subtree rooted from $\text{Current}_1$ is:

- $H_1$—height of the left subtree of $\text{Current}_1$
- $H_2$—height of the right subtree of $\text{Current}_1$

If $|H_1 - H_2| = 2$, then:

- Merge_and_Balance the two subtrees
- Root of the resulting subtree replaces $\text{Current}_1$

Otherwise, if $|H_1 - H_2| < 2$, then:

- Subtree rooted from $\text{Current}_1$ is a balanced subtree

If $\text{Current}_2$ is the left child of its parent, or the right child of $\text{CommonParent}$, then:

$\text{Current}_2 \leftarrow \text{Current}_2$.parent

Subtree rooted from $\text{Current}_2$ is:

- $H_1$—height of the left subtree of $\text{Current}_2$
- $H_2$—height of the right subtree of $\text{Current}_2$

If $|H_1 - H_2| = 2$, then:

- Merge_and_Balance the two subtrees
- Root of the resulting subtree replaces $\text{Current}_2$

Otherwise, if $|H_1 - H_2| < 2$, then:

- Subtree rooted from $\text{Current}_2$ is a balanced subtree

**Figure A.2** Adjust the left subtree.

**Figure A.3** Adjust the right subtree.
If A loses balance (original $|E| < |B|$)
apply LL rotation

If C loses balance (original $|C| > |E| < |B|$)
apply LR rotation

(a) Two subtrees to be processed
(b) A new node $BX$ is created
(c) Tree obtained from (b) by applying LL rotation to $A$
(d) Tree obtained from (c) by applying LR rotation to $C$

Figure A.4 Merge_and_Balance two subtrees.
Figure A.5 Right_Cut.

Figure A.6 Left_Cut.
REFERENCES


