1996

A performance model for parallel real-time systems

Il Kyu Lee
University of Wollongong

Recommended Citation
NOTE

This online version of the thesis may have different page formatting and pagination from the paper copy held in the University of Wollongong Library.

UNIVERSITY OF WOLLONGONG

COPYRIGHT WARNING

You may print or download ONE copy of this document for the purpose of your own research or study. The University does not authorise you to copy, communicate or otherwise make available electronically to any other person any copyright material contained on this site. You are reminded of the following:

Copyright owners are entitled to take legal action against persons who infringe their copyright. A reproduction of material that is protected by copyright may be a copyright infringement. A court may impose penalties and award damages in relation to offences and infringements relating to copyright material. Higher penalties may apply, and higher damages may be awarded, for offences and infringements involving the conversion of material into digital or electronic form.
A Performance Model for Parallel Real-Time Systems

A thesis submitted in fulfilment of the requirement for the award of the degree of Doctor of Philosophy

from
The University of Wollongong

by
Il Kyu Lee, MSc.

Supervisor: Phillip John McKerrow
Department of Computer Science

1996
Declaration

I hereby declare that I am the sole author of this thesis. I also declare that the material presented within is my own work except where duly acknowledged.

Il Kyu Lee

1996
I dedicate this thesis
to
my wife Sook Hee Lee,
my two daughters Mee Jin and Mee Eun Lee,
and my son Jong Do Lee.
Acknowledgments

I would like to express my deep feeling of gratitude to my supervisor Dr. Phillip J. McKerrow for his kind guidance, support, and encouragement during the course of this research. I would also like to thank prof. Doherty for his advice, prof. Hext for a careful review of this work and Dr. Jonathan Gray for giving me the knowledge of parallel processing.

Thanks are also due to my laboratory colleagues Shaomin, David, Pushkar, Pierre, Neil, and my Korean friends Sung Jin Cheong, Yeong Joon Lee, Chae Kyu Kim for their moral support. I am deeply indebted to my immediate and extended family in Korea for their concerns and support over the years. Most of all, many thanks to my good wife and lovely children for sharing my research journey and enduring the hardships with me.

I have regularly meditated upon the problems at the pond besides my laboratory. Insights and understandings have always come from an invisible source which I cannot resist. I thank Electronics and Telecommunications Research Institute (ETRI) in Korea for my experience of real-time systems development. Finally, I would like to thank the Department of Employment, Education and Training for providing me financial support in the form of fee-paying scholarship throughout my study.
Abstract

Real-time systems need to exploit high-speed parallel processing technology to meet their strict timing requirements. A general parallel architecture may have multiple stages connected in a pipeline, where each stage has multiple servers.

This thesis deals with performance analysis of such parallel real-time systems. A regular pipeline model is proposed as a mathematically tractable model. A concept of optimal instance blocking is introduced to achieve flow balance at each stage. A minimum-response-priority-assignment algorithm is presented to achieve the minimum end-to-end response time for each task. A concept of instance distribution is introduced to schedule a task whose execution time is greater than its period.

A general model for a pipelined multi-server parallel real-time system is developed in stages, starting from a single server model. Each modelling stage uses results from the previous stages. The final performance model provides a schedulability test; calculates end-to-end task response times, required buffer space and optimal hardware capacity; and includes an optimal scheduling algorithm for each stage.

The model has been validated by simulation. The limitations of the model are discussed. The model is theoretical rather than practical to achieve mathematical tractability. However it gives us predictable upper-bound estimates for the system performance.
Publications from Thesis

Table of Contents

Declaration..............................................................................................................i
Dedication................................................................................................................ii
Acknowledgments..................................................................................................iii
Abstract....................................................................................................................iv
Publications..............................................................................................................v
List of Algorithms..................................................................................................ix
List of Equations......................................................................................................x
List of Figures...........................................................................................................xi
List of Tables...........................................................................................................xii

Chapter 1 Introduction..........................................................................................1
  1.1 Background.......................................................................................................1
  1.2 Problem Analysis............................................................................................2
  1.3 Goal of Thesis..................................................................................................4
  1.4 Thesis Organisation.........................................................................................5

Chapter 2 Performance Indices.............................................................................7
  2.1 Workload and System Parameters................................................................8
  2.2 Schedulability..................................................................................................10
  2.3 Throughput......................................................................................................10
  2.4 Utilisation........................................................................................................14
  2.5 Priority............................................................................................................15
  2.6 Response Time...............................................................................................16
    2.6.1 Worst-Case Response Time....................................................................17
    2.6.2 Average Response Time........................................................................18
  2.7 Buffer Requirement.........................................................................................18
  2.8 Server Capacity.............................................................................................19

Chapter 3 A Single Server Model........................................................................22
  3.1 Introduction.....................................................................................................22
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2</td>
<td>System Model</td>
</tr>
<tr>
<td>3.3</td>
<td>Schedulability Test</td>
</tr>
<tr>
<td>3.4</td>
<td>Utilisation Factor</td>
</tr>
<tr>
<td>3.5</td>
<td>Response Time</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Joseph and Pandya's Response Time Model</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Extensions to Joseph and Pandya's Model</td>
</tr>
<tr>
<td>3.6</td>
<td>Best Priority Assignment</td>
</tr>
<tr>
<td>3.7</td>
<td>Buffer Space</td>
</tr>
<tr>
<td>3.8</td>
<td>Server Capacity</td>
</tr>
<tr>
<td>3.8.1</td>
<td>Capacity Planning</td>
</tr>
<tr>
<td>3.8.2</td>
<td>Optimal Server Capacity</td>
</tr>
<tr>
<td>3.8.3</td>
<td>Spare Execution Time for a Task</td>
</tr>
<tr>
<td>3.8.4</td>
<td>Upper-Bound Completion Time for an Instance of a Task</td>
</tr>
<tr>
<td>3.8.5</td>
<td>Deadline Adjustment</td>
</tr>
<tr>
<td>3.9</td>
<td>Examples</td>
</tr>
<tr>
<td>3.9.1</td>
<td>Two Examples for Preemptive Fixed Priority Scheduling</td>
</tr>
<tr>
<td>3.9.2</td>
<td>An Example for Non-preemptive Fixed Priority Scheduling</td>
</tr>
<tr>
<td>3.9.3</td>
<td>An Example for Preemptive Deadline Driven Scheduling</td>
</tr>
<tr>
<td>3.9.4</td>
<td>Two Examples for Server Capacity Estimation</td>
</tr>
<tr>
<td>3.10</td>
<td>Discussion</td>
</tr>
<tr>
<td><strong>Chapter 4</strong></td>
<td><strong>A Multi-Server Model</strong></td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>4.2</td>
<td>System Model</td>
</tr>
<tr>
<td>4.3</td>
<td>Condition for Multi-Server Processing</td>
</tr>
<tr>
<td>4.4</td>
<td>Performance Model</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Task Modification</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Modified-Task-First-Fit-Allocation (MTFFA)</td>
</tr>
<tr>
<td>4.5</td>
<td>Three Examples</td>
</tr>
<tr>
<td>4.6</td>
<td>Discussion</td>
</tr>
<tr>
<td><strong>Chapter 5</strong></td>
<td><strong>A Pipeline Model</strong></td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Flow Balance and End-to-End Responsiveness</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Problems</td>
</tr>
<tr>
<td>5.2</td>
<td>Optimal Instance Blocking</td>
</tr>
<tr>
<td>5.3</td>
<td>Regular Pipelining</td>
</tr>
</tbody>
</table>
List of Algorithms

2.1: Schedulability Test .................................................................11
2.2: Server Overflow Test ...............................................................14
2.3: Buffer Overflow Test ...............................................................19
2.4: Single Server Capacity ............................................................20
3.1: Schedulability Test for a Single Server System .................................29
3.2: Worst-Case Response Time with Critical Instant ................................35
3.3: Worst-Case Response Time with Busy Period ..................................37
3.4: Worst-Case Response Time with Non-Preemption ..............................39
3.5: Individual Instance Response Times of a Task ..................................44
3.6: Best-Priority-Assignment (BPA) ..................................................46
3.7: Optimal Server Capacity ...........................................................52
3.8: Spare Execution Time for a Task ..................................................54
3.9: Upper-Bound Completion Time for an Instance .................................57
3.10: Deadline Adjustment ...............................................................58
4.1: Modified-Task-First-Fit-Allocation (MTFFA) ...................................83
4.2: Schedulability Test for a Multi-Server System ..................................84
5.1: Schedulability Test for a Pipeline System .......................................113
5.2: Minimum-Response-Priority-Assignment (MRPA) ...............................115
6.1: Schedulability Test for a Pipelined Multi-Server System .......................133
6.2: Modified-Task-Minimum-Response-Allocation (MTMRA) .....................136
List of Equations

(2.1) ................................................................. 12
(2.2) ................................................................. 12
(2.3) ................................................................. 14
(2.4) ................................................................. 19
(3.1) ................................................................. 30
(3.2) ................................................................. 33
(3.3) ................................................................. 33
(3.4) ................................................................. 33
(3.5) ................................................................. 34
(3.6) ................................................................. 35
(3.7) ................................................................. 36
(3.8) ................................................................. 41
(3.9) ................................................................. 42
(3.10) ............................................................... 42
(3.11) ............................................................... 43
(3.12) ............................................................... 43
(3.13) ............................................................... 47
(3.14) ............................................................... 47
(3.15) ............................................................... 48
(3.16) ............................................................... 56
(4.1) ................................................................. 79
(4.2) ................................................................. 79
(4.3) ................................................................. 80
(4.4) ................................................................. 81
(4.5) ................................................................. 85
(4.6) ................................................................. 85
(4.7) ................................................................. 85
(4.8) ................................................................. 85
(4.9) ................................................................. 86
(5.1) ................................................................. 112
(5.2) ................................................................. 116
(5.3) ................................................................. 116
(5.4) ................................................................. 117
(5.5) ................................................................. 117
(5.6) ................................................................. 118
(5.7) ................................................................. 118
(5.8) ................................................................. 121
(5.9) ................................................................. 121
(5.10) ............................................................... 121
(5.11) ............................................................... 122
(6.1) ................................................................. 134
(6.2) ................................................................. 134
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A Performance Model</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>Hardware Configuration</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>Software Configuration</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>A Single Server System</td>
<td>28</td>
</tr>
<tr>
<td>3.2</td>
<td>Response Delay due to Interrupts by Higher-priority Tasks.</td>
<td>32</td>
</tr>
<tr>
<td>3.3</td>
<td>Illustration of Algorithm 3.3</td>
<td>37</td>
</tr>
<tr>
<td>3.4</td>
<td>Illustration of Algorithm 3.4</td>
<td>40</td>
</tr>
<tr>
<td>3.5</td>
<td>An Analogy</td>
<td>49</td>
</tr>
<tr>
<td>3.6</td>
<td>Optimal Server Capacity</td>
<td>51</td>
</tr>
<tr>
<td>3.7</td>
<td>Simulated CPU's Execution Process of Tables 3.1 and 3.2</td>
<td>64</td>
</tr>
<tr>
<td>3.8</td>
<td>The Execution Time Map for the Tasks in Table 3.7</td>
<td>68</td>
</tr>
<tr>
<td>4.1</td>
<td>A Multi-Server System</td>
<td>76</td>
</tr>
<tr>
<td>4.2</td>
<td>Execution Time Sequence Diagram for distribution of task $i$ to 4 servers</td>
<td>80</td>
</tr>
<tr>
<td>4.3</td>
<td>$T_i$ and $C_i$ for task $i$ from viewpoint of individual servers and whole system</td>
<td>81</td>
</tr>
<tr>
<td>5.1</td>
<td>A Pipeline System</td>
<td>96</td>
</tr>
<tr>
<td>5.2</td>
<td>Optimal Instance Blocking Factor</td>
<td>100</td>
</tr>
<tr>
<td>5.3</td>
<td>Example System for Optimal Instance Blocking Factor</td>
<td>102</td>
</tr>
<tr>
<td>5.4</td>
<td>Execution Map when Disk Instance Blocking Factor is 1</td>
<td>103</td>
</tr>
<tr>
<td>5.5</td>
<td>Effect on Task Utilisation factor by Instance Blocking Factor</td>
<td>104</td>
</tr>
<tr>
<td>5.6</td>
<td>Execution Map when Disk Instance Blocking Factor is 4</td>
<td>105</td>
</tr>
<tr>
<td>5.7</td>
<td>Task Execution Map at Stage 2 with No Release Time Control</td>
<td>109</td>
</tr>
<tr>
<td>5.8</td>
<td>Task Execution Map at Stage 2 with Modified Release Time</td>
<td>110</td>
</tr>
<tr>
<td>5.9</td>
<td>Instance Blocking Buffer for Example 5.1</td>
<td>119</td>
</tr>
<tr>
<td>5.10</td>
<td>Instance Blocking Buffer for Example 5.2</td>
<td>120</td>
</tr>
<tr>
<td>5.11</td>
<td>Minimum Required Number of Buffers</td>
<td>123</td>
</tr>
<tr>
<td>6.1</td>
<td>A Pipelined Multi-Server System</td>
<td>129</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Task Parameters of Example 3.1 .................................................................60
3.2 System Performance Indices of Example 3.1 .................................................60
3.3 Task Parameters of Example 3.2 .................................................................61
3.4 System Performance Indices of Example 3.2 .................................................62
3.5 Response Time for Each Instance of Task 2 ..................................................62
3.6 Task Parameters for Non-preemptive Fixed Priority Scheduling ....................66
3.7 Task Parameters for Preemptive Deadline Driven Scheduling .........................67
3.8 Task Parameters for Server Capacity Estimation ............................................70
3.9 Task Parameters for Excessive Workload .....................................................71
3.10 Updated Task Parameters ..........................................................71
4.1 Task Input of Example 4.1 ..............................................................................86
4.2 Task Performance of Example 4.1 .................................................................87
4.3 System Performance of Example 4.1 ............................................................88
4.4 Task Input of Example 4.2 ..............................................................................89
4.5 Task Performance of Example 4.2 .................................................................90
4.6 System Performance of Example 4.2 ............................................................90
4.7 Task Input of Example 4.3 ..............................................................................91
4.8 Task Performance of Example 4.3 in a Uniprocessor System .........................92
4.9 Task Performance of Example 4.3 in a Multiprocessor System .......................92
4.10 System Performance of Example 4.3 ............................................................93
5.1 Task Input Parameters for a 2-stage pipeline system .....................................108
5.2 Task Response Time with No Release Time Control ......................................109
5.3 Task Response Time with Modified Release Time ........................................111
6.1 A Given Set $TS$ of Tasks ...........................................................................138
6.2 A Set of Stages in the Given System ............................................................139
6.3 A Modified Set $TS'$ of Tasks ....................................................................139
6.4 Response Times of Modified Tasks ...............................................................140
6.5 Response Times of MRF Result .................................................................141
6.6 Response Times of BSE Result ....................................................................142
6.7 Minimum Required Hardware Configuration .................................................142
Chapter 1
Introduction

1.1 Background

Real-time systems are time-critical systems in which tasks have timing deadlines as well as their functional requirements (Krishna and Lee, 1994). Parallel processing architectures can help real-time systems to meet these timing requirements more effectively than single processing architectures can. To design and evaluate such systems, a performance model which can estimate the performance of parallel real-time systems is necessary.

There are two requirements for a real-time system to achieve its desired performance goal. The first is to provide a flow balance between the request rate of each task and its completion-rate (or throughput) during the peak-time interval: the two rates have to be the same. The second is to guarantee that the response times for all tasks must be less than or equal to their deadlines. If the system fails to meet the first requirement, a throughput failure occurs and the system overflows (or overloads), and some request(s) may be refused, which may result in the loss of the associated input data. Even if the first requirement is met, there is still a chance that the second requirement cannot be met. If the system fails to meet the second requirement, a response failure occurs. A throughput failure implies a response failure because the refused request will have an indefinite response time. A real-time system has a guaranteed capacity if it has no throughput failure and no response failure.

Given such time critical requirements, it is inevitable that real-time systems will exploit high-speed parallel processing technology. Single processing real-time systems have limitations in their processing power while parallel processing real-
time systems can schedule any set of tasks. Parallelism can be achieved by two methods of resource/task partitioning: *multi-staging* and *multi-server*. Van Zandt (1992) referred to multi-staging as *function partitioning* and multi-server as *data partitioning*. Multiple stages can be set up so that each stage executes an independent function from the other stages and can execute it simultaneously with the other stages. Mostly, a task which arrives at the system passes through a series of stages then leaves the system. The multi-stage approach exploits *pipeline parallelism* in which tasks are partitioned into subtasks for their dedicated stages (Park *et al.*, 1994). Multiple servers at a stage (e.g., disk arrays) enable instances of a task to execute in that stage simultaneously (e.g., disk stripping) (Reddy and Banerjee, 1989; Weikum *et al.*, 1991; Ganger *et al.*, 1994). Multiple instances of the same tasks can be dispatched to multiple servers to handle rapidly arriving periodic input data. Different tasks may be distributed to different servers as well. A *multi-stage multi-server system* (Lee and Aggarwal, 1990) combines these two concepts into a general parallel system. This combined approach helps to improve the throughput and the responsiveness of each task by shortening the queues of tasks waiting for access to a server.

Although parallel processing improves system performance, fast processing alone does not guarantee *predictability* (Stankovic, 1988). The objective of real-time systems is to meet the task deadlines. To meet the goal, the most important property of a real-time system should be predictability rather than just being fast. Therefore, we need a model for parallel real-time systems capable of predicting the behaviour of systems.

### 1.2 Problem Analysis

Performance models for parallel systems must calculate a set of performance metrics that characterise such systems. The metrics are determined from the
performance problems that occur in parallel real-time systems. Six problems must be addressed in performance modelling of parallel real-time systems.

(1) **Schedulability test**
The model must determine if the underlying system can schedule all tasks within their deadlines without losing any input during the system peak-time interval.

(2) **No input loss condition**
The model must determine if the underlying system can process all input requests during the system peak-time interval without losing any of them.

(3) **Buffer space**
The model must calculate the minimum buffer space required by each task on each server at each stage for the whole system to store the input data so that no loss occurs.

(4) **End-to-end system response time**
The model must calculate the end-to-end system response times (hereafter, end-to-end response times) for each task.

(5) **Optimal hardware configuration**
The model must determine the optimal hardware configuration which can schedule the given set of tasks, minimising the system cost provided that the resource types and the scheduling algorithms at all stages are pre-assumed. The optimal hardware configuration (OHC) can be characterised by \( \text{OHC} = \{ (j, N_j) \mid 1 \leq j \leq m \} \) where \( N_j \) is the minimum required number of servers at stage \( j \).
(6) **Optimal scheduling configuration**

The model must determine the optimal scheduling algorithm for each stage which can schedule the given set of tasks, minimising the system cost given a known set of hardware resource types and the scheduling algorithms at each stage. The optimal scheduling configuration (OSC) can be characterised by $\text{OSC} = \{ (j, \text{sch}_j) \mid 1 \leq j \leq m \}$ where $\text{sch}_j$ is the scheduling algorithm at stage $j$.

### 1.3 Goal of Thesis

The **focus of the thesis** is to develop a performance model for multi-stage multi-server real-time systems. The model determines the minimum required capacity of the system to guarantee all task deadlines. The model estimates the optimal server capacity and buffer space. It determines the maximum schedulable execution time for each task. It also estimates the worst and average-case response times of each task during the peak-time interval and determines the schedulability of a task set in a given system. The model analyses two real-time scheduling algorithms: fixed priority scheduling and deadline driven scheduling. Each scheduling algorithm deals with preemptive and non-preemptive resources.

It has been found that the schedulability test or the estimation of server capacity cannot be done only with a user specified deadline. It has to consider the machine restraint factors such as server overflow deadline and buffer overflow deadline as well. An algorithm for finding the optimal server capacity is proposed. It is based on the deadline balance theory which is introduced in this thesis. It can be applicable to any arbitrary set of tasks. The spare execution time for each task is calculated when a server has spare capacity. Otherwise the minimum required (i.e., optimal) server capacity is used. If the required single server capacity is not available, we need multiple servers where each server executes the workload of a
subset (i.e., a group of instances) of tasks. An algorithm to find the optimal number of such servers is presented.

The schedulability test for a pipelined multi-stage system is a more complex task. It has been found that a regular pipelined multi-stage model is the way to achieve a mathematically tractable model for finding performance characteristics. Every stage has regular task arrival times by deferring the release time of each instance of tasks at each stage. As the system capacity is composed of the capacity of each stage, the spare capacity for a server at a stage can be estimated on the assumption that the capacities of the servers at the other stages are constant.

In this thesis, I have presented theorems developed by others as a theoretical basis for my work (Theorems 2.2, 2.5-7). In addition, I have developed new theorems for multi-server and pipeline systems (Theorems 2.1, 2.3-4 and 2.8). Then starting from these theorems, I developed algorithms to model performance.

I chose to develop mathematically tractable models. As a result, these models deal with the worst-case scenarios for performance. Hence, they are theoretical models rather than practical models. I took this approach because I felt that developing tractable models was a necessary first stage in the modelling of such systems. As they give the bounds on performance parameters, they provide the limits for more detailed practical models (which often are not tractable).

1.4 Thesis Organisation

The rest of the thesis is organised as follows. First, I introduce the performance indices which my models aim to produce (Chapter 2). Second, I propose a single server model which estimates the task response times and the optimal server capacity and buffer space for a single server system (Chapter 3). Third, I extend the single server model to two parallel paradigms: a multi-server model where
multiple identical servers are processing in data parallelism (Chapter 4), and a pipeline model where a single server exists at each stage in a pipelined multi-stage system (Chapter 5). Fourth, I develop a pipelined multi-server model which combines these two parallel paradigms (Chapter 6). Finally, I conclude my thesis with a discussion of the models (Chapter 7). The notations and terminologies used throughout the thesis are summarised in Appendix A and Appendix B respectively. Detailed calculations of examples are put in Appendix C to avoid clutter. The source codes of the simulation programs for the validation of the models are included in Appendix D.
Chapter 2
Performance Indices

Performance indices represent the performance of a system. They are the output matrices from a performance model when we input the parameters of a workload and a system to model. In this chapter, the parameters of the workload and the system are defined. Then the standard performance indices are extended to parallel real-time systems domain.

![Diagram of a Performance Model]

**Figure 2.1**
A Performance Model
2.1 Workload and System Parameters

The input and output of our performance model is as shown in Figure 2.1. The workload is a set $TS = \{\tau_1, \ldots, \tau_n\}$ of tasks. The system is a set $RS = \{\text{Res}_1, \ldots, \text{Res}_M\}$ of resources. A resource consists of a server and its associated buffers for the queuing of subsequent requests (Figure 3.5). Each server can have a scheduling policy chosen from a set $SS = \{\text{Sch}_1, \ldots, \text{Sch}_L\}$ of scheduling algorithms. A task $\tau_i$ (hereafter, task $i$) may use a subset $rs \in RS$ of resources. It may process through a sequence of pipelined stages and may use a set of parallel servers at each stage. A resource $\text{Res}_x \in rs$ may be used as the $k^{th}$ server of stage $j$ with its associated buffers.

![Figure 2.2](image)

**Figure 2.2**

Hardware Configuration

We assume a $m$-stage parallel real-time system (Figure 2.2) with stage $j$ ($1 \leq j \leq m$) having $N_j$ identical servers. The $k^{th}$ server at stage $j$ has $B_{i,j,k}^{\text{install}}$ buffers for task $i$. Task $i$ ($1 \leq i \leq n$) is characterised by $\tau_i = (T_i, C_{i,j}, D_i, HS_i)$. The
symbols are defined below and in Appendix A. Task $i$ processes an infinite series of input data from a source. A continuous stream of the input arrives at the system synchronously or asynchronously. The period (or minimum interarrival time) between two subsequent arrivals of input is $T_i$. An instance $I_{i,q}$ processes the $q^{th}$ request (or instance) of task $i$. Task $i$ requests at most $C_{i,j}$ execution time on an identical server at stage $j$. $HS_i$ is a flag indicating the class of task $i$, that is, whether it is a hard or soft real-time task. When $HS_i = hard$, the task has a hard deadline requirement and when $HS_i = soft$, it has a soft deadline requirement. A hard real-time task $i$ (i.e., $HS_i = hard$) must complete processing within its deadline $D_i$ for every individual instance. A soft real-time task $i$ (i.e., $HS_i = soft$) need not do that. However its average response time during a peak-time interval must be less than or equal to its deadline $D_i$. The deadline $D_i$ is user specified end-to-end system deadline and measured relative to the time of the request. The peak-time interval $P = LCM (T_1, T_2, ... , T_n)$ where LCM is the least common multiple.

**Figure 2.3**

Software Configuration
A task allocator assigns instances of tasks to servers statically at compile time according to a data parallelism algorithm and each server has its own local scheduler where the servers at the same stage have the same scheduling algorithm (Figure 2.3).

The output of the model (Figure 2.1) is the performance of the system for a given workload. The characteristics of these seven performance indices are described in the following sections.

### 2.2 Schedulability

A workload, that is, a set of tasks is *schedulable* in a given system when it has no throughput failure and no response failure. Algorithm 2.1 is a *schedulability test* algorithm for a set of tasks in a system.

Note that the throughput failure occurs either from server overflow or buffer overflow. Server overflow implies buffer overflow. Some lower priority tasks in an overflowed server will have an indefinite response time, requiring an indefinite amount of buffer space for the subsequent arrivals of its input data. The finite amount of system buffer will eventually overflow and the associated input data will be lost. For these two overflow cases, it makes no sense to test the task deadlines because there is some input loss. This fundamental schedulability test model will be a framework for our further performance models.

### 2.3 Throughput

The throughput of a task is the completion rate (or output rate) of the task in the
Algorithm 2.1: Schedulability Test

{ tests if a given set of tasks is schedulable in a system }

1. test server overflow.
   
   if no server overflows in the system then
   
   test buffer overflow.
   
   if no buffer overflows in the system then
   
   test deadline miss.
   
   if no task fails to meet its deadline then
   
   the given set of tasks is schedulable.
   
   the underlying system has a guaranteed capacity.
   
   else /* deadline miss */
   
   response failure occurs.
   
   the given set of tasks is not schedulable.
   
   endif
   
   else /* buffer overflow */
   
   throughput failure occurs.
   
   the given set of tasks is not schedulable.
   
   endif
   
   else /* server overflow */
   
   throughput failure occurs.
   
   the given set of tasks is not schedulable.
   
   endif

2. stop.
system. The throughput \((X_i)\) of a task \(i\) is the same as its input or arrival rate \((1/T_i)\):

\[
X_i = 1/T_i
\]  

(2.1)

\(i.e., \) flow balanced \(\) when the system has no throughput failure. However, if the system has throughput failure, then flow balance is broken and some input may be lost. Thus, the throughput of some task will be less than its arrival rate:

\[
X_i < 1/T_i
\]  

(2.2)

when the system has throughput failure. It is impossible that \(X_i > 1/T_i\) because there cannot be an output which does not come from input. Therefore, for a multi-stage system to have flow balance, it is required that all stages must have flow balance, hence the throughputs of all stages must have the same value of \(1/T_i\).

**Theorem 2.1** If the goals of a multi-stage system are to give periodicity of release times of task \(i\) to stage \(j\) with the minimum release delay without missing any of the instances of task \(i\) from stage \(j-1\) and the following restrictions apply:

1. blocks of consecutive \(b_{i,j}\) (\(\geq 1\)) instances of task \(i\) has to be released at the same time at stage \(j\), and

2. \(b_{i,j} = \lambda b_{i,(j-1)}, \lambda = 1,2,...\) or \(b_{i,(j-1)} = \lambda b_{i,j}, \lambda' = 1,2,...\)

then these goals are achieved when the blocks are released at stage \(j\) on the worst-case completion time \((s_{i,j})\) of the \(b_{i,j}^{th}\) instance of task \(i\) at stage \(j-1\) for the first block and at the times of \(s_{i,j} + (k-1)b_{i,j} T_i\) for the \(k^{th}\) \((k = 2,3,...)\) blocks.

**Proof:**

1. minimum release delay: It is possible that the last instance in the block of task \(i\) has the worst-case response time \((W_{i,j})\) at stage \(j-1\). If the block is released before the worst-case completion time of the last instance, the instance cannot be processed
Chapter 2: Performance Indices

at stage $j$. If the block is released after the completion time of the last instance, it has unnecessary delay time. If the block is released on the completion time of the last instance, the last instance has no delay time and all preceding instances are also ready to be released because they were completed before the last instance. Therefore, it is proved that the goal of the minimum release delay is achieved.

(2) periodicity: Let $r_{i,k,j}$ be the release time for the instances of task $i$ in the $k^{th} (k = 1,2,3,...)$ block at stage $j$. Then the period $(T_{i,j})$ of the blocks of task $i$ at stage $j$ can be derived as follows:

**case 1. when** $b_{i,j} = \lambda b_{i,(j-1)}, \lambda = 1,2,...$

$$r_{i,k,j} = s_{i,j} + (k-1)b_{i,j}T_i$$

$$= r_{i,1,(j-1)} + (b_{i,j} - b_{i,(j-1)})T_i + W_{i,(j-1)} + (k-1)b_{i,j}T_i$$

$$= r_{i,1,(j-1)} + (kb_{i,j} - b_{i,(j-1)})T_i + W_{i,(j-1)}$$

$$r_{i,(k+1),j} = r_{i,1,(j-1)} + ((k+1)b_{i,j} - b_{i,(j-1)})T_i + W_{i,j-1}.$$  

$$\therefore T_{i,j} = r_{i,(k+1),j} - r_{i,k,j} = b_{i,j}T_i.$$  

**case 2. when** $b_{i,(j-1)} = \lambda'b_{i,j}, \lambda' = 1,2,...$

$$r_{i,k,j} = s_{i,j} + (k-1)b_{i,j}T_i$$

$$= r_{i,1,(j-1)} + W_{i,(j-1)} + (k-1)b_{i,j}T_i.$$  

$$r_{i,(k+1),j} = r_{i,1,(j-1)} + W_{i,(j-1)} + kb_{i,j}T_i.$$  

$$\therefore T_{i,j} = r_{i,(k+1),j} - r_{i,k,j} = b_{i,j}T_i.$$  

Therefore, it is proved that the goal of periodicity is achieved.

Note that the throughput $(X_{i,j})$ of task $i$ at any stage $j$ is the same achieving flow balance. This can be proved using the scheme in Theorem 2.1 because $X_{i,j} = b_{i,j}/b_{i,j}T_i = 1/T_i, 1 \leq i \leq n, 1 \leq j \leq m.$
2.4 Utilisation

To be able to test for server overflow, we need to calculate the utilisation factor of each server. The *utilisation factor* \( U_s \) is the proportion of the time the server \( s \) is occupied for processing tasks during the peak-time interval. The throughput \( X_{i,s} \) and execution time \( C_{i,s} \) of each task \( i \) at server \( s \) affects the utilisation factor of the server \( U_s \):

\[
U_s = \sum_{i=1}^{n} X_{i,s} C_{i,s}
\]

(Denning and Buzen, 1978). \hspace{1cm} (2.3)

\[
U_s = \frac{\sum_{i=1}^{n} (C_{i,s} / T_{i,s})}{U_s} \text{ when the system is flow balanced, since } X_{i,s} = 1/T_{i,s} \text{ (Equation (2.1)).}
\]

The utilisation factor of a server must be less than or equal to unity. Otherwise, some input may be lost as shown in Equation (2.2). Hence, Algorithm 2.2 gives us the server overflow condition.

---

**Algorithm 2.2: Server Overflow Test**

1. calculate the utilisation factor \( U_s = \sum_{i=1}^{n} (C_{i,s} / T_{i,s}) \).

2. if \( \exists U_s > 1 \) then

   server overflow occurs.

   else

   server overflow does not occur.

   endif

3. stop.
The utilisation factor of a server needs to be maximised for the economical use of the server. However, it must not overflow.

**Theorem 2.2** If the utilisation factor $U_s \leq 1$, then all instances of tasks have finite response times.

**Proof:** See Tindell (1992) and Joseph and Pandya (1986).

### 2.5 Priority

The priority assignment algorithm is important for a fixed priority scheduling policy because it affects the response time of each task and the maximum schedulable utilisation of the server. It gives as much critical effect as scheduling algorithm itself does on making a given set of tasks schedulable. Audsley (1991) proposed a schedulable priority assignment algorithm. His algorithm may be applicable to a single or a multi-server system. However, a pipeline or a pipelined multi-server system requires the highest schedulable priority at each stage for an inserted task to meet its end-to-end deadline. In this thesis, I present a best-priority-assignment (BPA) algorithm for a single or multi-server system (Algorithm 3.6) based on Theorem 2.3, and a minimum-response-priority-assignment (MRPA) algorithm (Algorithm 5.2) for a pipeline or pipelined multi-server system based on Theorem 2.4. These are extended versions of Audsley's algorithm.

**Theorem 2.3** If each stage has a single server and an inserted task $i$ is assigned the best (highest possible) priority at each stage while guaranteeing the deadlines of all the tasks other than task $i$, and the worst-case (end-to-end) response time $W_i$ exceeds the (end-to-end) deadline $D_i$, then the underlying system does not have a guaranteed capacity.
**Proof:** $W_i$ is the sum of all the worst-case stage response times of task $i$. The worst-case stage response for task $i$ is the sum of its release delay time and its worst-case response time on the server at the stage. The best priority at each stage gives the minimum worst-case response time on the server at the stage for task $i$. The release delay can be considered to be constant. Therefore, it is proved that if the minimum worst case response does not meet the deadline then the system does not have a guaranteed capacity. □

**Theorem 2.4** If each stage has multiple servers and an inserted task $i$ is allocated to the server $k_{min}$ ($1 \leq k_{min} \leq N_j$) at each stage $j$ which gives the task the minimum worst-case stage response time among $N_j$ servers while guaranteeing the deadlines of all the tasks other than task $i$, and the worst-case (end-to-end) response time $W_i$ exceeds the (end-to-end) deadline $D_i$, then the underlying system does not have a guaranteed capacity.

**Proof:** $W_i$ is the sum of all the worst-case stage response times of task $i$. The worst-case stage response for task $i$ is the sum of its release delay time and its worst-case response time on the server at the stage. The minimum-response server allocation gives the minimum worst-case response time at the stage for task $i$. The release delay can be considered to be constant. Therefore, it is proved that if the minimum worst case response does not meet the deadline then the system does not have a guaranteed capacity. □

### 2.6 Response Time

Response Time plays a key role in performance evaluation. The worst-case response time is needed for a hard real-time task to compare with its deadline. It is also needed to estimate the required buffer space for the subsequent instances of the
task so that the buffer does not overflow while the task suffers the worst-case response time which is larger than the task period. The *average response time* during the peak-time interval is needed for a soft real-time task to compare with its deadline.

### 2.6.1 Worst-Case Response Time

We need to know the worst-case response time \(W_i\) for hard real-time task \(i\) to determine if it meets its deadline \(D_i\).

\[
\begin{align*}
\text{If } (W_i \leq D_i) \text{ then} \\
\text{response requirement is met.} \\
\text{else} \\
\text{response failure occurs.}
\end{align*}
\]

**Definition 2.1** A *critical instant* for hard real-time task \(i\) is the instant when the task arrives at the system simultaneously with the arrivals of all higher-priority tasks.

**Theorem 2.5** If \(W_i \leq T_i\), then the worst-case response time \(W_i\) for hard real-time task \(i\) occurs at its critical instant.

**Proof:** See Liu and Layland (1973).  

**Definition 2.2** A *level-i busy period* is a time interval \((a, b)\) within which instances of task \(i\) or higher-priority tasks are processed but no instances of task \(i\) or higher-priority tasks are processed in \((a-\varepsilon, a)\) or \((b, b+\varepsilon)\) for sufficiently small \(\varepsilon > 0\).
Theorem 2.6 If \( W_i > T_i \), then the worst-case response time \( (W_i) \) for hard real-time task \( i \) occurs during a level-i busy period which is initiated by a critical instant.  

The worst-case response time \( (W_i) \) for hard or soft real-time task \( i \) is used to calculate its required number of buffers \( (B_i) \) for no input data loss, which will be explained in Sections 2.7 and 3.7.

2.6.2 Average Response Time

For a soft real-time task, we need to know the response time for each instance of a task during a peak-time interval. We then calculate the average response time \( (A_i) \) during that period to determine if it meets its deadline \( (D_i) \) requirement.

If \( (A_i \leq D_i) \) then  
response requirement is met  
else  
response failure occurs.
endif.

2.7 Buffer Requirement

When an input occurs, the input data is stored in a buffer. The buffer is allocated until its associated task completes processing the data. If an input arrives while the previous input is still being processed, the input needs another buffer. The minimum required number of buffers for a task \( i \) on a server \( s \) can be determined by its worst-case buffer holding time \( (H_{i,s}) \) and its input period \( (T_{i,s}) \):
\[ B_{i,s} = \lceil H_{i,s}/T_{i,s} \rceil \] (2.4)

Hence, Algorithm 2.3 determines if buffer overflow occurs.

Algorithm 2.3: Buffer Overflow Test

1. calculate the minimum required number of buffers for task \( i \) on server \( s \):
   \[ B_{i,s} = \lceil H_{i,s}/T_{i,s} \rceil. \]

2. let the installed number of buffers for task \( i \) on server \( s \) be \( B_{i,s}^{\text{install}} \).

3. if \( B_{i,s}^{\text{install}} < B_{i,s} \) then
   buffer overflow occurs.
   else
   buffer overflow does not occur.
   endif

4. stop.

Theorem 2.7 If the utilisation factor \( U_s \leq 1 \), then no input is lost when each task has a finite number of buffers.


2.8 Server Capacity

We may know if the underlying server has a guaranteed capacity or not by using schedulability test (Algorithm 2.1). However, a guaranteed capacity may be an overcapacity. If the system cannot schedule all tasks, it has an undercapacity. We need to estimate the optimal server capacity which is the minimum schedulable
capacity for a given set of tasks. We also need to calculate the *spare execution times for tasks* when the system has a guaranteed capacity.

If we can calculate the minimum schedulable common scaling factor $f_s$ (see Algorithm 3.7) for execution times of tasks on server $s$, the optimal server capacity will be $1/f_s$ times the underlying capacity. We can find the optimal capacity in two ways: a single server capacity (Chapter 3) or a multi-server capacity (Chapter 4). Algorithm 2.4 presents the overall procedure for estimation of single server capacity.

---

**Algorithm 2.4: Single Server Capacity**

1. calculate the minimum schedulable common scaling factor $f_s$ for execution times of tasks on server $s$.
2. the optimal server capacity is $1/f_s$ times the underlying capacity.
3. if $f_s < 1$ then
   - the underlying server has no guaranteed capacity.
   
   else
   - the underlying server has a guaranteed capacity.
   
   calculate the spare execution time ($\Delta C_i$) for each task $i$.

   endif

4. stop.

If the optimal single server capacity calculated in Algorithm 2.4 is not available then multiple identical servers which run concurrently are required. More extensive discussion which includes the development of the modified-task-first-fit-allocation (MTFFA) algorithm (Algorithm 4.1) when instance distribution is performed will be dealt with in Chapter 4. The MTFFA algorithm is developed from Theorem 2.8 which states the condition for scheduling tasks with multiple servers.
Theorem 2.8 If \( C_i \leq D_i, 1 \leq i \leq n \) in a single stage system, then a finite number of servers \( N \leq \sum_{i=1}^{n} \frac{P_i}{T_i} \) can schedule all \( n \) tasks regardless of the relation between \( C_i \) and \( T_i \).

Proof: In the worst-case situation where any server cannot share more than one instance of one task to meet the task deadlines, each instance of each task can be allocated to a dedicated (i.e., empty) server. The required number of servers in this case is \( N = \sum_{i=1}^{n} \frac{P_i}{T_i} \) and it is the upper limit. Then it is obvious that \( W_i = C_i, 1 \leq i \leq n \) and it follows that \( W_i \leq D_i, 1 \leq i \leq n \). Therefore, it is proved that all instances of all tasks can be scheduled. \( \blacksquare \)
Chapter 3
A Single Server Model

In this chapter, we propose a general single server performance model for real-time data processing systems. This model can apply to any resource (e.g., a CPU, a disk, or a communication network etc.). We developed this model by modifying existing single server models to be extendable to parallel paradigms, which we propose in later chapters. This model can assess the performance of systems with fixed or deadline driven preemptive and non-preemptive priority scheduling algorithms. Also the model can handle arbitrary task deadlines.

First, we propose an algorithm for the schedulability test. Then, we derive a general algorithm which estimates the worst-case and average response times during the peak-time interval for both hard and soft real-time tasks. For tasks of fixed priority scheduling, we use the best-priority-assignment algorithm. Also, we derive an equation to calculate the minimum amount of buffer space required for the server to guarantee that any real-time task does not lose its input data. Finally, we explain how to estimate the optimal server capacity for a set of given tasks. In addition, we estimate the maximum schedulable execution time for each task.

3.1 Introduction

The designers of real-time systems have to guarantee that they have enough server capacity to complete every task within its deadline. There are two classes of tasks in real-time systems: hard real-time and soft real-time (Civera et. al, 1983). Each instance of a hard real-time task must complete within its specified deadline: its worst-case response time must not exceed its deadline. In contrast, all instances
of a soft real-time task need not complete within its deadline but its average response time during its peak-time interval must be equal to or less than its deadline. These definitions guarantee that no real-time task loses input data.

Real-time data processing systems are computer systems which collect periodically and sporadically occurring real-world data from external devices, process them and store them in data structures in a time-critical manner. Data input is handled by hard real-time tasks. Some systems set off alarms on alarm panels and others send control commands to actuators as soon as they receive important data from input devices. A response failure in a hard real-time task may cause a disaster.

The data accumulated in the data structures may be saved on disk and displayed at the request of an operator. These data storage and retrieval tasks are not as time-critical, so they are soft real-time tasks with average response time requirements. In case of failure to meet the requirements, a disaster will not occur but operators may be inconvenienced either by information latency from the storage delay or by slow response from long retrieval time.

To determine if the underlying server can handle all the real-time tasks within their deadlines without losing any input data, we need a performance model. In this chapter, we develop a single server performance model. In later chapters we will expand this model to more general system architectures.

Many performance models (Allen, 1978 and 1980) have been devised for general-purpose computer systems. They are probabilistic models which assume a Poisson distribution for task interarrival times and an exponential distribution for task execution times. They cannot be applied to hard real-time systems. Some (Liu and Layland, 1973; Chetto H. and Chetto M., 1989a; Jeffay et al., 1991) define the conditions for schedulability of real-time tasks but do not estimate the exact response times for the tasks. Others (Mahjoub, 1984; Leinbaugh and Yamini, 1986; Stoyenko et al., 1991) studied on the bounds of system response
times for real-time tasks. However, they neither use an exact analysis for task response times nor attempt to estimate a required system capacity.

Based upon Liu and Layland's (1973) critical instant theorem (see Theorem 2.5), Joseph and Pandya (1986) proposed an exact analysis algorithm for real-time systems with preemptive priority scheduling. Their algorithm determines the condition for no input loss, estimates the worst-case response times, and calculates the minimum required number of input buffers. However, their analysis is restricted. First, it is confined to the case where the worst-case response times are less than or equal to the periods. Second, it can only estimate the worst-case response times for hard real-time tasks.

Using Lehoczky's (1990) busy period theorem, Tindel et al. (1992, 1994) proposed a worst-case response time analysis which estimates the worst-case response times for tasks where the worst-case response times exceed the periods. In practice, it is not uncommon to find hard and soft real-time tasks with arbitrary deadlines running together in a real-time system. Thus, many scheduling algorithms for soft real-time tasks have been proposed (Lehoczky et al., 1987; Sprunt et al., 1988b; Sprunt et al., 1989; Lehoczky and Ramos-Thuel, 1992; Davis et al., 1993) to minimise their average response times while meeting the deadlines of all hard real-time tasks in the system. However, they cannot estimate the average response times. Hong et al. (1989) derived the probability of soft real-time tasks missing their deadlines using probabilistic models. They did not use an exact analysis. We propose an exact analysis for soft real-time tasks by estimating a peak-time average response time.

The server that provides service may have a preemptive or non-preemptive scheduler. Thus, our analysis includes both types of schedulers. Tindell and Burns (1993) and Lee and McKerrow (1994) extended the analysis of Joseph and Pandya (1986) to non-preemptive disk drives. Our model generalises these models to apply to any non-preemptive resource.
Chapter 3: A Single Server Model

Liu and Layland (1973) gave an optimal fixed priority scheduling algorithm: the rate-monotonic-scheduling (RMS) for tasks where deadlines are equal to their periods. Their algorithm assigns a higher priority to a task which has a shorter period. Leung and Whitehead (1982) found an optimal fixed priority scheduling algorithm: the deadline-monotonic-scheduling for tasks whose deadlines are less than or equal to their periods. Their algorithm assigns a higher priority to a task which has a shorter deadline. Audsley (1991) proposed an optimal priority assignment algorithm for general fixed priority scheduling where task deadlines are arbitrary. The algorithm tries $(n^2+n)/2$ schedulability tests to find a schedulable priority assignment for a given set of $n$ tasks. However, the assignment does not guarantee that the underlying task has been assigned to have the minimum schedulable response time. We propose a best-priority-assignment algorithm which assigns the underlying task the highest schedulable priority to achieve the minimum schedulable response time. This is a modified version of Audsley's algorithm for better priority ordering of tasks. It is an important assignment policy to be extended to a pipelined multi-stage system.

All the above models were developed for fixed priority scheduling policies. As deadline driven scheduling has been proved to be optimal (Liu and Layland, 1973; Dertouzous, 1974) for a single server, it is an important scheduling policy for real-time systems. Falk (1988) took an interesting example to illustrate that a set of tasks which failed to be scheduled with rate-monotonic scheduling was schedulable with deadline driven scheduling. Some (e.g., Liu and Layland, 1973; Leung and Merill, 1980) have studied preemptive deadline driven scheduling while others (e.g., Johnson and Maddison, 1974; Jeffay et al., 1991) have worked on non-preemptive deadline driven scheduling. However, none seems to have estimated the exact analysis of the response times of tasks for deadline driven scheduling. Our model attempts to do so.

Klein et al. (1993) proposed a task capacity model. It estimates the
schedulable spare execution time for each task and the schedulable scaling factor for the execution times of all tasks in an underlying server of fixed preemptive priority scheduling when it has a guaranteed capacity. We extend their model to the case of non-preemptive and deadline scheduling when the underlying server has not a guaranteed capacity so that it can estimate the optimal (i.e., minimum schedulable) capacity of a server with a set of arbitrary tasks in an arbitrary scheduling algorithm.

In this chapter, we propose a general response time model which estimates not only the worst-case response time but also the average response time during the peak-time interval so that we can test the schedulability when both hard and soft real-time tasks are running concurrently. The model analyses both preemptive and non-preemptive fixed priority scheduling and both the preemptive and non-preemptive deadline driven scheduling for a single server. The assumptions of the model are set out in Section 3.2. Based upon these assumptions, a set of mathematical models for the performance evaluation of a single server will be developed. The performance indices calculated by the models include schedulability (Section 3.3); utilisation factor (Section 3.4); response time (Section 3.5); priority assignment (Section 3.6); buffer space (Section 3.7); and server capacity (Section 3.8). Examples are given to explain how these models work and simulation results are shown to verify the models (Section 3.9). Finally, extensions of these models to parallel systems (Section 3.10) will be discussed. A major part of this work are our extensions of the model of Joseph and Pandya (1986). We have extended their model to handle

1. worst-case response times longer than periods.
2. non-preemptive scheduling.
3. individual task response times.
4. soft real-time tasks.
3.2 System Model

The performance of a single server (Figure 3.1) is to be analysed in the following sections. The system model includes the following.

1. The system is a single server with its associated buffers for tasks.

2. The scheduler can have one of four scheduling policies: preemptive fixed priority, non-preemptive fixed priority, preemptive deadline driven, and non-preemptive deadline driven scheduling. For fixed priority scheduling, the priority of a task cannot change and a higher-priority task can preempt a lower-priority task at any time (except when the lower-priority task is executing a non-preemptable part). When two instances of tasks have the same priority, their priorities are determined on the FIFO (First-Come-First-Served) basis. For deadline driven scheduling, the priority of an instance of a task changes dynamically. The instance closest to its deadline has the higher priority. When two instances of tasks have the same deadline, their priorities are assigned randomly.

3. The total number of tasks which compete for the single server is $n$ where $n \geq 2$.

4. The system includes $n_h$ hard real-time tasks and $(n - n_h)$ soft real-time tasks where $0 \leq n_h \leq n$.

5. A task $i$ arrives at the server repeatedly with a period (or a minimum interarrival time if the task is aperiodic) of $T_i$, with each instance requesting at most $C_i (\leq D_i)$ execution time. Each task has to complete within a deadline $D_i$. The deadline $D_i$ is an upper-bound on the worst-case response time $W_i$ for a hard real-time task $i$ (i.e., $HS_i = hard$). For a soft real time task, $D_i$ is an
upper-bound on the peak-time average response time $A_i$ (i.e., $HS_i = soft$). All of these timing parameters have positive integer values.

(6) The priority of a task $i$ is $i$ (the highest priority is 1 and the lowest priority is $n$) if the server uses fixed priority scheduling.

(7) When a deadline scheduler is used, the deadlines $D_i < D_j$, $1 \leq i < j \leq n$.

Figure 3.1
A Single Server System

3.3 Schedulability Test

A set of tasks is schedulable in a given server when it has no throughput failure and no response failure. Algorithm 3.1 presents the schedulability test for a given set of tasks on a single server.

The calculation of each of the performance indices used in this algorithm, are presented in the following sections either as equations or as algorithms.
Chapter 3: A Single Server Model

Algorithm 3.1: Schedulability Test for a Single Server System

1. calculate the utilisation factor \( (U_s) \) for the server.

2. \( \textbf{if} \ (U_s > 1) \ \textbf{then} \) /* server overflow */
   
   throughput failure occurs and
   
   some task(s) is(are) not schedulable.

   \( \textbf{else} \)

   for each task \( i \) do
   
   calculate the worst-case response time \( (W_i) \).

   calculate the peak-time average response time \( (A_i) \).

   \( \textbf{if} \ (B_{\text{install}} < \left\lceil \frac{W_i}{T_i} \right\rceil) \ \textbf{then} \) /* buffer overflow */
   
   throughput failure occurs and
   
   task \( i \) is not schedulable.

   \( \textbf{else} \)

   \( \textbf{if} \ ((\text{task } i \ \text{is hard and } (W_i > D_i)) \ \text{or} \ ((\text{task } i \ \text{is soft and } (A_i > D_i))) \ \textbf{then} \) /* deadline miss */
   
   response failure occurs and
   
   task \( i \) is not schedulable.

   \( \textbf{else} \)

   task \( i \) is schedulable.

   \( \textbf{endif} \)

   \( \textbf{endif} \)

   \( \textbf{endfor} \)

\( \textbf{endif} \)

3. \( \textbf{if} \ (\text{all the tasks in the task set are schedulable}) \ \textbf{then} \)
   
   the task set is schedulable and
   
   the system has a guaranteed capacity.

\( \textbf{else} \)

the task set is not schedulable and

the system does not have a guaranteed capacity.

\( \textbf{endif} \)

4. stop.
3.4 Utilisation Factor

The utilisation factor of a server indicates the workload of the server. It is used to determine if the server has overflowed. When the utilisation factor of a server is greater than 1, some input data will be lost due to the server overflow. In this case, it will be impossible to schedule the given set of tasks on the server using any scheduling algorithm (Dhall and Liu, 1978). But if the utilisation factor of the server is less than or equal to 1, then no input data is lost due to server overflow. However, input data can still be lost if the system has insufficient buffers. We calculate the worst-case response times of the tasks to determine if the system has enough buffer space, and to determine if the tasks meet their deadlines.

The utilisation factor of a task \( U_i \) is the ratio \( \frac{C_i}{T_i} \). It is the part of the workload of the server that is due to task \( i \). The sum of the utilisation factors of all tasks is the utilisation factor of the server (Liu and Layland, 1973):

\[
U_s = \sum_{i=1}^{n} U_i,
\]

i.e.,

\[
U_s = \sum_{i=1}^{n} \frac{C_i}{T_i}. \tag{3.1}
\]

The utilisation factors of tasks can be used to locate the bottleneck task in a server. For example, if the utilisation factors of tasks 1, 2, and 3 are 0.3, 1.2, and 0.1, then the utilisation of the server is 1.6. Thus, we see that the server has overflowed and task 2 is the bottleneck task whose utilisation factor \( U_2 \) has to be reduced. \( U_2 \) can be reduced either by increasing the period \( T_2 \), or by reducing the execution time \( C_2 \).

If some tasks do not complete within their peak-time interval, then these tasks will eventually overrun the server. This implies that the utilisation factor of the server is greater than 1 and the server has excessive workload.
3.5 Response Time

Having determined that input data loss will not occur due to the server overflow, we can proceed to estimate the worst-case response times for all tasks and peak-time average response times for soft-real time tasks. Note that the worst-case response times for soft real-time tasks are also needed to calculate the number of input buffers required by each task.

3.5.1 Joseph and Pandya's Response Time Model

We introduce the concept of the calculation of the worst-case response time by modifying some notations of Joseph and Pandya (1986) which are rather difficult to understand. We note that their model can only be applied to tasks whose worst-case response times are less than or equal to their periods. In the following paragraphs we discuss their model which was applied to a single server with preemptive fixed priority scheduling.

The worst-case response time \( W_i \) of task \( i \) is the sum of the execution times of all higher-priority tasks arriving during the period \( (0,W_i) \) (these preempt task \( i \) ), and its own execution time \( C_i \). This calculation (Equation (3.5)) is formulated using the recursive function, 'Complete' (Equation (3.6)). We use the notation \((t,t')\) to indicate a time interval from \( t \) up to, but not including \( t' \).

The worst-case response time \( W_i \) for task \( i \) is calculated using the following rules.

(R1). The order of task execution is not important, just the number of instances of each task is important.
(R2). Start with the 1st period \((0, C_i)\), which is the worst-case response time for task \(i\) if no higher-priority task preempts it (See Figure 3.2 for rules 2-6). Note that we start from the time zero\((0)\) which is the *critical instant* (common release time) in this analysis.

(R3). At the critical instant (see Definition 2.1), all higher-priority tasks \(j < i\), interrupt task \(i\), increasing the response time for task \(i\) by \(\sum_{j=1}^{i-1} (\text{Inputs}_j (0, C_i) \times C_j)\). \(\text{Inputs}_j (t, t')\) is the number of instances of task \(j\) during the period \((t, t')\). We assume that an instance of a task is scheduled to handle each input. This delay time due to preemption is the 2nd period of time (Figure 3.2). It is the sum of preemption times of the instances of the higher-priority tasks during the 1st period.

\[
t_2 = C_i + \sum_{j=1}^{i-1} (\text{Inputs}_j (0, C_i) \times C_j)
\]

*Figure 3.2*

Response Delay due to Interrupts by Higher-priority Tasks
(R4). The 2nd period is from $C_i$ to $C_i + \sum_{j=1}^{i-1} (\text{Inputs}_j (0, C_i) \times C_j)$.

(R5). Now we must account for interrupts which occur during the 2nd period. This generates the 3rd period.

(R6). Repeat until no interrupt occurs during the current period. If no interrupt occurs during $n^{th}$ period $(t_{n-1}, t_n)$ then $W_i = t_n$ (i.e., the end of the $n^{th}$ period) as shown in Figure 3.2.

(R7). We can use a recursive formulation as follows:

(a) The number of instances of task $j$ during the period $(t, t')$ is:

$$\text{Inputs}_j (t, t') = \lceil t' / T_j \rceil - \lfloor t / T_j \rfloor$$  \hspace{1cm} (3.2)

where $\lceil x \rceil$ is the ceiling function which denotes the smallest integer equal to or greater than $x$.

(b) The total execution time of task $j$ during the period $(t, t')$ is

$$\text{Exec}_j (t, t') = \text{Inputs}_j (t, t') \times C_j$$  \hspace{1cm} (3.3)

i.e., the number of instances $\times$ execution time for task $j$.

(c) The total execution time of all higher-priority tasks, $j < i$, during the period $(t, t')$ is the preemption delay of task $i$ for the period:

$$\text{Preempt}_i (t, t') = \sum_{j=1}^{i-1} \text{Exec}_j (t, t')$$  \hspace{1cm} (3.4)
(d) Thus, the worst-case response time of task $i$, is the sum of the total execution time of all higher-priority tasks in the period $(0,W_i)$ and its own execution time $C_i$.

$$W_i = \text{Preempt}_i (0,W_i) + C_i \quad (3.5)$$

Unfortunately, the Equation (3.5) does not lend itself easily to analytical solution. There is another way of reasoning that leads to a more tractable solution by employing a more elaborate recursion formulation.

(R8) Here, we derive a different recursive formulation. The problem with rule 7(d) is that $W_i$ is in both sides of Equation (3.5). Rules 2-6 can be formulated using Figure 3.2 and rule 7 as follows:

$$t_0 = 0.$$  
$$t_1 = C_1.$$  
$$t_2 = t_1 + \text{Preempt}_i (t_0,t_1).$$  
$$t_3 = t_2 + \text{Preempt}_i (t_1,t_2).$$  
$$t_{n} = t_{n-1} + \text{Preempt}_i (t_{n-1},t_n).$$

Note that the iteration stops when $\text{Preempt}_i (t_{n-1},t_n) = 0$ and that the response time of the task $i$ is $t_n$.

(R9) Using the ideas in rule 8, we can reformulate rule 7(d) with a recursive equation in Algorithm 3.2.
Algorithm 3.2: Worst-Case Response Time with Critical Instant

\{ calculates the worst-case response time \( W_i \) of a task \( i \): valid only when \( W_i \leq T_i \} \)

\[
W_i = \text{Complete}_i (0, C_i) \tag{3.6}
\]

where \( \text{Complete}_i (t, t') = \)

\( t' \) when \( \text{Preempt}_i (t, t') = 0, \) or

\( \text{Complete}_i (t', t' + \text{Preempt}_i (t, t')) \) otherwise

and \( \text{Preempt}_i (t, t') = \sum_{j=1}^{i-1} (\text{Inputs}_j (t, t') \times C_j) \)

and \( \text{Inputs}_j (t, t') = \lceil t' / T_j \rceil - \lceil t / T_j \rceil. \)

The idea is to calculate the execution time of higher-priority tasks during the execution of task \( i \). By starting with a period \((0, C_i)\) equal to the execution time of task \( i \), we calculate the number of times each higher-priority task executes during that period. Then we calculate the additional execution time of those tasks to form the subsequent period. This process is repeated until a period is found where no higher-priority task is requested. At the end of this period, task \( i \) is completed and lower-priority tasks can commence. If the server does not overflow with a utilisation factor less than or equal to 1, the recursion will converge to a period of zero length (see Theorem 2.2). In Algorithm 3.2, the function 'Complete' finds the time when the pending execution (\( i.e., \) next period) is zero. \( \text{Complete}_i (t, t') \) returns the time at which task \( i \) will finish on the assumption that:

Just prior to time \( t \), all commitments to task \( i \) and higher priority tasks require a further \( t' - t \) time units to complete.

\( \text{Complete}_i (0, C_i) \) returns the completion time of task \( i \) at its critical instant (\( i.e., \) at
time 0). It is assumed that task $i$ initially acquires the server at time 0, but is immediately preempted by higher-priority tasks and delayed for the time duration of $\text{Preempt}_i (0, C_i)$. The proof of this equation is given by Joseph and Pandya (1986).

While Joseph and Pandya’s method for evaluating Equation (3.5) is presented by a recursive function (Equation (3.6)), Tindell et al. (1994) used an iterating approach to a solution which leads to the same result:

$$W_{i}^{m+1} = \sum_{j=1}^{i-1} \left( \left\lceil \frac{W_{i}^{m}}{T_j} \right\rceil \times C_j \right) + C_i$$ (3.7)

The iteration $(m \geq 0)$ begin with $W_{i}^{0} = C_i$, and ends when $W_{i}^{m+1} = W_{i}^{m}$. Then, $W_{i} = W_{i}^{m+1}$. Although the iteration approach seems simpler in this case, it becomes harder to understand and express than the recursive approach when we have to handle more complex cases (e.g., Algorithm 3.5). Hence, we use the recursive approach as the basis for the following extensions to Joseph and Pandya’s algorithm.

Now, we extend the Algorithm 3.2 to more general cases by relaxing task characterisations, resource types and scheduling algorithms.

### 3.5.2 Extensions to Joseph and Pandya's Model

If $W_{i} > T_{i}$ then Algorithm 3.2 cannot be used to calculate the worst-response time. In this situation, the subsequent instance of the task $i$ will be delayed not only by the higher-priority tasks but also by the previous instance(s) of the task $i$. Thus, we cannot tell if the 1st instance will have the longest response time or the 2nd or the 3rd etc. Lehoczky (1990) proved that the instance which has the worst-case response time occurs during a level-$i$ busy period (see Definition 2.2) initiated by a critical instant. Based upon the busy periods theorem (see Theorem 2.6), Tindell et
Chapter 3: A Single Server Model

Algorithm 3.3: Worst-Case Response Time with Busy Period

{ calculates the worst-case response time $W_i$ of task $i$: valid with an arbitrary $W_i$. The $q^{th}$ instance of the task $i$ arrives at $S_{i,q}$ and completes at $E_{i,q}$ }

1. $q = 1$ and $S_{i,q} = 0$.
2. $E_{i,q} = \text{Complete}_i(0,C_i)$.
3. While ($E_{i,q} > qT_i$) do /* still in level-i busy period */
   
   $E_{i,q+1} = \text{Complete}_i(E_{i,q}, E_{i,q}+C_i)$.

   $S_{i,q+1} = qT_i$.

   $q = q + 1$.

   endwhile.

4. $W_i = \max_{1 \leq x \leq q} \{ E_{i,x} - S_{i,x} \}$.

Figure 3.3

Illustration of Algorithm 3.3
al. (1994) produced an iterative equation to calculate the worst-case response times when \( W_i > T_i \). Algorithm 3.3 reproduces their algorithm using our notation for consistency with the rest of this thesis. Figure 3.3 depicts the algorithm.

We see in Algorithm 3.3 that the 1st instance of task \( i \) arrives at \( S_{i,1} = 0 \) (step 1) and acquires the server at the same time (step 2). However, the \((q+1)\text{th}\) \((q \geq 1)\) instance of task \( i \) arrives at \( S_{i,q+1} \) and acquires the server at \( E_{i,q} \) which is the completion time of the previous \((i.e., q\text{th})\) instance of task \( i \) (step 3). This means that an instance of a task cannot acquire the server before the previous instance of the same task completes. The busy period is not terminated while the completion time \( E_{i,q} \) of the current \((i.e., q\text{th})\) instance of task \( i \) is greater than the arrival time \((qT_i)\) of the next \((i.e., (q+1)\text{th})\) instance of the task \( i \) (step 3). Algorithm 3.3 is more general than Algorithm 3.2. However, it does not consider the non-preemptive cases.

Now we will generalise it into Algorithm 3.4 to include the non-preemptive case too. Let us decompose the execution time into \( C_i = C_i^{sw} + C_i^{preempt} + C_i^{non-preempt} \). The first part \( C_i^{sw} \) is the worst-case task switching overhead time required for task \( i \) to be initialised. The \( C_i^{sw} \) may be the longest system overhead time \((e.g.,\ disk\ access\ time\ or\ CPU\ context\ switching\ time)\). The higher-priority tasks which arrive during the period when the part \( C_i^{sw} \) is executing contribute to the delay time of task \( i \) even if the preemption cannot be made immediately when the higher-priority task arrives. The \( C_i^{preempt} \) is the preemptable part of the task. The part \( C_i^{preempt} \) is preempted whenever a higher-priority task arrives. The \( C_i^{non-preempt} \) is the non-preemptable part of task \( i \). This part is not preempted when a higher-priority task arrives. If \( (C_i^{preempt} \neq 0) \) and \( (C_i^{non-preempt} \neq 0) \), then the task \( i \) can be called partially non-preemptive. If \( (C_i^{preempt} = 0) \), we call the task fully non-preemptive \((i.e.,\ locking\ the\ whole\ part\ of\ the\ task)\). If \( (C_i^{non-preempt} = 0) \), we call the task preemptive. It is assumed that the non-preemptive part of task \( i \)
Algorithm 3.4: Worst-Case Response Time with Non-Preemption

\{ calculates the worst-case response time \( W_i \) of a task \( i \) with \( C_i = C_i^{sw} + C_i^{preempt} + C_i^{non-preempt} \) and a worst-case blocking time \( C_i^{block} \} \)

1. \( q = 1 \) and \( S_{i,q} = 0. \)
2. \( E_{i,q} = \text{Complete}_i (0, C_i^{block} + C_i - C_i^{non-preempt}) + C_i^{non-preempt}. \)
3. While \( (E_{i,q} > qT_i) \) do
   \( E_{i,q+1} = \text{Complete}_i ((E_{i,q}-C_i^{non-preempt}), E_{i,q}+(C_i-C_i^{non-preempt}))+C_i^{non-preempt}. \)
   \( S_{i,q+1} = qT_i. \)
   \( q = q+1. \)
endwhile.
4. \( W_i = \max_{1 \leq x \leq q} \{ E_{i,x} - S_{i,x} \}. \)

comes last. Therefore, \( C_i^{preempt} \) is followed by \( C_i^{non-preempt} \) for partially non-preemptive task \( i \).

Once a lower-priority task attains access to a non-preemptable server (e.g., a disk drive, a critical section of code etc.), it behaves as if it locks a semaphore guarding access to the server (Tindell and Burns, 1993) and it blocks even higher-priority tasks while accessing the server. This phenomenon is called priority inversion. Sprunt et al. (1988a) discussed logical (e.g., a critical section) and physical priority inversion (e.g., a disk drive). Hence, we have to consider the worst-case blocking time \( C_i^{block} \), the longest non-preemptable execution time (e.g., read/write time for a cluster of blocks in a disk, a scheduling clock interval, a packet transmission time, an execution time for a critical section etc.) of the lower-priority tasks as an initial delay time of a task (Algorithm 3.4, step 2). Using the priority ceiling protocol (Sha et al., 1990) a task can experience priority inversion
for at most $C_i^{block}$. The higher-priority tasks which arrive during the period when the $C_i^{block}$ is executing contribute to the delay time of task $i$ even if the preemption cannot be made immediately when the higher-priority task arrives. Algorithm 3.4 considers these effects. Figure 3.4 illustrates the algorithm.

Note that the amount $C_i^{non-preempt}$ is not included in the second parameter of the recursive function 'Complete' because it cannot be preempted by higher-priority tasks. Also note that the parameter $(E_{i,q} - C_i^{non-preempt})$ in step 3 takes into account the backlog of higher priority tasks during the non-preemptive execution time of task $i$. If $(C_i^{non-preempt} = 0)$ and $(C_i^{block} = 0)$, then Algorithm 3.4 becomes the same as Algorithm 3.3. Therefore, Algorithm 3.4 provides a more general worst-case response time model which can be applied to both preemptive and non-preemptive resources with tasks of arbitrary worst-case response times.

![Diagram](image-url)

**Figure 3.4**

Illustration of Algorithm 3.4
When it comes to finding either the worst-case task response time for deadline driven scheduling or the peak-time average task response times for any type of scheduling, we need a general algorithm to find the individual response time for each instance of a task. We cannot determine which instance of a task will have the longest response time in deadline driven scheduling because the task priority changes dynamically as time passes. Therefore, we must examine all instances for their individual response times to determine the longest one. We do the same thing to find the peak-time average response time for a task.

First of all, we introduce a more generalised concept of the busy period which can be applied to both fixed priority and deadline driven scheduling algorithms. Let \( I_{i,q} \) be the \( q^{\text{th}} \) (\( q \geq 1 \)) instance of task \( i \). The level-\((i,q)\) busy period is a time interval \((a, b)\) within which instances of \( I_{i,q} \) or of equal or higher-priority tasks are processed but no instances of those tasks are processed in the period \((a-\epsilon, a)\) or in the period \((b, b+\epsilon)\) for sufficiently small \( \epsilon > 0 \). In deadline driven scheduling, the higher-priority tasks can be interpreted as the tasks which are closer to their deadlines.

\[ \text{Busy}_{i,q}(a, a+C_Z) \] in Equation (3.8) returns the finishing time \( b \) of the level-\((i,q)\) busy period for either scheduling algorithm when its starting time is \( a \). \( C_Z \) is the execution time for the instance of task \( z \) that arrives at time \( a \) and has a priority equal to or higher than \( I_{i,q} \).

\[
b = \text{Busy}_{i,q}(a, a+C_Z) \\
\text{where} \quad \text{Busy}_{i,q}(t,t') =
\begin{cases} 
  t' & \text{when } \text{Delay}_{i,q}(t,t') = t'-t \text{ or} \\
  \text{Busy}_{i,q}(t, \text{Delay}_{i,q}(t,t')) & \text{otherwise}
\end{cases}
\]

(3.8)
Chapter 3: A Single Server Model

\[ \text{Delay}_{i,q}(t, t') = \begin{cases} \sum_{j=1}^{n} \text{Inputs}_j(t, t') \times C_j & \text{for fixed priority scheduling, or} \\ \sum_{j=1}^{n} \text{Inputs}_{j \in \text{ed}(i, q)}(t, t') \times C_j & \text{for deadline driven scheduling} \end{cases} \]

and

\[ \text{Inputs}_j(t, t') = \left\lceil \frac{t'}{T_j} \right\rceil - \left\lceil \frac{t}{T_j} \right\rceil \]

and

\[ \text{Inputs}_{j \in \text{ed}(i, q)}(t, t') = \min\left\{ \left\lceil \frac{(t'+T_j)}{T_j} \right\rceil , \left\lceil \frac{(q-1)T_i+D_i-D_j+T_j}{T_j} \right\rceil \right\} - \min\left\{ \left\lceil \frac{(t+T_j)}{T_j} \right\rceil , \left\lceil \frac{(q-1)T_i+D_i-D_j+T_j}{T_j} \right\rceil \right\} \]

Remember that \( \lceil x \rceil \) is the ceiling function which denotes the smallest integer equal to or greater than \( x \). For calculating the number of instances of a task in deadline driven scheduling, we define a new function:

\[ [x] = \begin{cases} 0, x \leq 0. \\ x-1, (x>0) \land (x\leq[x]). \\ \lfloor x \rfloor, \text{otherwise.} \end{cases} \] (3.9)

Note, the busy period is delayed recursively by the function 'Delay' until the delay time reaches the instant when no more tasks with the same or higher priority are pending.

Inputs_{j \in \text{ed}(i, q)}(t, t') calculates the number of executions of a task \( j \) during a period \( (t, t') \), where the deadline of the task is equal to or earlier than the deadline of the instance \( I_{i,q} \). The proposed calculation can be proved by the following reasonings. Let an instance \( I_{j,x} \) of task \( j \) satisfy the above condition, then it must satisfy both Equation (3.10) and (3.11), i.e.,

1) its arrival time must be during the period \( (t, t') \):

\[ t \leq (x-1)T_j < t' \] (3.10)
2) its absolute deadline must be equal to or less than that of instance \( I_{i,q} \):

\[
(x-1)T_j + D_j \leq (q-1)T_i + D_i
\]

(3.11)

Inputs \( j \in ed(i,q) \) \((t,t')\) is the number of positive integers \( x \) which satisfy the Equations (3.10) and (3.11). Therefore it is proved.

The peak-time interval \( P \) is the feasibility interval (Chetto, H and Chetto, M., 1989b; Audsley, 1991):

\[
P = \text{LCM} \left( T_1, T_2, \ldots, T_n \right)
\]

(3.12)

where LCM is the least common multiple. We need to estimate the response times only during the peak-time interval. Let us assume that a set of tasks \( TS \) is schedulable by either scheduling algorithm, then the server does exactly the same thing at \( t \geq 0 \) that it does at time \( t + kP \) \((k = 1, 2, \ldots)\). Hence, if the \( TS \) is schedulable in a \( P \) then it is also schedulable in any other \( P \) over an infinite time period.

Now, the response time of instance \( I_{i,q} \) which is the \( q^{th} \) instance of a task \( i \) can be calculated with Algorithm 3.5. In step 1, the algorithm finds the start time \( (a) \) of the current level-\((i,q)\) busy period. It then finds the finishing time of the busy period \( (b) \) using the function 'Busy' (see Equation (3.8)). \( N_{j_{\text{min}}} \) is the minimum of the next arrival times of instances of the same or higher-priority tasks. That is the start time of the next busy period. The subscript \( j_{\text{min}} \) is the value of \( j \) that minimises \( N_j \). It implies the same or higher-priority task number whose next arrival time is the start time of the next busy period. The algorithm repeats until \( b \) passes the arrival time \( (S_{i,q}) \) of instance \( I_{i,q} \). The initial delay \( (S'_{i,q}) \) of instance \( I_{i,q} \) occurs due to the higher-priority tasks which arrive during the period \( (a, S_{i,q}) \). It
Algorithm 3.5: Individual Instance Response Times of a Task

{ calculates the individual instance response times of a task $i$ for fixed priority and deadline driven scheduling }

1. initialise current level-$(i,q)$ busy period's start time $a = 0$ and its finishing time $b = 0$.
   for each arrival $q = 1$ to $P/T_i$ do /* $P$ is the peak-time interval */
   instance $I_{i,q}$'s arrival time $S_{i,q} = (q-1)T_i$.
   repeat /* find the level-$(i,q)$ busy period $(a, b)$ that overlaps with the arrival time $S_{i,q}$ of the instance $I_{i,q}$ */
   find next arrival time $N_j$ of an instance of task $j$ immediately after the current level-$(i,q)$ busy period. i.e., $N_j = \lceil b/T_j \rceil \times T_j$.
   $N_{jmin} = \min_{1 \leq j \leq n} \{N_j\}$ for fixed priority scheduling, or
   $\min_{1 \leq j \leq n} \{N_j, N_j+D_j \leq S_{i,q}+D_i\}$ for deadline driven scheduling.
   $a = N_{jmin} - b = Busy_{i,q} (a, a+C_{jmin})$.
   until $(b > S_{i,q})$.

   instance $I_{i,q}$'s initial delay time $S_{i,q}^* = S_{i,q} + Delay_{i,q} (a, S_{i,q})$.
   instance $I_{i,q}$'s completion time $E_{i,q} = Complete_{i,q} (S_{i,q}, S_{i,q}^*+C_i^{block}+C_i-C_i^{non-preempt}) + C_i^{non-preempt}$ when $q = 1$, or
   $Complete_{i,q} (S_{i,q}, S_{i,q}^*+C_i-C_i^{non-preempt}) + C_i^{non-preempt}$ otherwise.
   instance $I_{i,q}$'s response time $R_{i,q} = E_{i,q} - S_{i,q}$.
   endfor

2. $W_i = \max_{1 \leq q \leq P/T_i} R_{i,q}$.
   $A_i = \text{avg}_{1 \leq q \leq P/T_i} R_{i,q}$.

3. stop.
is calculated using the function 'Delay' (see Equation (3.8)). Then the completion time of instance $I_{i,q}$ is calculated using the function 'Complete' (cf. Algorithm 3.4). The response time $(R_{i,q})$ of instance $I_{i,q}$ is the time interval between its arrival time $(S_{i,q})$ and its completion time $(E_{i,q})$.

In step 2, the worst-case response time $(W_i)$ of task $i$ is the maximum of the response times of all its instances. The peak-time average response time $(A_i)$ of task $i$ is the average of the response times of all its instances. Note that the number of instances of task $i$ during the peak-time interval $P$ is $P/T_i$ (see Equation (3.12)).

Algorithm 3.5 does the most general response time analysis. It is a superset of the Algorithms 3.2-3.4. Note that the algorithm takes into account arbitrary task parameters, non-preemptive resources and deadline driven scheduling.

### 3.6 Best Priority Assignment

The model specifies that the priorities of tasks for fixed priority scheduling are determined with a proposed best-priority-assignment (BPA) algorithm. It assigns an inserted task the highest schedulable priority that will achieve the minimum schedulable response time for the task while guaranteeing all task deadlines (see Algorithm 3.6).

The algorithm successively tries to find tasks that are schedulable at priority $n$ to 1 (step 2). Firstly, it attempts to find a task $\tau_i$ schedulable at priority $n$. Next, priority $n-1$ is considered. If, for any priority $p$ a schedulable task cannot be found (step 2(c)), no schedulable priority assignment exists. Finding if a task is schedulable when it is assigned priority $p$ involves testing the schedulability (step 2(b)) of a maximum of $p$ tasks. For example, when you try to find if a task is schedulable at priority $n$, you need to test the schedulability of a maximum of $n$
Algorithm 3.6: Best-Priority-Assignment (BPA)
{ inserts a task $\tau_n$ into a set $\mathcal{T}_S = \{\tau_1, \ldots, \tau_{n-1}\}$ of $(n-1)$ schedulable tasks ($n \geq 1$) and makes a new set $\mathcal{T}_S' = \{\tau_1, \ldots, \tau_n\}$ of $n$ schedulable tasks ($n \geq 1$) giving it the highest schedulable priority }

1. initialise priority assignment $\psi_i = 0$ for $1 \leq i \leq n$.

2. for $p = n$ to $1$ do
   /* priority $p$ ($n$ is the lowest priority) */
   (a) set unassigned = TRUE.
   (b) for $i = 1$ to $n$ do
      if $\psi_i = 0$ then
         /* task $\tau_i$ has no priority assigned yet */
         assign task $\tau_i$ the priority $p$ temporarily.
         if task $\tau_i$ is schedulable then
            set unassigned = FALSE.
            assign the task $\tau_i$ the priority $p$ permanently.
            $\psi_i = p$.
            exit the for loop.
      endif
   endif
   endfor
   (c) if (unassigned) then
      no schedulable priority assignment exists for $\mathcal{T}_S'$:
      stop.
   endif
   endfor

3. $\mathcal{T}_S'$ has a schedulable priority assignment and
   the highest schedulable priority for task $\tau_i$ is $\psi_i$.

4. stop.
tasks. When you assign a task to priority 1 then only the task is required to be tested for its schedulability. Testing the schedulability of the tasks with lower priority is unnecessary because the order of the higher priority does not affect their schedulability (see Theorem 2 in Audsley, 1991). Thus, the number of schedulability tests $N_{\text{test}}$ required for all priority levels is given by:

$$N_{\text{test}} = \sum_{p=1}^{n} p.$$  \hspace{1cm} (3.13)

Therefore,

$$N_{\text{test}} = \frac{n^2 + n}{2}. \hspace{1cm} (3.14)$$

Note that the worst-case required number of schedulability tests in Algorithm 3.6 is $(n^2 + n)/2$.

### 3.7 Buffer Space

If it is guaranteed that a server does not overflow (see Section 3.4), then a finite number of buffers will ensure that no input data is lost even when the tasks have worst-case response times which are larger than their period (see Theorem 2.7). The buffers are used to hold the input data which cannot be processed immediately after it arrives at the server when a task (of higher or lower priority) seizes the server. For any task, the minimum number of buffers is required for the worst-case situation. Thus, the calculation of the minimum required number of buffers ($B_i$) for any hard or soft real-time task is based on the worst-case response time.
(Wi) and the period (Ti) of that task:

\[ B_i = \lceil W_i / T_i \rceil, \quad 1 \leq i \leq n \]  

(3.15)

This equation reproduces the buffer calculation by Joseph and Pandya (1986). Other people (Lehoczky and Sha, 1986; Tindell and Burns, 1993; Reddy and Wyllie, 1994) analysed the I/O buffer space problem in a similar way. They used the producer-consumer (Gray, 1987; Stallings, 1992) concept. Equation (3.15) will be extended to a pipelined multi-stage system in Section 5.5.4.

Let the size of a buffer (e.g., a stream of a record, a block, a bucket of blocks, or a packet etc. depending on the application) for task i be \( Z_i \) (bytes) then the minimum required buffer space \( M_i \) for task i will be \( B_i Z_i \) (bytes). The server requires \( \sum_{i=1}^{n} B_i Z_i \) (bytes) of buffer space not to lose any input data for any task. Note that allocating a buffer space larger than the minimum required buffer space will cause no problem of either throughput or response failure. However, the extra buffer space will be wasted.

### 3.8 Server Capacity

An algorithm for calculating the optimal capacity of a server is developed. The spare execution time for each task is also calculated. Both of these require the calculation of the upper-bound completion time of an instance of a task. In addition, the potential for adjusting a task's deadline is examined.

#### 3.8.1 Capacity Planning

The maximum schedulable utilisation (Dhall and Liu, 1978; Sprunt et al., 1988a) of
a server is an important issue in real-time systems. Our first aim is to formulate a server capacity model which can estimate the optimal (i.e., minimum schedulable) capacity of a server. Any decrease of the capacity of the server from the optimal capacity will make the given set of tasks unschedulable.

![Diagram of tasks, buffers, and server](image)

**Figure 3.5**

An Analogy

Our second aim is a task capacity model which can estimate the maximum schedulable execution time of a task in a given set of tasks. In a fully utilised task according to a certain scheduling algorithm, any increase in the execution time of the task will make the given set of tasks not schedulable with respect to that algorithm. The execution time in a fully utilised task is called the maximum schedulable execution time of the task.

An analogy for our capacity model is the estimation of the capacity of a barber shop. Imagine the barbershop in Figure 3.5. Let us assume that there are \( n \) tasks (classes of customers). Task \( i \) (a class of customers) \( (1 \leq i \leq n) \) arrives periodically to the resource (barbershop) requiring \( C_i \) execution time (service time).
for the server (barber). Its period is $T_i$ and the $q^{th}$ instance (customer) of task $i$ (a class of customers) must finish its execution by the time of $(q-1)T_i + D_i$ where $D_i$ is the deadline for task $i$.

Now three questions arise for the given set of tasks (customers). First, what capacity of a single server (barber) is required, or how many servers (barbers) are required to meet all task deadlines if a fast server (barber) is not available. Second, how many buffers (waiting seats) for each task (customer class) are required. Third, how do we answer the two questions above if the customers (tasks) want to have a series (pipelined multi-stage) of services, e.g., barber (e.g., CPU), then hair washer (e.g., disk) etc.

A buffer estimation was presented in Section 3.7. The capacity of a multi-server system will be presented in Chapter 4. The third problem will be handled in Chapters 5 and 6. The estimation of the optimal capacity for a single server is explained in the following section.

3.8.2 Optimal Server Capacity

We are aiming to find the optimal capacity of a server (point C in Figure 3.6). The optimal capacity occurs when the worst-case completion time for at least one instance of a task is the same as its upper-bound completion time (see Section 3.8.4) while all tasks are schedulable. This situation is called deadline balanced.

The optimal capacity is the minimum schedulable capacity of a server where any decrease in the capacity will cause the given set of tasks to be unschedulable because at least one task misses its effective deadline (point A in Figure 3.6).

If you have determined the optimal server capacity on the assumption that sufficient buffer space is available, you can also determine the required buffer space (point B in Figure 3.6) on that optimal server with Equation (3.15). The required buffer space increases as the server capacity decreases because the worst-case task response times increase. The required buffer space decreases as the
server capacity increases because the worst-case task response times decrease.

![Diagram of server capacity and worst-case response time](image)

**Figure 3.6**
Optimal Server Capacity

Algorithm 3.7 calculates the optimal server capacity for a set of tasks. The goal of Algorithm 3.7 is to calculate the common scaling factor $f_s$ for the execution times of all tasks which results in:

1. at least one instance's worst-case completion time equal to its upper-bound completion time, and
2. all tasks meeting their effective deadlines.

Let the underlying capacity of the server be $S$, then the optimal server capacity $S' = S/f_s$. The flow of the algorithm is as follows. In step 1, it calculates the initial common scaling factor $f_L$ for the execution times of all tasks for a server with a utilisation factor 1. This step is necessary to get the finite response times for all instances of tasks because a server with its utilisation factor 1 does not
Algorithm 3.7: Optimal Server Capacity
{ calculates the optimal server capacity $S'$ assuming that the underlying server
capacity is $S$ }

1. calculate the initial common scaling factor $f_\ell$ that makes the server utilisation
equal to 1, i.e.,
   \[ f_\ell \left( \frac{\sum_{i=1}^{n} C_i / T_i}{n} \right) = 1. \]

2. for tasks $i = 1$ to $n$ do
   for instances $q = 1$ to $P / T_i$ do /* $P$: peak-time interval */
   initialise the temporary scaling factor $f = f_\ell$.
   repeat
   update the values of $C_i$ with the temporary scaling factor $f$,
   i.e.,
   \[ C_i = f C_i, 1 \leq i \leq n. \]
   find the upper-bound completion time $\Phi_{i,q}$ for instance $I_{i,q}$.
   determine the next approximation $\Delta f$ for the extension
   scaling factor that makes the instance $I_{i,q}$ complete by the
time $\Phi_{i,q}$, i.e.,
   \[ \Phi_{i,q} = \Delta f \left( \left( \sum_{j=1}^{i-1} Inputs_j (0, \Phi_{i,q}) \times C_j \right) + q C_i \right) \]
   for fixed priority scheduling, or
   \[ \Phi_{i,q} = \Delta f \left( \left( \sum_{j=1}^{i} Inputs_{j \in (i,q)} (0, \Phi_{i,q}) \times C_j \right) + q C_i \right) \]
   for deadline driven scheduling.
   update the temporary scaling factor $f = (\Delta f) f$.
   until ($\Delta f = 1$). /* scaling factor adjustment converges */
   $f_{i,q} = f$. /* scaling factor guaranteeing instance $I_{i,q}$ */
endfor /* instances $q$ */
endfor /* tasks $i$ */

3. scaling factor for a set of tasks $f_s = \min_{1 \leq i \leq n, 1 \leq q \leq P / T_i} f_{i,q}$.

4. required capacity $S' = S / f_s$.

5. stop.
overflow.

In step 2, it extends the execution times of all tasks so that each instance of each task may complete exactly at its upper-bound completion time (see Section 3.8.4). The extension scaling factor $\Delta f$ is multiplied by the initial common scaling factor $f_L$ to produce the scaling factor $f_{i,q}$ which is the scaling factor that guarantees that instance $I_{i,q}$ of task $i$ will meet its deadline. Note, the upper-bound completion time of an instance of a task is the sum of the preemption time of higher-priority tasks and the execution time of the previous instance and its own execution time. When execution times of tasks are updated, the upper-bound completion time of the instance is updated too. Thus, the extension scaling factor $\Delta f$ is updated until $\Delta f$ converges to 1.

In step 3, it calculates the common scaling factor for a set of tasks $f_s = \min_{1 \leq i \leq n, 1 \leq q \leq T_i} f_{i,q}$. Then in step 4, it calculates the optimal server capacity $S' = S/f_s$.

3.8.3 Spare Execution Time for a Task

If the server has a guaranteed capacity then we can calculate the spare execution time for each task. Let $C'_i$ be the maximum schedulable execution time for each task $i$ and $C_i$ be the current execution time for task $i$ on the server. Then the spare execution time $\Delta C_i = C'_i - C_i$.

Algorithm 3.8 calculates the spare execution time for a task on a given server. The flow of the algorithm is as follows. In step 1, it finds the upper-bound completion times for all instances of tasks using Algorithm 3.9 in the following section. In step 2, it extends the execution time of task $i$ so that instance $I_{i,q}$ (the $q^{th}$ instance of task $i$) may complete exactly at its upper-bound completion time. The extension scaling factor is stored in $f_{i,i,q}$.

In step 3, it extends the execution time of task $i$ so that lower-priority instance
Algorithm 3.8: Spare Execution Time for a Task
{ calculates the spare execution time $\Delta C_i$ for task $i$ assuming that the underlying server has a guaranteed capacity }

1. find the upper-bound completion times $\Phi_{i,z}$ for instance $I_{z,q}$
   (1 ≤ $z$ ≤ $n$, 1 ≤ $q$ ≤ $P/T_z$) using Algorithm 3.9.

2. determine the $f_{i,i,q}$ for the scaling factor of task $i$ that makes instance $I_{i,q}$
   complete by the time $\Phi_{i,q}$:
   $\Phi_{i,q} = \left( \sum_{j=1}^{i-1} Inputs_j(0, \Phi_{i,q}) \times C_j \right) + q f_{i,i,q} C_i$
   1 ≤ $q$ ≤ $P/T_i$ for fixed priority scheduling, or
   $\Phi_{i,q} = \left( \sum_{j=1}^{n} Inputs_{jed(i,q)}(0, \Phi_{i,q}) \times C_j \right) + q f_{i,i,q} C_i$
   1 ≤ $q$ ≤ $P/T_i$ for deadline driven scheduling.

3. determine and update the $f_{i,x,q}$ for the scaling factor of task $i$ that makes
   lower-priority instance $I_{x,q}$ complete by the time $\Phi_{x,q}$:
   $\Phi_{x,q} = \left( \sum_{j=1}^{x-1} Inputs_{j}(0, \Phi_{x,q}) \times C_j \right) + q f_{i,x,q} C_i$
   $i < x \leq n$, 1 ≤ $q$ ≤ $P/T_x$ for fixed priority scheduling, or
   $\Phi_{x,q} = \left( \sum_{j=1}^{x} Inputs_{jed(i,x)}(0, \Phi_{x,q}) \times C_j \right) + q C_x$
   $1 \leq x \leq n$, 1 ≤ $q$ ≤ $P/T_x$ for deadline driven scheduling.

4. maximum schedulable scaling factor for task $i$:
   $f_i = \min_{i, 1 \leq z \leq n, 1 \leq q \leq P/T_z} f_{i,z,q}$
   for fixed priority scheduling, or
   $f_i = \min_{i, 1 \leq x \leq n, 1 \leq q \leq P/T_x} f_{i,x,q}$
   for deadline driven scheduling.

5. maximum schedulable execution time for task $i$: $C'_i = f_i C_i$.

6. spare execution time for task $i$: $\Delta C_i = C'_i - C_i$.

7. stop
$I_{x,q}$ will complete exactly at its upper-bound completion time. The extension
scaling factor is stored in $f_{i,x,q}$. In step 4, it calculates the maximum schedulable
scaling factor $f_i$ and the maximum schedulable execution time $C'_i$. Then in step
5, it calculates the spare execution time $\Delta C_i = C'_i - C_i$.

From the result of Algorithm 3.8, we can expand the execution time of task $i$
up to $C'_i$. Otherwise, we can execute up to $\lfloor \Delta C_i \rfloor$ instances of task $i$ on the server.
$\lfloor x \rfloor$ is the floor function which denotes the greatest integer equal to or less than $x$.

### 3.8.4 Upper-Bound Completion Time for an Instance of a Task

Our goal is to find the upper-bound completion time $\Phi_{i,q} \leq d_{i,q}$ for the $q^{th}$
instance of task $i$ (i.e., instance $I_{i,q}$) which was used in Algorithms 3.7 and 3.8.
The effective deadline $d_{i,q}$ for instance $I_{i,q}$ will be discussed later in this section.
Two important facts are observed to solve this problem.

First, the upper-bound completion time for an instance of a task is not
necessarily the same as its effective deadline ($d_{i,q}$) because the effective deadline
can be located inside a higher-priority busy period. A higher-priority busy period
is a continuous time interval during which the higher-priority tasks execute and
forms a higher-priority preemption zone. In this case, the instance of task $i$ must
complete before the instance of higher-priority tasks arrived. A peak-time interval
consists of an alternate sequence of a higher-priority busy period and the higher-
priority idle periods. If a deadline is located at a higher-priority idle period then the
upper-bound completion time for an instance of the task is exactly the same as its
effective deadline. However, if the effective deadline is located at a higher-priority
busy period then, the upper-bound completion time may be shorter than the
effective deadline. It must be the start time of the higher-priority busy period. This
is due to the fact that the higher-priority tasks cannot be preempted by task $i$. Thus,
the algorithm must determine if the deadline is inside a higher-priority busy period.
We define a level-\((i,q)\) busy period as a time interval \((a, b)\) within which instances of tasks whose priority are higher than that of instance \(I_{i,q}\) are processed. No instances of those tasks are processed in the period \((a-\epsilon, a)\) or in the period \((b, b+\epsilon)\) for sufficiently small \(\epsilon > 0\).

Second, the upper-bound completion time for the \(q^{th}\) instance of task \(i\) must meet not only the user specified external deadline \((q-1)T_i + D_i\) but also the resource constrained internal deadlines. The internal deadlines are the server overflow deadline and the buffer overflow deadline. The effective deadline \(d_{i,q}\) for instance \(I_{i,q}\) must be the minimum value of these three internal deadlines. The upper-bound completion time \(\Phi_{i,q}\) of instance \(I_{i,q}\) cannot be larger than the end of the \(P\) (peak-time interval). If any instance does not complete within \(P\), server overflow occurs. Thus, \(P\) is the server overflow deadline for any instance. The effective deadline is \(P\) when the external deadline \((q-1)T_i + D_i\) is larger than \(P\). Otherwise the effective deadline is the same as \(D_i\).

From Equation (3.15) we see that the buffer overflow deadline for the \(q^{th}\) instance of task \(i\) is \((q-1)T_i + B_i T_i\). Even if the server overflow deadline is met, if the buffer overflow deadline is less than the server overflow deadline, then the effective deadline must be updated to be the buffer overflow deadline. Thus, the effective deadline \(d_{i,q}\) for an instance \(I_{i,q}\) of a hard task \(i\) can be calculated as follows:

\[
\begin{align*}
    d_{i,q} &= \min \{ (q-1)T_i + D_i, P, (q-1)T_i + B_i T_i \} \\
    &\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad

As soft real-time tasks do not have deadlines for individual instances, it seems that it is hard to estimate the optimal server capacity when they are running on a server. Thus, Algorithms 3.7 and 3.8 are restricted to hard real-time tasks.

To find this upper-bound completion time, Algorithm 3.9 checks the start and end times of higher-priority busy periods progressively from time zero (0) until the
Algorithm 3.9: Upper-Bound Completion Time for an Instance

{ finds the upper-bound completion time $\Phi_{i,q}$ for an instance $I_{i,q}$ with its start time $S_{i,q} = (q-1)T_i$ and its effective deadline $d_{i,q}$ assuming that the underlying server does not overflow }

1. initialise the start and end time of the current busy period $a = b = 0$.
2. while $(b < d_{i,q})$ do /* deadline has not passed */
   find next arrival time $N_j$ of an instance of task $j$ immediately after the current level-$(i,q)^h$ busy period. i.e., $N_j = \left\lfloor b/T_j \right\rfloor \times T_j$.
   
   $N_{j_{\text{min}}} = \min_{1 \leq j \leq i-1} \{ N_j \}$ for fixed priority scheduling, or
   $\min_{1 \leq j \leq n} \{ N_j | N_j+D_j < S_{i,q}+D_i \}$ for deadline driven scheduling.

   $a = N_{j_{\text{min}}}$. /* start time of next busy period */
   $b = \text{Busy}_{i,q} ( a, a+C_{j_{\text{min}}})$. /* end time of next busy period */

endwhile
3. if $a \geq d_{i,q}$ then $\Phi_{i,q} = d_{i,q}$ else $\Phi_{i,q} = a$.
4. stop.

Effective deadline is passed. The checking of the higher-priority busy period continues while the current busy period has not passed the effective deadline (step 2). If the current busy period has passed the effective deadline (step 3) then the effective deadline is the upper-bound completion time ($\Phi_{i,q} = d_{i,q}$). Otherwise, the start time of the current busy period is the completion time ($\Phi_{i,q} = a$). The function 'Busy' is well described in Equation (3.8). The algorithm works on the assumption that the underlying server does not overflow, in which case the algorithm cannot find the finite length of upper-bound completion time for some instance(s).
3.8.5 Deadline Adjustment

If a given set of tasks are schedulable, such tasks may have spare deadlines. The worst-case response time $W_i$ for hard real-time task $i$ and the peak-time average response time $A_i$ for soft real-time task $i$ is the lower-bound deadline $D'_i$. Thus, the spare deadline $\Delta D_i = D_i - D'_i$. Algorithm 3.10 presents the deadline adjustment procedure.

Algorithm 3.10: Deadline Adjustment

{ determines the amount of marginal deadline to be adjusted }

if the underlying server overflows then
    deadline cannot be adjusted
else
    if (deadline $D_i < W_i$ for a hard task) or ($D_i < A_i$ for a soft task) then
        the deadline can be lengthened by the amount of $(W_i - D_i)$ if the task is hard or $(A_i - D_i)$ if it is soft.
    else /* ($D_i \geq W_i$ for a hard task) or ($D_i \geq A_i$ for a soft task) */
        the deadline can be shortened by the amount of $(D_i - W_i)$ if the task is hard or $(D_i - A_i)$ if it is soft.
endif
endif

3.9 Examples

Examples are provided to illustrate these performance evaluation algorithms for
• preemptive (Section 3.9.1) and non-preemptive (Section 3.9.2) fixed priority scheduling,
• preemptive deadline driven scheduling (Section 3.9.3), and
• for calculating the optimal server capacity and spare execution times (Section 3.9.4).

3.9.1 Two Examples for Preemptive Fixed Priority Scheduling

Example 3.1 (Table 3.1) is for a system with 4 hard and 1 soft real-time tasks where the system has a guaranteed capacity. Example 3.2 (Table 3.3) is for a system with 2 hard real-time tasks where the system does not have a guaranteed capacity. Both systems have a single CPU for which tasks are competing. The CPU uses preemptive fixed priority scheduling. It is assumed that the system provides enough buffer space for each task. The minimum CPU interarrival time for task \( i \) is \( T_i \). Task \( i \) has a priority \( i \). Task \( i \) is either hard real-time (hard) or soft real-time (soft) as defined by the variable \( HS_i \) in Table 3.1. Task \( i \) consumes a maximum CPU time \( C_i \). The CPU deadline of task \( i \) is \( D_i \). The deadline \( D_i \) applies to the worst-case response time \( W_i \) when \( HS_i \) is hard, and applies to the peak-time average response time \( A_i \) when \( HS_i \) is soft. The task parameters are given in Tables 3.1 and 3.3 for Examples 3.1 and 3.2 respectively. Initially all tasks arrive at time 0 (a critical instant). The task parameters (Table 3.3) for Example 3.2 are taken and modified from Lehoczky (1990).

The task performance indices (i.e., output parameters) calculated with Algorithm 3.5 and Equation (3.15) are given in Tables 3.1 and 3.3 for Examples 3.1 and 3.2 respectively. The performance indices include the worst-case CPU response time \( W_i \), the peak-time average CPU response time \( A_i \), and the minimum required number of input buffers \( B_i \) for each task.
Chapter 3: A Single Server Model

Table 3.1
Task Parameters of Example 3.1

<table>
<thead>
<tr>
<th>Task</th>
<th>Input Parameter</th>
<th>Output Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i$</td>
<td>$T_i$</td>
<td>$C_i$</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>18</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>2</td>
</tr>
</tbody>
</table>

The system performance indices calculated with Equations (3.12), (3.1) and Algorithm 3.1 are given in Tables 3.2 and 3.4 for Examples 3.1 and 3.2 respectively. The performance indices include the duration of the peak-time interval ($P$) and CPU utilisation factor ($U_s$). They also include the determination of whether no input is lost ($NIL$) due to throughput failure and whether the underlying system has a guaranteed CPU capacity ($GCC$).

Table 3.2
System Performance Indices of Example 3.1

<table>
<thead>
<tr>
<th>$P$</th>
<th>$U_s$</th>
<th>$NIL$</th>
<th>$GCC$</th>
</tr>
</thead>
<tbody>
<tr>
<td>180</td>
<td>.91</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

60
From this data, we can see that the system in Example 3.1 has a guaranteed capacity ($GCC$ in Table 3.2), while the system in Example 3.2 does not have (Table 3.4). In Example 3.1, the system has a guaranteed capacity because the utilisation ($U_s$) is less than 1 and no input is lost ($NIL$ in Table 3.2) and all tasks meet their deadlines (Table 3.1). Task 5 is a soft real-time task. Its peak-time average CPU response time ($A_5 = 15$ in Table 3.1) is less than its deadlines ($D_5 = 17$ in Table 3.1). Also, the four hard real-time tasks meet their deadlines.

In contrast, we see from Table 3.4 that Example 3.2 does not have a guaranteed CPU capacity for its workload. This is because task 2 does not meet its CPU deadline ($W_2 > D_2$ in Table 3.3) even though the utilisation is less than 1 and no input is lost. The utilisation is less than 1 because all instances of tasks complete within the peak-time interval ($P$) but not within their deadlines.

Also, we note that tasks 4 and 5 in Example 3.1 (Table 3.1) and task 2 in Example 3.2 (Table 3.2) need multiple input buffers (i.e., more than double buffering) to ensure that no input is lost because their worst-case CPU response times ($W_i$) are longer than their respective minimum interarrival times ($T_i$).

<table>
<thead>
<tr>
<th>Task</th>
<th>Input Parameter</th>
<th>Output Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T_i$</td>
<td>$C_i$</td>
</tr>
<tr>
<td>1</td>
<td>70</td>
<td>26</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>62</td>
</tr>
</tbody>
</table>

Table 3.3

Task Parameters of Example 3.2
Table 3.4
System Performance Indices of Example 3.2

<table>
<thead>
<tr>
<th>P</th>
<th>$U_s$</th>
<th>NIDL</th>
<th>GCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>700</td>
<td>.99</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

Table 3.5
Response Time for Each Instance of Task 2

To determine why task 2 in Example 3.2 does not meet its deadline, we calculate the response times of all instances during the level-2 busy period when they are preempted by higher-priority tasks. The results of these calculation with Algorithm 3.3 are given in Table 3.5. From these results, we see that the level-2 busy period is from time 0 to time 694 (the completion time of the last (7th) instance). The largest response time is 118 for the 5th instance which is greater...
than the deadline of 117. Note, only this instance misses the deadline. Also, if the deadline can be extended by 1 the system will have a guaranteed capacity. In this case, a system upgrade may be saved by reexamining the deadlines.

Note that the response times of all instances in the level-2 busy period must be checked. If we checked only the first instance, we would draw the erroneous conclusion that the deadline \(D_2 = 117\) would be met because \(R_{2,i} = 114\). Having discussed the performance of these two examples we now illustrate how the performance indices are derived by working through the calculation done by Algorithm 3.3 for Example 3.1. The detailed calculations for Tables 3.1 and 3.2 are walked through in Appendices C.1 and C.2.

To test if the model's calculations are correct, we coded a simulation program (Appendix D.1) which reads the input parameters and simulates the execution of the tasks every unit of time. This simulation is a deterministic, discrete-time simulation. The intermediate output of this simulation for Example 3.1 is shown in Figure 3.7. The simulated system follows all the assumptions we made previously in exactly the same way. The two examples in the previous section were run on the simulator.

To verify Algorithm 3.5 we compared the results of the calculation by our analytic model with the results of the simulation model. The results of the simulation were exactly the same as the results of the algorithm (given in Tables 3.1-3.5). At each time step, the simulation program output the number of the executing task so we could view the CPU execution process clearly. We see (Figure 3.7) that tasks 1-5 have their worst-case response times 2, 5, 8, 19, and 29 respectively when they arrive at the common critical instant \((i.e., \text{at time zero})\). Their completion times are shown by outlining their task numbers.

These two examples are good test cases for our model because they contain both hard and soft real-time tasks and also include all four possible cases of situations that can occur when an instance of task \(i\) arrives at the CPU:
Chapter 3: A Single Server Model

Peak-time interval = 180.
CPU busy time = 165.
CPU utilisation factor = 0.91.

Note:
+ marks in every 5 time units.
Numbers means the task numbers executed at the time.
Blank means no task executed at the time.
Outlined number indicates its worst-case completion time.
# means the end time of the peak-time interval.

Figure 3.7
Simulated CPU’s Execution Process of Tables 3.1 and 3.2

(case 1) the situation where the previous instance of task $i$ is still being processed (e.g. at time 18 in Figure 3.7 when the 2nd instance of task 4 arrives).

(case 2) the situation where processing of the previous instance of task $i$ is finished, and a higher-priority task which arrived after the previous instance of task $i$ was finished, is still being processed (e.g., at time 40 in Figure 3.7 when the 3rd instance of task 5 arrives).
Chapter 3: A Single Server Model

(case 3) the situation where processing of the previous instance of task $i$ is finished, and no higher-priority task is executing, i.e., a lower-priority task is executing or the CPU is idle. (e.g., at time 10 in Figure 3.7 when the 2nd instance of task 1 arrives).

(case 4) A higher-priority task preempts task $i$ (e.g., at time 30 in Figure 3.7 when the 3rd instance of task 3 arrives).

The fact that the results of our analytic model are exactly same as the results of the simulation model for the two test examples which contains all possible cases, confirms that the algorithm is correct.

3.9.2 An Example for Non-preemptive Fixed Priority Scheduling

4 hard real-time tasks (Table 3.6) are competing for a disk drive which uses non-preemptive fixed priority scheduling and has enough buffer space. It is assumed that once a task acquires disk service it uses the disk drive exclusively until it completes its data transfer. Thus, we model the task execution times with $C_i = C_{t_{sw}} + C_{preempt} + C_{t_{non-preempt}}$ where $C_{preempt} = 0$. We take the longest execution time of lower-priority tasks $\max_{j \in \mathcal{L}(i)} C_j$ for the worst-case blocking time $C_{t_{block}}$ (Table 3.6) for task $i$.

The utilisation factor of the disk $U_s = \frac{\sum_{i=1}^{4} U_i}{4} = \frac{\sum_{i=1}^{4} C_i/T_i}{4} = (10+20)/60 + (5+45)/200 + (25+45)/6000 + (30+30)/1500 = 0.8.$

The disk does not overflow because $U_s < 1.$

Thus, the worst-case response time ($W_i$) for task $i$ is bounded and can be calculated as in Table 3.6 by substituting its input parameter values into Algorithm 3.4. The critical instant of the disk for a task is the instant when the task arrives at the disk concurrently with all higher-priority tasks and when the lower-priority task with the longest disk execution time has just started its disk service (Lee and
Mckerrow, 1994). If the task has an arbitrary disk deadline (e.g., $D_2 > T_2$ for task 2), the worst-case disk response time for task $i$ occurs during the level-$i$ busy period starting at the critical instant of the disk.

All four tasks meet their deadlines with $W_i < D_i$, $1 \leq i \leq 4$ (see Table 3.6). The minimum required number of buffers $B_i$ is calculated for each task (Table 3.6). Thus, we can judge that the 4 tasks are schedulable on the underlying disk drive. The detailed calculations done by Algorithm 3.4 for Table 3.6 are provided in Appendix C.3.

<table>
<thead>
<tr>
<th>Task</th>
<th>Input Parameter</th>
<th>Output Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i$</td>
<td>$T_i$ $C_i^{\text{block}}$ $C_i^{s_w}$ $C_i^{\text{non-preempt}}$</td>
<td>$D_i$ $HS_i$ $W_i$ $B_i$</td>
</tr>
<tr>
<td>1</td>
<td>60  70  10  20</td>
<td>150   hard 100  2</td>
</tr>
<tr>
<td>2</td>
<td>200 70  5  45</td>
<td>250   hard 210 2</td>
</tr>
<tr>
<td>3</td>
<td>6000 60  25  45</td>
<td>500   hard 440 1</td>
</tr>
<tr>
<td>4</td>
<td>1500 0  30  30</td>
<td>600   hard 550 1</td>
</tr>
</tbody>
</table>

Table 3.6
Task Parameters for Non-preemptive Fixed Priority Scheduling

3.9.3 An Example for Preemptive Deadline Driven Scheduling
4 hard real-time tasks (Table 3.7) are competing for a CPU which uses preemptive deadline driven scheduling and has enough buffer space.
Chapter 3: A Single Server Model

Task Parameters for Preemptive Deadline Driven Scheduling

<table>
<thead>
<tr>
<th>Task</th>
<th>Input Parameter</th>
<th>Output Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T_i$</td>
<td>$C_i$</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>600</td>
<td>20</td>
</tr>
</tbody>
</table>

Table 3.7

The utilisation factor of the CPU $U_s = \sum_{i=1}^{4} U_i = \sum_{i=1}^{4} C_i/T_i = 1/10 + 2/12 + 8/30 + 20/600 = 0.56$

The CPU does not overflow ($U_s < 1$).

Thus, all task response times are bounded and can be calculated as in Table 3.7 by substituting the input parameter values into Algorithm 3.5. Notice the task deadlines are arbitrary with respect to their periods, i.e., $D_1 = T_1$, $D_2 = T_2$, $D_3 > T_3$, and $D_4 < T_4$.

All 4 tasks meet their deadlines with $W_i < D_i$, $1 \leq i \leq 4$ (see Table 3.7). The minimum required number of buffers for each task $B_i$ is also calculated (Table 3.7). Thus, we judge that the 4 tasks are schedulable on the CPU. From this data, we can see that the shorter deadline task does not necessarily have the shorter worst-case response time (e.g., see $D_1 < D_2$ and $W_1 > W_2$) although the peak-time average response times $A_i$ seems to increase proportionally with the deadlines $D_i$ ($1 \leq i \leq 4$).

To validate Algorithm 3.5, we coded a simulator (Appendix D.2) for deadline driven scheduling on a CPU. The simulation breaks the ties of deadlines between
Chapter 3: A Single Server Model

Figure 3.8

The Execution Time Map for the Tasks in Table 3.7.

two instances of tasks with a FIFO. When the simulator executed the given tasks of Table 3.7, it produced their execution time map as in Figure 3.8.
Let us check the worst-case response times for each task. Task 1 which arrives at time 30 completes at time 38 (see outlined number 1) with $W_1 = 8$. Task 2 which arrives at time 24 completes at time 29 (see outlined number 2) with $W_2 = 5$. Task 3 which arrives at time 0 completes at time 37 (see outlined number 3) with $W_3 = 37$. Task 4 which arrives at time 0 completes at time 26 (see outlined number 4) with $W_4 = 26$. As we track down each task's worst-case response time, we find that this simulator output and our model's calculation in Table 3.7 for the worst-case response times of all tasks are the same except the worst-case response times for task 3 and 4. For them, our model estimates 1 time unit more than the simulator (i.e., $W_3 = 38$ and $W_4 = 27$) because our model considers the worst case delay when two instances of different tasks have the same deadlines. Note that the ties are broken arbitrarily by the actual deadline scheduler whereas this simulator is coded simply to break the tie with a FIFO policy for the tractability of the task executions. In the actual worst-case situation, task 1 which arrives at 20 with the same deadline 30 as task 4, delays task 4 for 1 time unit in our model. Similarly, task 1 which arrives at 30 with the same deadline 40 as task 3, delays task 3 for 1 time unit in our model.

### 3.9.4 Two Examples for Server Capacity Estimation

A set of 3 hard real-time tasks (Table 3.8) is running on a CPU which uses fixed priority scheduling and has enough buffer space.

The utilisation factor of the CPU $U_s = \sum_{i=1}^{3} \frac{U_i}{C_i/T_i} = \sum_{i=1}^{3} \frac{C_i}{T_i} = \frac{40}{100} + \frac{40}{150} + \frac{100}{350} = 0.95$

The CPU does not overflow ($U_s < 1$).

Thus, all task response times are bounded and can be calculated as in Table 3.8 by substituting the input parameter values into Algorithm 3.3. All three tasks
Chapter 3: A Single Server Model

### Table 3.8
Task Parameters for Server Capacity Estimation

<table>
<thead>
<tr>
<th>Task</th>
<th>Input Parameter</th>
<th>Output Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>$T_i$</td>
<td>$C_i$</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>40</td>
</tr>
<tr>
<td>2</td>
<td>150</td>
<td>40</td>
</tr>
<tr>
<td>3</td>
<td>350</td>
<td>100</td>
</tr>
</tbody>
</table>

meet their deadlines with $W_i < D_i$, $1 \leq i \leq 3$ (see Table 3.8). The minimum required number of buffers for each task $B_i$ is also calculated. Thus, we can judge that the set of tasks is schedulable on the CPU. Now, using the Algorithm 3.10, the spare deadline for task $i$ is calculated: $\Delta D_i = D_i - W_i$ (e.g., $\Delta D_2 = D_2 - W_2 = 150 - 80 = 70$). The spare execution time for task $i$ ($\Delta C_i = C'_i - C_i$) is calculated with Algorithm 3.8. We see that task 2 has spare execution time ($\Delta C_2$) of 3.3 (Table 3.8). Thus, we can expand the execution time of task 2 up to 43.3 ($C'_2$). Otherwise, we can execute up to 3 (i.e., $\lceil 3.3 \rceil$) instances of task $i$ on the server.

If we estimate the required CPU capacity for this set of tasks using Algorithm 3.7, it yields the scaling factor $f_s = 1.0263$. Thus the required CPU capacity $S' = S/f_s = 0.97S$ when the underlying capacity of the CPU is $S$. This means that the underlying capacity can be reduced to a smaller (i.e., 0.97) capacity for the given set of tasks to be schedulable.

We take another example (Table 3.9) for the case when the underlying capacity is insufficient and a CPU overflows due to the excessive workload of the two tasks. We assumed that there exists a bigger CPU with enough capacity to schedule the given set of tasks.
The utilisation factor of the CPU \( U_s = \frac{2}{i=1} U_i = \frac{2}{i=1} C_i/T_i = 500/100 + 320/200 = 6.6. \) The CPU overflows \((U_s > 1)\).

Thus, task response times (for some or all tasks) cannot be bounded and cannot be calculated using the Algorithm 3.3. The tasks do not have spare execution times either. In this situation, we have to calculate the optimal required CPU capacity. Using the Algorithm 3.7, we obtain the scaling factor \( f_s = 0.1 \) (step 3). Thus the required CPU capacity \( S' = S/f_s = 10S \) (step 4) when the underlying capacity of the CPU is \( S \).

<table>
<thead>
<tr>
<th>Task</th>
<th>Input Parameter</th>
<th>Output Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i )</td>
<td>( T_i )</td>
<td>( C_i' )</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>50</td>
</tr>
<tr>
<td>2</td>
<td>200</td>
<td>32</td>
</tr>
</tbody>
</table>

**Table 3.9**
Task Parameters for Excessive Workload
If we move the set of tasks in Table 3.9 to a CPU with the required capacity then the updated $C'_i = f_i C_i = 0.1 C_i$ (Table 3.10). Now, the worst-case response time (50) of the task 1 is the same as its deadline (i.e., deadline balanced) while task 2 has a spare deadline ($\Delta D_2 = D_2 - W_2 = 38$). This implies that the required capacity is the optimal capacity for the given set of the two hard real-time tasks.

### 3.10 Discussion

In this chapter, we have generalised Joseph and Pandya’s model (1986) of real-time systems to cover (1) both hard and soft real-time tasks, (2) both preemptive and non-preemptive scheduling, (3) both the fixed priority and deadline driven scheduling, and (4) a greater range of task parameters. In doing so, we have obtained additional information about the execution of the tasks which is useful for analysing the performance of individual tasks as well as for capacity planning.

A general response time model is derived to estimate not only the worst-case response times but also the average response times for both hard and soft real-time tasks. Using our model, we can determine whether the underlying system has the guaranteed capacity when hard real-time tasks and soft real-time tasks are running together in the system. In particular we have presented an algorithm for calculating the completion time of every instance of every task executing in a real-time system. An algorithm to estimate the optimal server capacity is also presented. We have validated this model using a deterministic discrete-time simulation model.

However, the model is limited to a single server (e.g., a CPU). The model cannot handle multi-server (e.g., multiprocessor) or multi-stage system (e.g., CPU-disk pipeline). More general models may be available by extending this model. Multiple servers are required when a single-server cannot schedule a given
set of tasks. In the following chapter, a performance model for multi-server systems is presented.
Chapter 4
A Multi-Server Model

In this Chapter, we propose an allocation algorithm and its performance model for a multi-server system executing real-time tasks. An allocation algorithm statically assigns tasks to servers, and in each server a scheduler dynamically assigns resource time to tasks. We have developed the modified-task-first-fit-allocation (MTFFA) algorithm to allocate a set of tasks with arbitrary parameters (e.g., execution times, periods, deadlines) while meeting all task deadlines. An instance of a task executes on only one server so the allocator may allocate all instances to one server or spread instances (in groups) over multiple servers.

The performance model embedded inside the algorithm calculates the required number of servers for the system to meet all task deadlines. As each instance of a task is executed on only one server, the worst-case and the peak-time average response times and the minimum required number of input buffers for each task is calculated with the single server model in Chapter 3.

4.1 Introduction

The availability of inexpensive microprocessors has made it practical to employ large numbers of processors in real-time applications (Mok and Dertouzos, 1978). In a hard real-time system, distributing the workload to many processors in a multiprocessor can solve both the throughput failure problem and the response failure problem, which may arise in a uniprocessor system.

It can be extended to any kind of resource (e.g., disk, communication system etc.). If either the throughput failure problem or the response-failure problem
occurs when we use a state-of-the-art single server system, the only solution is to use a multi-server system.

In a multi-server system, an allocator allocates a task to a server. Then a local scheduler in the server schedules the task. While Chapter 3 dealt with scheduling algorithms in a server, this chapter focuses on allocation algorithms. To minimise the required number of servers and to maximise the schedulable utilisation of each server, we need an optimal allocation algorithm. We also need a performance model embedded in the algorithm to estimate the performance of the system.

Dynamic real-time allocation algorithms (Ramamritham and Stankovic, 1984; Ramamritham et al, 1989; Ramamritham et al, 1990) have been proposed. They aim to provide flexibility and better resource utilisation. However, the dynamic allocation algorithms do not provide an exact analysis of performance.

Dhall and Liu (1978) studied static real-time allocation problems on multiprocessor systems and investigated the performances of two multiprocessor allocation algorithms: the first-fit-allocation algorithm and the next-fit-allocation algorithm. Each processor used a rate-monotonic-scheduler (Liu and Layland, 1973). Both allocation algorithms can estimate the required number of processors. The first algorithm does a better job of reducing the required number of processors than the second algorithm.

We propose a performance model which extends the single server model in Chapter 3 to a multi-server system. It is embedded in the proposed modified-task-first-fit-allocation (MTFFA) algorithm. The algorithm is modified from, and more general than, Dhall and Liu's first-fit-allocation algorithm and is applicable to tasks with arbitrary time parameters and allows any internal scheduling algorithm for a server. In the MTFFA, the instances of tasks whose execution times are longer than their periods are distributed to an appropriate number of servers to solve the server overflow problem which occurs when all their instances are allocated to a single server. If a task is not schedulable on a server, it is allocated to another
server which can schedule the task. In the model, we derive the required number of servers for a system to schedule a given set of tasks.

![Diagram of a multi-server system](image)

**Figure 4.1**

A Multi-Server System

### 4.2 System Model

The system model (Figure 4.1) whose performance is to be estimated has the following properties.

1. The system has multiple, identical servers. Each server has enough buffers for each allocated task.

2. Task $i$ ($1 \leq i \leq n$) has the properties described in Chapter 3: period $T_i$, execution time $C_i$ and deadline $D_i$. Note, all servers run at the same speed.
The execution time $C_i$ for task $i$ has been measured on one of the servers.

3. The instance $q$ of task $i$ is the $q^{th}$ ($q \geq 1$) arrival of the task $i$. The first arrival of all tasks occur together at the critical instant.

4. Instances of tasks are independent from each other. Several instances of a task may be executing concurrently on several servers. Also, instances of different tasks can be executing independently on the same server. An instance executes on only one server. The instances of task $i$ executing on the same server are considered to be in group $g$ and are referred to as a modified task $\tau_{i,g}$ (see Section 4.4.1). All modified tasks assigned to the same server can arrive simultaneously at their critical instants.

4.3 Condition for Multi-Server Processing

The condition for scheduling tasks on multiple servers are given in Theorem 2.8. Let the maximum capacity of a server in a single server system be $S$, and assume that a set of $n$ tasks are not schedulable and $C_i$ ($1 \leq i \leq n$) is measured on that server, then a server in a multi-server system must have capacity $S' \geq \max_{1 \leq i \leq n} (C_i/D_i)S$ because $C_i(S/S') \leq D_i, 1 \leq i \leq n$ is required.

If such $S'$ is not available then multi-processing is impossible because even if the task with $\max_{1 \leq i \leq n} (C_i)$ alone is executed in the server of $S'$, it cannot be scheduled due to deadline miss. Otherwise, the set of tasks is schedulable with a finite number of servers in the multi-server system for any scheduling algorithm. From Theorem 2.8, we develop a new scheduling algorithm (Algorithm 4.1).
4.4 Performance Model

A sequence of three actions is required to schedule instances of tasks to execute on servers in a multi-server system. The stages are: task modification at system build time, allocation of the modified tasks among servers at system build time, and dynamic scheduling of the tasks within individual servers at system run time.

The first stage is task modification. All instances of a task may execute on one server if it is schedulable. Otherwise instances of a task may be spread across several servers in groups. Task modification is the process of dividing the instances of tasks into groups for allocation to multiple servers. They are divided into the minimum number of groups where each group can be schedulable on an empty server. An empty server is a server on which no task is allocated yet. A group of instances of a task is called a modified task.

The second stage is the allocation of the modified tasks to servers. The allocation of a modified task to a server is based upon its execution and response time requirements. If a modified task cannot be scheduled on a server with other modified task(s) then it is allocated to an empty server. The MTFFA algorithm produces a table containing the allocation of the instances of tasks to servers. The loader uses this information to determine which tasks to load into the local memory of each server. This allocation is static throughout the life of the system.

The third stage is the execution of the instances of tasks on servers. During execution, a multi-server scheduler monitors the arrivals of tasks and accumulates the instance number \( q \) for each task. Using the instance number, and the table set up by the static allocation, the multi-server scheduler determines which server is to execute this instance of the task and signals the server to schedule the task. Each server may handle one or more tasks. So internal to each server, a fixed priority or deadline driven scheduler is used at system run-time. This dynamic scheduling can
be considered as subscheduling (Dasarathy and Feridun, 1984) of MTFFA.

This approach to scheduling is used in order to meet real-time requirements. Allocating tasks to servers at system build time enables all the object code to be loaded once, which is essential for embedded systems. Also it reduces the execution requirement of dynamic scheduling to that required to fetch information from a lookup table. Disadvantage is that it is inflexible. However, an important advantage is that we base the allocation of instances to server on a performance model. In fact the MTFFA algorithm evaluates the performance of each task at allocation time. In the following two sections we describe the task modification and the MTFFA algorithm in detail.

4.4.1 Task Modification

When the execution time \( C_i \) of task \( i \) is longer than its period \( T_i \), its instances are alternately (and equally) distributed to different servers. Otherwise, the task alone can cause server overflow. The distribution relaxes the period, that is it increases the interarrival time of instance of a task on a server because other instances go to another server. Thus, the minimum number of servers \( (N_i) \) required for task \( i \) is the smallest integer satisfying \( C_i \leq N_iT_i \):

\[
N_i = \lceil \frac{C_i}{T_i} \rceil
\]  

(4.1)

where \( \lceil x \rceil \) is the ceiling function. Now, the period \( (T_i) \) multiplied by the number of servers \( (N_i) \) is equal to or greater than the execution time \( (C_i) \). This number is the period \( (T_{i,g}) \) of the modified task \( \tau_{i,g} \):

\[
T_{i,g} = N_iT_i = \lceil \frac{C_i}{T_i} \rceil T_i
\]  

(4.2)

The instances of a task which are allocated to the same server form a modified task.
A different modified task is allocated to a different server. Thus, the number of modified tasks for task \(i\) is same as the minimum required number of servers for task \(i\) as given by Equation (4.1). The group number \(g\) for the \(q^{th}\) instance of task \(i\) is:

\[
g = ((q - 1) \mod N_i) + 1 = ((q - 1) \mod \lceil C_i/T_i \rceil) + 1
\] (4.3)

The result of distributing the processing in this way is that task \(i\) has a virtual period \(T_{i, virtual} = T_{i,g}\) at each server (Equation (4.2)). Thus, the system can now achieve the desired throughput. For example, consider a task with \(T_i = 10\) and \(C_i = 32\). From Equation (4.2), the minimum required number of servers, \(N_i = \lceil C_i/T_i \rceil = 4\). The sequence of execution of the instances of the task on 4 servers \((p_1, \ldots, p_4)\) is shown in Figure 4.2.

![Figure 4.2](image)

Execution Time Sequence Diagram for distribution of task \(i\) to 4 servers

From a system point of view, the effect of parallel processing is to reduce the task execution time to be less than the period. This virtual execution time \(C_{i, virtual} = C_i/N_i = C_i \lceil C_i/T_i \rceil\) as shown in Figure 4.3. The virtual period at each server...
is $T_i^{\text{virtual}} = T_{i,g} = \lceil C_i / T_i \rceil T_i = 40$.

\begin{figure}[h]
\centering
\includegraphics[width=0.7\textwidth]{system_diagram.png}
\caption{T_i and C_i for task i from viewpoint of individual servers and whole system}
\end{figure}

The utilisation factor ($U_{i,g}$) of the modified task $\tau_{i,g}$ executing on server $k$ is:

$$U_{i,g} = C_i / T_{i,g} = \lceil C_i / T_i \rceil (C_i / T_i) \quad (4.4)$$

\subsection{4.4.2 Modified-Task-First-Fit-Allocation (MTFFA)}

The MTFFA algorithm is a two-phase procedure. \textit{In the first phase}, it transforms a set of original tasks to a set of modified tasks. This is a necessary step for tasks whose execution time exceeds their periods. The algorithm divides those overrunning tasks into a minimum number of modified tasks which do not
overflow. A task overflows when its utilisation factor is greater than 1.

In the second phase, it assigns the modified tasks one by one to a server while each task is schedulable in the server. A task is schedulable in a server when the server can schedule the task itself and all the other tasks already assigned in the server according to the local scheduling algorithm. For each task, it examines earlier servers used. When the task is not schedulable in the first server, it moves on to the subsequent servers until a server is found which can schedule the task. If no used server can schedule the task then it assigns a new server for the task. Once a modified task is assigned to a server then it is scheduled by the local scheduling algorithm (e.g., fixed priority scheduling or deadline driven scheduling) on the local server.

The static allocation of tasks to servers by the MTFFA procedure is described in Algorithm 4.1. In this algorithm, tasks are first modified according to instance groups (step 1). Then, for each modified task, every server is examined, starting at server 1 (step 3(a)), to see if it can schedule the task. If it can schedule the modified task then the task is allocated to it (step 3(c)), otherwise the scheduler moves to the next server (step 3(b)).

At the completion of the algorithm, all tasks have been allocated to servers and the number of servers used by the tasks is $N$. Note that $N$ is the required number of servers for the given tasks. $\psi_{i,k}$ (step 3(c)) is an integer variable which is 1 when the modified task $\tau_{i,g}$ is assigned to server $k$, and is 0 otherwise (step 2).

The schedulability test (step 3(b)) of a modified task ($\tau_{i,g}$) on a server ($p_k$) includes the calculation of the utilisation factor of the server ($U_k$), the required number of input buffers for each task on the server ($B_{i,k}$), the worst-case response time for the task on the server ($W_{i,k}$), and the peak-time average response time for the task on the server ($A_{i,k}$). Algorithm 4.2 describes the details of the schedulability test.
Algorithm 4.1: Modified-Task-First-Fit-Allocation (MTFFA)

{ produces an allocation matrix \( \psi \), the required number of servers \( N \) }

1. create the modified tasks: \( \tau_{i,g}(\lceil C_i / T_i \rceil T_i, C_i, D_i) \) for \( 1 \leq i \leq n, 1 \leq g \leq \lceil C_i / T_i \rceil \).

2. initialise the required number of servers \( N = 1 \).
   initialise the allocation matrix \( \psi_{i,k} = 0 \) for \( 1 \leq i \leq n, 1 \leq k \).

3. for \( i = 1 \) to \( n \) do
   for \( g = 1 \) to \( \lceil C_i / T_i \rceil \) do
     (a) set current server number \( k = 1 \).
     (b) while \( \tau_{i,g} \) is not schedulable on server \( p_k \) (Algorithm 4.2) do
         set \( k = k + 1 \).
     endwhile
     (c) allocate \( \tau_{i,g} \) to \( p_k \) and
         set allocation matrix \( \psi_{i,k} = 1 \).
     if \( k > N \) then
         set \( N = k \).
     endif
   endfor  /* for each group */
   endfor  /* for each task */

4. stop.
Algorithm 4.2: Schedulability Test for a Multi-Server System.
{ determines if a modified task $\tau_{i,g}$ is schedulable on server $p_k$ ($1 \leq k \leq N$) in a multi-server system with $N$ used servers so far }

1. set allocation matrix $\psi_{i,k} = 1$. /* temporarily allocate to $p_k$ for the test */
2. set execution time $C_{j,k} = \psi_{j,k}C_j$, period $T_{j,k} = \lceil C_j/T_j \rceil T_j$, and deadline $D_{j,k} = D_j$, for $1 \leq j \leq n, 1 \leq k \leq N$.
3. calculate the utilisation factor for the server, $U_k = \sum_{j=1}^{i} (\psi_{j,k}C_j)/(C_j/T_j)T_j$.
4. if $(U_k > 1)$ then /* server overflow */
   throughput failure occurs and $\tau_{i,g}$ is not schedulable on server $p_k$.
   else
   for each task $j$ allocated to server $p_k$ do
      calculate the worst-case response time ($W_{j,k}$).
      calculate the peak-time average response time ($A_{j,k}$).
      if ($B_{j,k}^{install} < \lceil W_{j,k}/T_{j,k} \rceil$) then /* buffer overflow */
      throughput failure occurs and $\tau_{i,g}$ is not schedulable on server $p_k$.
      else
      if ((task $j$ is hard) and ($W_{j,k} > D_{j,k}$)) or ((task $j$ is soft) and ($A_{j,k} > D_{j,k}$)) then /* deadline miss */
      response failure occurs and $\tau_{i,g}$ is not schedulable on server $p_k$.
      else
      $\tau_{i,g}$ is schedulable on server $p_k$.
      endif
   endif
   endfor
   if all task $j$ is schedulable then
   $\tau_{i,g}$ is schedulable on server $p_k$.
   else
   $\tau_{i,g}$ is not schedulable on server $p_k$.
   endif
5. set allocation matrix $\psi_{i,k} = 0$. /* back to the deallocation status after test */
6. stop.
Note that the schedulability test in the single server model (Algorithm 3.1) is a special case of the Algorithm 4.2 when there is only one server \( i.e., N = 1 \) and the allocation matrix \( \psi_{i,N} = 1, 1 \leq i \leq n \) in the system.

The worst-case response time \( W_i \) and the peak-time average response time \( A_i \) of task \( i \) can be calculated at the completion of Algorithm 4.1 as follows:

\[
W_i = \max_{1 \leq k \leq N} W_{i,k} \quad (4.5)
\]

\[
A_i = \text{avg} \ A_{i,k} \quad (4.6)
\]

\( W_i \) is the longest response time of the instances of the task \( i \) among all servers while \( A_i \) is the peak-time average response time of the instances of the task \( i \) among all servers. Note, \( W_{i,k} \) and \( A_{i,k} \) is obtained by substituting \( C_{i,k}, T_{i,k} \) and \( P_{i,k} \) into Algorithm 3.5.

\( B_{j,k} \) buffers (Algorithm 4.2, step 4) are required for task \( j \) on server \( k \) not to lose any input data.

\[
B_{j,k} = \left\lceil W_{j,k}/T_{j,k} \right\rceil = \left\lceil W_{j,k}/(\lceil C_j/T_j \rceil T_j) \right\rceil \quad (4.7)
\]

Let \( Z_j \) be the size of an input buffer for task \( j \), then the required buffer space for tasks executing on server \( k \) is the sum of the required buffer space for all the tasks assigned to the server.

\[
M_k = \sum_{j=1}^{n} (\psi_{j,k} B_{j,k}) Z_j \quad (4.8)
\]

The buffer space required to store all the input in the whole system is the sum of the required buffer spaces for all the servers:
\[ M_S = \sum_{k=1}^{N} M_k \]  

(4.9)

### 4.5 Three Examples

To illustrate the application of the theory developed in the previous sections, we will work through three simple examples.

**Example 4.1**

A multiprocessor system has five periodic hard real-time tasks with enough processors and sufficient buffer space. Each processor uses preemptive fixed priority scheduling. The period \((T_i)\), the execution time \((C_i)\), the deadline \((D_i)\), and the size of an input buffer \((Z_i)\) for each task \(i\) are given in Table 4.1. This example shows the effect on performance of one task overrunning its deadline. The execution time of task 2 (18) exceeds its period (12). Therefore, task 2 cannot be scheduled on one server as it will eventually miss the deadline (25).

<table>
<thead>
<tr>
<th>(i)</th>
<th>(T_i)</th>
<th>(C_i)</th>
<th>(D_i)</th>
<th>(Z_i)</th>
<th>(HS_i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>1</td>
<td>10</td>
<td>2</td>
<td>hard</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>18</td>
<td>25</td>
<td>1</td>
<td>hard</td>
</tr>
<tr>
<td>3</td>
<td>150</td>
<td>15</td>
<td>30</td>
<td>4</td>
<td>hard</td>
</tr>
<tr>
<td>4</td>
<td>600</td>
<td>30</td>
<td>35</td>
<td>7</td>
<td>hard</td>
</tr>
<tr>
<td>5</td>
<td>30</td>
<td>18</td>
<td>40</td>
<td>3</td>
<td>hard</td>
</tr>
</tbody>
</table>

**Table 4.1**

Task Input of Example 4.1
To solve this overrun problem, we substitute the input parameters for the system (Table 4.1) into Algorithm 4.1. The results of Algorithm 4.1 are given in Table 4.2. It contains the specification and the performance indices for each modified task calculated by Algorithm 4.1. The specification includes task number \(i\), instance group number \(g\), period \((T_{i,g})\), execution time \((C_i)\), deadline \((D_i)\), and the allocated processor number \((k)\). The task performance indices are the utilisation factor of the modified task \((U_{i,g})\), the worst-case system response time \((W_{i,k})\) and the required number of buffers for the task on the allocated processor \((B_{i,k})\). Note that task 2 is divided into two groups where each group has a shorter execution time (18) than its period (24) and a shorter worst-case response time (20 and 18 respectively) than its deadline (25). Task 2 is now scheduled on 2 processors (i.e., processor 1 and 2), and now meets its deadline because the virtual period (see \(T_{2,1}\) and \(T_{2,2}\)) is lengthened to 24 which is longer than the execution time of 18.

<table>
<thead>
<tr>
<th>(i)</th>
<th>(g)</th>
<th>(T_{i,g})</th>
<th>(C_i)</th>
<th>(D_i)</th>
<th>(k)</th>
<th>(U_{i,g})</th>
<th>(W_{i,k})</th>
<th>(B_{i,k})</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td>.10</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>24</td>
<td>18</td>
<td>25</td>
<td>1</td>
<td>.75</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>24</td>
<td>18</td>
<td>25</td>
<td>2</td>
<td>.75</td>
<td>18</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>150</td>
<td>15</td>
<td>30</td>
<td>3</td>
<td>.10</td>
<td>15</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>600</td>
<td>30</td>
<td>35</td>
<td>4</td>
<td>.05</td>
<td>30</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>30</td>
<td>18</td>
<td>40</td>
<td>3</td>
<td>.60</td>
<td>33</td>
<td>3</td>
</tr>
</tbody>
</table>

**Table 4.2**  
Task Performance of Example 4.1
Chapter 4: A Multi-Server Model

The system performance values which are also calculated by Algorithm 4.1 are given in Table 4.3. The system performance indices are the utilisation factor ($U_k$), the required buffer space for each processor ($M_k$) and the system ($M_s$), and the optimal number of processors ($N$) required to guarantee the deadlines of the given tasks.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>$k$</th>
<th>$U_k$</th>
<th>$M_k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Individual Processors</td>
<td>1</td>
<td>.85</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>.75</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>.70</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>.05</td>
<td>7</td>
</tr>
<tr>
<td>$N$</td>
<td>4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$M_s$</td>
<td>-</td>
<td>-</td>
<td>21</td>
</tr>
<tr>
<td>Uniprocessor</td>
<td>-</td>
<td>2.35</td>
<td>-</td>
</tr>
<tr>
<td>Multiprocessor</td>
<td>-</td>
<td>.59</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 4.3
System Performance of Example 4.1

The same workload was applied to a uniprocessor, and its utilisation factor was calculated using Equation (3.1). The uniprocessor is hopelessly overloaded ($U_s = 2.35$) for the given tasks (Table 4.3). While the utilisation factor of the uniprocessor implies that 3 processors should be sufficient for no server overflow, we have seen that 4 processors are required to guarantee that these tasks meet their deadlines. The fourth processor is very lightly loaded ($U_d = 0.05$). Thus, the system has capacity for additional load. The overall utilisation of the multiprocessor system, which is the average of the utilisations of individual
processors, is 0.59 to guarantee the response times of these particular tasks.

**Example 4.2**

A different workload (Table 4.4) is applied to the multiprocessor system of Example 4.1. This example shows that multiple processors are needed when the tasks will overload a uniprocessor even though their execution times are shorter than their periods (*i.e.*, the utilisation factor for each task is less than 1).

<table>
<thead>
<tr>
<th>$i$</th>
<th>$T_i$</th>
<th>$C_i$</th>
<th>$D_i$</th>
<th>$Z_i$</th>
<th>$HS_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>1</td>
<td>10</td>
<td>2</td>
<td>hard</td>
</tr>
<tr>
<td>1</td>
<td>12</td>
<td>6</td>
<td>15</td>
<td>1</td>
<td>hard</td>
</tr>
<tr>
<td>2</td>
<td>150</td>
<td>15</td>
<td>20</td>
<td>4</td>
<td>hard</td>
</tr>
<tr>
<td>3</td>
<td>600</td>
<td>30</td>
<td>35</td>
<td>7</td>
<td>hard</td>
</tr>
<tr>
<td>4</td>
<td>30</td>
<td>18</td>
<td>40</td>
<td>3</td>
<td>hard</td>
</tr>
</tbody>
</table>

**Table 4.4**

Task Input of Example 4.2

To guarantee that all the tasks meet their deadlines, Algorithm 4.1 allocates tasks 1 and 2 to processor 1, tasks 3 and 5 to processor 2, and task 4 to processor 3 as shown in Table 4.5, requiring a total 3 processors in the system as shown in Table 4.6.

From Table 4.6 we see that a uniprocessor is overloaded ($U_s = 1.35$) while a 3 processor system is not. Three processors are required to guarantee that all tasks meet their deadlines.
### Chapter 4: A Multi-Server Model

<table>
<thead>
<tr>
<th>(i)</th>
<th>(g)</th>
<th>(T_{i,g})</th>
<th>(C_i)</th>
<th>(D_i)</th>
<th>(k)</th>
<th>(U_{i,g})</th>
<th>(W_{i,k})</th>
<th>(B_{i,k})</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td>.10</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>12</td>
<td>6</td>
<td>15</td>
<td>1</td>
<td>.50</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>150</td>
<td>15</td>
<td>20</td>
<td>2</td>
<td>.10</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>600</td>
<td>30</td>
<td>35</td>
<td>3</td>
<td>.05</td>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>30</td>
<td>18</td>
<td>40</td>
<td>2</td>
<td>.60</td>
<td>33</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>(k)</th>
<th>(U_k)</th>
<th>(M_k)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Individual processors</strong></td>
<td>1</td>
<td>.60</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>.70</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>.05</td>
<td>7</td>
</tr>
<tr>
<td><strong>(N)</strong></td>
<td>3</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td><strong>(M_s)</strong></td>
<td>–</td>
<td>–</td>
<td>20</td>
</tr>
<tr>
<td><strong>Uniprocessor</strong></td>
<td>–</td>
<td>1.35</td>
<td>–</td>
</tr>
<tr>
<td><strong>Multiprocessor</strong></td>
<td>–</td>
<td>.45</td>
<td>–</td>
</tr>
</tbody>
</table>

**Table 4.5**

Task Performance of Example 4.2

**Table 4.6**

System Performance of Example 4.2
Example 4.3

A workload containing hard and soft real-time tasks (Table 4.7) is applied to the multiprocessor system in Example 4.1. For soft real-time tasks the performance calculations are different (Algorithm 3.5). This example shows that we may need a multiprocessor system when the utilisation factor of a uniprocessor is less than 1 (Table 4.10) because some hard real-time tasks miss their deadlines.

<table>
<thead>
<tr>
<th>$i$</th>
<th>$T_i$</th>
<th>$C_i$</th>
<th>$D_i$</th>
<th>$Z_i$</th>
<th>$HS_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>1</td>
<td>10</td>
<td>2</td>
<td>hard</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>3</td>
<td>15</td>
<td>1</td>
<td>hard</td>
</tr>
<tr>
<td>3</td>
<td>150</td>
<td>15</td>
<td>20</td>
<td>4</td>
<td>hard</td>
</tr>
<tr>
<td>4</td>
<td>600</td>
<td>30</td>
<td>35</td>
<td>7</td>
<td>hard</td>
</tr>
<tr>
<td>5</td>
<td>30</td>
<td>6</td>
<td>15</td>
<td>3</td>
<td>soft</td>
</tr>
</tbody>
</table>

Table 4.7
Task Input of Example 4.3

If the tasks are executed on a uniprocessor, then, according to Algorithm 3.5, the worst-case response times for hard real-time tasks 3 and 4 are 24 and 70 respectively and the peak-time average response time for soft real-time task 5 is 21, which are all longer than their deadlines (Table 4.8). This occurs even though the utilisation factor of the uniprocessor is 0.7 (Table 4.10).
Table 4.8
Task Performance of Example 4.3 in a Uniprocessor System

To meet their deadlines, Algorithm 4.1 assigns task 3 to processor 2 and task 4 to processor 3 as shown in Table 4.9.

Table 4.9
Task Performance of Example 4.3 in a Multiprocessor System
Chapter 4: A Multi-Server Model

<table>
<thead>
<tr>
<th>Parameter</th>
<th>$k$</th>
<th>$U_k$</th>
<th>$M_k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Individual processors</td>
<td>1</td>
<td>.55</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>.10</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>.05</td>
<td>7</td>
</tr>
<tr>
<td>$N$</td>
<td>3</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>$M_s$</td>
<td>–</td>
<td>–</td>
<td>17</td>
</tr>
<tr>
<td>Uniprocessor</td>
<td>–</td>
<td>.70</td>
<td>–</td>
</tr>
<tr>
<td>Multiprocessor</td>
<td>–</td>
<td>.23</td>
<td>–</td>
</tr>
</tbody>
</table>

Table 4.10  
System Performance of Example 4.3

4.6 Discussion

We developed a performance model to estimate the guaranteed capacity for a multi-server system executing hard and soft real-time tasks. We proposed a modified-task-first-fit-allocation (MTFFA) algorithm to schedule a set of tasks with arbitrary parameters on a multi-server system. It distributes a group of instances of a task to multiple servers. The performance model is embedded in the MTFFA algorithm. It calculates the required number of servers and buffer space to guarantee task deadlines. It also calculates the worst-case response time and the peak-time average response time of each task.

The MTFFA algorithm has an advantage over other allocation algorithms. It can schedule periodic real-time tasks whose execution times exceed their periods. It can calculate the performance indices before it allocates the tasks to servers.
However, we modelled a single stage system where the system assumes the previous stage as its input source and the next stage as its output sink and the input period (or the minimum interarrival time) is regular. In reality, this is not the case. Multiple stages are connected in a pipeline and output periods of tasks will be their input periods at the next stage. The input periods are irregular because they depend on completion times of the tasks at the previous stage. An obvious extension of our work is to devise a performance model which considers those pipeline systems.
In chapter 3, we developed a model for a single server. In chapter 4, we extended this model to handle multiple servers, but for a single resource type. In this chapter we extend the single server model to handle multiple resources connected in a pipeline. To do this, we introduce the concepts of the optimal instance blocking and the regular pipelining. The proposed pipeline model calculates the minimum required number of buffers for each task at each stage of the pipeline for flow balancing. It also calculates the worst-case and peak-time average end-to-end response times for hard and soft real-time tasks in a pipeline system. The calculations of the required server capacity for a stage and the spare task execution times are developed as well.

5.1 Introduction

In a pipeline system, tasks execute through a sequence of resources (CPU, communication network, a disk etc.). It is a multi-stage system where each stage has a single server (Figure 5.1). In the following two sections, the requirements of such systems (Section 5.1.1) and the problems for modelling their performance are addressed (Section 5.1.2)

5.1.1 Flow Balance and End-to-End Responsiveness

Tasks flow through all stages, each of which may have different speeds which are inherent to their device characteristics. Two strict requirements must be satisfied in such a situation: flow balance and end-to-end responsiveness. First, the flow of all tasks has to be balanced through all stages so that the throughput for each task is
the same at all stages. Otherwise, input data loss will take place. That is, flow unbalance will occur due to either server overflow or buffer overflow at a slow stage. In real-time situations, input requests arrive asynchronously at the start of the pipeline invoking associated tasks. The arrival of the tasks at subsequent stages in the pipeline is also asynchronous. Flow balance may be achieved by buffering between stages.

Second, even if no input loss occurs through flow balance, a task may have a longer end-to-end response time than its end-to-end deadline. By end-to-end we mean the time interval from when a task enters the pipeline to when it exits the pipeline. The worst-case end-to-end response time of each hard real-time task and the peak-time average end-to-end response-time of each soft real-time task must be equal to or less than its end-to-end deadline. To meet these requirements, we need a performance model to estimate all task performance indices. However, there are some problems in performance modelling for a pipeline system.

5.1.2 Problems

Three problems must be addressed in performance modelling for a pipeline system: speed matching, release jitter, and priority and subdeadline assignment.
Chapter 5: A Pipeline Model

(1) **Speed matching problem**

If we have multiple stages in a pipeline then the output rate of all stages must be the same as their input rate. Otherwise backlog occurs at a bottleneck stage which causes either server overflow or buffer overflow at that stage.

However, when tasks go through a pipeline of resources (e.g., CPU stage then disk stage), the difference of resource speeds at different stages may cause throughput failure for a task at a slow stage. Rummler and Wilkes (1994) and others address the cache mechanism between two servers with different speeds. However, we need to extend this concept to the hard real-time situation.

(2) **Release jitter problem**

The sharing of a server among several tasks in a stage causes the completion times of each task to vary. As the completion interval of a task is irregular, the input period of the task at the next stage is also irregular. This irregularity of task arrivals at a stage makes it hard to estimate the response times of tasks at each stage. Hence, their end-to-end system response times are complex to calculate.

Audsley *et. al* (1993), Tindell *et. al* (1994) and Klein *et. al* (1993) discuss the release jitter problem for the situation where task release times are varied and the delayed task execution affects the response times of lower-priority tasks. They focus on single-stage single-server situations where a task may not be released as soon as it arrives. For example, a task may be delayed until the next tick of a tick scheduler, or it may be awaiting the arrival of a message. The release jitter problem occurs when the worst-case time between successive releases of a task is shorter than its arrival period. They claim that this inter-arrival ‘compression’ causes the longer response times for lower-priority tasks. They calculate the worst-case response time of a task considering this. However, their models are restricted only to a single-stage.
One solution (Klein et al., 1993) to variations in input period is to delay the release of the subtask at the next stage until the beginning of the next period. However, this causes a maximum release delay time of one task period. We need to find the optimal release time for each instance of each task to improve resource utilisation.

(3) Priority and subdeadline assignment problem

As each stage has its own scheduler, we have to assign a priority for each task at each stage when it uses a fixed priority scheduler. We need an optimal priority assignment algorithm to meet the end-to-end deadlines for tasks.

When all stages use deadline driven scheduling, the subdeadline assignment problem occurs (Kao and Garcia-Molina, 1993). The end-to-end deadline is from arrival time at the system to the exit time from the system. However, the system is faced with the problem of assigning subdeadlines to each task at each stage for the local deadline driven scheduler. For example, the deadline driven scheduler may require the subdeadline at a CPU, and a subdeadline at a disk for each task. Simply applying the same subdeadline to each stage is not optimal because each stage has a different processing load and processing speed. Thus, the deadline of later stages depends on the performance of the previous stages.

To solve these three problems we introduce the concept of instance blocking (Section 5.2), the restriction of regular pipelining (Section 5.3), and the system model (Section 5.4). We then propose a mathematically tractable pipeline performance model (Section 5.5) which includes a test for an end-to-end schedulability (Section 5.5.1), a minimum-response-priority-assignment (MRPA) algorithm (Section 5.5.2), a calculation of end-to-end response time (Section 5.5.3), a determination of buffer requirements (Section 5.5.4), and an estimation of server capacity (Section 5.5.5). Finally, we discuss the restrictions of our model and how to extend it (Section 5.6).
5.2 Optimal Instance Blocking

The problem of speed mismatch among stages in a pipeline system can be solved in two ways. First, if a server has not enough speed then the capacity of the server has to be upgraded. Second, flow balance buffering can prevent both server overflow and buffer overflow for a slower server when the server is either a block device or has a large portion of task switching time in its execution time. Two level flow balance buffering is required: instance block buffering and over-period response buffering.

Over-period response buffering (the upper-level buffering) is needed when task response times exceed their interarrival times. Otherwise buffer overflow will occur. Over-period response buffering is already discussed in Section 3.7. In this section, we introduce the instance block buffering.

The purpose of instance block buffering is to match the speed between two stages. In previous chapters, we assumed that a buffer holds only one instance. However, we will allow a buffer to hold several instances. The instance blocking factor is the number of input instances stored in a buffer. This number is calculated by the system designer. The buffer is processed when the buffer is filled up or a time-out occurs. The time-out protects against the situation when an instance does not arrive for a long time.

Instance block buffering is an extension of the concept of cache buffering to general resources. Instead of a stage executing each task on an instance basis, it executes a block of instances. By executing a block of instances the overhead time for the execution of each instance is reduced, and hence the task utilisation factor of the server at the stage is reduced. The larger the task switching time the greater the impact of instance blocking on performance. So it is expected to have a greater
impact on slow resources like disks than on the CPU (Lewis and Smith, 1976; Ruemmler and Wilkes, 1994; Reddy and Wyllie, 1994). However, a large instance blocking factor may increase the task response time because the task has to wait until the buffer is filled up before it is ready to execute.

There is a trade off between the instance blocking factor and the utilisation factor of a server. The more we increase the instance blocking factor the less server utilisation we can achieve. We must increase the instance blocking factor ($b_i$) of task $i$ at least up to the point ($K_1$ in Figure 5.2) where we have no server overflow ($U_s = 1$). It is due to the fact that the speed balance (or flow balance) is achieved if and only if all servers in the pipeline do not overflow. However, we cannot increase the instance blocking factor to the point ($K_2$ in Figure 5.2) where the task deadline ($D_i$) is not met. The more we increase the instance blocking factor, the larger the worst-case response time ($W_i$) we have due to the blocking delay. This delay is the time the system waits for the buffer to fill up. Thus the optimal blocking factor must be between $K_1$ and $K_2$ when we consider other task deadlines as well.

![Figure 5.2](image)

**Figure 5.2**

Optimal Instance Blocking Factor
Chapter 5: A Pipeline Model

We need to determine the optimal instance blocking factor of a buffer for each task to balance the flow of tasks through stages while meeting the end-to-end deadlines of all tasks. If an optimal blocking factor exists, we can then determine the worst-case response time and the peak-time average response time of tasks and the minimum required number of the buffers. Otherwise, we have to expand server capacity to avoid server overflow or deadline miss.

Let the execution time of a block of $b_{ij}$ ($\geq 1$) instances of task $i$ at stage $j$ be $C_{ij}(b_{ij}) = C_{ij}^{sw} + b_{ij}C_{ij}^{non-sw}$ where $C_{ij}^{sw}$ is the task switching overhead time, $C_{ij}^{non-sw}$ is non-overhead execution time for an instance and $b_{ij}$ is the instance blocking factor. Then the optimal instance blocking factor for a task at a stage can be determined by finding the minimum blocking factor which meets the end-to-end deadlines of all tasks. This calculation is performed by substituting $C_{ij} = C_{ij}(b_{ij})$, $T_{ij} = b_{ij}T_i$ into the end-to-end schedulability test algorithm (Algorithm 5.1 in Section 5.5.1).

We take a simple example to illustrate how the optimal instance blocking factor can be found. A pipeline system (Figure 5.3) consists of a CPU and a disk with enough buffer space. The system has only one task whose period ($T_j$) is 10. Each instance of the task requires 2 units of CPU execution time ($C_{i,1}$) and then 20 units of disk execution time ($C_{i,2}$). The instance blocking factor of the task at CPU ($b_{i,1}$) is 1. Let us assume that the task switching time (i.e., disk access time) at disk ($C_{i,2}^{sw}$) is 14 and the data transfer time for an instance ($C_{i,2}^{non-sw}$) is 6 which makes the total disk execution time of an instance 20. The end-to-end deadline (i.e., CPU-disk) for the task ($D_j$) is 75 time units.
Chapter 5: A Pipeline Model

Figure 5.3
Example System for Optimal Instance Blocking Factor

Let the instance blocking factor \( b_{1,2} \) of the task at the disk be 1. Then we see that CPU utilisation factor is 2/10 which is less than 1 and that disk utilisation factor is 20/10 which is greater than 1. The end-to-end deadline cannot be met because disk overflow will cause some later instances to have indefinitely large end-to-end response time (Figure 5.4). How can we solve the problem? We notice that the task switching time takes a large proportion (70%) of the disk execution time.

Let us now make the instance blocking factor 4 for the task at the disk. Figure 5.5 shows that the disk utilisation factor is \( (14+4\times6)/(4\times10) = 38/40 \), while the CPU utilisation is still 2/10 \( (i.e., \) less than 1). Thus, now no server overflow occurs in the pipeline because of the instance blocking effect at the disk.
Chapter 5: A Pipeline Model

Figure 5.4
Execution Map when Disk Instance Blocking Factor is 1

The worst-case CPU response time of the task is 2. The worst-case disk response time of the task (see Figure 5.6) is the sum of instance blocking delay time \((4-1) \times 10\) and the disk response time \((14+4 \times 6)\). Therefore, the worst-case disk response time of the task is 68. The end-to-end worst-case response time is \((2+68) = 70\). Hence, the end-to-end deadline of 75 is met. We see that an instance blocking factor of 4 solves both the server overflow and the deadline miss problems.
If we change the instance blocking factor to 3 for the task at the disk, then the disk utilisation factor is \((14+3\times6)/(3\times10) = 32/30\) which is greater than 1 and disk overflow occurs. If we change the instance blocking to 5 for the task at the disk then the disk utilisation factor is \((14+5\times6)/(5\times10) = 44/50\) which is less than 1 and no server overflow occurs. However, the worst-case disk response time of the task is the sum of the instance blocking delay time \(((5-1)\times10)\) and the disk response time \((14+5\times6)\). Therefore, the worst-case disk response time for the task is 84. The end-to-end worst-case response time is \(2+84 = 86\) which is greater than the deadline 75. In this case the instance blocking factor of 5 solves the server overflow problem but fails to solve the deadline miss problem. Thus, a blocking factor of 4 is the only solution to the schedulability for the task.
However, what if the deadline is 45? In this case we have to enlarge the capacity of the server itself because instance blocking is unable to solve the problem. Note that the lower-bound deadline is 70 with the instance blocking factor of 4. What if the deadline is 90? Then blocking factors of both 4 and 5 will do. However, the blocking factor 4 is the optimal value because it requires less buffer space and gives a higher disk utilisation factor and a shorter disk response time.

This very simple example illustrates the concept of the optimal blocking factor well. Algorithm 5.1 (Section 5.5.1) determines the schedulability of a pipeline system using the concept. When we extend this example to multiple tasks \((n \geq 2)\),
finding the optimal instance blocking factor follows the same procedure, except Algorithm 5.1 considers other tasks in the system when calculating the utilisation factors of servers and the worst-case response times of tasks.

The blocking factor sometimes is restricted by task factors, or by the available buffer space in the system. Thus, we assume that the instance blocking factor for each task at each stage has been determined by the system designer, and will be used as an input parameter in our performance model (Assumption (5) in Section 5.4).

5.3 Regular Pipelining

It is hard to determine that the output rate is the same as the input rate through all stages in a pipeline system when the completion times of the instances of a task at a stage is so diverse that the interarrival time of the next stage will not be as regular as the one at the first stage. Note that the completion time of an instance at a stage is the same as the arrival time of the instance at the next stage. This is a real problem for us, both to make sure whether flow balance is achieved and to estimate the worst-case task response times at the next stage. If the task arrival times at the next stage are not controlled, the system is indeterminate.

We propose a regular pipeline model which has regular arrival times for all tasks at all stages. The instances of a task arrive synchronously at the first stage, but each instance has a different response time. To achieve a general model of a stage where instances arrive synchronously and leave asynchronously we delay instances between stages to re-synchronise them. Thus, we achieve a regular pipeline where instances of a task arrive synchronously at each stage. The model is extended to handle asynchronous arrival of instances for a task at the first stage by assuming that the synchronous period for the task is its minimum interarrival time.
As above, the delays cause the instances to arrive at the subsequent stage synchronously.

The regular pipeline model is mathematically tractable which makes the system deterministic and thus controllable. Our work is related to the traditional periodic flow-shop model by Bettatti and Liu (1992). They introduced the concept of postponing the release time of every instance of a task at a stage until its predecessor subtask is surely completed at the previous stage. However, their model is restricted to the situation where the task deadlines are equal to the task periods with rate-monotonic scheduling. We extend their model to include arbitrary task parameters and scheduling algorithms.

The optimal modified release time can be obtained with the minimum release delay for each instance of a task. By using the modified release times, the model transforms all stages which are non-regular into regular stages. The model modifies the task release time to the next stage by inserting a release delay time. However, note that the release delay is an overhead time which increases the response time delay. Thus, the model calculates the optimal release time for each instance of a task at each stage so that the task interarrival times are synchronised as shown in Theorem 2.1. We claim that this release time is optimal because no instance block of the task can complete its execution after the end of the worst-case block response time and there is at least an instance block that completes at the end of the worst-case block response time (see Theorem 2.1). That is, the minimal release delay for a task is achieved when at least one instance block of a task has zero (0) release delay time.

We extend the Ye Ding's example (Lehoczky, 1990) of a single stage system to a 2-stage pipeline system (Table 5.1) to illustrate the regular pipeline model and the effect of the model on the performance of the tasks. The server at stage 1 is called server-1 and the server at stage 2 is called server-2. Three tasks are running in the system. Task 1 executes only on server-1. Task 2 executes first on server-1
and then server 2. Task 3 executes only on server-2. That is only task 2 is doing the pipeline. All three tasks are hard real-time and the end-to-end deadline $D_2$ for task 2 is 190. Both stages use preemptive fixed priority scheduling. All tasks have an instance blocking factor of 1 in their stages. This simple situation is presented to enable us to understand the process of the regular pipeline model.

\[
\begin{array}{|c|c|c|c|c|c|c|}
\hline
\text{Task} & \text{T}_i & \text{C}_{i,1} & \text{Priority} & \text{C}_{i,2} & \text{Priority} & \text{D}_i & \text{HS}_i \\
\hline
\text{1} & 70 & 26 & 1 & - & - & 30 & \text{hard} \\
\text{2} & 100 & 62 & 2 & 70 & 1 & 190 & \text{hard} \\
\text{3} & 400 & - & - & 20 & 2 & 120 & \text{hard} \\
\hline
\end{array}
\]

**Table 5.1**

Task Input Parameters for a 2-stage pipeline system

The input parameters for this example are given in Table 5.1. Task 1 has period of 70 time units and processing requirement of 26 at stage 1. Task 2 has period of 100 time units and processing requirement of 62 at stage 1.

Task 1 has higher priority than task 2 and meets its deadline because $W_1 < D_1$ (Tables 5.1 and 5.2). The completion times of instances for task 2 at stage 1 are 114, 202, 316, 404, 518, 606, 694, etc. which are also their arrival times at stage 2 (Figure 5.7) with no release time modification. Thus, the response times of instances for task 2 at stage 1 is 114, 102, 116, 104, 118, 106, 94, etc. The worst-case response time of task 2 at stage 1 is 118 ($W_{2,1}$ in Table 5.2). Task 2 also meets its deadline because $W_2 < D_2$ (Tables 5.1 and 5.2).

However, the interarrival times of task 2 at stage 2 are 88, 114, 88, 114, 88, 88, etc. in the uncontrolled situation. They are irregular. When we have irregular release times of task 2 at stage 2, task 3 misses its deadline (see Tables 5.1 and 5.2).
at the critical instant 114 (Figure 5.7). Why does it miss its deadline? Because task 2 preempts task 3 at time 202 with the interarrival time 88 which is shorter than the period (100) of task 2 at stage 1.

![Figure 5.7](image)

**Figure 5.7**
Task Execution Map at Stage 2 with No Release Time Control

<table>
<thead>
<tr>
<th>Task</th>
<th>Stage 1</th>
<th>Stage 2</th>
<th>End-to-End</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>$W_{i,1}$</td>
<td>$W_{i,2}$</td>
<td>$W_i$</td>
</tr>
<tr>
<td>1</td>
<td>26</td>
<td>-</td>
<td>26</td>
</tr>
<tr>
<td>2</td>
<td>118</td>
<td>70</td>
<td>188</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>160</td>
<td>160</td>
</tr>
</tbody>
</table>

**Table 5.2**
Task Response Time with No Release Time Control
Now, we control the release time of the instances for task 2 at stage 2 to be 118, 218, 318, 418, 518, 618, 718, etc. (Figure 5.8). These modified release times give the interarrival times of task 2 at stage 2 the regular value of 100 which is the same as the interarrival time of the stage 1. This scheme gives the instances of the task the optimal release delay of 4, 16, 2, 14, 0, 12, 24, etc. Note that the 5th instance has zero (0) release delay time which makes the amount of release delay optimal. When we control the release times of the task 2 at stage 2 with the optimal modified release time, task 3 meets its deadline (see Tables 5.1 and 5.3) because the delay of the release time of task 2 from time 202 to time 218 enables task 3 to complete before it is preempted.

![Figure 5.8](image)

**Figure 5.8**

Task Execution Map at Stage 2 with Modified Release Time

This example shows that a regular release time makes the period of the task at the next stage regular which enables easy calculation of the task response time at the next stage. We also note that the regular release time seems to help remove the release jitter problem.
5.4 System Model

The performance of a pipeline system (Figure 5.1) is analysed in the following sections. The system model is defined as follows.

1. The system consists of $m$ pipelined stages where each stage has a single server with its associated buffers for tasks.

2. Each server has a local scheduler. The scheduler can have one of four scheduling policies: preemptive fixed priority, non-preemptive fixed priority, preemptive deadline, or non-preemptive deadline driven scheduling.

3. The total number of tasks which compete for the pipelined servers is $n$ where $n \geq 2$.

4. The system includes $n_h$ hard real-time tasks and $(n - n_h)$ soft real-time tasks where $0 \leq n_h \leq n$.

5. A task $i$ ($1 \leq i \leq n$) arrives at the system synchronously with a period (or a minimum interarrival time if the task is asynchronous) of $T_i$ requesting at most $C_{i,j}$ execution time at stage $j$ ($1 \leq j \leq m$). Each task must meet its end-to-end deadline $D_i$. The instance blocking factor of task $i$ at stage $j$ is $b_{i,j}$.
hence, the period of task \( i \) at stage \( j \) is \( T_{i,j} = b_{i,j}T_i \). We assume that \( b_{i,0} = 1 \) for calculations at stage 1. This enables general equations to be applied to all stages. We also assume that \( b_{i,j} = \lambda b_{i,(j-1)} \), \( \lambda = 1, 2, \ldots \) or \( b_{i,(j-1)} = \lambda b_{i,j}, \lambda' = 1, 2, \ldots \)

(6) When a deadline driven scheduler is used, the subdeadline for each task \( i \) at stage \( j \) is given as \( D_{i,j} \leq D_j \).

### 5.5 Performance Model

A schedulability test algorithm for a pipeline system is presented. A minimum-response-priority-assignment (MRPA) algorithm for each task at each stage is proposed. The model calculates the end-to-end task response times, the minimum required number of buffers for each task at each stage. Finally it calculates the required server capacity at each stage.

#### 5.5.1 End-to-End Schedulability Test

A set of tasks is *schedulable* in a pipeline system when it has no throughput failure at any stage and no end-to-end response failure. Algorithm 5.1 provides the procedure of *schedulability test* for a given set of tasks in a pipeline system. This algorithm is an extension of the schedulability test algorithm validated in Chapter 3.

The *utilisation factor of the server* at stage \( j \) is:

\[
U_j = \frac{\sum_{i=1}^{n} C_{i,j}}{T_{i,j}}. \tag{5.1}
\]

The remaining performance parameters used in this algorithm are presented in the following sections either as equations or as algorithms.
Algorithm 5.1: Schedulability Test for a Pipeline System
{ determines if a set of tasks is schedulable in a pipeline of \( m \) stages }

1. calculate the utilisation factor \((U_j)\) for the server at stage \( j \), \( 1 \leq j \leq m \).

2. if \(( \exists_j U_j > 1 )\) then /* server overflow */
   
   throughput failure occurs and
   some task(s) is(are) not schedulable.

   else
   
   calculate the worst-case stage response time \((W_{i,j})\) for task \( i \) at stage \( j \).

   \[
   \text{if } \left( \exists_{i,j} B_{i,j}^{\text{install}} < \left[ \frac{W_{i,j}}{b_{i,j} T_i} \right] \right) \text{ then} \\
   \] /* buffer overflow */

   throughput failure occurs and
   task \( i \) is not schedulable.

   else
   
   calculate the worst-case end-to-end response time \((W_i)\) for task \( i \).

   calculate the peak-time average end-to-end response time \((A_i)\).

   if ((task \( i \) is hard) and \((W_i > D_i)\)) or
   
   ( (task \( i \) is soft) and \((A_i > D_i)\) ) then /* deadline miss */
   
   end-to-end response failure occurs and
   task \( i \) is not schedulable.

   else
   
   task \( i \) is schedulable.

   endif

endif

endif

3. if (all the tasks in the task set are schedulable) then
   
   the task set is schedulable and
   the system has a guaranteed capacity.

else

   the task set is not schedulable and
   the system does not have a guaranteed capacity.

endif

4. stop.
5.5.2 Minimum-Response-Priority-Assignment (MRPA)

We observe that the global priority assignment for all stages is not optimal for fixed priority scheduling. For example, two hard real-time tasks are running in a pipeline system of three stages which have a preemptive fixed priority scheduling policy. Let the task 1 have the execution requirements $C_{1,i} = 2$, $C_{1,2} = 3$, $C_{1,3} = 5$ for each stage and the end-to-end deadline $D_1 = 20$. Let the task 2 have its execution requirements $C_{2,i} = 4$, $C_{2,2} = 6$, $C_{2,3} = 8$ and the end-to-end deadline $D_2 = 25$. If task 1 has higher priority than task 2 globally for all the three stages, then their worst-case response times will be $W_1 = 10$, $W_2 = 28$ and as a result task 2 fails to meet its end-to-end deadline. However, if task 1 can be assigned the higher priority only at stage 3, with task 2 having the higher priority at stages 1 and 2, then their worst-case response times will be $W_1 = 20$, $W_2 = 23$, and all tasks meet their respective end-to-end deadlines. Thus, we need an optimal priority assignment algorithm for each task at each stage to meet its end-to-end deadline.

We propose a minimum-response-priority-assignment (MRPA) algorithm for a pipeline model by extending Algorithm 3.6 to a pipeline system. This algorithm assumes that the servers at all stages use fixed priority scheduling. It assigns an inserted task the highest schedulable priority at each stage which will achieve the minimum schedulable end-to-end response time for the task as in Algorithm 5.2.

In step 1, the algorithm starts by assigning an inserted task the highest possible priority which guarantees the end-to-end deadlines for all tasks. If the task is not assigned any priority at a stage then the system does not have a guaranteed capacity because the server at that stage does not have sufficient capacity. Thus, there is no need to examine the further stages. Otherwise, we move on to the next stage and do the same thing.
Algorithm 5.2: Minimum-Response-Priority-Assignment (MRPA)
{ inserts a task $\tau_A$ on the server at each stage $j$ ($1 \leq j \leq m$) giving it the highest schedulable priority }

1. set assigned = TRUE.
   for stage $j = 1$ to $m$ do
     assign the inserted task $\tau_A$ the highest schedulable priority on
     the server at stage $j$ (Algorithm 3.6).
     if no schedulable priority assignment exists then
       set assigned = FALSE.
       exit the for loop.
     endif
   endfor

2. if assigned then
   calculate the worst-case end-to-end response time ($W_A$) for task $\tau_A$
   calculate the peak-time average end-to-end response time ($A_A$).
   if ($\text{task } \tau_A \text{ is hard} \text{ and } (W_A > D_A)$) or
   ($\text{task } \tau_A \text{ is soft} \text{ and } (A_A > D_A)$) then  /* end-to-end deadline miss */
     end-to-end response failure occurs and
     task $\tau_A$ is not schedulable and
     no schedulable priority assignment exists.
   else
     task $\tau_A$ is schedulable and
     the highest schedulable priority is assigned for
     task $\tau_A$ at each stage.
   endif
   else
     the task $\tau_A$ is not schedulable and
     no schedulable priority assignment exists.
   endif

3. stop.

In step 2, when the task has been assigned its priority at all stages, the end-to-end response time for the task is calculated according to the priorities assigned at each stage. If it is equal to or less than its end-to-end deadline, then the task is
schedulable and the priorities assigned at each stage are optimal. Otherwise, it is not schedulable because no priority assignment is feasible, in which case we determine that the system does not have a guaranteed capacity.

Note that the algorithm assigns an inserted task at each stage the priority which leads to its minimum response time while guaranteeing the end-to-end deadlines for all tasks. Thus, if the inserted task is not schedulable with this priority assignment algorithm then there is no other priority assignment algorithm to schedule the task (see Theorem 2.3).

In the Algorithm 5.2, the number of schedulability tests \( M_{\text{test}} \) required for all priority levels at all stages is given by (cf., \( N_{\text{test}} \) in Equation (3.13)):

\[
M_{\text{test}} = m N_{\text{test}}. \tag{5.2}
\]

Therefore,

\[
M_{\text{test}} = m(n^2+n)/2. \tag{5.3}
\]

Note that the worst-case required number of schedulability tests in the algorithm is \( m(n^2+n)/2 \).

While we claim that a subtask deadline is not required for fixed priority scheduling, it is required for deadline scheduling. We now consider the situation where all stages use deadline driven scheduling. In a deadline driven scheduler, every subtask requires a subdeadline to be used by the scheduler. We need to determine the subdeadline for each subtask. However, it seems that no one has yet found the optimal subdeadline assignment algorithm for deadline scheduling. Ho et. al (1990) discussed slack distribution policies for multiple segment tasks. Kao and Garcia-Molina (1993) proposed four heuristic equations for subdeadline estimation. All of them heuristically determine the slack times of subtasks by
Chapter 5: A Pipeline Model

considering the computation times or utilisations of the remaining subtasks. They are not the exact slack times except for the subtask at the last stage. Thus, our model simply assumes that the designer has chosen a good subdeadline assignment algorithm and has determined the subdeadlines of all subtasks for the deadline driven scheduler.

Consider the situation when a pipeline consists of hybrid stages where some stages use fixed priority scheduling and others use deadline driven scheduling. Our model will apply the MRPA algorithm to the fixed priority stages and will assume the subdeadlines of tasks at the deadline driven stages. Designers may need to find the sub-end-to-end deadline of a task for the group of deadline driven stages by subtracting the worst-case end-to-end response time of the task spent at the group of fixed priority stages from the end-to-end system deadline of the task.

5.5.3 End-to-End Response Time

The end-to-end response time $R_{i,q}$ for the $q^{th}$ instance of task $i$ is the sum of its response times $R_{i,j,q}$ at each stage $j$.

$$R_{i,q} = \sum_{j=1}^{m} R_{i,j,q}$$  \hspace{1cm} (5.4)

The response time $R_{i,j,q}$ for the $q^{th}$ instance of task $i$ at stage $j$ is the sum of its instance blocking delay $V_{i,j,q}$ and the worst-case block response time $W_{i,j}$ of task $i$ at stage $j$.

$$R_{i,j,q} = V_{i,j,q} + W_{i,j}$$  \hspace{1cm} (5.5)

$W_{i,j}$ can be calculated using Algorithm 3.5 when we know $C_{i,j}$ and $T_{i,j}$ for each task $i$ (Assumption (5) in Section 5.4)

The calculation of the instant blocking delay $V_{i,j,q}$ for the $q^{th}$ instance of task $i$
at stage $j$ depends on the ratio $\lambda$ of the two instance blocking factors $b_{i,(j-1)}$ and $b_{i,j}$ at the two adjacent stages $j-1$ and $j$ as shown in the following equations.

$$V_{i,j,q} = \left( \lambda - 1 - \left( \left\lfloor \frac{q}{b_{i,(j-1)}} \right\rfloor - 1 \right) \mod \lambda \right) b_{i,(j-1)} T_i$$

when $b_{i,j} = \lambda b_{i,(j-1)}$, $\lambda = 1,2,\ldots$ \hspace{1cm} (5.6)

or,

$$V_{i,j,q} = \left( \left\lfloor \frac{q}{b_{i,j}} \right\rfloor - 1 \right) \mod \lambda' \right) b_{i,j} T_i$$

when $b_{i,(j-1)} = \lambda' b_{i,j}$, $\lambda' = 1,2,\ldots$ \hspace{1cm} (5.7)

Note that we assumed that $b_{i,0} = 1$ (Assumption (5) in Section 5.4). The following two examples with Figures 5.9 and 5.10 illustrate Equations (5.6) and (5.7) respectively.

**Example 5.1**

Let the instance blocking factors $b_{i,(j-1)} = 3$ and $b_{i,j} = 9$ for task $i$ with a period of $T_i$. If we want to know the instance blocking delay of the 11th instance of task $i$ at stage $j$, then substitute $q = 11$, $\lambda = 3$ in Equation (5.6) as follows.

$$V_{i,j,11} = (3-1-((\left\lfloor \frac{11}{3} \right\rfloor -1) \mod 3)) \times 3 \times T_i$$

$$= 6T_i$$

We can see in Figure 5.9 that the 11th instance completes its execution at stage $j-1$ at the same time with the 10th and the 12th instances because the instance blocking factor $b_{i,(j-1)}$ at stage $j-1$ is 3.
Chapter 5: A Pipeline Model

Figure 5.9

Instance Blocking Buffer for Example 5.1

The 11th instance has to wait for its execution at stage $j$ until the 18th instance completes its execution at stage $(j-1)$ because the instance blocking factor $b_{i,j}$ at stage $j$ is 9.

Example 5.2

Let the instance blocking factors $b_{i,(j-1)} = 4$ and $b_{i,j} = 2$ for task $i$ with a period of
$T_i$. We can calculate the instance blocking delay of the 7th instance of task $i$ at stage $j$ by substituting $q = 7$, $\lambda' = 2$ in Equation (5.7) as follows.

$$V_{i,j,7} = ((\lceil 7/2 \rceil - 1) \mod 2) \times 2 \times T_i = 2T_i$$

We can see in Figure 5.10 that the 7th instance completes its execution at stage $j - 1$ at the same time with the 5th, 6th and 8th instances because the instance blocking factor $b_{i,(j-1)} = 4$.

We also can see that the 7th instance is forced to delay its execution at stage $j$ for $2T_i$ to make the pipeline regular.

Using the results of Equation (5.4), the worst-case end-to-end response time $W_i$ and the peak-time average end-to-end response time $A_i$ can be calculated as
follows.

\[ W_i = \max_{1 \leq q \leq P/T_i} R_{i,q} \]  
\[ A_i = \text{avg}_{1 \leq q \leq P/T_i} R_{i,q} \]

where \( P \) is the peak-time interval.

### 5.5.4 Buffer Requirement

We assumed that the instance blocking factor of a buffer for each task at each stage is given by designers for their application situations (see Section 5.2). However, we need to calculate how many buffers are needed to avoid buffer overflow due to the over-period task response.

Joseph and Pandya (1986) estimated the minimum required number of buffers applicable only in the situation of single instance block buffering in a single stage system (see Equation (3.15)).

We extend their model to multiple instance blocking in a multi-stage pipeline system to calculate the required number of buffers. The minimum required number of buffers for task \( i \) at the stage \( j \) can be calculated as follows. Note that each buffer stores the number of instances specified by the instance blocking factor.

\[ B_{i,j} = \left\lfloor \frac{W_{i,j}}{T_{i,j}} \right\rfloor = \left\lfloor \frac{W_{i,j}}{b_{i,j} T_i} \right\rfloor \]  

Equation (3.15) is a special case of Equation (5.10) when \( b_{i,j} = 1 \). However, we claim that Joseph and Pandya's (1986) assumption of \( b_{i,j} = 1 \) has to be generalised in a multi-stage situation where \( b_{i,j} \geq 1 \). Thus Equation 5.10 is the general equation for buffer requirements. Example 5.3 and Figure 5.11 illustrate the Equation (5.10).
Example 5.3

Suppose that a task $i$ goes through a 2-stage periodic pipeline with $T_i = 10$, $b_{i,1} = 1$, $W_{i,1} = 15$, $b_{i,2} = 3$, $W_{i,2} = 70$ and $b_{i,3} = 1$. Then, the task period at each stage (Figure 5.11) is calculated with model characteristic (5) in Section 5.4:

$$T_{i,1} = b_{i,1} T_i = 1 \times 10 = 10$$
$$T_{i,2} = b_{i,2} T_i = 3 \times 10 = 30$$

The minimum required number of over-period buffers for task $i$ at stage 2 is calculated with Equation (5.10) as follows:

$$B_{i,2} = \left\lfloor \frac{W_{i,2}}{b_{i,2} T_i} \right\rfloor = \left\lfloor \frac{70}{3 \times 10} \right\rfloor = 3$$

This buffer situation at stage 2 is illustrated in Figure 5.11. In a similar manner, we can obtain the number of buffers at stage 1:

$$B_{i,1} = \left\lfloor \frac{W_{i,1}}{b_{i,1} T_i} \right\rfloor = \left\lfloor \frac{15}{1 \times 10} \right\rfloor = 2$$

5.5.5 Server Capacity

We can calculate the minimum required capacity for the server at stage $j$ if we know the capacities of the servers at all other stages are sufficient. We focus on the server at stage $j$ alone, as if it is a single server system. The effective deadline $d_{i,j,q}$ (cf. Equation (3.16)) for the $q^{th}$ instance of task $i$ at stage $j$ is as follows:

$$d_{i,j,q} = \min\left(d_{i,j,q}^{user}, d_{i,j,q}^{server}, d_{i,j,q}^{buffer}\right)$$  \hspace{1cm} (5.11)
where user specified deadline $d_{i,j,q}^{user} = D_i - \left( V_{i,j,q} + \sum_{s=1}^{m} R_{i,s,q} \right)$

and server overflow deadline $d_{i,j,q}^{server} = P$

and buffer overflow deadline $d_{i,j,q}^{buffer} = B_{i,j} b_{i,j} T_i - W_{i,(j-1)} - (b_{i,j} - 1) T_i$

Figure 5.11
Minimum Required Number of Buffers

Note that the buffer overflow deadline $d_{i,j,q}^{buffer}$ can be derived from Equation (5.10). This deadline is for the worst-case block response time. Substituting the effective deadline $d_{i,j,q}$, $C_{ij}$, and $T_{ij}$ to $d_{i,q}$, $C_i$, and $T_i$ in Algorithm 3.8 and 3.9 we can calculate the optimal server capacity and spare execution time for task $i$ at stage $j$. 
5.6 Discussion

We have proposed a regular pipeline model which can estimate the performance of a pipelined system for a range of task parameters and local scheduling algorithms. We have justified the model of synchronisation with minimum response delay by optimal modified release time. We have shown that existing single server models can be applied or extended to the pipeline model easily. Also, we proved that by controlling the release times of tasks at each stage that we can calculate the end-to-end response times of the tasks.

However, the model makes the response time of every instance block for task $i$ the same value as the worst-case block response time $W_i$. Thus it may cause a larger average response time than the non-regular pipeline model because every instance block must be delayed to make a regular pipeline. The model is restricted to a pipeline where each stage has only one server. We extend this model to a pipelined multi-server model where each stage can have multiple identical servers in the following chapter.
Chapter 6
A Pipelined Multi-Server Model

A general model of parallel real-time systems can be formulated by combining multi-server parallelism with multi-stage parallelism. To achieve this model we extend the pipeline model in Chapter 5 to have multiple servers in each stage. We modelled multiple servers for a single stage in Chapter 4. Thus, in this chapter we combine the models developed in the previous two chapters.

In this model, a set of real-time tasks pass through a pipeline of stages where each stage has multiple servers. A performance model is presented for such parallel real-time systems. This model provides a high-level unified framework for studying the performance of general parallel real-time systems with a range of task parameters, resource types and scheduling algorithms.

6.1 Introduction

The timing requirements of real-time systems are different from those of non-real-time systems. Batch systems require good average throughputs while time-sharing systems require good average response times. However, neither of them has the strict worst-case performance requirement that real-time systems do. The requirement is that all tasks must meet their deadlines without losing any input data.

To meet such critical timing requirement, fast parallel-processing technology is exploited. Parallel approaches help to improve both the throughput and the response times for tasks. Parallelism can be achieved by two methods of resource/task partitioning: multi-server (Chapter 4) and multi-staging (Chapter 5). A multi-stage multi-server system in this chapter combines theses two concepts into
a general parallel system.

Real-time system designers and users will aim to maximise schedulable utilisation of such parallel systems to achieve economical use of their resources. This goal can be attained by both optimal task allocation and optimal resource scheduling. Task allocation is the process of mapping a task to a resource, while resource scheduling is the process of allocating control of a resource to a task. To evaluate these problems for a given set of tasks and resources, and to find out the minimal cost schedulable system configuration, we need a performance model.

The rest of this paper flows as follows. First, we review what researchers have done on these problems (Section 6.2). We then assume a system model (Section 6.3) to analyse. After that, we describe the strategies of our performance model (Section 6.4). Based on these foundations we delve into the detailed algorithms (Section 6.5) of the model with an example (Section 6.6). Finally, we conclude by discussing the constraints of the model (Section 6.7).

### 6.2 Related work

Liu and Layland (1973) derived a simple sufficient condition to determine if a set of periodic tasks can be scheduled in a single processor. They found that any set of $n$ periodic tasks can be scheduled by the rate-monotonic-scheduling (RMS) algorithm if the utilisation factor of the processor is no greater than $n(2^{1/n-1})$. As the number $(n)$ of tasks increase, the utilisation bound converges to $ln2$ (69%). The RMS algorithm is a preemptive fixed priority scheduling policy where the task with the shortest period has the highest priority. They also found that any set of periodic tasks scheduled by the preemptive deadline driven algorithm will meet all task deadlines if, and only if, the utilisation factor of the processor is no greater than 1 (100%). However, both of these schedulability tests are restricted to uniprocessor
systems and the task deadlines are assumed to be equal to the task periods.

Extensions of the Liu and Layland theory were done in many ways to derive more exact criteria for testing schedulability that can be used in more general circumstances. These state-of-the-art uniprocessor performance models are well summarised in Lehoczky et al. (1991), Sha et al. (1991) and Lehoczky (1994).


2. Task deadlines larger than task periods were studied by Lehoczky (1990) and Shih et al. (1990).

3. A mixture of periodic and aperiodic tasks was studied by Lehoczky et al. (1987), Sprunt et al. (1988b), Sprunt et al. (1989), Sprunt (1990), and Lehoczky and Ramos-Thuel (1992).

4. Transient overload was studied by Sha et al. (1986).

5. Task synchronisation was studied by Sha et al. (1990) and Baker (1990, 1991).

6. The situation in which tasks are divided into subtasks of different priorities was studied by Gonzalez Harbour et al. (1991).

7. Non-preemptive scheduling was studied by Jeffay et al. (1991).

The following issues are addressed by Joseph and Pandya (1986): the condition for no input loss; the required buffer space; the schedulability test; and the worst-case task response time. However, their model is restricted to uniprocessor systems with the fixed preemptive priority scheduling algorithm.

Some studied the guaranteed scheduling of multiprocessors (Johnson and Maddison, 1974; Dhall and Liu, 1978; Lawler and Martel, 1981; Bertossi and
Bonuccelli, 1983). Others investigated the guaranteed scheduling of resources other than processors, such as communication subsystems (Lehoczky and Sha, 1986; Strosnider, 1988) and I/O controllers (Sprunt et al., 1988a).

Meanwhile, for more practical situations where tasks require both processor time and I/O time, several papers focused on the integration of processor scheduling and I/O scheduling (Klein et al., 1994; Bettati and Liu, 1990, 1992; Ho et al., 1990; Sha et al., 1986; Zhao and Ramamritham, 1985; Ramamritham and Stankovic, 1984; Leinbaugh, 1980; Leinbaugh et al., 1982). However, these models do not deal with multiple servers at a stage.

It is obvious that we need to use multiple servers at each stage when a set of tasks is not schedulable in a pipeline where each stage has only one server. Thus, we develop a performance model which can handle pipelined multi-server real-time systems.

### 6.3 System Model

The following system model is defined for the later discussions of strategies in Section 6.4, and the development of a performance model in Section 6.5.

1. The system is a pipeline of $m$ stages where each stage $j$ ($1 \leq j \leq m$) has $N_j$ servers with its associated buffers for tasks as shown in Figure 6.1. Each stage has enough servers and all servers have enough buffer space to schedule the given set of tasks.

2. All the servers at stage $j$ have the same resource type and scheduling algorithm. As before, the model handles a range of resource types and scheduling algorithm. The unit costs of a server at all stages are the same.
Chapter 6: A Pipelined Multi-Server Model

Figure 6.1
A Pipelined Multi-Server System

(3) The total number of tasks which compete for the pipelined servers is \( n \geq 2 \).

(4) The system includes \( n_h \) hard real-time tasks and \( (n - n_h) \) soft real-time tasks where \( 0 < n_h < n \).

(5) A task \( i \) arrives at the system synchronously with a period (or a minimum interarrival time if the task is asynchronous) of \( T_i \), with each instance requesting at most \( C_{ij} \) execution time at stage \( j \). Each task \( i \) must complete within its end-to-end deadline \( D_i \). The instance blocking factor (see Section 5.2) of task \( i \) at stage \( j \) is \( b_{ij} \) where \( b_{i0} = 1 \).

(6) The sum of the execution times of a task at all stages does not exceed its end-to-end deadline, \( i.e., \sum_{j=1}^{m} C_{ij} \leq D_i, \forall (1 \leq i \leq n) \). Therefore, if a task executes on empty servers at all stages it is schedulable.

(7) The task modification concept (Section 4.4.1), and the regular pipelining concept (Section 5.3) is applied to the system.

(8) When a deadline driven scheduler is used, the subdeadline for each task \( i \) at
stage $j$ is given as $D_{ij} \leq D_j$.

(9) The lower-level performance models (Chapters 3, 4 and 5) are already developed and available to our model as submodels.

### 6.4 Strategies

Our model evaluates the performance of pipelined multi-server systems. The systems have a set of stages and each stage has a set of identical servers of the same resource type. For each resource type, a set of scheduling algorithms may be applied. Resource types may include processors, buses, memory modules, I/O channels, disk drives, network links, network nodes, etc. Scheduling algorithms may include the preemptive/non-preemptive fixed priority scheduling, preemptive/non-preemptive deadline driven scheduling, etc. The resource types and their scheduling algorithms are well described elsewhere (Klein et. al, 1993).

Our model is more abstract than any other performance models because it is at the highest level in the model hierarchy and is independent of the hardware resource types and the scheduling algorithms. The model is able to determine the schedulability for a given set of tasks. The model can also determine the minimal cost schedulable hardware configuration (Mills, 1976) and the minimal cost scheduling algorithm for each stage. Calculation of buffer space and worst-case response times are done by this model. We subsume the lower-level performance models already developed in Chapters 3, 4 and 5. To devise the performance model, basic strategies are set up beforehand as follows.

(1) **High-level Performance Model**

A long-term (or global) allocator for the system and short-term (local) schedulers for each server are required. The long-term allocator assigns a task to a server
statically at compile time. The short-term schedulers schedule a set of tasks in a server dynamically at execution time. The goal of the long-term allocator is to attain the minimal cost hardware configuration for the whole system whereas the goal of the short-term scheduler is to achieve the maximum schedulable utilisation for each server. The allocating algorithms in a parallel real-time system play a key role in achieving the performance goal of the system. Thus, the performance models can be implemented in those algorithms. We develop a long-term allocating algorithm in which our high-level performance model is embedded.

(2) Instance Distribution
A given set $TS$ of tasks has to be converted to an allocatable set $TS'$ of tasks before the actual allocation takes place. The instance distribution (see Section 4.4.1) is necessary when the execution time of a task at a stage exceeds its period at the stage.

(3) Minimum Response Fit
A task is allocated on the server at each stage which causes the minimum schedulable response time at the stage. If a task fails to meet its end-to-end deadline, then any allocation of the task to the existing system is infeasible (see Theorem 2.4). This is the best way to obtain the minimum end-to-end response time with existing servers. If this allocation fails then system expansion is required.

(4) Bottleneck Stage Expansion
When the system needs to be expanded, the bottleneck stages are expanded until all the end-to-end task deadlines are met.
6.5 Performance Model

The performance model is an extension of the multi-server model (Chapter 4) and the pipeline model (Chapter 5). The model is embedded in a long-term allocator called the modified-task-minimum-response-allocation (MTMRA) algorithm. It is a performance-guaranteeing policy which allocates the modified allocatable tasks to servers at compile time to achieve the following goals:

1. maximise the schedulable utilisation factor for each server,
2. minimise the end-to-end task response times, and
3. minimise the required number of servers,

while meeting the end-to-end deadlines for the given set of tasks. These tasks have a range of execution times and deadlines independent of their periods.

At first, an end-to-end schedulability test algorithm for a pipelined multi-server system is presented. Then the MTMRA algorithm is described. The model calculates the end-to-end task response times and the minimum required number of buffers for each task at each server. It also determines the minimum required number of servers at each stage.

6.5.1 End-to-End Schedulability Test

A set of tasks is *schedulable* in a pipelined multi-server system when it has no throughput failure and no end-to-end response failure. Algorithm 6.1 is a schedulability test algorithm for a given set of tasks in a pipelined multi-server system.
Algorithm 6.1: Schedulability Test for a Pipelined Multi-Server System
{ determines if a set of tasks is schedulable in a pipelined multi-server system }

1. calculate utilisation factor $U_{j,k}$ for the $k^{th}$ server at stage $j$, $1 \leq k \leq N_j$, $1 \leq j \leq m$.

2. if ($\exists_{j,k} U_{j,k} > 1$) then /* server overflow */

   throughput failure occurs and
   some task(s) is(are) not schedulable.

   else

   calculate the worst-case response time ($W_{i,j,k}$) for task $i$ on the $k^{th}$
   server at stage $j$.

   $\left\{ \exists_{i,j,k} B_{i,j,k}^{\text{install}} < \left[ \frac{W_{i,j}}{b_{i,j}} \left( b_{i,j} T_i \right) \right] \right\}$

   then

   /* buffer overflow */

   throughput failure occurs and
   task $i$ is not schedulable.

   else

   calculate the worst-case end-to-end response time ($W_i$) for task $i$.
   calculate the peak-time average end-to-end response time ($A_i$).
   if ((task $i$ is hard) and ($W_i > D_i$)) or
   ((task $i$ is soft) and ($A_i > D_i$)) then /* deadline miss */

   end-to-end response failure occurs and
   task $i$ is not schedulable

   else

   task $i$ is schedulable.

   endif

endif

endif

3. if (all the tasks in the task set are schedulable) then

   the task set is schedulable and
   the system has a guaranteed capacity.

else

   the task set is not schedulable and
   the system does not have a guaranteed capacity.

endif.

4. stop
To determine schedulability, the algorithm first calculates the utilisation factor for every server in the system.

The utilisation factor of the $k^{th}$ server at stage $j$ is:

$$U_{j,k} = \sum_{i=1}^{n} \frac{\psi_{i,j,k} C_{i,j}}{T_{i,j}}$$  \hspace{1cm} (6.1)

where $T_{i,j} = \left[ \frac{C_{i,j}}{b_{i,j} T_i} \right] b_{i,j} T_i$.

Note, in this calculation the original period $T_i$ is virtually lengthened to the modified period $\left[ \frac{C_{i,j}}{b_{i,j} T_i} \right] T_i$ because of the instance distribution (Section 4.4.1). The allocation matrix $\psi_{i,j,k} = 1$ when task $i$ is allocated to the $k^{th}$ server at stage $j$, otherwise $\psi_{i,j,k} = 0$.

In its second step, the worst-case end-to-end response time ($W_i$) for task $i$ is calculated using Equation (5.8) by substituting $\left[ \frac{C_{i,j}}{b_{i,j} T_i} \right] T_i$ for $T_i$ in Equations (5.6) and (5.7). The peak-time average end-to-end response time ($A_i$) for task $i$ is calculated using Equation (5.9) by substituting $\left[ \frac{C_{i,j}}{b_{i,j} T_i} \right] T_i$ for $T_i$ in Equations (5.6) and (5.7).

The second step also requires the calculation of the minimum required number of buffers for each task on every server at every stage. The minimum required number of buffers for task $i$ on the allocated $k^{th}$ server at stage $j$ is:

$$B_{i,j,k} = \frac{W_{i,j}}{b_{i,j} \left( \left[ \frac{C_{i,j}}{b_{i,j} T_i} \right] T_i \right)}.$$ \hspace{1cm} (6.2)
Note that the Equation (6.2) is derived from Equation (5.10) by substituting \[ \frac{C_{t,i}}{b_{t,i}T_i} \] for \( T_i \). Based on the calculation in steps 1 and 2, step 3 determines the schedulability of the set of tasks.

### 6.5.2 Modified-Task-Minimum-Response-Allocation (MTMRA)

The MTMRA algorithm (Algorithm 6.2) optimally allocates instances of tasks to servers at each stage. It consists of three sub-algorithms called the Instance Distribution algorithm ID, the Minimum Response Fit algorithm MRF and the Bottleneck Stage Expansion algorithm BSE. The Instance Distribution algorithm ID divides all original tasks into allocatable tasks (step 1). The Minimum Response Fit algorithm MRF allocates the task to the server which gives the minimum response time at the stage (step 2). The Bottleneck Stage Expansion algorithm BSE determines which stages need additional capacity when the existing system has insufficient capacity (step 3).

First, the MRF algorithm attempts to allocate the modified task to a server that is in use. If the modified task is not allocatable due to server overflow then it is allocated to an empty server (step 2(b)). A task has its optimal schedulable priority (Algorithm 5.2) when it is allocated to the server which executes it with the minimum response time. The end-to-end response times of the tasks are calculated when all tasks are allocated at all stages. If they meet their end-to-end deadlines, the allocation is optimal. Otherwise, the BSE algorithm determines the bottleneck stage which has given the largest preemption time to the task which has missed its deadline is expanded by adding an empty server. Then the algorithm executes again from the step 2(b). However, the algorithm will eventually stop because all tasks are schedulable on empty servers (assumption (6) in Section 6.3). The optimal hardware configuration (OHC) is obtained at the completion of the algorithm where \( \text{OHC} = \{(j, N_j) \mid 1 \leq j \leq m \} \).
Chapter 6: A Pipelined Multi-Server Model

Algorithm 6.2: Modified-Task-Minimum-Response-Allocation (MTMRA)

\{ produces an allocation matrix $\psi$ and the required number of servers $N_j$ at each stage $j$ \}

1. **Instance Distribution (ID) Algorithm**

   - set stage period for task $i$ at stage $j$:
     \[ T_{i,j} = \frac{C_i}{b_{i,j}T_i}. \]
     
   - create the subtasks: $\tau_{i,j,g}(T_{i,j}, C_{i,j})$ for $1 \leq i \leq n$, $1 \leq j \leq m$, $1 \leq g \leq \sum_{1 \leq i \leq n} \frac{C_i}{b_{i,j}T_i}$.

2. **Minimum Response Fit (MRF) Algorithm**

   (a) set the total number of servers at each stage $j$: $N_j = 1$.
     
   (b) for each stage $1 \leq j \leq m$ do
     
     - for each modified task $\tau_{i,j,g}$, $1 \leq i \leq n$, $1 \leq g \leq \sum_{1 \leq i \leq n} \frac{C_i}{b_{i,j}T_i}$ do
       
       - if it is allocatable in an existing server then
         
         allocate the task to the minimum-response-server $k$ and
         
         set the allocation matrix $\psi_{i,j,k} = 1$.
       
       - else
         
         allocate the task to a new server and
         
         set $N_j = N_j + 1$.
         
         set the allocation matrix $\psi_{i,j,k} = 1$.
       
       endif
     
     - calculate the worst-case response time $W_{i,j,g}$ for group $g$ of task $i$ at stage $j$.
     
   endfor
     
   - calculate the worst-case response time for task $i$ at stage $j$:
     \[ W_{i,j} = \max_{1 \leq g \leq \sum_{1 \leq i \leq n} \frac{C_i}{b_{i,j}T_i}} W_{i,j,g}. \]

   endfor

3. **Bottleneck Stage Expansion (BSE) Algorithm**

   - calculate the worst-case end-to-end response time ($W_i$) for task $\tau_i$.
     
   - calculate the peak-time average end-to-end response time ($A_i$).
     
   - if (($task \ \tau_i \ is \ hard$) and ($W_i > D_i$)) or
     
   - (($task \ \tau_i \ is \ soft$) and ($A_i > D_i$)) then /* end-to-end deadline miss */
     
   - end-to-end response failure occurs and task $\tau_i$ is not schedulable.
     
   - set bottleneck stage $J_{\text{max}} = \max_{1 \leq j \leq m} (W_{i,j} - C_{i,j})$.
     
   - install new server at the bottleneck stage: $N_{j_{\text{max}}} = N_{j_{\text{max}}} + 1$.
     
   - go to step 2(b).
   
   else
     
     task $\tau_i$ is schedulable and
     
     task $\tau_i$ is allocated to the optimal server at each stage.
   
   endif

4. total required number of servers for the system $N_s = \sum_{j=1}^{m} N_j$.

5. stop.
The algorithm has the following characteristics to maximise the schedulable utilisation factor of each server while minimising the end-to-end task response times.

1. The minimum response time at each stage leaves the largest slack time for the rest of the stages so that they may have more chance to utilise a used server instead of requiring an empty server.

2. If a task still cannot meet its end-to-end deadline then we can readily allocate empty servers for bottleneck stages.

3. As we assumed that each stage has enough servers to provide to tasks (assumption (1) in Section 6.3), if we assign each task to a dedicated server at each stage then all tasks are schedulable (see assumption (6) in Section 6.3). Therefore, the schedulability of the tasks is already secured. What counts for us then is to maximise the schedulable utilisation of servers. To achieve that, the model tries to minimise the required number of servers at each stage with the Minimum Response Fit (MRF) algorithm while expanding the bottleneck stages when the servers in use cannot schedule the tasks.

The optimal scheduling configuration (OSC) is determined by choosing the one which requires the minimum number of servers \( N_s \) for the system (see step 4 in Algorithm 6.2) among the set of available local scheduling algorithms for each stage. The optimal scheduling configuration can be characterised by \( \text{OSC} = \{(j, sch_j) \mid 1 \leq j \leq m\} \) where \( sch_j \) is the scheduling algorithm at stage \( j \).
6.6 An Example

Table 6.1 shows an example set $TS$ of three hard real-time tasks to illustrate our model. All tasks have the instance blocking factor 1. Note that the end-to-end deadline of task 1 is the same as its period. Tasks 2 and 3 have their end-to-end deadlines larger than their periods.

<table>
<thead>
<tr>
<th>$i$</th>
<th>$T_i$</th>
<th>$C_{i,1}$</th>
<th>$C_{i,2}$</th>
<th>$C_{i,3}$</th>
<th>$D_i$</th>
<th>$HS_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>10</td>
<td>hard</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>15</td>
<td>25</td>
<td>3</td>
<td>60</td>
<td>hard</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>23</td>
<td>30</td>
<td>6</td>
<td>75</td>
<td>hard</td>
</tr>
</tbody>
</table>

**Table 6.1**

A Given Set $TS$ of Tasks

The sum of the execution times of all stages for tasks 2 and 3 are larger than their period, but less than their end-to-end deadlines. Note that the execution times $C_{2,1}$ and $C_{2,2}$ for task $\tau_2$ are larger than its period ($T_2$). We will show how our unified model successfully allocates all these tasks while meeting their deadlines.

For example, stage 2 has three identical servers ($N_2$) of resource type $RY_2$ ($Rtype_2$) with its scheduling algorithm $SH_2$ ($Sch_2$). Each server has enough buffer space for its tasks. These three tasks will be executed on a 3-stage regular pipeline system (Table 6.2). Stage $j$ has $N_j$ servers of resource type $Rtype_j$ and scheduling algorithm $Sch_j$.

The first step in the model is to determine the modified allocatable set $TS'$ of tasks using the algorithm ID in algorithm 6.2. The modified allocatable set of tasks is given in Table 6.3. Note, task 2 (Table 6.1) is divided into two allocatable modified tasks at stage 1, three at stage 2 and one at stage 3 (Table 6.3) with their
arrival periods at these stages of 24, 36 and 12. Note, \( g \) is an instance group number.

The result of applying the end-to-end schedulability test (Algorithm 6.1) to these modified tasks is presented in Table 6.4. Worst-case response times \( W_{ijg} \) are calculated using the algorithms presented for lower-level models in previous
chapters. Note, the worst-case end-to-end response time $W_2$ is infinite ($\infty$) and $W_3$ is 86 ($= 30 + 45 + 11$). Both of them are larger than their end-to-end deadlines 60 ($D_2$) and 75 ($D_3$). Thus, tasks 2 and 3 cannot meet their deadlines. The response time is considered to be infinite when a server overflows.

<table>
<thead>
<tr>
<th>task</th>
<th>stage 1</th>
<th>stage 2</th>
<th>stage 3</th>
<th>end-to-end</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i$</td>
<td>$g$</td>
<td>$W_{i,1,e}$</td>
<td>$W_{i,1}$</td>
<td>$W_{i,2,e}$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>20</td>
<td>32</td>
<td>$\infty$</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>30</td>
<td>45</td>
<td>11</td>
</tr>
</tbody>
</table>

**Table 6.4**

Response Times of Modified Tasks

Thus, it is shown that the set of modified tasks are not schedulable. To make them schedulable, we need additional servers in one or more stages. That is we have to modify the hardware configuration. So, next we determine the optimal hardware configuration using the Algorithm 6.2. Table 6.5 shows the intermediate result from the MRF algorithm.

From the result in Table 6.4, we can see that the modified task $\tau_{2,2,3}$ is not schedulable. However, after the MRF algorithm, this task is schedulable because it is allocated to an empty server at stage 2 (Tables 6.1 and 6.5). This result was achieved by increasing the number of servers at stage 2 from 3 (Table 6.2) to 4. This also results in a reduction of the worst-case response time of task 2 at stage 2.
(w_{2,2}) to 32 and the end-to-end worst-case response time (w_2) to 59. Now task 2 satisfies its deadline (W_2 < D_2).

<table>
<thead>
<tr>
<th>task</th>
<th>stage 1</th>
<th>stage 2</th>
<th>stage 3</th>
<th>end-to-end</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>g</td>
<td>W_{1,i,g}</td>
<td>W_{1,i}</td>
<td>W_{2,i,g}</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>20</td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>22</td>
<td>22</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>30</td>
<td>30</td>
<td>45</td>
</tr>
</tbody>
</table>

**Table 6.5**

Response Times of MRF Result

However, the problem of task 3 still remains with W_3 > D_3. To solve this problem, the BSE algorithm (see Algorithm 6.2) is executed. Table 6.6 shows the result of the BSE algorithm. The BSE algorithm identifies that stage 2 is the bottleneck because W_{3,2,1} (Table 6.5) has the largest preemption time (45–30) (Table 6.1). Thus, the algorithm installs a new server at stage 2 and allocates task τ_{3,2,1} to the empty server. Now the W_{3,2,1} is reduced 30 (Table 6.6) and W_3 is reduced 71 which is less than D_3 (75). Thus, all the tasks met their deadlines.

We have shown that hardware configuration defined in table 6.2 is unable to schedule the workload defined in Table 6.1. To solve this problem, the MRF algorithm added a new server at stage 2, which had a server overflow problem. However, Task τ_3 still had an end-to-end deadline miss problem. Thus the model again added a new server to the bottleneck stage 2 through the BSE algorithm. As a result, the minimum required hardware configuration was found (Table 6.7). We
can see from Table 6.7 that the bottleneck stage 2 (Table 6.2) has expanded by 2 servers to schedule the given set $TS$ of tasks in Table 6.1.

<table>
<thead>
<tr>
<th>task</th>
<th>stage 1</th>
<th>stage 2</th>
<th>stage 3</th>
<th>end-to-end</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>g</td>
<td>$W_{i,1,r}$</td>
<td>$W_{i,1,l}$</td>
<td>$W_{i,2,r}$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>20</td>
<td>30</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>22</td>
<td>22</td>
<td>25</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>30</td>
<td>30</td>
<td>30</td>
</tr>
</tbody>
</table>

**Table 6.6**
Response Times of BSE Result

<table>
<thead>
<tr>
<th>$j$</th>
<th>$Rtype_j$</th>
<th>$Sch_j$</th>
<th>$N_j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>RY1</td>
<td>SH1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>RY2</td>
<td>SH2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>RY3</td>
<td>SH3</td>
<td>2</td>
</tr>
</tbody>
</table>

**Table 6.7**
Minimum Required Hardware Configuration

Now that we have found the optimal hardware for a given scheduling configuration. The next step is to find the optimal scheduling configuration (OSC), *i.e.*, to determine which scheduling algorithm from a set of scheduling algorithms available at each stage. To do this, we schedule the workload (Table 6.1) with
combinations of scheduling configurations. The scheduling configuration which requires the minimum number of servers for the system by Algorithm 6.2 will be the optimal one.

6.7 Discussion

We proposed a high-level performance model which can determine the optimum hardware configuration and the optimum scheduling configuration in a multi-stage multi-server real-time systems. This model is based on the lower-level performance models developed in previous chapters. However, our model assumes that the unit costs of a server at all stages are the same but in reality a bottleneck stage may have cheaper servers than other stages. Thus, the Bottleneck Stage Expansion (BSE) algorithm does not consider the minimal cost expansion in general.

The model can be extended to include the intra-task structure with AND/OR precedence constraints (Gillies and Liu, 1990; Bettati et al., 1991), where an AND subtask begins its execution when all its predecessors are complete while an OR subtask is ready to execute when just one of its predecessors is complete. If we include another level of group for these AND/OR subtasks to Algorithm 6.2, we may calculate the worst-case response time for the previous stage of an AND subtask by taking the maximum value of the worst-case response times for all the predecessors. Likewise, the worst-case response time for the previous stage of an OR subtask may be the minimum value of the worst-case response times for all the predecessors.
Chapter 7
Conclusions

As discussed in Chapters 1 and 2, we need a performance model for a general parallel real-time system which can estimate

- the schedulability of the proposed workload,
- the worst-case and peak-time average end-to-end response times for each task,
- the buffer requirements, and
- the optimal system capacity.

In this thesis, I have developed such a performance model. This development involved developing four performance models for increasingly complex systems. The fourth and most general model is based on the other models. These models are for:

1. single server systems (Chapter 3),
2. multi-server systems (Chapter 4),
3. pipeline systems (Chapter 5), and
4. pipelined multi-server systems (Chapter 6).

All these models except the pipelined multi-server model are the extensions of existing models to a range of task parameters, resource types, scheduling algorithms and hardware configurations. The single server model is the extension of the work of Joseph and Pandya (1986). Their model was restricted to preemptive fixed priority scheduling and tasks whose deadlines do not exceed their
periods. My extension is inspired by other work such as

- Liu and Layland's (1973) critical instant theorem,
- Lehoczky's (1990) busy period theorem,
- Tindell et al.'s (1992, 1994) response time model for preemptive fixed priority scheduling,
- Tindell and Burns' (1993) response time model for non-preemptive fixed priority scheduling, and

My single server model is an extension of Joseph and Pandya's model to include:

a. worst-case response times longer than periods,
b. non-preemptive scheduling,
c. individual task response times,
d. soft real-time tasks, and
e. deadline driven scheduling.

The first type of parallel systems to be considered is a multi-server system. Based upon the single server model, I built a multi-server model with a first-fit-allocation algorithm similar to the rate-monotonic-first-fit-allocation algorithm of Dhall and Liu (1978). My performance model is embedded in the first-fit-allocation algorithm which determines the required number of servers to schedule a set of tasks. My model handles the following situations, which are not addressed by Dhall and Liu:

a. execution times longer than periods,
b. deadlines less than or greater than periods,
c. soft real-time tasks, and
d. deadline driven scheduling.

The second type of parallel systems to be considered is a pipeline system. My model for a regular pipeline is an extension of:

- Audsley's (1991) optimal priority assignment algorithm, and

My model introduces the optimal instance blocking concept for speed balancing through stages and the optimal modified release time concept for regular pipelining. The extensions in my model are:

a. minimum-response-priority-assignment (MRPA) algorithm,
b. deadlines less than or greater than periods,
c. end-to-end task response times,
d. soft real-time tasks, and
e. deadline driven scheduling.

Finally, a pipelined multi-server model is developed as the highest-level abstract model by combining the multi-server model and the pipeline model. It uses all the lower-level models and estimates the optimal schedulable system configuration. This new model can determine:

a. the performance,
b. the optimal hardware configuration, and
c. the optimal scheduling configuration for multi-stage multi-server systems.
In summary, my contributions to the performance modelling of parallel real-time systems in this thesis are:

1. a general response time analysis which
   - can estimate the worst-case response time for hard real-time tasks and
     the peak-time average response time for soft real-time tasks,
   - is applicable to fixed priority scheduling and to deadline driven scheduling, and
   - is applicable for both preemptive and non-preemptive resources,
2. a general schedulability test for a set of tasks on a single server with a range of task parameters, resource types and scheduling algorithms,
3. a best-priority-assignment algorithm which assigns an inserted task the highest schedulable priority to achieve the minimum schedulable response time for the task,
4. an estimation of the optimal server capacity and spare task execution times,
5. an instance distribution model for tasks which have longer execution times than their periods,
6. a modified-task-first-fit-allocation (MTFFA) algorithm to determine the required number of servers in a pipeline system to schedule a set of periodic tasks which may have longer execution times than their periods,
7. an optimal instance blocking model for speed balancing through all stages in a regular pipeline,
8. a regular pipelining model to calculate the end-to-end response times and the buffer space for tasks, and
9. a modified-task-minimum-response-allocation (MTMRA) algorithm to determine the required number of servers in a pipelined multi-server system to schedule a set of periodic tasks with end-to-end deadlines.
Chapter 7: Conclusion

The important results of the thesis are as follows.

- These models are theoretical rather than practical because they estimate the upper-bound performance indices based upon a tightly controlled situation (e.g., regular pipelining).
- These models are mathematically tractable.
- The basic algorithms of these models are verified by discrete-time simulation for preemptive fixed priority scheduling and preemptive deadline driven scheduling (Appendix D).
- These models enable us to:
  (a) determine the schedulability of a set of tasks in one of the four types of hardware configuration (i.e., single server, multi-server, pipeline, pipelined multi-server),
  (b) estimate the optimal system capacity both for a single server or for multiple servers when the task is not schedulable,
  (c) allocate an inserted task to a server at each stage to minimise its worst-case stage response time,
  (d) assign the best schedulable priority to an inserted task to minimise its worst-case response time at a server,
  (e) estimate the worst-case and the peak-time average response time for each task, and
  (f) estimate the spare execution time for each task.
- The advantages of these models are that they allow us to
  (a) calculate the performance indices easily and quickly by mathematical equations rather than simulation, and
  (b) calculate the upper-bound values on performance indices to guarantee robust design of real-time systems.
Chapter 7: Conclusion

However, these models have their limitations. The limitations of my models which have to be overcome through future studies are discussed as follows.

- The response time analysis is restricted to four scheduling algorithms which are preemptive/non-preemptive fixed priority scheduling and preemptive/non-preemptive deadline driven scheduling. Thus, the analysis may need to be extended to include other scheduling algorithms (e.g. FIFO) in case of a restricted hardware situation, even if those algorithms may not be appropriate to real-time system. It can be extended to include a more detailed analysis of tasks (see, Leinbaugh and Yamini, 1982) to be more precise.

- The calculation algorithm for the optimal system capacity and spare task execution times is restricted to a system with hard real-time tasks only. Extension of the algorithm to be applicable to soft real-time tasks is needed for a hybrid system where hard and soft real-time tasks are running concurrently. Spare task periods cannot be estimated by this algorithm. The algorithms to calculate them are to be studied in the future.

- My model assumes that the unit costs of a server at all stages are the same. However, a bottleneck stage may have cheaper servers than other stages in reality. Thus, the Bottleneck Stage Expansion (BSE) subalgorithm in Chapter 6 does not consider the minimal cost expansion in general. This extension is to be studied in the future.

- My model does not consider the intra-task structure with AND/OR precedence constraints, where an AND subtask begins its execution when all its predecessors are complete while an OR subtask is ready to execute when just one of its predecessors is complete. It needs to be extended to include those parameters.

- My model does not allow re-visit of a task at the same stage. For example, it
does not consider the complex situations where a task executes at a CPU then at a disk and again at the CPU etc. The model needs an extension for these cases.

- Since our model adopts the regular pipeline model in Chapter 5, the worst-case end-to-end response time and the peak-time average end-to-end response time are calculated in a modelled situation where every instances of tasks suffers the worst-case response time at all stages which is the upper-bound. This gives a greater peak-time average response time for soft real-time tasks than in actual non-regular pipeline situations. Hence, the model is tractable but theoretical rather than practical.

- Our model adopts a rather pessimistic evaluation as the worst-case end-to-end response time is calculated to be the sum of the worst-case stage response times which is the upper-bound. This model calculation may require more capacity than is actually required. However, it is NP-hard to determine the instant for each task which causes the worst-case end-to-end response time in multi-stage multi-server systems. Thus, the precise estimation of the end-to-end task response times is an open question.

Although this thesis is rather abstract and theoretical, it has brought up several issues and problems in the modelling of parallel real-time systems for their performance analysis. It has proposed or suggested some modelling solutions to them. They may be used as a basis to more precise parallel real-time models which will appear in the future. Performance modelling for parallel real-time systems with a multi-stage multi-server architecture is still to be explored.
### Appendix A

#### Notation Summary

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_i$</td>
<td>peak-time average (end-to-end) response time for task $i$</td>
</tr>
<tr>
<td>$A_{i,k}$</td>
<td>peak-time average response time for task $i$ on server $k$</td>
</tr>
<tr>
<td>$B_i$</td>
<td>minimum required number of buffers for task $i$</td>
</tr>
<tr>
<td>$B_{i,j}$</td>
<td>minimum required number of buffers for task $i$ at stage $j$</td>
</tr>
<tr>
<td>$b_{i,j}$</td>
<td>instance blocking factor for task $i$ at stage $j$</td>
</tr>
<tr>
<td>$B_{i,j,k}$</td>
<td>minimum required number of buffers for task $i$ on the $k^{th}$ server at stage $j$</td>
</tr>
<tr>
<td>$B_{i,k}$</td>
<td>minimum required number of buffers for task $i$ on server $k$</td>
</tr>
<tr>
<td>$B_i^{install}$</td>
<td>current installed number of buffers for task $i$</td>
</tr>
<tr>
<td>$\text{Busy}_{i,q}(t,t')$</td>
<td>calculates the finishing time of level-$i,q$ busy period during the period $(t, t')$ - Equation (3.8)</td>
</tr>
<tr>
<td>$C_i'$</td>
<td>maximum schedulable execution time for task $i$</td>
</tr>
<tr>
<td>$C_i$</td>
<td>execution time for task $i$</td>
</tr>
<tr>
<td>$\Delta C_i$</td>
<td>spare execution time for task $i$</td>
</tr>
<tr>
<td>$C_{i,j}$</td>
<td>execution time for task $i$ at stage $j$</td>
</tr>
<tr>
<td>$C_{i}^{block}$</td>
<td>the worst-case blocking time for task $i$</td>
</tr>
<tr>
<td>$C_{i}^{non-preempt}$</td>
<td>non-preemptable part of the execution time for task $i$</td>
</tr>
<tr>
<td>$C_{i}^{preempt}$</td>
<td>preemptable part of the execution time for task $i$</td>
</tr>
<tr>
<td>$C_{i}^{sw}$</td>
<td>worst-case task switching overhead time for task $i$</td>
</tr>
<tr>
<td>$C_{i}^{virtual}$</td>
<td>virtual execution time for task $i$</td>
</tr>
<tr>
<td>$\text{Complete}_{i}(t,t')$</td>
<td>calculates the completion time of task $i$ during the period $(t,t')$ - Equation (3.6)</td>
</tr>
<tr>
<td>Term</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Delay(<em>{i,q}(t,t')</em>)</td>
<td>calculates the total execution time by the tasks with the same or higher priority than instance (I_{i,q}_) - Equation (3.8)</td>
</tr>
<tr>
<td>(D_{i})</td>
<td>deadline of task (i)</td>
</tr>
<tr>
<td>(D_{i,j})</td>
<td>subdeadline for task (i) at stage (j)</td>
</tr>
<tr>
<td>(d_{i,i,q})</td>
<td>effective deadline for instance (I_{i,q}) at stage (j)</td>
</tr>
<tr>
<td>(d_{i,i,q buffer})</td>
<td>effective buffer overflow deadline for instance (I_{i,q}) at stage (j)</td>
</tr>
<tr>
<td>(d_{i,i,q server})</td>
<td>effective server overflow deadline for instance (I_{i,q}) at stage (j)</td>
</tr>
<tr>
<td>(d_{i,i,q user})</td>
<td>effective user specified deadline for instance (I_{i,q}) at stage (j)</td>
</tr>
<tr>
<td>(d_{i,q})</td>
<td>effective deadline for instance (I_{i,q})</td>
</tr>
<tr>
<td>(ed(i,q))</td>
<td>the earlier deadline tasks than instance (I_{i,q})</td>
</tr>
<tr>
<td>(E_{i,q})</td>
<td>completion time of instance (I_{i,q})</td>
</tr>
<tr>
<td>(Exec_j(t,t'))</td>
<td>calculates the total execution time of task (j) during the period ((t,t')) - Equation (3.3)</td>
</tr>
<tr>
<td>(f_i)</td>
<td>execution scaling factor for task (i)</td>
</tr>
<tr>
<td>(f_{i,q})</td>
<td>execution scaling factor for instance (I_{i,q})</td>
</tr>
<tr>
<td>(f_s)</td>
<td>execution scaling factor for a given set (s) of tasks</td>
</tr>
<tr>
<td>(g)</td>
<td>instance group number</td>
</tr>
<tr>
<td>(hp(i))</td>
<td>higher priority tasks than task (i)</td>
</tr>
<tr>
<td>(HS_i)</td>
<td>a flag of hard for hard real-time or soft for soft real-time task (i)</td>
</tr>
<tr>
<td>(i)</td>
<td>task number</td>
</tr>
<tr>
<td>(i,j)</td>
<td>indicating task (i) at stage (j)</td>
</tr>
<tr>
<td>(i,j,g)</td>
<td>indicating an instance group (g) of task (i) at stage (j)</td>
</tr>
<tr>
<td>(i,j,k)</td>
<td>indicating task (i) on the (k^{th}) server at stage (j)</td>
</tr>
<tr>
<td>(i,j,q)</td>
<td>indicating instance (I_{i,q}) at stage (j)</td>
</tr>
<tr>
<td>(I_{i,q})</td>
<td>the (q^{th}) instance of task (i)</td>
</tr>
<tr>
<td><strong>Inputs</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>------------</td>
<td>----------------</td>
</tr>
<tr>
<td>$j(t,t')$</td>
<td>calculates the number of instances of task $j$ during the period $(t,t')$ - Equation (3.2)</td>
</tr>
<tr>
<td>$j$</td>
<td>stage number</td>
</tr>
<tr>
<td>$k$</td>
<td>server number</td>
</tr>
<tr>
<td>$m$</td>
<td>total number of stages</td>
</tr>
<tr>
<td>$M_i$</td>
<td>minimum required buffer space for task $i$</td>
</tr>
<tr>
<td>$M_k$</td>
<td>minimum required buffer space for server $k$</td>
</tr>
<tr>
<td>$mod$</td>
<td>mathematical function modulus</td>
</tr>
<tr>
<td>$M_s$</td>
<td>minimum required buffer space for a system</td>
</tr>
<tr>
<td>$N$</td>
<td>minimum required number of servers</td>
</tr>
<tr>
<td>$n$</td>
<td>total number of tasks</td>
</tr>
<tr>
<td>$n_h$</td>
<td>number of hard real-time tasks</td>
</tr>
<tr>
<td>$N_i$</td>
<td>required number of servers for task $i$</td>
</tr>
<tr>
<td>$N_j$</td>
<td>required number of servers at stage $j$</td>
</tr>
<tr>
<td>$N_s$</td>
<td>total required number of servers for the system</td>
</tr>
<tr>
<td>$P$</td>
<td>peak-time interval</td>
</tr>
<tr>
<td>$p_k$</td>
<td>server $k$</td>
</tr>
<tr>
<td>Preempt$_i(t,t')$</td>
<td>calculates the total preemption time of task $i$ by the higher priority tasks during the period $(t,t')$ - Equation (3.4)</td>
</tr>
<tr>
<td>$q$</td>
<td>instance number</td>
</tr>
<tr>
<td>$R_{i,i,a}$</td>
<td>response time for instance $I_{i,a}$ at stage $j$</td>
</tr>
<tr>
<td>$R_{i,a}$</td>
<td>end-to-end response time for instance $I_{i,a}$</td>
</tr>
<tr>
<td>Rtype$_j$</td>
<td>resource type of stage $j$</td>
</tr>
<tr>
<td>$S$</td>
<td>underlying server capacity</td>
</tr>
<tr>
<td>$S'$</td>
<td>optimal server capacity</td>
</tr>
<tr>
<td>Sch$_j$</td>
<td>scheduling algorithm for stage $j$</td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>$S_{i,q}$</td>
<td>arrival time for the $q^{th}$ instance of task $i$</td>
</tr>
<tr>
<td>$T_i$</td>
<td>period (or minimum interarrival time) of task $i$</td>
</tr>
<tr>
<td>$\tau_i$</td>
<td>task $i$</td>
</tr>
<tr>
<td>$T_{i,g}$</td>
<td>period of modified task $\tau_{i,g}$</td>
</tr>
<tr>
<td>$\tau_{i,g}$</td>
<td>modified task which is instance group $g$ of task $i$</td>
</tr>
<tr>
<td>$T_{i,j}$</td>
<td>period for task $i$ at stage $j$</td>
</tr>
<tr>
<td>$\tau_{i,j,g}$</td>
<td>a modified task which is instance group $g$ of task $i$ at stage $j$</td>
</tr>
<tr>
<td>$T_i^{virtual}$</td>
<td>virtual period for task $i$</td>
</tr>
<tr>
<td>$U_i$</td>
<td>utilisation factor for task $i$</td>
</tr>
<tr>
<td>$U_{i,g}$</td>
<td>utilisation factor for modified task $\tau_{i,g}$</td>
</tr>
<tr>
<td>$U_j$</td>
<td>utilisation factor for a server at stage $j$</td>
</tr>
<tr>
<td>$U_{i,k}$</td>
<td>utilisation factor for the $k^{th}$ server at stage $j$</td>
</tr>
<tr>
<td>$U_s$</td>
<td>utilisation factor for server $s$</td>
</tr>
<tr>
<td>$V_{i,i,q}$</td>
<td>instance blocking delay for instance $I_{i,q}$ at stage $j$</td>
</tr>
<tr>
<td>$W_i$</td>
<td>worst-case (end-to-end) response time for task $i$</td>
</tr>
<tr>
<td>$W_{i,j}$</td>
<td>worst-case response time for task $i$ at stage $j$</td>
</tr>
<tr>
<td>$W_{i,j,g}$</td>
<td>worst-case response time for group $g$ of task $i$ at stage $j$</td>
</tr>
<tr>
<td>$W_{i,k}$</td>
<td>worst-case response time for task $i$ on server $k$</td>
</tr>
<tr>
<td>$X_i$</td>
<td>throughput of task $i$</td>
</tr>
<tr>
<td>$Z_i$</td>
<td>size of a buffer for task $i$</td>
</tr>
<tr>
<td>$\Phi_{i,q}$</td>
<td>upper bound completion time for instance $I_{i,q}$</td>
</tr>
<tr>
<td>$\psi_{i,j,k}$</td>
<td>1 if task $i$ is allocated to the $k^{th}$ server at stage $j$, otherwise 0</td>
</tr>
<tr>
<td>$\psi_{i,k}$</td>
<td>1 if task $i$ is allocated to server $k$, otherwise 0</td>
</tr>
<tr>
<td>$\lceil x \rceil$</td>
<td>ceiling function which denotes the smallest integer greater than or equal to $x$</td>
</tr>
<tr>
<td>\lfloor x \rfloor</td>
<td>floor function which denotes the greatest integer equal to or less than x</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>\lfloor x \rfloor</td>
<td>\begin{cases} 0, &amp; x \leq 0. \ x-1, &amp; (x &gt; 0) \land (x = \lfloor x \rfloor). \ \lfloor x \rfloor, &amp; otherwise. \end{cases} - Equation (3.9)</td>
</tr>
</tbody>
</table>
### Appendix B

**Terminology Summary**

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Absolute deadline</strong></td>
<td>When an instance of a task has to complete its processing.</td>
</tr>
<tr>
<td><strong>Allocatable task</strong></td>
<td>A task whose utilisation factor does not exceed the unity.</td>
</tr>
<tr>
<td><strong>Allocation matrix</strong></td>
<td>A matrix indicating which task is assigned to which server.</td>
</tr>
<tr>
<td><strong>Analytic model</strong></td>
<td>A mathematical model, <em>i.e.</em>, not a simulation model.</td>
</tr>
<tr>
<td><strong>Arbitrary task deadline</strong></td>
<td>A deadline which is independent of the period of a task.</td>
</tr>
<tr>
<td><strong>Arrival rate</strong></td>
<td>Number of arrivals of instances of a task per unit time.</td>
</tr>
<tr>
<td><strong>Asynchronous input</strong></td>
<td>Input which occurs independently of the clock.</td>
</tr>
<tr>
<td><strong>Batch system</strong></td>
<td>A computer system which executes a series of jobs in a sequential manner.</td>
</tr>
<tr>
<td><strong>Best-priority-assignment (BPA)</strong></td>
<td>Assignment of priorities to tasks which not only guarantees the schedulability of all tasks but also gives the minimum schedulable response times for tasks.</td>
</tr>
<tr>
<td><strong>Blocking time</strong></td>
<td>A time interval during which a higher priority task waits for a lower priority task to complete the execution of its non-preemptable part.</td>
</tr>
<tr>
<td><strong>Bottleneck stage expansion (BSE)</strong></td>
<td>A policy that the bottleneck stage expands its capacity when the end-to-end deadline of a task is not met.</td>
</tr>
</tbody>
</table>
Appendices

**Buffer**
A storage device used to compensate for a difference in data handling rates when transmitting data from one stage to another, or a place to hold the input data to a task while the task is waiting to execute.

**Buffer overflow**
The state where available buffer space cannot hold some input data because the amount of the space is not enough.

**Buffer overflow deadline**
The deadline by when an instance of a task must complete so that it does not cause buffer overflow.

**Buffer requirement**
The minimum required amount of buffer space which guarantees that no buffer overflows.

**Buffer space**
The size of the buffer area used to hold input data.

**Capacity planning**
Estimating the required capacity of a system for a given workload.

**Cluster of blocks**
A series of consecutive disk blocks whose access time is not preemptable.

**Common scaling factor**
A scaling factor of the execution time which applies to all tasks.

**Completion rate**
Number of completions of instances of a task per unit time.

**Concurrent servers**
Multiple identical servers which execute tasks in parallel.

**Context switching time**
The time a CPU takes to switch when a higher-priority task preempts a lower-priority task.

**CPU**
Central processing unit.
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical instant</td>
<td>A time instant when all tasks arrive at the system simultaneously.</td>
</tr>
<tr>
<td>Critical section</td>
<td>The section of code which cannot be interrupted when it is processed.</td>
</tr>
<tr>
<td>Data partitioning</td>
<td>Partitioning of instances of a task each of which executes on independent input data.</td>
</tr>
<tr>
<td>Deadline</td>
<td>Required upper bound response time of a task.</td>
</tr>
<tr>
<td>Deadline balanced capacity</td>
<td>The capacity where any decrease will cause a task to be unschedulable.</td>
</tr>
<tr>
<td>Deadline miss</td>
<td>A state when the response time of a task is longer than its deadline.</td>
</tr>
<tr>
<td>Deadline-monotonic-scheduling</td>
<td>A preemptive fixed priority scheduling policy which assigns a higher priority to a task with a shorter deadline. The task deadlines are equal to or less than the task periods.</td>
</tr>
<tr>
<td>Deterministic</td>
<td>A state when there are unique outputs for a given set of inputs, <em>i.e.</em>, not probabilistic.</td>
</tr>
<tr>
<td>Discrete-time</td>
<td>Time which is incremented by the next tick of the clock with natural numbers, <em>i.e.</em>, not continuous time.</td>
</tr>
<tr>
<td>Dynamic scheduling</td>
<td>Scheduling which is used at system run-time.</td>
</tr>
<tr>
<td>Effective deadline</td>
<td>An absolute deadline for an instance of a task which meet the user specified deadline and server overflow deadline and the buffer overflow deadline.</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>End-to-end response time</strong></td>
<td>The response time from the arrival time at the first stage to the departure time at the last stage.</td>
</tr>
<tr>
<td><strong>End-to-end schedulability</strong></td>
<td>A test to see if all tasks meet their end-to-end deadlines.</td>
</tr>
<tr>
<td><strong>Exact analysis</strong></td>
<td>A calculation of task response times with a mathematical solution.</td>
</tr>
<tr>
<td><strong>Execution map</strong></td>
<td>The simulated sketch of the execution of tasks by a CPU.</td>
</tr>
<tr>
<td><strong>Execution scaling factor</strong></td>
<td>The ratio of the maximum schedulable execution time factor for a task to its current execution time.</td>
</tr>
<tr>
<td><strong>External deadline</strong></td>
<td>User specified deadline.</td>
</tr>
<tr>
<td><strong>Feasibility interval</strong></td>
<td>The time interval during which the schedulability test is performed.</td>
</tr>
<tr>
<td><strong>FIFO</strong></td>
<td>First-in-first-out.</td>
</tr>
<tr>
<td><strong>First-fit</strong></td>
<td>Examine servers in sequential order and the first server which can schedule the task is selected.</td>
</tr>
<tr>
<td><strong>Fixed priority</strong></td>
<td>Priority of a task is fixed throughout its execution.</td>
</tr>
<tr>
<td><strong>Flow balance</strong></td>
<td>The state of equality between input rate and output rate.</td>
</tr>
<tr>
<td><strong>Fully utilised server</strong></td>
<td>A server on which any increase in the execution time of a task will make the given set of tasks not schedulable.</td>
</tr>
<tr>
<td><strong>Function partitioning</strong></td>
<td>Resource partitioning.</td>
</tr>
<tr>
<td><strong>GCC</strong></td>
<td>Guaranteed CPU capacity.</td>
</tr>
<tr>
<td><strong>General-purpose computer system</strong></td>
<td>A computer system in which non-real-time tasks are running.</td>
</tr>
<tr>
<td><strong>Guaranteed capacity</strong></td>
<td>The capacity which guarantees that response times of all tasks are equal to or less than their deadlines.</td>
</tr>
<tr>
<td><strong>Guaranteed scheduling</strong></td>
<td>The scheduling which guarantees that all tasks are scheduled.</td>
</tr>
<tr>
<td><strong>Hard deadline</strong></td>
<td>The deadline for a hard real-time task.</td>
</tr>
<tr>
<td><strong>Hard real-time task</strong></td>
<td>A task whose worst-case response time must be less than or equal to its deadline.</td>
</tr>
<tr>
<td><strong>High-level Unified framework</strong></td>
<td>The performance model which is applicable to a range of task parameters, resource types and scheduling algorithms by using layered lower-level models.</td>
</tr>
<tr>
<td><strong>Higher-priority busy period</strong></td>
<td>The period during which the higher priority tasks are processed.</td>
</tr>
<tr>
<td><strong>Higher-priority idle period</strong></td>
<td>The period during which no higher priority task is processed.</td>
</tr>
<tr>
<td><strong>Higher-priority preemption zone</strong></td>
<td>The period during which the higher priority tasks are processed, preempts the lower priority tasks.</td>
</tr>
<tr>
<td><strong>ID</strong></td>
<td>Instance distribution.</td>
</tr>
<tr>
<td><strong>Input rate</strong></td>
<td>Arrival rate.</td>
</tr>
<tr>
<td><strong>Input request</strong></td>
<td>A task invocation by an input.</td>
</tr>
<tr>
<td><strong>Instance</strong></td>
<td>One invocation of a task to process an input. An instance will continually require to use a server until its completion, i.e., instances do not suspend themselves.</td>
</tr>
<tr>
<td><strong>Instance blocking</strong></td>
<td>Storing a number of instances in a buffer to be processed on a multi-instance basis.</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Instance blocking delay</strong></td>
<td>The time to store the instances in the buffer.</td>
</tr>
<tr>
<td><strong>Instance blocking factor</strong></td>
<td>The number of instances to be stored in a buffer.</td>
</tr>
<tr>
<td><strong>Instance distribution</strong></td>
<td>Instances of a task are distributed to multiple servers to make it schedulable.</td>
</tr>
<tr>
<td><strong>Instance group</strong></td>
<td>An allocatable group of instances of a task.</td>
</tr>
<tr>
<td><strong>Internal deadline</strong></td>
<td>Server overflow and buffer overflow deadline.</td>
</tr>
<tr>
<td><strong>Job allocator</strong></td>
<td>A software component which assigns instances of tasks to servers.</td>
</tr>
<tr>
<td><strong>LCM</strong></td>
<td>Least common multiple.</td>
</tr>
<tr>
<td><strong>Level-(i,q) busy period</strong></td>
<td>The period during which instance ( I_{i,q} ) (the ( q^{th} ) instance of task ( i )) and the higher priority tasks are processed.</td>
</tr>
<tr>
<td><strong>Level-i busy period</strong></td>
<td>The period during which task ( i ) and the higher priority tasks are processed.</td>
</tr>
<tr>
<td><strong>Local scheduler</strong></td>
<td>The process which schedules instances of tasks within a server.</td>
</tr>
<tr>
<td><strong>Lower-level model</strong></td>
<td>The sub-model which is used for a higher-level models.</td>
</tr>
<tr>
<td><strong>Marginal execution time</strong></td>
<td>Spare execution time.</td>
</tr>
<tr>
<td><strong>Maximum schedulable execution time</strong></td>
<td>The maximum execution time of a task which guarantees the schedulability of all tasks in the system.</td>
</tr>
<tr>
<td><strong>Maximum schedulable utilisation</strong></td>
<td>The maximum utilisation factor for a system which guarantees all task deadlines.</td>
</tr>
<tr>
<td><strong>Minimum interarrival time</strong></td>
<td>The minimum interval between the two consecutive input.</td>
</tr>
<tr>
<td><strong>Minimum response fit</strong></td>
<td>An allocation policy which allocates a task to the server which produces the minimum response time.</td>
</tr>
<tr>
<td><strong>Minimum schedulable response time</strong></td>
<td>The minimum possible response time for a task which guarantees the schedulability of all tasks in the system.</td>
</tr>
<tr>
<td><strong>Modified release time</strong></td>
<td>The controlled release time at each stage for regular pipelining.</td>
</tr>
<tr>
<td><strong>Modified task</strong></td>
<td>An allocatable group of instances of a task whose utilisation factor does not exceed unity.</td>
</tr>
<tr>
<td><strong>MRF</strong></td>
<td>Minimum response fit.</td>
</tr>
<tr>
<td><strong>MRPA</strong></td>
<td>Minimum-response-priority-assignment.</td>
</tr>
<tr>
<td><strong>MTFFA</strong></td>
<td>Modified-task-first-fit-allocation.</td>
</tr>
<tr>
<td><strong>MTMRA</strong></td>
<td>Modified-task-minimum-response-allocation.</td>
</tr>
<tr>
<td><strong>Multi-processor</strong></td>
<td>Multiple processors executing tasks concurrently.</td>
</tr>
<tr>
<td><strong>Multi-server</strong></td>
<td>Multiple identical servers at a stage.</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Multi-stage multi-server system</strong></td>
<td>A system of pipelined stages where each stage has multiple identical servers.</td>
</tr>
<tr>
<td><strong>Multi-staging</strong></td>
<td>An architecture in which independent resources are partitioned into a pipelined multiple stages.</td>
</tr>
<tr>
<td><strong>NIL</strong></td>
<td>No input loss.</td>
</tr>
<tr>
<td><strong>Non-preemptive</strong></td>
<td>A status when a task cannot be preempted by higher priority tasks while it is executing.</td>
</tr>
<tr>
<td><strong>NP-hard</strong></td>
<td>A problem which cannot be solved exactly in polynomial time, <em>i.e.</em>, is intractable.</td>
</tr>
<tr>
<td><strong>Optimal deadline</strong></td>
<td>The minimum schedulable deadline.</td>
</tr>
<tr>
<td><strong>Optimal hardware configuration</strong></td>
<td>The hardware configuration which requires the minimum number of servers to schedule a set of tasks.</td>
</tr>
<tr>
<td><strong>Optimal priority assignment</strong></td>
<td>Assignment of priorities for tasks which guarantees the schedulability of all tasks.</td>
</tr>
<tr>
<td><strong>Optimal scheduling algorithm</strong></td>
<td>The scheduling algorithm which requires the minimum number of servers to schedule a set of tasks.</td>
</tr>
<tr>
<td><strong>Optimal server capacity</strong></td>
<td>The minimum schedulable server capacity.</td>
</tr>
<tr>
<td><strong>Output rate</strong></td>
<td>Completion rate.</td>
</tr>
<tr>
<td><strong>Over-period buffering</strong></td>
<td>Buffering the subsequent instances of a task when the response time of an instance of a task exceeds its period.</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Overrun</strong></td>
<td>Overflow.</td>
</tr>
<tr>
<td><strong>Parallel processing</strong></td>
<td>Executing multiple tasks concurrently on a multiple servers and/or at multiple stages.</td>
</tr>
<tr>
<td><strong>Peak-time interval</strong></td>
<td>The least common multiple of all task periods.</td>
</tr>
<tr>
<td><strong>Performance goal</strong></td>
<td>Goal to meet the performance requirements of the system.</td>
</tr>
<tr>
<td><strong>Performance indices</strong></td>
<td>The indices which indicate the performance of tasks and system.</td>
</tr>
<tr>
<td><strong>Performance model</strong></td>
<td>A model which takes a system and a workload as input and produces performance indices.</td>
</tr>
<tr>
<td><strong>Performance modelling</strong></td>
<td>The process of building a model which produces the performance indices of a system.</td>
</tr>
<tr>
<td><strong>Periodic task</strong></td>
<td>An infinite series of the same type of instances occurring at regular intervals.</td>
</tr>
<tr>
<td><strong>Pipeline</strong></td>
<td>A series of stages where the output of the previous stage is the input of the next stage.</td>
</tr>
<tr>
<td><strong>Pipeline parallelism</strong></td>
<td>Technique of executing tasks through a series of stages where all the stages perform their independent functions simultaneously.</td>
</tr>
<tr>
<td><strong>Precedence constraint</strong></td>
<td>A constraint on the order of task executions.</td>
</tr>
<tr>
<td><strong>Predecessor</strong></td>
<td>A task which must complete before the next task begins.</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Preemptive</td>
<td>A status when a task can be preempted when a higher priority task arrives.</td>
</tr>
<tr>
<td>Probabilistic model</td>
<td>An analytical model which uses the probability theory to produce the indices.</td>
</tr>
<tr>
<td>Producer-consumer</td>
<td>A circular buffer model where a producer fills the buffer while a consumer empties the buffer.</td>
</tr>
<tr>
<td>Rate-monotonic-scheduling</td>
<td>A preemptive fixed priority scheduling policy which assigns a higher priority to a task with a shorter period. The task deadlines are equal to task periods.</td>
</tr>
<tr>
<td>Real-time system</td>
<td>A system in which all tasks must meet their deadlines.</td>
</tr>
<tr>
<td>Real-time task</td>
<td>A task which must meet its deadline.</td>
</tr>
<tr>
<td>Recursive formulation</td>
<td>A formulation in which a model or process makes itself as a self-model or process.</td>
</tr>
<tr>
<td>Regular pipeline</td>
<td>A pipeline where each stage has a regular input period and all stages are flow balanced.</td>
</tr>
<tr>
<td>Release deference Time</td>
<td>The time interval between the task arrival time and task release time.</td>
</tr>
<tr>
<td>Release jitter</td>
<td>Irregularity of task release times due to its irregular completion times at the previous stage.</td>
</tr>
<tr>
<td>Release time</td>
<td>The time when a task is ready to execute.</td>
</tr>
<tr>
<td>Request rate</td>
<td>the number of inputs during a unit time.</td>
</tr>
<tr>
<td>Resource</td>
<td>A server with its associated buffer.</td>
</tr>
<tr>
<td><strong>Resource partitioning</strong></td>
<td>Dividing resources of different functionalities into multiple stages so that tasks can execute simultaneously through pipeline parallelism.</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Resource type</strong></td>
<td><em>e.g.</em>, CPU, disk, communication network, <em>etc.</em></td>
</tr>
<tr>
<td><strong>Response failure</strong></td>
<td>A status when task response time is greater than task deadline.</td>
</tr>
<tr>
<td><strong>Response time</strong></td>
<td>The time interval between task arrival and task completion.</td>
</tr>
<tr>
<td><strong>RMS</strong></td>
<td>Rate-monotonic-scheduling.</td>
</tr>
<tr>
<td><strong>Scaling factor</strong></td>
<td>The ratio of maximum schedulable execution time and the underlying execution time.</td>
</tr>
<tr>
<td><strong>Schedulability</strong></td>
<td>The ability of a system to meet all task deadlines.</td>
</tr>
<tr>
<td><strong>Schedulability test</strong></td>
<td>Test if a set of tasks are schedulable.</td>
</tr>
<tr>
<td><strong>Schedulable</strong></td>
<td>A set of tasks is schedulable when all the tasks meet their deadlines.</td>
</tr>
<tr>
<td><strong>Semaphore guarding access</strong></td>
<td>Using a flag to access a server exclusively so that the task may not be preempted by higher-priority tasks during its execution.</td>
</tr>
<tr>
<td><strong>Server</strong></td>
<td>An object which gives service to a set of requesting tasks.</td>
</tr>
<tr>
<td><strong>Server capacity</strong></td>
<td>Speed of a server (<em>e.g.</em>, 10 MIPS for a CPU).</td>
</tr>
<tr>
<td><strong>Server overflow</strong></td>
<td>A server which has tasks where the sum of the utilisation factors of the tasks exceed the unity.</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Server overflow deadline</td>
<td>The deadline by when a task must complete not to cause server overflow.</td>
</tr>
<tr>
<td>Simulation</td>
<td>A process of mimicking the behaviour of a system.</td>
</tr>
<tr>
<td>Simulation model</td>
<td>A model built by a simulator.</td>
</tr>
<tr>
<td>Simulator</td>
<td>A computer program which performs simulation.</td>
</tr>
<tr>
<td>Slack distribution</td>
<td>The remaining time to end-to-end deadline is divided and distributed to each stage forming the subdeadlines of the stages.</td>
</tr>
<tr>
<td>Soft deadline</td>
<td>The deadline for a soft real-time task.</td>
</tr>
<tr>
<td>Soft real-time task</td>
<td>The task whose peak-time average response time must be less than or equal to its deadline.</td>
</tr>
<tr>
<td>Spare deadline</td>
<td>Reducible time for deadline while guaranteeing the schedulability of all tasks.</td>
</tr>
<tr>
<td>Spare execution time</td>
<td>An expandable execution time for a task which guarantees the schedulability of all tasks in the system.</td>
</tr>
<tr>
<td>Spare interarrival time</td>
<td>An expandable interarrival time for a task which guarantees the schedulability of all tasks in the system.</td>
</tr>
<tr>
<td>Speed matching</td>
<td>Controlling the flow of each stage with different device speed characteristics by instance blocking so that all stages may be flow balanced.</td>
</tr>
<tr>
<td>Sporadic task</td>
<td>A sequence of the same type of instance occurring asynchronously at irregular interval with a minimum interarrival time.</td>
</tr>
<tr>
<td>Stage</td>
<td>A step of a pipeline which consists of a series of steps.</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>--------------------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Stage response time</td>
<td>The response time of a task at a stage.</td>
</tr>
<tr>
<td>Static allocation</td>
<td>The allocation of a task to a server at compile time.</td>
</tr>
<tr>
<td>Subdeadline</td>
<td>The deadline of a task at a stage.</td>
</tr>
<tr>
<td>Synchronous input</td>
<td>Input whose operation is clock-controlled.</td>
</tr>
<tr>
<td>System</td>
<td>A set of resources with their scheduling policies.</td>
</tr>
<tr>
<td>System overflow</td>
<td>The utilisation factor of a server in a system is greater than unity.</td>
</tr>
<tr>
<td>System overload</td>
<td>System overflow.</td>
</tr>
<tr>
<td>System performance</td>
<td>e.g., utilisation factors for servers, no input loss, guaranteed server capacity etc.</td>
</tr>
<tr>
<td>Task</td>
<td>A process to handle inputs from a source.</td>
</tr>
<tr>
<td>Task capacity</td>
<td>The maximum schedulable execution time for a task.</td>
</tr>
<tr>
<td>Task characterisation</td>
<td>Task parameters.</td>
</tr>
<tr>
<td>Task modification</td>
<td>A task is divided into allocatable groups of instances.</td>
</tr>
<tr>
<td>Task parameter</td>
<td>e.g., period, execution time and deadline.</td>
</tr>
<tr>
<td>Task partitioning</td>
<td>Division of a periodic task into allocatable instance groups for distributing to multiple servers.</td>
</tr>
<tr>
<td>Task performance</td>
<td>e.g., response time, required number of buffers, etc.</td>
</tr>
<tr>
<td><strong>Task switching</strong></td>
<td>Initial access time to a server when a task gains control of the server.</td>
</tr>
<tr>
<td><strong>Overhead time</strong></td>
<td>The synchronous communication between two tasks for an exclusive resource.</td>
</tr>
<tr>
<td><strong>Task synchronisation</strong></td>
<td>The number of output during a unit time (cf. completion rate).</td>
</tr>
<tr>
<td><strong>Throughput</strong></td>
<td>A status when output rate is less than input rate.</td>
</tr>
<tr>
<td><strong>Throughput failure</strong></td>
<td>Time needed by an algorithm expressed as a function of its input size.</td>
</tr>
<tr>
<td><strong>Time complexity of an algorithm</strong></td>
<td>A computer system which shares its time.</td>
</tr>
<tr>
<td><strong>Time-sharing system</strong></td>
<td>Solvable by a polynomial-time-bounded algorithm.</td>
</tr>
<tr>
<td><strong>Tractable</strong></td>
<td>The overload of the system during a short period until a steady-state load is reached.</td>
</tr>
<tr>
<td><strong>Transient overload</strong></td>
<td>The maximum possible completion time of a task which meets its deadline.</td>
</tr>
<tr>
<td><strong>Upper-bound completion time</strong></td>
<td>The deadline by when the user wants a task to complete.</td>
</tr>
<tr>
<td><strong>User specified deadline</strong></td>
<td>The proportion of busy time, which is the ratio of the execution time to the period.</td>
</tr>
<tr>
<td><strong>Utilisation factor</strong></td>
<td>The effective execution time of a task from viewpoint of system when instance distribution is made.</td>
</tr>
<tr>
<td><strong>Virtual execution time</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Virtual period</strong></td>
<td>The effective period of a task when instance distribution is made.</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Workload</strong></td>
<td>A set of tasks requesting system services.</td>
</tr>
<tr>
<td><strong>Worst-case response time</strong></td>
<td>The longest response time.</td>
</tr>
</tbody>
</table>
Appendices

Appendix C
Example Calculations

Detailed calculations for Tables 3.1, 3.2 and 3.6 are provided in the following Appendix C.1, C.2 and C.3 respectively.

C.1 Detailed Calculations for Table 3.1
We illustrate how the performance indices are derived by working through the calculations in Algorithm 3.3 for Table 3.1.

\[ E_{4,1} = \text{Complete}_4 (0,3) \]
\[ = 3 \text{ when } \text{Preempt}_4 (0,3) = 0, \text{ or } \]
\[ \text{Complete}_4 (3,3 + \text{Preempt}_4 (0,3)) \text{ otherwise.} \]

\[ \text{Preempt}_4 (0,3) = \text{Inputs}_1 (0,3) \times C_1 + \text{Inputs}_2 (0,3) \times C_2 + \]
\[ \text{Inputs}_3 (0,3) \times C_3 \]
\[ = 2 + 3 + 3 \]
\[ = 8 \]

\[ \text{Preempt}_4 (3,11) = \text{Inputs}_1 (3,11) \times C_1 + \text{Inputs}_2 (3,11) \times C_2 + \]
\[ \text{Inputs}_3 (3,11) \times C_3 \]
\[ = 2 + 0 + 0 \]
\[ = 2 \]

\[ \text{Preempt}_4 (11,13) = \text{Inputs}_1 (11,13) \times C_1 + \text{Inputs}_2 (11,13) \times C_2 + \]
\[ \text{Inputs}_3 (11,13) \times C_3 \]
\[ = 0 + 3 + 0 \]
\[ = 3 \]
Preempt4 (13,16) = Inputs1 (13,16) × C1 + Inputs2 (13,16) × C2 + 
Inputs3 (13,16) × C3
= 0 + 0 + 3
= 3

Preempt4 (16,19) = Inputs1 (16,19) × C1 + Inputs2 (16,19) × C2 + 
Inputs3 (16,19) × C3
= 0 + 0 + 0
= 0

Hence, $E_4,1 = 19$. Since $(E_4,1 > T_4)$ where $T_4 = 18$ (Table 3.1), we further proceed to calculate $E_4,2$.

$E_4,2 = \text{Complete}_4 (E_4,1 + C_4)$
= $\text{Complete}_4 (19,19+3)$
= 24.

Since $(E_4,2 < 2T_4)$ where $T_4 = 18$ (Table 3.1), we stop to proceed to calculate $E_4,3$.

Note that the level-4 busy period is finished at time 24.

Therefore $W_4 = \max \{(E_4,1 - 0T_4), (E_4,2 - 1T_4)\} = \max \{(19-0), (24-18)\} = 19$.

By similar calculation, we can obtain $W_1 = 2, W_2 = 5, W_3 = 8$ and $W_5 = 29$.

Using Equation (3.15),
$B_5 = \lceil 29 / 20 \rceil = 2$.

By similar calculation, we can obtain $B_1 = 1, B_2 = 1, B_3 = 1$, and $B_4 = 2$.

C.2 Detailed Calculations for Table 3.2

Using Equation (3.10),
P = LCM (10, 12, 15, 18, 20)
Using Equation (3.1),
\[ U_s = \frac{((180 / 10) \times 2 + (180 / 12) \times 3 + (180 / 15) \times 3 + (180 / 18) \times 3 + (180 / 20) \times 2)}{180} = 0.91 \]

Using Algorithm 3.1, we see that no input loss will occur, i.e., \( \text{NIL} = \text{TRUE} \) because no server overflows \( (U_s < 1) \) and no buffer overflows (we assumed that the required buffer space is provided). The current capacity of the CPU is enough, i.e., the guaranteed CPU capacity \( \text{GCC} = \text{TRUE} \) because \( W_1 < D_1, W_2 < D_2, W_3 < D_3, W_4 < D_4, \) and \( A_5 < D_5. \)

C.3 Detailed Calculations for Table 3.6
We illustrate how the performance indices are derived by working through the calculations done by Algorithm 3.4 for Table 3.6.
\[ E_{2,1} = \text{Complete}_2(0, C_{2_{\text{block}}+C_{2_{\text{sw}}}}) + C_{2_{\text{non-preempt}}} \]
\[ = \text{Complete}_2(0, (70+5)) + 45 \]
\[ \text{Complete}_2(0,75) = 75 \text{ when } \text{Preempt}_2(0,75) = 0, \text{ or } \]
\[ \text{Complete}_2(75, 75 + \text{Preempt}_2(0,75)) \text{ otherwise.} \]
\[ \text{Preempt}_2(0,75) = \text{Inputs}_1(0,75) \times C_1 \]
\[ = 2 \times (10+20) \]
\[ = 60 \]
\[ \text{Preempt}_2(75,135) = \text{Inputs}_1(75,135) \times C_1 \]
\[ = 1 \times 30 \]
\[ = 30 \]
\[ \text{Preempt}_2(135,165) = 0 \]
Hence, \( E_{2,1} = 165+45 = 210 \). Since \( (E_{2,1} > 1T_2) \) where \( T_2 = 200 \) (Table 3.6), we further proceed to calculate \( E_{2,2} \).

\[
E_{2,2} = \text{Complete}_2 \ (E_{2,1}-C_{2\text{non-preempt}}, E_{2,1}+C_{2\text{sw}}) + C_{2\text{non-preempt}} \\
= \text{Complete}_2 \ ((210-45), (210+5)) + 45 \\
= 275 + 45 \\
= 320.
\]

Since \( (E_{2,2} < 2T_2) \) where \( T_2 = 200 \) (Table 3.6), we stop to proceed to calculate \( E_{2,3} \). Note that the level-2 busy period is finished at time 320.

Therefore \( W_2 = \max \{(E_{2,1}-0T_2), (E_{2,2}-1T_2)\} = \max \{(210-0), (320-200)\} = 210.\)

By similar calculation, we can obtain \( W_1 = 100, W_3 = 440 \) and \( W_4 = 550.\)

Using Equation (3.15),

\[
B_2 = \left\lceil \frac{210}{200} \right\rceil = 2.
\]

By similar calculation, we can obtain \( B_1 = 2, B_3 = 1, \) and \( B_4 = 1.\)
Appendices

Appendix D
Source Codes

The source codes of the two CPU simulators are provided for both preemptive fixed priority scheduling (D.1) and preemptive deadline driven scheduling (D.2).

The codes are written in C on SUN 4/690MP.

D.1 CPU Simulator for Preemptive Fixed Priority Scheduling

/* Program name: priority_cpu_simulator.c */
* CPU simulation program for real-time systems
* assuming preemptive fixed priority scheduling.
* Input
*   i: priority.
*   T[i]: period of task i.
*   C[i]: execution time of task i.
* Output
*   CPU execution map.
*   U_cpu: CPU utilisation factor.
*   W[i]: worst-case CPU response time for task i.
*   A[i]: peak-time average CPU response time for task i.
*   B[i]: min. required number of buffers for task i.
*/

#include <stdio.h>
#include <math.h>
#define max_task 20
#define max_queue 20
#define max_arrival 10000
#define max_print_column 50

int T[max_task], C[max_task];  /* input parameters */
double U_cpu;
int W[max_task], A[max_task], B[max_task];  /* output parameters */
int no_of_tasks;
int lcm; /* peak-time interval */

int time; /* time tick */

int arrival_time[max_task][max_queue];
int arrival_queue[max_task];
int execution_remain[max_task];
int tot_cpuresponse[max_task];
int complete_instancecount[max_task];

main()
{
    input_process();
    serveroverflow_check();
    execution_simulation();
}

} /* end of main() */

input_process()
{
    int i;

    printf("Number of tasks? [1..%d] ", max_task); scanf("%d", &no_of_tasks);
    for (i = 1; i <= no_of_tasks; i++) {
        printf("Priority %d task:
", i);
        printf("Min. interarrival time ? ");  scanf("%d", &T[i]);
        printf("Max. processing time ? "); scanf("%d", &C[i]);
    }

} /* end of input_process() */

serveroverflow_check()
{
    int i,
        busy_time;

    lcm = T[1];
    for (i = 2; i <= no_of_tasks; i++) {
        lcm = least_common_multiple(lcm, T[i]);
    }
    printf("\nPeak-time interval = %d
", lcm);

    busy_time = 0;
    for (i = 1; i <= no_of_tasks; i++) {
        busy_time = busy_time + lcm*T[i]*C[i];
    }
    printf("CPU busy time = %d
", busy_time);

    U_cpu = (double)(busy_time)/(double)(lcm);
    printf("CPU utilisation factor = %f (%f%%)
", U_cpu, (U_cpu* 100.0));

    if (U_cpu > 1.0) 
    {

176
printf("Server overflow occurs.");
exit(1);
}
} /* end of serveroverflow_check() */

execution_simulation()
{
    int i;

    /* head printout */
    for (i = 1; i <= max_print_column; i++) {
        if ((i%5) == 0) printf("+");
        else printf(" ");
    }
    printf("\n");

    /* data initialization */
    time = 0;
    for (i = 1; i <= no_of_tasks; i++) {
        W[i] = 0;
        B[i] = 1;
        arrival_time[i][1] = time;
        arrival_queue[i] = 1;
        execution_remain[i] = C[i];
        tot_cpuresponse[i] = 0;
        complete_instancecount[i] = 0;
    }

    /* simulate until time reaches to lcm (i.e., peak-time interval) */
    do{
        draw_cpu_execution_map();
        time++;
        if ((time%max_print_column)  == 0) printf("\n");
    } while (time < lcm);
    printf("#\n\n\n"); /* terminating time mark */

    /* calculate peak-time average response time */
    for (i = 1; i <= no_of_tasks; i++)
        A[i] = tot_cpuresponse[i] / complete_instancecount[i];

    /* printout min, max, avg response times for tasks */
    for (i = 1; i <= no_of_tasks; i++) {
        printf("%d%d%d%d%d\n",
            i, T[i], C[i], W[i], A[i], B[i]);
    }
} /* end of execution_simulation() */
Appendices

```c
int least_common_multiple(num1, num2) /* num1 and num2 should be positive */
int num1, num2;
{
    int i;
    for (i = 1; i <= num2; i++) {
        if (((i*num1) % num2) == 0) return (i*num1);
    }
} /* end of least_common_multiple(num1, num2) */

draw_cpu_execution_map()
{
    #define non_exist 0
    #define exist 1

    int i,
    j,
    interrupt,
    arrivalqueue_size,
    current_time,
    current_cpuresponse;

    interrupt = non_exist;

    /* check if any task arrivals */
    if (time != 0) {
        for (i = 1; i <= no_of_tasks; i++) {
            if ( (time % T[i]) == 0 ) {
                arrival_queue[i]++;
                arrivalqueue_size = arrival_queue[i];
                arrival_time[i][arrivalqueue_size] = time;
                if (arrival_queue[i] > B[i])
                    B[i] = arrival_queue[i];
        }
    }

    /* pop the highest priority task */
    for (i = 1; i <= no_of_tasks; i++) {
        if (arrival_queue[i] > 0) {
            interrupt = exist;
            break;
        }
    }

    /* printout the cpu seizure task number */
    if (interrupt == exist) {
        printf("%d", i);
        execution_remain[i]--;
        if (execution_remain[i] == 0) { /* execution completed */
            arrival_queue[i]--;
            arrivalqueue_size = arrival_queue[i];
            current_time = time + 1;
        }
    }
```

current_cpuresponse = current_time - arrival_time[i][1];

for (j = 1; j <= arrivalqueue_size; j++)
    arrival_time[i][j] = arrival_time[i][j+1];

tot_cpuresponse[i] = tot_cpuresponse[i] + current_cpuresponse;

complete_instancecount[i]++;
if (current_cpuresponse > W[i])
    W[i] = current_cpuresponse;
    execution_remain[i] = C[i];

    } /* if execution_remain[i] == 0 */
} /* if interrupt == exist */
else /* interrupt == non_exist */ { 
    printf(" ");
}

} /* end of draw_cpu_execution_map() */
Appendices

D.2 CPU Simulator for Preemptive Deadline Driven Scheduling

/*
* Program name: deadline_cpu_simulator.c
* CPU simulation program for real-time systems
* assuming preemptive deadline driven scheduling.
*
* Input
* T[i]: period of task i.
* C[i]: execution time of task i.
* D[i]: deadline of task i.
*
* Output
* CPU execution map.
* U_cpu: CPU utilisation factor.
* W[i]: worst-case CPU response time for task i.
* A[i]: peak-time average CPU response time for task i.
* B[i]: min. required number of buffers for task i.
*/

#include <stdio.h>
#include <math.h>
#define max_task 20
#define max_queue 20
#define max_arrival 10000
#define max_print_column 50
int T[max_task], C[max_task], D[max_task]; /* input parameters */
double U_cpu;
int W[max_task], A[max_task], B[max_task]; /* output parameters */
int no_of_tasks; /* peak-time interval */
int time; /* time tick */
int arrival_time[max_task][max_queue];
int arrival_queue[max_task];
int execution_remain[max_task];
int deadline[max_task][max_queue];
int tot_cpuresponse[max_task];
int complete_instancecount[max_task];

main()
{
    input_process();
    serveroverflow_check();
}
Appendices

execution_simulation();

} /* end of main() */

input_process()
{
    int i;

    printf("Number of tasks? [1..%d] ",max_task); scanf("%d", &no_of_tasks);
    for (i = 1; i <= no_of_tasks; i++) {
        printf("Min. interarrival time ? "); scanf("%d", &T[i]);
        printf("Max. processing time ? "); scanf("%d", &C[i]);
        printf("Deadline ? "); scanf("%d", &D[i]);
    }
}

} /* end of input_process() */

serveroverflow_check()
{
    int i,
    busy_time;

    lcm = T[1];
    for (i = 2; i <= no_of_tasks; i++) {
        lcm = least_common_multiple(lcm, T[i]);
    }
    printf("Peak-time interval = %d\n", lcm);

    busy_time = 0;
    for (i = 1; i <= no_of_tasks; i++) {
        busy_time = busy_time + lcm/T[i]*C[i];
    }
    printf("CPU busy time = %d\n", busy_time);

    U_cpu = (double)(busy_time)/(double)(lcm);
    printf("CPU utilisation factor = %f (%f\%\n", U_cpu, (U_cpu* 100.0));

    if (U_cpu > 1.0) {
        printf("Server overflow occurs.\n");
        exit(1);
    }
}

} /* end of serveroverflow_check() */

execution_simulation()
{
    int i;

    /* head printout */
    for (i = 1; i <= max_print_column; i++) {
        printf(" ");
    }
if ((i%5) == 0) printf("+");
else printf(" ");
}
printf("\n");

/* data initialization */
time = 0;
for (i = 1; i <= no_of_tasks; i++) {
    W[i] = 0;
    B[i] = 1;
    arrival_time[i][1] = time;
    arrival_queue[i] = 1;
    execution_remain[i] = C[i];
    deadline[i][1] = time + D[i];

tot_cpuresponse[i] = 0;
    complete_instancecount[i] = 0;
}

/* simulate until time reaches to lcm (i.e., peak-time interval) */
do {
    draw_cpu_execution_map();
    time++;
    if ((time%max_print_column) == 0) printf("\n");
} while (time < lcm);

printf("\#\n\n\n\n"); /* terminating time mark */

/* calculate peak-time average response time */
for (i = 1; i <= no_of_tasks; i++)
    A[i] = tot_cpuresponse[i] / complete_instancecount[i];

/* printout min, max, avg response times for tasks */
printf("%dT[i]%dC[i]%dD[i]%dW[i]%dA[i]%dB[i]\n"),
for (i = 1; i <= no_of_tasks; i++) {
    printf("%d%dt%dt%dt%dt%dn", i, T[i], C[i], D[i], W[i], A[i], B[i]);
}

} /* end of execution_simulation */

int least_common_multiple(num1, num2) /* num1 and num2 should be positive */
int num1, num2;
{
    int i;
    for (i = 1; i <= num2; i++)
        if (((i*num1) % num2) == 0) return (i*num1);
}

} /* end of least_common_multiple(num1, num2) */
draw_cpu_execution_map()
{
  #define non_exist 0
  #define exist 1

  int i,
      j,
      interrupt,
      arrivalqueue_size,
      earliest_deadline,
      current_time,
      current_cpuresponse;

  interrupt = non_exist;

  /* check if any task arrivals */
  if (time != 0) {
    for (i = 1; i <= no_of_tasks; i++) {
      if ((time % T[i]) == 0) {
        arrival_queue[i]++;
        arrivalqueue_size = arrival_queue[i];
        arrival_time[i][arrivalqueue_size] = time;
        deadline[i][arrivalqueue_size] = time + D[i];
        if (arrival_queue[i] > B[i])
          B[i] = arrival_queue[i];
      }
    }
  }

  /* pop the earliest deadline task */
  earliest_deadline = 1000000;
  for (j = 1; j <= no_of_tasks; j++) {
    if (arrival_queue[j] > 0) {
      interrupt = exist;
      if (deadline[j][1] < earliest_deadline) {
        earliest_deadline = deadline[j][1];
        i = j;
      } else if /* same deadline then FIFO */
        ((deadline[j][1] == earliest_deadline) &&
         (arrival_time[j][1] < arrival_time[i][1])) { i = j; }
    }
  }

  /* printout the cpu seizure task number */
  if (interrupt == exist) {
    printf("%d", i);
    execution_remain[i]--;
    if (execution_remain[i] == 0) { /* execution completed */
      arrival_queue[i]--;
      arrivalqueue_size = arrival_queue[i];
      current_time = time + 1;
  }
}
current_cpuresponse = current_time - arrival_time[i][1];

for (j = 1; j <= arrivalqueue_size; j++) {
    arrival_time[i][j] = arrival_time[i][j+1];
    deadline[i][j] = deadline[i][j+1];
}

tot_cpuresponse[i] = tot_cpuresponse[i] + current_cpuresponse;

complete_instancecount[i]++;
if (current_cpuresponse > W[i])
    W[i] = current_cpuresponse;

execution_remain[i] = C[i];

} /* if execution_remain[i] == 0 */
} /* if interrupt == exist */
else /* interrupt == non_exist */ {
    printf("\"");
}
} /* end of draw_cpu_execution_map() */
Bibliography


Bibliography


Davis, R.I., Tindell, K.W. and Burns, A., 1993. 'Scheduling Slack Time in


Gotlieb, C.C. and MacEwen, G.H., 1973. 'Performance of Movable-Head Disk


Johnson, H.H. and Madison, M.S., 1974. 'Deadline Scheduling for a Real-Time Multiprocessor', *NTIS (N76-15843)*, Springfield, VA.
Bibliography


Mahjoub, A., 1984. 'On the Static Evaluation of Distributed Systems


Ramamritham, K., Stankovic, J.A. and Shiah, P.-F., 1990. 'Efficient Scheduling


Bibliography


