©Copyright by

Jun Sun

1997

# FIXED-PRIORITY END-TO-END SCHEDULING IN DISTRIBUTED REAL-TIME SYSTEMS

BY

JUN SUN

B.S., Shanghai Jiao Tong University, 1989

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1997

Urbana, Illinois

# Abstract

In recent years more and more real-time applications run on multiprocessor or distributed systems. In such systems, a task may execute sequentially on many different processors. Such a task can be viewed as a linear chain of subtasks, each of which represents a segment of the task that executes on one of those processors. The response time of the task is measured from the release of its first subtask to the completion of its last subtask and is called the end-to-end response time. A task is schedulable if its end-to-end response time is never greater than the specified end-to-end relative deadline.

This thesis deals with the problem of scheduling periodic tasks to meet their end-to-end deadlines. Specifically, the thesis focuses on fixed-priority scheduling algorithms, where each subtask is assigned a fixed priority and is scheduled preemptively. According to this approach, three related problems need to be solved.

**Priority Assignment :** How we assign the priorities to subtasks so that the system schedulability can be maximized.

**Execution Synchronization :** How we synchronize the releases of subtasks so that the precedence constraints among subtasks are satisfied, and the system schedulability is optimized.

**Schedulability Analysis :** Given a certain priority assignment and an execution synchronization method, how we compute the worst-case end-to-end response time of each task, so that we can verify the schedulability of the task and hence the schedulability of the system.

As solutions to these three problems, the thesis describes five synchronization protocols, several deadline-based priority assignment methods, and corresponding schedulability analysis al-

gorithms. These solutions form an integrated end-to-end scheduling framework for scheduling tasks with end-to-end deadlines and analyzing the schedulability of an end-to-end system.

As an application of the framework, the thesis proposes an end-to-end scheduling based approach to the resource contention problem in distributed real-time systems. Simulation results show that the end-to-end scheduling approach performs better in many cases than an existing approach, known as the Multiprocessor Priority Ceiling Protocol. In the appendix, the thesis includes an in-depth discussion of sporadic server algorithms.

To my parents,
- who made me possible

To Hsiu-Fen,
- who makes me complete

# Acknowledgments

My deepest thanks go to Professor Jane Liu, who helped me grow not only in the academic research but also in many other dimensions of my life. Since becoming my advisor in 1993, she made a diligent effort to lead me into the real-time computing field, teach me the knowledge, and guide me to successfully complete my degree. By her example of hard work and her pursuit of excellence, she showed me the essentials of being a successful researcher. Her valuable advice and help have kept me confident in times of doubt and led me to success from places of difficulties. She often shared with me her values and experience, which was so precious then and is even more so now.

I also thank Professor Dave Liu and Professor Rajesh Gupta who kindly agreed to be members of my thesis committee. They provided many insightful comments, read through my thesis draft, and directed my research in the right direction.

Many members of the Real-Time Research Lab at University of Illinois helped me to make this thesis possible. I thank Riccardo Bettati who opened the door to the end-to-end scheduling research, Matthew Storch who offered many inspiring opinions as well as his valuable friendship, and Wu Feng whose review of an early draft of my thesis picked up many embarrassing mistakes. David Hull answered my innumerable questions on Unix, Latex and other tools. I thank many other members in the research lab who helped me in various ways and places, including Too-Seng Tia, Arjun Shanker, Ian Philp, Rhan Ha, Mark Gardener and Changwen Liu. Special thanks go to Zhong Deng for helping me submit the thesis and on many other things.

I thank Sean Sullivan, Carolyn Pater and Larry Pater for reading through the final draft and fixing my "Chinese" usage of English. I am also thankful to Xingbin (Ben) Zhang who, as a friend, offered his valuable opinions and assistance on many occasions.

Lastly, I would like to thank my wife, Hsiu-Fen, who has gone through the whole arduous journey of writing a Ph.D. thesis with me. She tasted every piece of joy and bitterness that I tasted. She understood the meaning of every word beyond the scope that I meant. For many days and nights, she sacrificed her time to give me comfort and let me concentrate on the work. Without her love, care and support, my finishing the thesis would never have been possible. To her I will always be grateful.

# Table of Contents

**Chapter**


**1 Introduction** . . . . . 1

1.1 Background and Motivation . . . . . 1

1.2 Summary of Contributions . . . . . 5

1.3 Organization of the Thesis . . . . . 6

**2 Related Work** . . . . . 9

2.1 Real-Time Scheduling Theories . . . . . 9

2.1.1 Uniprocessor Scheduling Theories . . . . . 10

2.1.2 End-to-End Scheduling Theories . . . . . 10

2.2 Resource Access Control Protocols . . . . . 11

2.3 Real-Time Communication . . . . . 12

2.4 Classical Scheduling Theory . . . . . 14

**3 System Model** . . . . . 16

3.1 Assumptions . . . . . 16

3.2 Problems in End-to-End Scheduling . . . . . 21

**4 Synchronization Protocols** . . . . . 25

4.1 Introduction . . . . . 25

4.2 The Synchronization Protocols . . . . . 26

4.2.1 Direct Synchronization (DS) Protocol . . . . . 27

4.2.2 Phase Modification (PM) Protocol . . . . . 28

4.2.3 Modified Phase Modification (MPM) Protocol . . . . . 31

4.2.4 Release Guard (RG) Protocol . . . 34
4.2.5 Sporadic Server (SS) Protocol . . . 36
4.3 Complexity and Run-Time Overhead . . . 37
4.4 Execution Control . . . 39

**5 Schedulability Analysis** . . . 40
5.1 Busy Period Analysis . . . 41
5.2 Schedulability Analysis for the PM and MPM Protocols . . . 43
5.2.1 Algorithm SA/PM . . . 44
5.2.2 An Improvement of Algorithm SA/PM . . . 46
5.2.3 A Further Improvement of Algorithm SA/PM . . . 56
5.3 Schedulability Analysis for the RG Protocol . . . 58
5.4 Schedulability Analysis for the SS Protocol . . . 62
5.5 Schedulability Analysis for the DS Protocol . . . 63
5.5.1 Clumping Effect . . . 64
5.5.2 Overall Structure of Algorithm SA/DS . . . 66
5.5.3 Algorithm IEERT . . . 66
5.5.4 Algorithm SA/DS . . . 69
5.5.5 Termination of Algorithm SA/DS . . . 73
5.5.6 Relation with Previous Work . . . 74
5.6 Summary . . . 75

**6 Priority Assignment** . . . 76
6.1 Priority Assignment Problem . . . 76
6.2 Heuristic Priority Assignment Methods . . . 78
6.3 Local Assignment Methods vs. Global Assignment Methods . . . 81
6.4 Summary . . . 81

**7 Performance of End-to-End Scheduling Algorithms** . . . 83
7.1 Experiment I - Deadline-Based Priority Assignment Methods . . . 84
7.1.1 Workload Generation . . . 84
7.1.2 Performance Criteria . . . 85

7.1.3 Simulation Results . . . 86
7.2 Experiment II - Algorithm SA/DS vs. Algorithm SA/PM . . . 87
7.2.1 Workload Generation . . . 88
7.2.2 Performance Criteria . . . 88
7.2.3 Simulation Results . . . 89
7.3 Experiment III - Algorithm SA/PM vs. Algorithm SA/IPM . . . 90
7.3.1 Workload Generation . . . 90
7.3.2 Performance Criteria . . . 91
7.3.3 Simulation Results . . . 91
7.4 Experiment IV - Average Task EER Times . . . 92
7.4.1 Workload Generation . . . 94
7.4.2 Performance Criterion . . . 94
7.4.3 Simulation Results . . . 95
7.5 Summary . . . 97

**8 End-to-End Scheduling Approach to the Resource Contention Problem . . . 99**
8.1 Background . . . 100
8.2 System Model . . . 101
8.3 MPCP Approach . . . 103
8.3.1 Priority Assignment . . . 104
8.3.2 Upper Bounds of Blocking Times . . . 104
8.3.3 Schedulability Analysis . . . 107
8.3.4 Corrections of the MPCP . . . 108
8.3.5 Improvement to the MPCP . . . 110
8.4 End-to-End Scheduling Approach . . . 112
8.4.1 Mapping an MPCP System to an End-to-End System . . . 113
8.4.2 Priority Assignment and Execution Synchronization . . . 114
8.4.3 Schedulability Analysis . . . 115
8.5 Performance Comparison . . . 116
8.5.1 Workload Generation . . . 116
8.5.2 Performance Criteria . . . 118

8.5.3 Overall Performance Comparison . . . 119
8.5.4 Improvement to the MPCP . . . 122
8.5.5 Performance of End-to-End Scheduling Algorithms . . . 126
8.5.6 The MPCP Approach vs. the DS Synchronization Protocol . . . 126
8.5.7 Overall Performance with a Mixed Workload . . . 127
8.6 Summary . . . 129

**9 Future Research Directions** . . . 131
9.1 General Subtask Precedence Constraints . . . 131
9.1.1 Multiple Adjacent Subtasks Execute on the Same Processor . . . 131
9.1.2 The Precedence Graph is a Directed Acyclic Graph (DAG) . . . 133
9.2 Hybrid End-to-End Scheduling . . . 134
9.3 End-to-End Scheduling with Dynamic Priorities . . . 135

**Appendix**

**A Sporadic Server Algorithms** . . . 136
A.1 Background . . . 137
A.2 The Structure of a Sporadic Server . . . 139
A.3 Algorithm SS1 . . . 141
A.4 Algorithm SS2 . . . 144
A.5 Algorithm SS3 . . . 147
A.6 Scheduling Sporadic Requests with Unknown Execution Times (Algorithm SS4) . 152
A.7 Discussion . . . 154

**Bibliography** . . . 157

**Vita** . . . 163

# List of Tables


4.1 Comparison of Protocols with Respect to Complexity and Overhead . . . . . 37

5.1 Subtask Parameters of an End-to-End System . . . . . 46
5.2 Subtask Parameters for the Example in Figure 5.4 . . . . . 48
5.3 An Example to Illustrate the Interference among Sibling Subtasks . . . . . 54
5.4 One-to-One Mapping of Events in the RG Protocol and the SS Protocol . . . . . 63

6.1 Subtasks with Calculated Relative Deadlines . . . . . 80

7.1 Schedulability Indices Yielded by Different Priority Assignment Methods . . . . . 87
7.2 Failure Rates for Algorithm SA/DS . . . . . 89
7.3 Worst-Case Schedulability Index Ratios of Algorithm SA/DS over Algorithm SA/PM . . . . . 90
7.4 Success Rates for Algorithm SA/IPM . . . . . 92
7.5 Worst-Case Schedulability Index Ratios of Algorithm SA/PM over Algorithm SA/IPM . . . . . 93
7.6 Average Schedulability Index Ratios of Algorithm SA/PM over Algorithm SA/IPM 93

8.1 Task Parameters of the System in Example 1 . . . . . 105
8.2 Task and GCS Server Priorities in Example 1 . . . . . 105
8.3 Blocking Times of Tasks in Example 1 . . . . . 107
8.4 The Bounds on the Response Times of Tasks in Example 1 According to the MPCP . . . . . 108
8.5 Task Parameters in Example 2 . . . . . 109

8.6 The Upper Bounds on the Blocking Times and Response Times of Tasks in Example 2 . . . 109
8.7 Blocking Times of Tasks in Example 1 According to the Corrected MPCP . . . . 110
8.8 Blocking Times of Tasks in Example 1 According to the Improved MPCP . . . . . 111
8.9 Subtask Parameters of the System in Example 1 . . . . . 114
8.10 Different Deadlines for Subtasks in Example 1 . . . . . 114
8.11 Different Deadlines for Subtasks in Example 1 . . . . . 115
8.12 Upper Bounds on Task EER Times in Example 1 . . . . . 116
8.13 The Success Rates When the Processor Utilization is 0.5 . . . . . 120
8.14 The Success Rates When the Processor Utilization is 0.6 . . . . . 123
8.15 The Success Rates When the Processor Utilization is 0.7 . . . . . 124
8.16 The Success Rates When the Processor Utilization is 0.8 . . . . . 125
8.17 The Average Success Rates of the Fixed MPCP and the Improved MPCP . . . . . 125
8.18 The Average Success Rates of Algorithms SA/DS, SA/PM and SA/IPM . . . . . 126
8.19 The Average Success Rates of the MPCP Approach, Algorithm SA/DS and the Combined Approach . . . . . 127
8.20 Success Rates of the MPCP Approach and the End-to-End Scheduling Approach for Mixed Workload . . . . . 128

# List of Figures


1.1 An End-to-End Task . . . . . 3

3.1 An End-to-End System . . . . . 18
3.2 Mapping a Prioritized Bus to a "Link" Processor . . . . . 20
3.3 An Example to Illustrate the Problems in End-to-End Scheduling . . . . . 21
3.4 The Schedule of the System in Figure 3.3 when Every Instance of $T_{2,2}$ is Released Immediately after the Corresponding Instance of $T_{2,1}$ Completes . . . . . 23

4.1 Illustration of the Phase Modification Protocol . . . . . 28
4.2 A Simple End-to-End System (reproduced from Figure 3.3) . . . . . 29
4.3 The Schedule According to the PM Protocol . . . . . 30
4.4 Illustration of the Modified Phase Modification Protocol . . . . . 32
4.5 The Schedule of the System in Figure 4.2 According to the RG Protocol . . . . . 35
4.6 Classification of Scheduling Protocols . . . . . 39

5.1 Two $\phi_{1,1}$-Level Busy Periods . . . . . 41
5.2 A $\phi_{i,j}$-Level Busy Period . . . . . 42
5.3 Pseudo-Code of Algorithm SA/PM . . . . . 47
5.4 A Simple Recurrent End-to-End System . . . . . 48
5.5 Pseudo-Code of Algorithm SA/IPM . . . . . 51
5.6 The Time Demand Functions of $T_1$ in the Example in Figure 5.4 . . . . . 52
5.7 Pseudo-Code of Algorithm SA/IPM/IF1 . . . . . 53
5.8 The Schedule of the System in Table 5.3 . . . . . 54
5.9 An Example to Illustrate the Second Improvement of Algorithm SA/PM . . . . . 56
5.10 Time Demand Functions of $T_1$ in Figure 5.9 . . . . . 57

5.11 Pseudo-Code of Algorithm SA/IPM/IF2 . . . . . 59
5.12 The EER Time of a Task in a System Using the RG Protocol . . . . . 60
5.13 An Example to Illustrate the Clumping Effect . . . . . 64
5.14 A Schedule of the System in Figure 5.13 . . . . . 65
5.15 The Flow Control Diagram of Algorithm SA/DS . . . . . 67
5.16 Maximal Duration of a $\phi_{i,j}$-Level Busy Period (DS Protocol) . . . . . 68
5.17 Pseudo-Code of Algorithm IEERT . . . . . 70
5.18 Pseudo-Code of Algorithm SA/DS . . . . . 71
5.19 An Example to Show that Algorithm SA/DS May Never Terminate . . . . . 73

7.1 The PM/DS Ratios . . . . . 95
7.2 The RG/DS Ratios . . . . . 96
7.3 The PM/RG Ratios . . . . . 97

8.1 The Simple System in Example 1 . . . . . 101
8.2 The GCS Server of $T_1$ in Example 1 . . . . . 102
8.3 The GCS Server of $T_4$ in Example 1 . . . . . 103
8.4 The Schedule of the System in Example 2 . . . . . 110
8.5 An End-to-End View of the System in Example 1 . . . . . 112

9.1 The Precedence Graph of Subtasks is a DAG . . . . . 133

A.1 The Schedule if $T_s$ Were a Periodic Task . . . . . 138
A.2 A Schedule of the Example System . . . . . 138
A.3 The Pseudo-Code of Event Handlers in Algorithm SS1 . . . . . 143
A.4 The Schedule of the Simple System According to Algorithm SS1 . . . . . 143
A.5 Another Schedule of the Simple System According to Algorithm SS1 . . . . . 145
A.6 The Pseudo-Code of Event Handlers in Algorithm SS2 . . . . . 146
A.7 The Schedule of the Simple System According to Algorithm SS2 . . . . . 147
A.8 Two Schedules of the Simple System According to Algorithm SS2 . . . . . 148
A.9 Description of Operations on the Free Budget Function . . . . . 150
A.10 The Pseudo-Code of Algorithm SS3 . . . . . 151
A.11 The Pseudo-Code of Algorithm SS4 . . . . . 155

# List of Symbols


$\alpha$ : recurrence degree of a task ..... 90

$\eta$ : factor to determine the duration of a critical section (CSD factor) ..... 117

$\kappa$ : number of subtasks in each task ..... 88

$\kappa_{cs}$ : number of critical sections in a task ..... 117

$\lambda$ : method to generate the duration of a critical section ..... 117

$\mu$ : processor utilization ..... 88

$\tau_i$ : maximum execution time of task $T_i$ ..... 79

$\tau_{i,j}$ : maximum execution time of subtask $T_{i,j}$ ..... 17

$\phi_{i,j}$ : the priority level of subtask $T_{i,j}$ ..... 17

$B_i$ : the blocking time of task $T_i$ ..... 105

$B_{i,j}$ : the blocking time of subtask $T_{i,j}$ ..... 115

$C_{i,j}(m)$ : the completion time of subtask instance $_{i,j}(m)$ ..... 42

$D_i$ : the relative deadline of task $T_i$ ..... 17

$D_{i,j}$ : the maximum duration of a $\phi_{i,j}$-level busy period ..... 43

$f_i$ : the phase of task $T_i$ ..... 17

$f_{i,j}$ : the phase of subtask $T_{i,j}$ ..... 28

$g_{i,j}$ : the release guard of subtask $T_{i,j}$ ..... 34

$H_{i,j}$ : the set of subtasks that have priorities higher than or equal to $T_{i,j}$ and are the same processor as $T_{i,j}$ ..... 44

$M_{i,j}$ : the maximum number of instances of $T_{i,j}$ that can be in a $\phi_{i,j}$-level busy period ... 45

$M_k^{i,j}(t)$ : the interference function of task $T_k$ with respect to subtask $T_{i,j}$ during interval $[t_0, t_0 + t)$ ..... 50

$P_i$ : the $i$th processor in the system ..... 16

$p_i$ : the period of task $T_i$ ..... 17

$p_{i,j}$ : the period of subtask $T_{i,j}$ ..... 17

$R_i$ : an upper bound on the response time of $T_i$ ..... 46

$R_{i,j}$ : an upper bound on the response time of $T_{i,j}$ ..... 28

$r_{i,j}(m)$ : the release time of subtask instance $T_{i,j}(m)$ ..... 42

$T_i$ : the $i$th task in the system ..... 16

$T_{i,j}$ : the $j$th subtask in task $T_i$ ..... 16

$T_{i,j}(m)$ : the $m$th instance of subtask $T_{i,j}$ ..... 17

$V_{i,j}$ : an upper bound on the IEER time of subtask $T_{i,j}$ ..... 66

$W(t)$ : time demand function for interval $[t_0, t_0 + t)$ ..... 42

# List of Abbreviations

**Algorithm IEERT :** An algorithm that takes as input a set of end-to-end tasks and upper bounds on the IEER times of subtasks and computes a set of new bounds on IEERT times (needed by Algorithm SA/DS) ..... 66

**Algorithm SA/DS :** an algorithm that analyzes the schedulability of an end-to-end system using the DS protocol ..... 66

**Algorithm SA/IPM :** an improved version of Algorithm SA/PM, which takes advantage of recurrent tasks ..... 46

**Algorithm SA/IPM/IF1 :** an algorithm to compute the interference function (needed by Algorithm SA/IPM) ..... 53

**Algorithm SA/IPM/IF2 :** an improved version of Algorithm SA/IPM/IF1 ..... 58

**Algorithm SA/PM :** an algorithm that analyzes the schedulability of an end-to-end system using the PM, MPM, RG or SS protocol ..... 44

**CSD factor :** the critical section duration factor (used to determine the duration of a critical section in simulation) ..... 117

**DS protocol :** the Direct Synchronization Protocol ..... 27

**EDM assignment :** the effective-deadline-monotonic priority assignment ..... 79

**EER time :** the end-to-end response time ..... 40

**GCS server :** the global critical section server ..... 102

**GDM assignment :** the global-deadline-monotonic priority assignment ..... 79

**IEER time :** the intermediate end-to-end response time of a subtask ..... 60

**MPCP :** the Multiprocessor Priority Ceiling Protocol ..... 99

**MPM protocol :** the Modified Phase Modification Protocol ..... 31

**NPDM assignment :** the normalized-proportional-deadline-monotonic priority assignment ..... 79

**PCP :** the Priority Ceiling Protocol ..... 12

**PDM assignment :** the proportional-deadline-monotonic priority assignment ..... 79

**PM protocol :** the Phase Modification Protocol ..... 28

**RG protocol :** the Release Guard Protocol ..... 34

**SBP :** the Stack-Based Protocol ..... 12

**SS protocol :** the Sporadic Server Protocol ..... 36

# Chapter 1

# Introduction

A real-time system demands that its results be both logically correct and delivered on time. A failure to obtain some result by a certain time may have disastrous effects. These real-time systems are often called *hard real-time systems*. In a hard real-time system, multiple threads execute concurrently. This thesis describes a suite of scheduling techniques to arrange the order and choose the time of execution for different threads in a distributed real-time system. It also describes a suite of schedulability analysis algorithms to determine whether every timing requirement of the system is met when threads in the system are scheduled via these techniques.

## 1.1 Background and Motivation

The multiple concurrent execution threads in a real-time system are often modeled as a set of *periodic tasks* [1]. In a single-processor system, a periodic task is an infinite stream of execution requests with the interval between any two consecutive requests being no shorter than the period of the task. We call each request an *instance* of the task, and we say an instance of a task is released when one execution request of the task arrives. A task is *strictly periodic* if the interval between the releases of every two consecutive instances is exactly equal to the period. Clearly, a task that is strictly periodic is a periodic task, but not vice versa. In a single-processor system, there is a ready queue to hold the released but not yet completed task instances. All the released task instances compete for the system resources (e.g., the CPU), following the order decided by the scheduler, which is an implementation of a certain scheduling algorithm.

*Priority-driven scheduling* is an effective approach to scheduling periodic tasks on processors. According to priority-driven scheduling, every released but not completed task instance has a priority. At any time, the task instance with the highest priority executes. An executing task instance continues its execution until either it voluntarily gives up the processor (e.g., due to the completion of the execution) or a higher priority task instance preempts it.

Among priority-driven scheduling schemes, *fixed-priority scheduling* has gained popularity due to its simplicity of implementation, ease of schedulability analysis and predictable behavior during transient overload situations. According to fixed-priority scheduling, all the instances of the same task have the same priority. For this reason, we often say that we assign a priority to a task rather than to a task instance. This thesis focuses on fixed-priority scheduling.

The *response time* of a task instance is the length of time from the instant when it is released to the instant when it completes. Oftentimes we use the "response time of a task" to mean the response time of an arbitrary instance of the task. Each task has a *relative deadline*, or simply *deadline*, which specifies the maximum allowed response time of the task. We say a task is schedulable if the response time of the task is no longer than its relative deadline, and a system is schedulable if every task in the system is schedulable.

In a hard real-time system, we need to ensure that every task is schedulable. For this purpose, we developed *schedulability analysis algorithms*. Such algorithms take the system parameters as input (e.g., periods, priorities, maximal execution times, and relative deadlines of tasks) and report whether the system is schedulable. We say that a schedulability analysis algorithm is *correct* if every system that the algorithm finds schedulable is indeed schedulable, and a schedulability analysis algorithm is *optimal* if it is correct and every system that the algorithm finds not schedulable is indeed not schedulable. (By a system "indeed not schedulable", we mean that we can construct a valid schedule where at least one task instance misses its deadline.) When it is impossible to obtain an optimal schedulability analysis algorithm, it suffices for us to have correct schedulability analysis algorithms.

In recent years, more and more real-time applications run on multiprocessor or distributed systems. The periodic task model needs to be extended accordingly to represent tasks in a distributed or multi-processor system. For example, in the system shown in Figure 1.1, the execution of a *monitoring* task takes three steps: (1) sampling sensor data on a field processor, (2) sending the sampled data over the communication link and (3) displaying the data on the

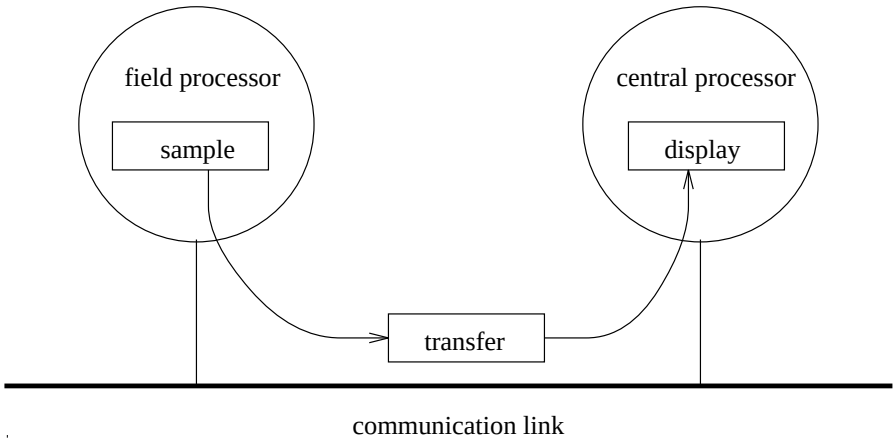



**Figure 1.1**: An End-to-End Task

central control processor. If we model the communication link as a processor and the message transmission as a task on the processor, the three steps can be viewed as three *subtasks* of the whole *monitoring* task. In this thesis, we focus our attention on tasks where each task consists of a chain of subtasks executing on different processors. Furthermore, each task is periodic in the sense that its first subtask is periodic, i.e., the interval between two consecutive releases of its first subtask is no shorter than the period of the task. After one instance of the first subtask completes, a corresponding instance of the second subtask can be released, and so on. We refer to the corresponding instances of all the subtasks of a task collectively as an instance of the task. For example, suppose that we call the three subtasks in Figure 1.1 *sample*, *transfer* and *display*. After one instance of subtask *sample* completes, the data that need to be transferred are ready and we may release an instance of subtask *transfer*. Similarly when the data arrive at the central processor, we may release an instance of subtask *display* to process the data. The interval between any two consecutive releases of subtask *sample* should be no shorter than the period.

We note that the timing constraint of the task in Figure 1.1 is not specified as a relative deadline of subtask *sample*, *transfer*, or *display*. Rather, it is specified as the maximum allowed interval from the time when a new sample of sensor data is taken on the field processor until the data is displayed on the central processor. In general, we call the time interval from the release of an instance of the first subtask until the completion of the corresponding instance of the last subtask the *end-to-end response time* of a task. The timing constraint for such a task is specified by an *end-to-end relative deadline*, or simply *end-to-end deadline*, which is the maximum allowed

end-to-end response time of the task. We call a task that consists of a chain of subtasks and has an end-to-end deadline an *end-to-end task* and a system that contains end-to-end tasks an *end-to-end system*. An end-to-end task is schedulable if its end-to-end response time is never greater than its end-to-end relative deadline, and the system is schedulable if every task in it is schedulable. This thesis deals with the *end-to-end scheduling problem*, i.e., how we schedule tasks with end-to-end deadlines in a distributed or multiprocessor real-time system so that the end-to-end system is schedulable.

In the context of fixed-priority scheduling, a rich set of solutions exist for scheduling uniprocessor real-time systems. There are several priority assignment methods that are optimal for various kinds of uniprocessor systems [1, 2, 3] and various schedulability analysis algorithms that verify if a system of periodic tasks scheduled according to a fixed-priority algorithm is schedulable [1, 4, 5, 6, 7, 8]. However, none of this work is suitable to solve the end-to-end scheduling problem. For example, according to the rate-monotonic priority assignment [1], all subtasks in a same parent task have the same priority, which is inversely proportional to the period. By doing so, we lose the freedom of assigning different priorities to different subtasks, and potentially decrease the schedulability of the system. Another problem is that some subtasks may not be periodic in a distributed real-time system, while the schedulability analysis methods mentioned above require that every subtask to be periodic. Consider the task in Figure 1.1 for example. If we release an instance of subtask *transfer* as soon as an instance of subtask *sample* finishes, subtask *transfer* is not periodic due to the possible varied response times of subtask *sample*. In this case, we either develop new scheduling analysis algorithms for this new situation or control the release of the subtasks so that they fit the periodic task model and the existing algorithms are applicable.

In general, three problems arise in fixed-priority, end-to-end scheduling.

**Priority Assignment :** How we assign the priorities to subtasks so that the system schedulability can be maximized.

**Execution Synchronization :** How we synchronize the execution of subtasks so that the precedence constraints are satisfied and the system schedulability, as well as other performance concerns, are optimized.

**Schedulability Analysis :** How we verify that a system of end-to-end tasks is schedulable, given a certain priority assignment method and execution synchronization method.

Existing work on fixed-priority end-to-end scheduling either does not address these problems together or addresses them inadequately. For example, Tindell et al. [9, 10] discussed the priority assignment problem in an end-to-end system, but they ignored the execution synchronization and schedulability analysis problems. Kao et al. [11, 12] studied the end-to-end scheduling problem in *soft real-time systems*, where the possibility of a task missing deadlines is minimized but allowed. Consequently, the schedulability of a system is expressed by a statistical value rather than a true/false assertion. Bettati addressed all three problems in his Ph.D. dissertation [13]. He proposed a solution for the execution synchronization problem, together with schedulability analysis algorithms, but full exploration of these problems is left open in his thesis. This thesis addresses these open problems and develops an end-to-end scheduling framework that gives an integrated, analytical and comprehensive treatment of all three problems.

## 1.2 Summary of Contributions

The foundation of the end-to-end scheduling framework described here is a simple but flexible end-to-end system model. In a distributed or multiprocessor system, the processors can be connected through a communication network, dedicated communication links or shared-memory architecture. Instead of dealing with each specific application and system architecture in an ad hoc manner, we deal with abstract tasks and processors and leave communication issues to be addressed separately. This not only broadens the application of the end-to-end scheduling framework, but also leads to a better understanding of the end-to-end scheduling problem.

A new formulation of the end-to-end scheduling problem is the second contribution of this thesis. Based on the end-to-end system model, three distinct but closely related problems are identified, namely, priority assignment, execution synchronization and schedulability analysis. As we briefly mentioned before, the approaches for uniprocessor real-time systems are not suitable for the end-to-end systems. Previous research work on fixed-priority end-to-end scheduling typically does not address and solve the end-to-end scheduling problem adequately. This thesis proposes an integrated framework to define and solve the end-to-end scheduling problem.

Solutions to the three individual problems mentioned previously are the building blocks of the end-to-end scheduling framework. We demonstrate that in the context of end-to-end scheduling the priority assignment problem is "NP-hard". We then propose several heuristic assignment methods. We present five *synchronization protocols* that govern the release and the execution of subtasks, including the one proposed by Bettati [13], as solutions to the execution synchronization problem. Several schedulability analysis algorithms are developed for end-to-end systems that use different priority assignment methods and different synchronization protocols. With these building blocks, the end-to-end scheduling framework offers a rich set of choices for scheduling a distributed or multiprocessor real-time system that fits the end-to-end system model.

The end-to-end scheduling framework also provides a solution to the resource contention problem in a distributed or multiprocessor real-time system. During its execution on a local processor, a task may need to access a resource that is on a remote processor. Resource contention arises when multiple tasks need to have exclusive access to the same resource, and therefore the execution of these tasks can be delayed. To be able to verify the schedulability of such a system, we must control the access to the shared resources and bound the delay that each task can experience. In this thesis we propose an end-to-end scheduling based approach to the resource contention problem. Compared with the existing solution, known as the *Multiprocessor Priority Ceiling Protocol (MPCP)* [14], the end-to-end scheduling approach is more flexible and yields better performance in most cases.

## 1.3 Organization of the Thesis

In Chapter 2, we provide an overview of the related work. The work presented in this thesis is closely related to the real-time scheduling theory for uniprocessor systems and existing work on fixed-priority end-to-end scheduling. Specifically, the schedulability analysis algorithms in this thesis owe their fundamental techniques to those for uniprocessor systems. The end-to-end scheduling approach to the resource contention problem builds on the Priority Ceiling Protocol (PCP) [15] and the Stack Based Protocol (SBP) [16] for a uniprocessor system, and is related to the MPCP for a distributed or multiprocessor system. This thesis is also related to classical scheduling theory in that the end-to-end system model is a periodic version of the *job-shop* model studied in the classical scheduling theory.

In Chapter 3, we formally present the end-to-end system model. The model is simple enough to allow the analysis of timing properties and yet flexible enough to model most real-time applications and system architectures. We formulate the three problems in the end-to-end scheduling, namely the priority assignment problem, the execution synchronization problem, and the schedulability analysis problem, based on this model.

We discuss the execution synchronization problem and the schedulability analysis problem in Chapters 4 and 5, and leave the discussion of the priority assignment problem to Chapter 6. In Chapter 4, we present five synchronization protocols as solutions to the execution synchronization problem. Two performance issues of the synchronization protocols, the algorithm complexity and the run-time overhead, are also addressed.

For each synchronization protocol, we develop schedulability analysis algorithms that can be used to determine whether an end-to-end system using the specific synchronization protocol is schedulable. In general, the verification of the schedulability of a system is achieved by bounding the worst-case end-to-end response time of a task and comparing the bound with the end-to-end deadline of the task. We present these algorithms in Chapter 5.

In Chapter 6, we demonstrate that the priority assignment problem is NP-hard. We proceed to present several deadline-based heuristic assignment methods. Related work on the priority assignment is discussed in Chapter 6.

Given such a rich set of solutions to the end-to-end scheduling problem, it is necessary for us to understand the strengths and weaknesses of each solution to each individual problem. In Chapter 7, we describe a series of experiments that were done to evaluate the performance of different end-to-end scheduling algorithms.

In Chapter 8, we address the resource contention problem in a distributed system. We first point out some of the weaknesses of the MPCP, and then introduce the end-to-end scheduling approach towards the same problem. Simulation was conducted to compare the performance of the end-to-end scheduling approach with the MPCP approach. Simulation results show that in many cases the end-to-end scheduling approach has better performance.

In Chapter 9, we list a few future directions for the study of end-to-end scheduling, including scheduling with relaxed precedence constraints among subtasks, hybrid end-to-end scheduling and dynamic-priority end-to-end scheduling. An in-depth discussion of the Sporadic Server algorithms

is attached to the thesis as Appendix A. These algorithms are pertinent to the implementation of one of the synchronization protocols.

# Chapter 2

# Related Work

The related work to this thesis can be roughly classified into four areas:

1. real-time scheduling theory,
2. resource access protocols in real-time systems,
3. real-time communications, and
4. classical scheduling theory.

Among these four areas, this thesis is most directly related to real-time scheduling theory and resource access protocols. It is closely related to real-time communication, and more remotely related to classical scheduling theory. This chapter gives an overview of existing work in these areas.

## 2.1 Real-Time Scheduling Theories

Many real-time scheduling theories have been developed in the last twenty years. With a few exceptions, existing theories are on uniprocessor systems. In many cases, approaches and solutions in end-to-end scheduling presented in this thesis are extensions to or based on existing real-time scheduling algorithms. This is especially true for the solutions of the priority assignment problem and the schedulability analysis problem.

### 2.1.1 Uniprocessor Scheduling Theories

Two related problems arise on fixed-priority scheduling of periodic tasks in a uniprocessor system: priority assignment and schedulability analysis. In their classic paper [1], Liu and Layland proposed the rate-monotonic priority assignment for tasks with relative deadlines equal to their periods. They proved that the rate-monotonic assignment is optimal among all fixed-priority assignments in the sense that if such a system is not schedulable according to the rate-monotonic assignment, no other fixed-priority assignment can make it schedulable. The schedulability can be verified by comparing the sum of the total utilization factor of the tasks with a single utilization bound. Leung and Whithead [2] looked into the cases where tasks may have arbitrary relative deadlines. They proposed deadline-monotonic priority assignment, which is an optimal priority assignment among fixed-priority assignments as long as the relative deadlines are shorter than their respective periods. For tasks with relative deadlines longer than their periods, both the rate-monotonic assignment and deadline-monotonic assignment are not optimal. Audsley [3] devised a pseudo-polynomial algorithm that finds a feasible priority assignment if one exists.

On schedulability analysis, Joseph and Pandya [4] proposed an algorithm to compute the response time of a task instance which is released at the same time as the instances of all other tasks. This response time is the worst-case response time of the task if it is no greater than the period. Lehoczky [5] proposed a sufficient and necessary test to verify the schedulability of tasks if the relative deadline of the task is no greater than its period. For tasks whose response times and relative deadlines are longer than the periods, Lehoczky [6] formulated the concept of an *i-level busy period* to analyze the worst-case behavior of a task. The busy period analysis method was further extended by Audsley et al. [7, 8, 17] to account for bursty activities, various blocking times, and release jitters.

### 2.1.2 End-to-End Scheduling Theories

We have not seen much work on fixed-priority scheduling of periodic tasks with end-to-end deadlines. A common approach to this problem is to decompose an end-to-end system into a set of uniprocessor systems [18] and then apply some existing real-time scheduling algorithms for uniprocessor systems. By doing so, an integrated end-to-end scheduling framework is not needed. However, two issues are not addressed. The first issue is how subtasks in the same task

communicate and synchronize with each other. Loose synchronization between two adjacent subtasks, such as in the producer/consumer style, is sometimes assumed. The other issue is how to guarantee that tasks meet their end-to-end deadlines, especially when adjacent subtasks are loosely synchronized. The existing answers to this question are generally pessimistic.

Various aspects of the end-to-end scheduling problems have been examined by people in different contexts. For example, Tindell et al. [19] attempted to compute upper bounds on the end-to-end response times when the *Direct Synchronization* protocol is used (This protocol will be defined in Chapter 4.). However, the interference of instances of the same subtask is ignored in Tindell's work, and the result in [19] is not correct. Our approach, presented in Chapter 5, takes this factor into account, and yields correct results. Tindell et al. [19] studied the priority assignment problem in a distributed system where tasks have end-to-end deadlines. Garcia and Harbour [10] proposed a refined solution to a similar problem. This work complements our work on heuristic priority assignment methods.

Bettati [13] studied the problem of scheduling a set of tasks with arbitrary release times and end-to-end deadlines in a flow-shop system. He then extended this work to the scheduling of periodic flow-shop tasks. His *Phase Modification* technique is incorporated as one of the synchronization protocols in this thesis. In this thesis, the task model is extended to periodic job-shop tasks. In addition to the Phase Modification synchronization protocol, we present four additional protocols. We improve the schedulability analysis algorithms for the Phase Modification protocol and develop new ones for the new protocols. The priority assignment problem is also addressed in this thesis, and two heuristic priority assignment methods, which are more suitable for end-to-end systems according to our simulation results, are proposed.

## 2.2 Resource Access Control Protocols

We apply the end-to-end scheduling framework to solve the resource contention problem in a distributed or multiprocessor real-time system. This is closely related to previous work on the resource contention problem in a uniprocessor system or a multiprocessor system.

In a real-time system, the exclusive access to a resource is typically achieved by semaphore-like operations. Sha [15] and Rajkumar [20] have shown that careless use of semaphores can lead to unpredictable timing behavior in a real-time system and a resource access control protocol

is needed to keep the blocking times of tasks bounded. Moreover, the blocking times of tasks can be easily taken into account in the schedulability analysis. For a uniprocessor system, Sha et al. [15] developed the Priority Ceiling Protocol (PCP). According to the PCP, the blocking time that each task instance can experience is no more than the duration of one critical section of a lower-priority task. Baker [16] developed the Stack Based Protocol (SBP) which gives the same upper bound on the blocking time of each task as with the PCP. In addition, the SBP is applicable to both fixed-priority scheduling and dynamic-priority scheduling.

In a distributed system, Rajkumar et al. [21] looked at the case where tasks execute locally on their host processors but need to access resources on remote processors. They extended the PCP to the Multiprocessor Priority Ceiling Protocol (MPCP) and developed a way to compute an upper bound on the blocking time of each task. However, the performance of the MPCP is sometimes poor.

Bettati recognized that the problem addressed by the MPCP can be solved by an end-to-end scheduling based approach [13], but did not explore this direction of research further. In this thesis, we pursue a full exploration of the end-to-end scheduling approach to the resource contention problem and compare the performance of the end-to-end scheduling approach and the MPCP approach.

## 2.3 Real-Time Communication

Due to the recent development in multi-media applications and communication networks, there is an increasing amount of research effort on real-time communications. This thesis is related to the work on scheduling message transmissions in packet-switching or cell-switching networks, such as ATM networks. The transmitted messages are from real-time sources, such as video conferencing and MPEG movies. They are continuous and satisfy certain burstiness constraints. The design objectives of the network often include (1) a bounded end-to-end delay on a message delivery, (2) a bounded buffer size at each switch or hop, and sometimes (3) a small end-to-end delay jitter. To achieve these objectives, real-time scheduling techniques, such as prioritized messages and rate-based scheduling, are used to schedule the messages at each switch.

A *service discipline* refers to a message scheduling scheme used to switch the packets or cells. Zhang and Ferrari [22] divided service disciplines into two categories, *work-conserving* disciplines,

which never allow an outgoing link to idle when there are messages waiting to be sent on the link, and *non-work-conserving* disciplines, which may allow outgoing links to be idle in the presence of waiting messages. Rene [23, 24] proposed a *rate-controlled* service discipline by using a $(\sigma, \rho)$-regulator. This work-conserving scheme yields good performance in terms of the end-to-end delay and the buffer size. Many similar schemes have been proposed, including Delay Earliest-Due-Date (Delay-EDD) [25, 26, 27], Virtual Clock [28], Fair Queuing [29], and Generalized Processor Sharing [30, 31]. On the other side, non-work-conserving disciplines include Jitter Earliest-Due-Date (Jitter-EDD) [32], Stop-and-Go [33], Hierarchical Round Robin (HRR) [34], and Rate-Controlled Static Priority (RCSP) [22].

In most of the above-mentioned schemes, each message stream is assumed to satisfy certain burstiness constraints, such as the $(\sigma, \rho)$ burstiness constraint[1] and the leaky bucket constraint, when it enters the network. In cases where applications are ill-behaved, a $(\sigma, \rho)$-regulator can be used to enforce the burstiness constraint. At each switch, we either try to preserve the original burstiness patterns, such as in non-work-conserving disciplines, or keep the burstiness of data flow under control, such as in $(\sigma, \rho)$-regulator and HRR schemes.

Overlaps exist between the message scheduling problem and the end-to-end task scheduling problem. For example, in both studies, a common objective is to obtain a bounded end-to-end delay. Not surprisingly, we see that many solutions to these problems are similar in spirit, especially on the execution synchronization problem in the end-to-end task scheduling. For example, one conclusion drawn in this thesis is that a system has a better schedulability if the release of subtasks is controlled. This conclusion corresponds to the observation in real-time communications that the rate-controlled discipline yields shorter overall end-to-end delays than the uncontrolled version.

Despite the commonalities, many fundamental differences between these two areas exist, and these differences prevent the solutions to one problem from being directly applicable to the other. In a modern packet-switching network, the message scheduling is performed on-the-fly and implemented in hardware. The scheduling scheme must be simple and straightforward. The situation is much better in task scheduling, where the task execution times are generally much

[1] A message stream satisfies $(\sigma, \rho)$ constraint if and only if for any time interval with duration of $d$, the total amount of messages that need to be transmitted is less than $\sigma + \rho d$.

larger than the scheduling overhead. Consequently, it is possible to implement more complicated scheduling algorithms which may not be suitable for real-time communication networks.

Another difference between the two areas is that the precedence constraints between subtasks in an end-to-end task is by nature different from the precedence constraints of a message traversing through a series of switches. In message transmission, when the first packet of a message is sent from switch A to switch B, switch B can immediately forward the packet to switch C. In the case of an end-to-end task, however, the first subtask has to complete entirely on the first processor before the second subtask can start on the second processor. As a result, round-robin-based schemes, which are effective in message transmission scheduling, can lead to quite long end-to-end response times of end-to-end tasks.

## 2.4 Classical Scheduling Theory

A rich set of scheduling theories has been developed in the context of operations research. In this context, the problem is how to schedule a set of jobs on a set of machines, and the goal of scheduling is to maximize the productivity. An excellent survey was done by Lawler et al. [35], who classified the problems and provided an overview of the known solutions. Many topics are also covered in [36] by Blazewicz. Xu and Parnas [37] gave insights on how classical scheduling theories can be used to solve real-time scheduling problems. Recently, Stankovic et al. [38] pointed out some implications of classical scheduling theories in the design of real-time systems. Nevertheless, classical scheduling theories are in general not directly applicable to real-time scheduling due to different assumptions.

Almost all work on classical scheduling assumes a *static workload* where there are a finite number of tasks and the task parameters are known *a priori*. Each task has only one instance. Oftentimes, all tasks have the same release times. Consequently, the resultant schedule has a finite length. In contrast, the workload in a real-time application typically consists of a set of periodic or sporadic tasks, and the schedule lasts forever. It may be argued that a periodic task set can be transformed to a static set of tasks by confining our attention to the task instances in a *hyper-period*, an interval whose duration is equal to the least common multiple of all the periods. After this transformation, it becomes possible to apply classical scheduling theories to the transformed task set. Two issues still need to be addressed. First, a periodic task may

not be released periodically, and thus a definite transformation is impossible. Secondly, even if this transformation is possible, we need to consider Graham's anomalies [39] when trying to determine whether every task instance will meet its deadline. Because the task instances have variable execution times, examining the schedule where each task has its maximum execution time may not be sufficient to draw a conclusion about the schedulability of tasks. The general problem of predicting the completion times of task instances in the presence of anomalies has not been solved [40].

The end-to-end scheduling problem studied in this thesis resembles the classical *job-shop* scheduling problem. In the job-shop model, each task needs to execute on a set of processors in a certain order, and each task may require a different order. A similar but more restricted model, called the *flow-shop* model, where all tasks have the same orders, was studied and extended to end-to-end real-time scheduling by Bettati [13]. Like many other works in classical scheduling, efforts on job-shop scheduling problems have concentrated on minimizing the length of the schedule for a static workload. Johnson [41] provided an algorithm to find a minimum length schedule for a flow-shop system with two processors in $O(n \log n)$ time. A simple extension provided by Jackson [42] yields a solution to the 2-processor job-shop scheduling problem with the same time complexity. Hefetz and Adiri [43] provided an algorithm for computing a minimum length schedule for a two-processor job-shop system with unit-length tasks in linear time. Brucker solved the similar problem of 2-processor unit-time job-shop scheduling to minimize the maximum lateness [44, 45]. However, the problems of scheduling of any non-trivial job-shops with more than two processors are NP-hard [46, 47, 48]. To the best of our knowledge, there is no algorithm that produces *zero unit penalty* schedules for a job-shop system, where a zero unit penalty schedule ensures that every task meets its deadline.

# Chapter 3

# System Model

In this chapter, we formally describe the end-to-end system model that serves as the basis of our work. We also discuss several modeling techniques that map real-world applications to the end-to-end system model, to show the flexibility and usefulness of the model. Through a simple example, we then illustrate and define the three problems in end-to-end scheduling, namely the priority assignment problem, the execution synchronization problem and the schedulability analysis problem.

## 3.1 Assumptions

A task in a distributed real-time system is called an *end-to-end* task if it consists of a chain of subtasks and has an *end-to-end* deadline. We call a real-time system an *end-to-end system* if it consists of more than one processor and a set of end-to-end tasks. The following list further explains the notion of an end-to-end task and end-to-end system, as well as other aspects of an end-to-end system.

1. The system consists of a set $\{P_i\}$ of processors and a set $\{T_i\}$ of tasks.

2. Each task $T_i$ consists of a chain of $n_i$ subtasks, $T_{i,1}$, $T_{i,2}$, ..., $T_{i,n_i}$. $T_i$ is referred to as the *parent task* to its subtasks, and subtasks are referred to as *sibling subtasks* to each other if they have the same parent task.

3. Each request for execution of a subtask is called an instance of that subtask and the corresponding instances of all subtasks are collectively called an instance of their parent task.

4. Subtask $T_{i,j}$ is a *predecessor (successor)* of subtask $T_{i,k}$ if $j < k$ $(j > k)$, and $T_{i,j}$ is the *immediate predecessor (immediate successor)* of $T_{i,k}$ if they are also adjacent (i.e., $|j-k| = 1$). Similarly, if $T_{i,j}(m)$ denotes the $m$th instance of $T_{i,j}$, $T_{i,j}(m)$ is a *predecessor (successor)* of $T_{i,k}(m)$ if $j < k$ $(j > k)$, and $T_{i,j}(m)$ is the *immediate predecessor (immediate successor)* of $T_{i,k}(m)$ if $T_{i,j}$ and $T_{i,k}$ are also adjacent. An instance of a later subtask $T_{i,j}$ for $j > 1$ cannot be released until its immediate predecessor completes.

5. Each task $T_i$ is a periodic task with period $p_i$, meaning that the first subtask $T_{i,1}$ is a periodic subtask with the period $p_i$. A subtask that is later in the subtask chain may or may not be released periodically, depending on the particular synchronization protocol used in the system. For the sake of convenience, we still say that every subtask has a period $p_{i,j}$ which is equal to the period $p_i$ of its parent task.

6. The release time of the first instance of $T_{i,1}$ is the *phase* $f_i$ of task $T_i$.

7. If an instance of $T_{i,j}$ is released before the previous instances of $T_{i,j}$ complete, there are multiple instances of $T_{i,j}$ in the ready queue. In this case, all instances of $T_{i,j}$ are scheduled on the FIFO basis, i.e., an instance of $T_{i,j}$ cannot start to execute before the previous instances of $T_{i,j}$ complete.

8. Each task $T_i$ has a relative end-to-end deadline $D_i$. In other words, for each instance of $T_{i,1}$ released at time $t$, the corresponding instance of $T_{i,n_i}$ must complete by time $(t + D_i)$. Otherwise, we say that this instance of the task misses the deadline.

9. Each subtask $T_{i,j}$ has a maximum execution time $\tau_{i,j}$, meaning that the execution time of each of its instances is no greater than $\tau_{i,j}$.

10. Subtasks are statically assigned to processors.

11. Each subtask has a fixed priority $\phi_{i,j}$, i.e., all instances of the subtask have the same priority. A preemptive, fixed-priority scheduling algorithm governs the execution of subtasks. In other words, when an instance of a subtask is released, it is put in a ready queue. At any

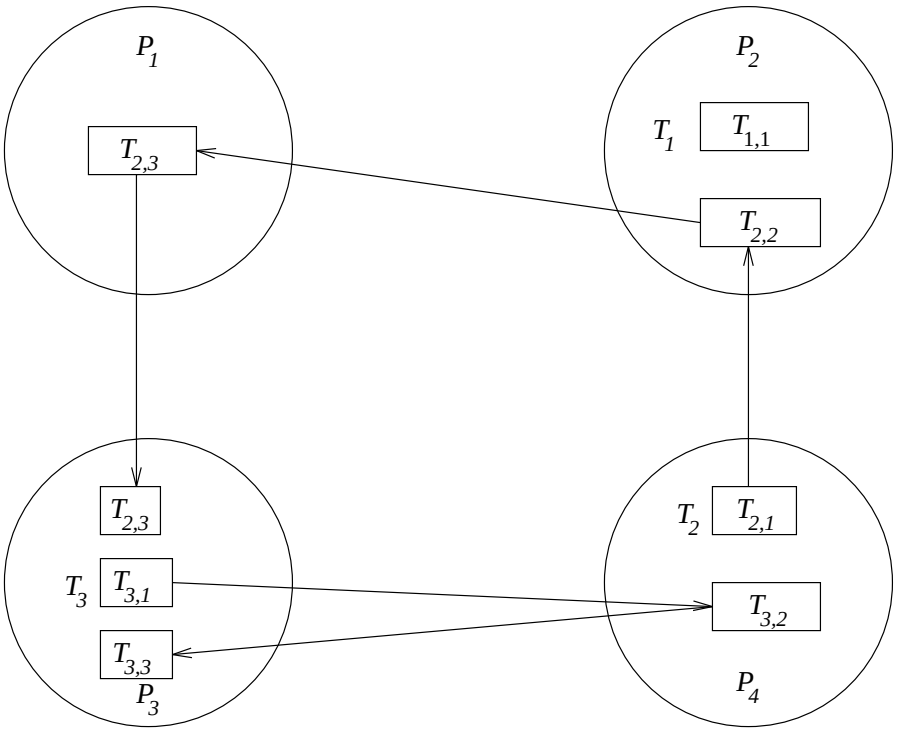



**Figure 3.1**: An End-to-End System

time, the scheduler chooses to execute the subtask instance with the highest priority from the ready queue. The current executing subtask instance executes to completion, unless it is preempted by a newly-released subtask instance with a higher priority.

Figure 3.1 shows a simple end-to-end system. There are four processors and three tasks in the system. (Each circle represents a processor, and each box in a circle represents a subtask that executes on the processor represented by the circle.) Task $T_1$ has only one subtask, which is the trivial case. Task $T_2$ has four subtasks; they execute in turn on processor $P_4$, $P_2$, $P_1$ and $P_3$. Task $T_3$ executes on processor $P_3$ first, then on $P_4$ and then back to $P_3$ again. We call a task, such as $T_3$, a *recurrent end-to-end task* because it executes on the same processor more than once, or, equivalently, it has more than one subtask on the same processor.

We do not explicitly prohibit two or more adjacent subtasks from being on the same processor. The schedulability analysis algorithms, which will be described in Chapter 5, remain correct when such a situation arises. In practice, we can hardly find a reason for doing so. Adjacent subtasks on the same processor can always be treated as one, and performance will be better. Harbour et

al. [49] studied the case, in which subtasks with different fixed priorities executing on a single processor, and developed schedulability analysis algorithms for bounding the task response times.

In the previous list of assumptions, we deliberately avoided issues on resource contention. It is possible, however, to take resource contention into account in this model. If the resource contention is among the subtasks on the same processor, the PCP or SBP can be used to control access to the resources. The maximum blocking time that a subtask instance can experience is bounded, and the blocking time can be counted into the schedulability analysis algorithms to obtain correct timing bounds. If the contended resource is a global resource, i.e., it is accessed by subtasks on different processors, Chapter 8 presents a solution to this problem. Until Chapter 8, we ignore the resource contention problem.

The system model makes no assumptions about the communication overhead. Obviously, the overhead may not be negligible in practice. The model can take it into account in several ways. First of all, any implementation of a communication link must guarantee a maximum delay in a message delivery in order to obtain a bound on the end-to-end response time of a task. Provided that this is true, the following examples show how to capture communication overhead within the model.

**Prioritized Bus** In the case when the communication link is a prioritized bus, such as CAN [50], each message has a fixed priority. Processors that want to send messages compete for the bus by putting the priorities of the messages on the bus, and the processor with highest message priority gains the control over the bus. A higher priority message may experience some blocking time when it is ready to be sent while a lower priority message is being sent at that time. Such a communication link can be modeled as a "link" processor, and the transmission of each message is modeled as a "message" subtask on the "link" processor. The maximum execution time of a "message" subtask is equal to the maximum time needed to deliver the message when it is transmitted alone. The blocking time of the "message" subtask is equal to the maximum delay it can experience due to non-preemptive segments in lower priority messages. Therefore, such a system fits the end-to-end system model described in the earlier sections, and we need not consider the communication cost separately. For example, if the communication link in Figure 1.1 is a prioritized bus, the system can be modeled as an end-to-end system with three processors, as shown in Figure 3.2.

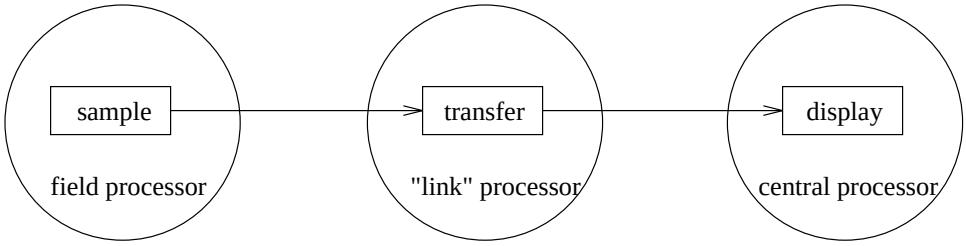



**Figure 3.2**: Mapping a Prioritized Bus to a "Link" Processor

**Dedicated Links** In many embedded control systems, dedicated links are used to connect processors. Interference exists only among messages that are sent from the same processor, while there is no interference among messages sent from different processors. Two cases can be further identified. In some implementations, a message transmission needs the intervention of the CPU, such as in polling or synchronized message sending. In this case, subtasks do not relinquish the CPU during message transmission. Consequently, the communication link can be treated as a local, shared resource, and sending a message is a critical section in the sender subtask. The communication cost can thus be treated as the duration of the critical section and can be incorporated as part of the execution time for the purpose of schedulability analysis. In the other case, sending messages does not need the intervention of the CPU, such as message transmissions controlled by a DMA controller. In this case, the maximum time needed to transmit a message, including the possible blocking time experienced by the sender, can be added into the blocking time of the sender subtask. After such care is taken, our schedulability analysis algorithms will remain correct.

**Other Cases** In the case of networks, such as FDDI [51], where the maximum delay of each message transmission is bounded but the networks are neither prioritized buses nor dedicated links, we again model the communication link as a "link" processor and the message transmissions as the subtasks on the "link" processor. The only difference with respect to our end-to-end system model is that the "link" processor uses a scheduling algorithm different from a fixed-priority scheduling algorithm. This presents a new problem, called *hybrid end-to-end scheduling*, where a mixture of different scheduling paradigms are used on different processors. A thorough study of this problem is beyond the scope of this thesis. In Chapter 9, we will discuss briefly how

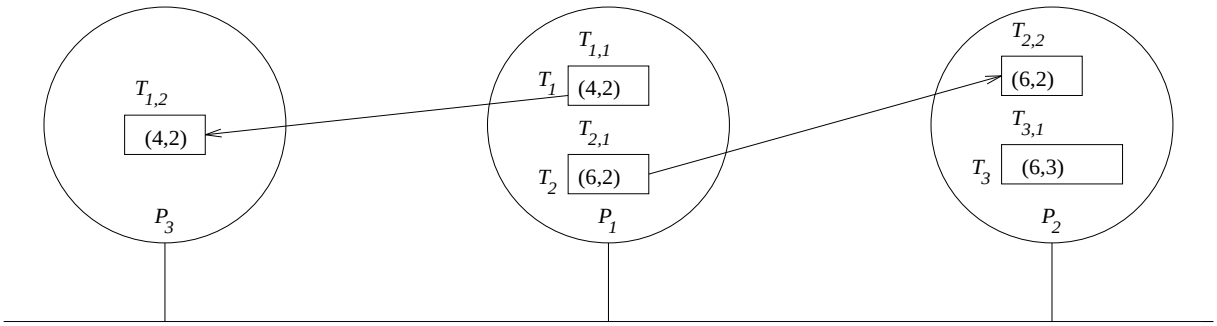



**Figure 3.3**: An Example to Illustrate the Problems in End-to-End Scheduling

to extend the algorithms presented in this thesis to provide solutions to the hybrid end-to-end scheduling problem.

## 3.2 Problems in End-to-End Scheduling

To illustrate the problems in scheduling an end-to-end system, let us follow the development of a simple end-to-end system. Suppose that after preliminary system design, coding and workload partitioning, we obtain an implementation shown in Figure 3.3. There are three processors and three tasks in the system. Again, each circle represents a processor, and a box in a circle represents a subtask that executes on the processor represented by the circle. Tasks $T_1$ and $T_2$ have two subtasks each; these subtasks are $T_{1,1}$, $T_{1,2}$, $T_{2,1}$ and $T_{2,2}$. Task $T_3$ has only one subtask. For the sake of consistency, we call it subtask $T_{3,1}$. Subtask parameters are given in their corresponding boxes. The first number is the period and the second number is the execution time. The deadline of each task is equal to its period.

**Priority Assignment** When the subtasks are to be scheduled on a fixed priority basis, the first issue we face is assigning priorities to the subtasks. On processor $P_1$, if we assign a higher priority to $T_{2,1}$, then an instance of $T_{1,1}$ can be preempted once by an instance of $T_{2,1}$, causing the response time of the instance of $T_{1,1}$ to be 4 and leaving no time for the corresponding instance of $T_{1,2}$ to execute before the deadline. This instance of $T_1$ will miss its deadline. Therefore we should assign priorities the other way around (i.e., let $T_{1,1}$ have a higher priority) to prevent $T_1$ from missing deadlines. Similarly, on processor $P_2$ we should assign a higher priority to subtask $T_{2,2}$ than $T_{3,1}$.

In general, in a fixed-priority end-to-end system, assigning priorities to subtasks so that the system is schedulable is called the *priority assignment problem*. We say that a priority assignment is *feasible* if according to the assignment the system is schedulable. (A more precise definition of feasible priority assignment can be found in Chapter 6.) Solutions to the priority assignment problem are priority assignment methods, where an *optimal* method always finds a feasible priority assignment if such an assignment exists, and a good heuristic method is more likely to yield feasible assignments than a random assignment. In Chapter 6 we will see that the priority assignment problem is in fact a family of sub-problems which are not likely to have efficient optimal solutions. We will present several heuristic methods there.

**Execution Synchronization** The second problem in end-to-end scheduling is concerned with the releases of subtask instances which are not of the first subtask in a chain. Consider subtask $T_{2,2}$ in Figure 3.3 for example. When an instance of $T_{2,1}$ completes, we can release a corresponding instance of $T_{2,2}$ on $P_2$ immediately. However, we are not obligated to do so. In fact, we may want to intentionally delay the release of the corresponding instance of $T_{2,2}$ if by doing so we can improve the schedulability of the system.

Figure 3.4 shows the first 10 time units in the schedules on $P_1$ and $P_2$ if we release an instance of $T_{2,2}$ immediately after the corresponding instance of $T_{2,1}$ completes. Each dotted arrow in the figure represents a synchronization signal sent from $P_1$ to $P_2$, indicating that an instance of $T_{2,1}$ has just completed. Upon receipt of this signal, an instance of $T_{2,2}$ is released immediately. In the schedule, we let the phase of $T_3$ be 4 and the phases of all other tasks be 0. We notice that $T_3$ misses its deadline at time 10 because the first instance of $T_{3,1}$ is preempted twice by $T_{2,2}$. This situation happens because the completion time of subtask $T_{2,1}$ is not periodic, either due to interference of $T_{1,1}$ or due to the fact that the actual execution time can be shorter than the maximum execution time. Since an instance of $T_{2,2}$ is released immediately after the completion of the corresponding instance of $T_{2,1}$, two instances of $T_{2,2}$ may be released within a 6-time-unit interval, as shown in Figure 3.4, causing $T_{3,1}$ to miss its deadline. In contrast, if instances of $T_{2,2}$ were released periodically at the period of 6, $T_{3,1}$ surely would never miss any deadline because $T_{2,2}$ and $T_{3,1}$ have the same periods and their total utilization is less than 1.

One way to improve the schedulability of $T_3$ is to always release an instance of $T_{2,2}$ 4 time units after the release of the corresponding instance of $T_{2,1}$. It is always feasible for us to do so,

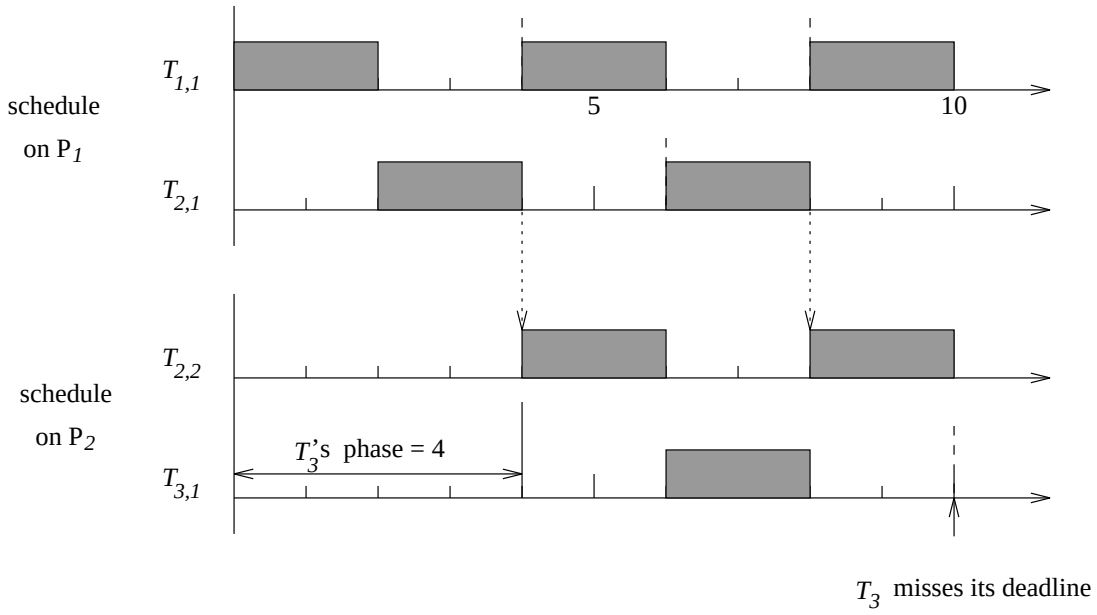



**Figure 3.4**: The Schedule of the System in Figure 3.3 when Every Instance of $T_{2,2}$ is Released Immediately after the Corresponding Instance of $T_{2,1}$ Completes

because the worst-case response time of $T_{2,1}$ is no more than 4 time units. In other words, 4 time units after the release of an instance of $T_{2,1}$ the instance has surely completed. Consequently we are able to release the corresponding instance of $T_{2,2}$ at that time without violating the precedence constraint between $T_{2,1}$ and $T_{2,2}$. Thus, the release of $T_{2,2}$ is made periodic, and any instance of $T_{3,1}$ can only be preempted once by an instance of $T_{2,2}$. The worst-case response time of $T_{3,1}$ is no more than 5 time units, which is less than the relative deadline. $T_3$ becomes schedulable.

In general, the *execution synchronization problem* is concerned with when instances of a subtask should be released when the subtask is not the first in the chain. As solutions to this problem, we propose several *synchronization protocols*, which will be presented in detail in Chapter 4. In addition to the impact on system schedulability, which we have seen in the previous paragraphs, synchronization protocols also have impact on other aspects of system performance. For example, tasks may have different average end-to-end response times when different protocols are used. Also, different protocols may have different overhead and require different scheduling support. Qualitative comparisons of some aspects of their performance are discussed in Chapter 4, while a quantitative performance comparison of these issues is covered in Chapter 7.

**Schedulability Analysis** Once we have assigned the priorities to subtasks and chosen an appropriate synchronization protocol, the next question is whether every instance of each task will meet its deadline. Earlier, we determined the schedulability of tasks in Figure 3.3 in an *ad hoc* fashion. This is possible only because the system is small and tasks are simple. We need a general, systematic method to determine the system schedulability. This is the third problem in the end-to-end scheduling, namely the *schedulability analysis problem*, which entails verifying the schedulability of an end-to-end system for a given priority assignment and a given synchronization protocol that is used to schedule the system.

To solve this problem, we need *schedulability analysis algorithms*. We may need different schedulability analysis algorithms for different synchronization protocols because the protocols yield different timing behavior of a system, as we have demonstrated by the simple example above. Oftentimes, the schedulability analysis algorithm is not optimal in the sense that it does not give a necessary and sufficient condition for schedulability. In other words, the algorithm may report that the system is not schedulable when it actually is. For this reason, another important issue in schedulability analysis is how to refine and improve the algorithms to yield more accurate predication about the system schedulability.

**Relation Among the Three Problems** We have identified and illustrated the three problems in end-to-end scheduling. These problems are closely related. For example, given a priority assignment alone, we cannot assert that a system is schedulable without knowing the synchronization protocol used in the system, and *vice versa*. Even if a system using a particular priority assignment and a particular synchronization protocol is indeed schedulable, we cannot tell it from systems which are indeed not schedulable if the best existing schedulability analysis algorithm reports that the system is not schedulable. From this perspective, the performance of a priority assignment method and a synchronization protocol is with respect to the best schedulability analysis algorithm, not with respect to the actual system schedulability they yield. In the following three chapters, we will address these three problems in detail. Due to the logical dependency between these three problems, we will discuss the execution synchronization and schedulability analysis problems first and the priority assignment problem last. In our discussion on the execution synchronization and schedulability analysis, we will assume that a certain fixed priority assignment is given for the system.

# Chapter 4

# Synchronization Protocols

## 4.1 Introduction

In this chapter, we describe five synchronization protocols that govern the release of subtasks and control the execution of subtasks. A valid protocol must satisfy two necessary conditions.

1. The precedence constraints among subtasks in the same task must be satisfied. No protocols should release an instance of a subtask before the corresponding instance of its predecessor has completed.

2. The synchronization protocol must render itself to schedulability analysis. In other words, we must be able to bound the end-to-end response times of tasks in a system using the protocol. For easy reference in later discussion, we abbreviate the end-to-end response times as the EER times.

We use the following criteria to measure the performance of each valid protocol.

**Bounds on Task EER Times :** When the best schedulability analysis algorithm for a synchronization protocol can yield finite upper bounds on task EER times, we refer to the bounds as the bounds yielded by the synchronization protocol. For a hard real-time system, we prefer synchronization protocols to yield small bounds on task EER times.

**Average EER Times :** In many applications, we want the average EER times to be as short as possible while in other applications we see no advantage if a task finishes earlier than the worst case.

**Complexity :** A more complex algorithm is more difficult to implement, and oftentimes incurs a greater *run-time overhead*.

**Run-Time Overhead :** The overhead includes the number of interrupts and the number of context switches associated with each subtask instance. Ideally these numbers should be bounded and small for each subtask instance.

**Output Jitter :** For an individual task, the output jitter is the difference between the worst-case EER time (or the upper bound on the EER time) and the best-case EER time (or the lower bound on the EER time). The smaller the difference, the smoother the output. Large output jitter often leads a large amount of buffering space required to hold the intermediate and final results computed by the subtasks.

It is impossible to optimize a synchronization protocol with respect to all these performance measures simultaneously. For example, given a fixed worst-case EER time, a smaller average EER time can imply a larger output jitter. Consequently, no protocol is superior to another with respect to all the above performance measures. Our study intends to reveal the strengths and the weaknesses of the protocols.

## 4.2 The Synchronization Protocols

In the following description of each protocol, we discuss the complexity of the algorithm and the run-time overhead while leaving other performance issues to Chapter 7. For the run-time overhead, we include the number of context switches and the number of interrupts associated with each subtask instance. Interrupts are further classified into timer interrupts and synchronization interrupts. A timer interrupt happens at each time instant set by the scheduler. A synchronization interrupt happens when an instance of a subtask completes, and a synchronization signal is sent to notify the scheduler on the processor where the immediate successor of the subtask executes. By assumption, each instance of the first subtask in each task is triggered by a timer interrupt[1]. For example, suppose a task $T_i$ has two subtasks, $T_{i,1}$ and $T_{i,2}$, on two processors $P_1$ and $P_2$. A timer interrupt happens periodically with the interval equal to $p_i$ on $P_1$. In response to each

[1]In practice, the instance of the first subtask might be triggered by an external event. As far as cost is concerned, the overhead is the same as a timer interrupt.

interrupt, the scheduler on $P_1$ releases an instance of $T_{i,1}$ and puts it in the ready queue for execution. After an instance of $T_{i,1}$ completes, depending on the synchronization protocol, the scheduler on $P_1$ immediately or sometime later sends a synchronization signal to the scheduler on $P_2$. A synchronization signal causes a synchronization interrupt on $P_2$. Upon receipt of the synchronization signal, the scheduler on $P_2$ may release an instance of $T_{i,2}$ immediately or later, depending on the synchronization protocol used.

### 4.2.1 Direct Synchronization (DS) Protocol

Recall that in the previous chapter we discussed a simple way to control the release of subtasks, i.e., an instance of a subtask is released as soon as its immediate predecessor completes. We call this protocol the Direct Synchronization Protocol, or simply the DS protocol. One way to implement this protocol is as follows. When an instance of a subtask completes, the scheduler of the subtask sends a synchronization signal to the scheduler on the processor where the immediate successor of the subtask executes. Upon receipt of this synchronization signal, the scheduler on the later processor releases one instance of the successor subtask and puts it in the ready queue. As mentioned earlier, each instance of the first subtask in a task is released upon a timer interrupt.

This protocol is obviously easy to implement. Each subtask instance incurs the cost of one interrupt. Due to the nature of priority-driven scheduling, one subtask instance preempts at most one other subtask instance with a lower priority. Therefore, each subtask instance involves an overhead of at most two context switches.

Despite the simple implementation and low overhead, there are two problems associated with the DS protocol, making it a poor choice for a hard real-time system. As we have seen in the example in the previous chapter, the DS protocol may yield longer worst-case EER times and cause tasks to miss their deadlines. As a matter of fact, we will show in the next chapter that the upper bound on the EER time of a task yielded by the DS protocol is always greater than or equal to the upper bound yielded by other protocols. The second problem is that the schedulability analysis of a system using the DS protocol is very complicated. In fact, we may not obtain finite upper bounds on the task EER times for some systems using the DS protocol. These shortcomings motivated us to design new protocols presented in the rest of this section.

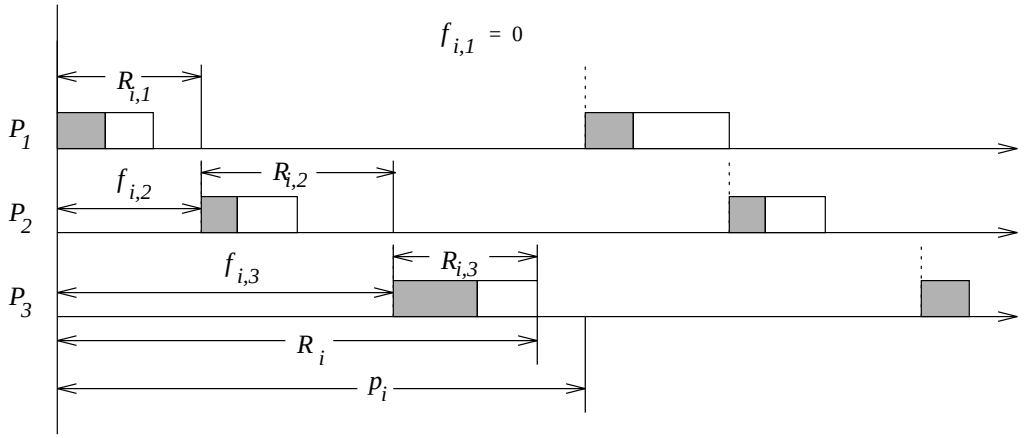



**Figure 4.1**: Illustration of the Phase Modification Protocol

### 4.2.2 Phase Modification (PM) Protocol

The Phase Modification protocol was proposed by Bettati [13] to schedule periodic flow-shop tasks. The Phase Modification protocol, or the PM protocol, requires that we first bound the worst-case response time of each subtask $T_{i,j}$. Let $R_{i,j}$ denote an upper bound on the response time of $T_{i,j}$. If the phase of $T_i$ is $f_i$, then according to the PM protocol we adjust the phase $f_{i,j}$ of each subtask $T_{i,j}$ to be $f_i + \sum_{k=1}^{j-1} R_{i,k}$, and at run-time we release each subtask periodically according to its adjusted phase and the period of its parent task. This is illustrated in Figure 4.1. Each shaded box represents the execution time of the corresponding subtask instance, and the shaded box plus the adjacent white box represents the response time of the corresponding subtask instance. We see from the absence of the synchronization signals that an instance of $T_{i,2}$ is released independent of the completion time of the corresponding instance of $T_{i,1}$. Instead, it is released according to the adjusted phase and the period of $T_i$. It is easy to verify that whenever an instance of a subtask is released, the immediate predecessor of the subtask instance has completed, (i.e., the precedence constraints are satisfied,) provided that $T_{i,1}$ is strictly periodic and the bound $R_{i,j}$ is a correct upper bound on the response time of $T_{i,j}$.

From the above description, we can see the PM protocol is tightly intertwined with the schedulability analysis algorithm. They are mutually dependent. To schedule an end-to-end system using the PM protocol, we first use the schedulability analysis algorithm to obtain upper bounds on the response times of subtasks. We then adjust the phases of subtasks and schedule them according to the adjusted phases. On the other hand, as will become clear in Chapter 5,

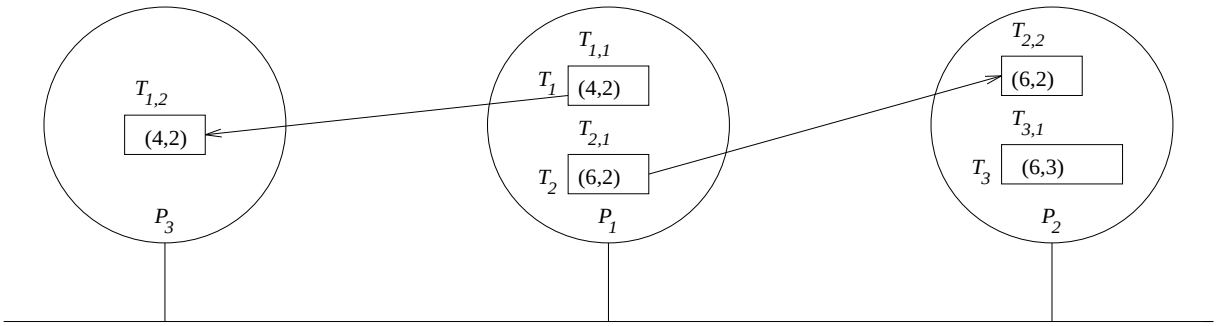



**Figure 4.2**: A Simple End-to-End System (reproduced from Figure 3.3)

when bounding the response times of subtasks we need to assume that all subtasks are periodic so that we can treat subtasks on different processors independently.

In implementation, each subtask instance is triggered by a timer interrupt, and no synchronization interrupt is necessary. The interrupt cost associated with each subtask instance is one. Similar to the DS protocol, we can verify that the number of context switches associated with each subtask instance is at most two.

For convenience, we reproduce Figure 3.3 in Figure 4.2, and use it to illustrate the PM protocol. Recall that subtask $T_{1,1}$ has a higher priority than $T_{2,1}$ on $P_1$, and $T_{2,2}$ has a higher priority than $T_{3,1}$. The phases of $T_1$ and $T_2$ are 0 and the phase of $T_3$ is 4. The relative deadlines of the tasks are equal to their periods. Using the schedulability analysis algorithm which will be presented in the next chapter, we obtain a bound on the response time of $T_{2,1}$, which is 4 time units. Hence we adjust the phase of $T_{2,2}$ to 4. We also obtain an upper bound on the response time of $T_{1,1}$, which is equal to its maximum execution time, 2. Thus $T_{1,2}$ has a phase of 2. The phases of all other subtasks are equal to the phases of their corresponding parent tasks. Figure 4.3 shows the schedule $P_1$ and $P_2$ for the first 10 time units when the PM protocol is used. According to this schedule, the first instance of $T_{3,1}$ completes at time 9, and hence the first instance of $T_3$ meets its deadline. Compared with the schedule in Figure 3.4 when the DS protocol is used, the difference in this schedule is the release time of the second instance of $T_{2,2}$. Although the second instance of $T_{2,1}$ completes at time 8, the second instance of $T_{2,2}$ is not released until time 10, because the adjusted phase of $T_{2,2}$ is 4 and the period of $T_{2,2}$ is 6. Thus we ensure that the first instance of $T_{3,1}$ is preempted only once by $T_{2,2}$. In fact, every instance of $T_{3,1}$ will be preempted at most once by an instance of $T_{2,2}$, and thus $T_3$ becomes schedulable.

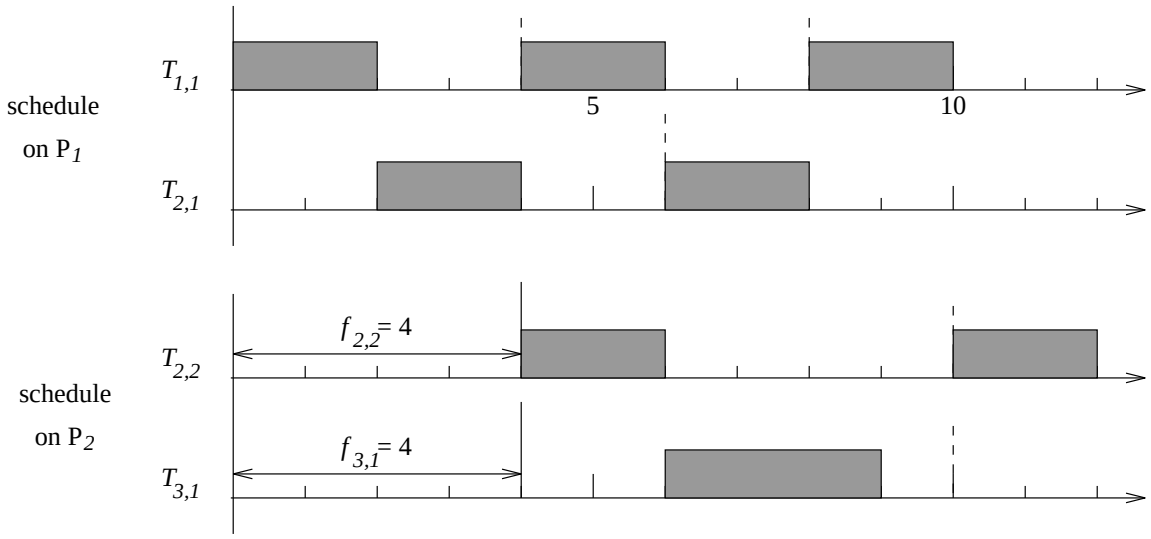



**Figure 4.3**: The Schedule According to the PM Protocol

As we will show in the next chapter, the bounds yielded by the PM protocol are always smaller than or equal to the bounds yielded by the DS protocol.

The PM protocol has its own shortcomings, however. First, the release time of each subtask instance is critical. If an instance is released a little earlier, its immediate predecessor may not have completed yet, and the precedence constraint will be violated. On the other hand, if it is released a little later, this instance may not complete before its immediate successor is released. As a consequence, no clock skewing is allowed in the system; either the schedulers on all processors use a centralized clock, or the clocks on all processors are strictly synchronized. In practice, either case can be difficult to achieve in a distributed computing environment.

Second, the PM protocol will not work if the inter-release time of the first subtask can be greater than the period. For example, in Figure 4.1, if the inter-release time between the first and the second instance of $T_{i,1}$ is greater than $p_i$ while the second instance of $T_{i,2}$ is still released "on time" according to its phase and period, the precedence constraint between the second instance of $T_{i,1}$ and the second instance of $T_{i,2}$ can be violated. This problem is rather limiting because, according the periodic task model, the period of a task means the minimum inter-release time of two consecutive instances. In the context of end-to-end scheduling, the period of an end-to-end task is the minimum inter-release time of two consecutive instances of its first subtask. This

shortcoming of the PM protocol limits its application to a smaller set of end-to-end systems where the first subtasks of all tasks are strictly periodic.

The third problem arises if a subtask occasionally overruns. In this case, the precedence constraint between the subtask and its immediate successor can be violated, due to the same reason mentioned above.

All these problems root from the same fact, i.e., the lack of explicit synchronization between the completion of a subtask instance and the release of its immediate successor in the PM protocol. In many situations it is not sufficient to rely solely on timing bounds and timer interrupts. This observation leads us to a modified version of the PM protocol, called the *Modified Phase Modification* protocol, or the *MPM* protocol, which overcomes these problems at a slightly higher cost.

### 4.2.3 Modified Phase Modification (MPM) Protocol

According to the MPM protocol, when a subtask instance completes, the scheduler checks if the response time of the instance is less than the upper bound on its response time. Suppose that the response time of subtask $T_{i,j}$ is upper-bounded by $R_{i,j}$. According to the MPM protocol, if the completion time of an instance of $T_{i,j}$ released at time $t$ is earlier than $t + R_{i,j}$, a timer interrupt is set to $t + R_{i,j}$. When the timer interrupt is due, a synchronization signal is sent to the processor where subtask $T_{i,j+1}$ executes. On the other hand, if the completion time of this instance is greater than or equal to $t + R_{i,j}$, a synchronization signal is sent immediately to the processor where $T_{i,j+1}$ executes. In either case, upon receipt of the signal, the scheduler of $T_{i,j+1}$ releases an instance of $T_{i,j+1}$ immediately and puts it into the ready queue.

Figure 4.4 illustrates the MPM protocol when it is applied to task $T_i$ in Figure 4.1. A dotted arrow represents the sending of a synchronization signal, and the line with double arrows represents the duration from the completion of a subtask instance until the sending of a corresponding synchronization signal. Although the schedule in Figure 4.4 is identical to the one in Figure 4.1, the underlying mechanism to achieve this schedule is totally different. For example, in Figure 4.1, the release time of the first instance of $T_{i,2}$ is solely dependent on the phase of $T_{i,2}$, while in Figure 4.4 it depends on when the synchronization signal from processor $P_1$ is received.

The following lemma states the relation between the PM and MPM protocols.

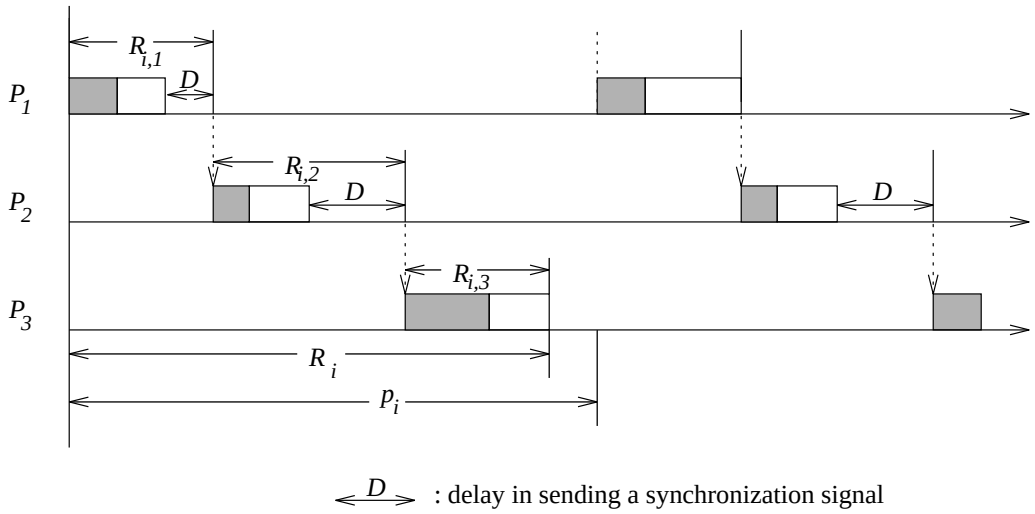



**Figure 4.4**: Illustration of the Modified Phase Modification Protocol

**Lemma 1** *Under the following conditions, a system using the MPM protocol has exactly the same schedule as the system using the PM protocol: (1) schedulers on different processors use a centralized clock or the clocks on different processors are strictly synchronized; (2) the response time of every instance of subtask $T_{i,j}$ does not exceed the upper bound on the response time of $T_{i,j}$; (3) the first subtasks are strictly periodic; and (4) interrupt overhead is negligible.*

To show that Lemma 1 is correct, let us examine the release times of subtask instances. If we use the PM protocol, the $g$th instance of a subtask $T_{i,j}$ is released at time $f_i + \sum_{k=1}^{j-1} R_{i,k} + (g-1) \times p_i$ under the conditions stated in the lemma. If we use the MPM protocol, the $g$th instance of $T_{i,j}$ $(j > 1)$ is always $R_{i,j-1}$ time units after the release of the $g$th instance of $T_{i,j-1}$, provided that the response time of $T_{i,j-1}$ is never greater than $R_{i,j-1}$. Given that the first subtask is strictly periodic, we have that the $g$th instance of $T_{i,1}$ is released at $f_i + (g-1) \times p_i$. Hence the $g$th instance of $T_{i,j}$ is released at time $f_i + \sum_{k=1}^{j-1} R_{i,k} + (g-1) \times p_i$, the same time as it is released when the PM protocol is used. As a result, we have identical schedules for both the PM protocol and the MPM protocols.

When one or more above conditions do not hold, the PM and MPM protocols produce different schedules. If a system is scheduled according to the MPM protocol, we observe the following facts when one of the conditions does not hold.

**Clocks on different processors are not synchronized :** The release time of a subtask which is not the first subtask in the chain depends solely on when the synchronization signal is

received and does not depend on the local clock. It does not matter if the clocks are synchronized or not.

**The response time of a subtask instance exceeds the upper bound :** This situation can happen when subtasks occasionally overrun, the system experiences a transient overload, or the schedulability analysis is incorrect. Although this may cause some schedulability problem, the precedence order between the overrun subtask and its successor is preserved gracefully.

**The first subtask of each task is not strictly periodic :** Again, the precedence order among subtasks is preserved. Furthermore, we can verify that the minimum inter-arrival time of a subtask $T_{i,j}$ for $j > 1$ is greater than or equal to $p_i$ as well.

**Interrupts have overhead :** In most cases, there are two interrupts associated with each subtask instance. One is the timer interrupt for completing before the worst-case completion time, and the other is the synchronization interrupt. By contrast, only one interrupt cost is associated with each subtask instance according to the PM protocol. If the overhead of interrupt cannot be ignored, the additional cost of using the MPM protocol instead of the PM protocol will eventually lead to longer EER times of tasks.

Even though the MPM protocol overcomes some problems of the PM protocol, two aspects of these protocols can still be problematic. First, the PM and MPM protocols depend on a schedulability analysis algorithm to compute the bounds on the response times of subtasks in order to schedule the subtasks properly. Consequently, whenever the work load changes, e.g., when a task is added to the system, the upper bounds on the subtask response times may change, and accordingly the schedulers must adjust the phases of the subtasks in the case of the PM protocol or the intervals for setting timer interrupts in the case of the MPM protocol. In other words, the PM and MPM protocols are not easily adaptable to workload changes. Secondly, according to the PM and MPM protocols, the EER time of task $T_i$ is at least $\sum_{l=1}^{n_i-1} R_{i,l}$, which is close to the worst-case EER time, i.e., $\sum_{l=1}^{n_i} R_{i,l}$. Therefore, we expect that the average EER time of a task is very close to the worst-case EER time as well. If shorter average EER times are desired, such as in many interactive applications, the performance of the PM and MPM protocols may be unacceptable in terms of average EER times.

### 4.2.4 Release Guard (RG) Protocol

The motivation behind the Release Guard protocol, or the RG protocol, is to yield the same upper bounds on task EER times as the PM and MPM protocols but overcomes the two shortcomings of these protocols. The idea in the RG protocol is to release instances of each subtask such that the interval between any two consecutive releases is no shorter than the period of the subtask. By doing so, we can treat each subtask as a periodic subtask in schedulability analysis and obtain the same tight upper bounds on the response times of subtasks and eventually the same upper bounds on task EER times as the PM and MPM protocols. To yield shorter average EER times of tasks, the RG protocol allows the scheduler to release a subtask instance at the earliest possible time while still ensures that the previous constraint on the inter-release times is satisfied.

To describe the RG protocol, we introduce a variable $g_{i,j}$ called the *release guard*, for every subtask $T_{i,j}$. At time $t$, the release guard of a subtask specifies the earliest possible time instant when the next instance of the subtask can be released. We say that a time instant $t$ is a *processor idle point*, or simply an *idle point*, if all subtask instances that are released before $t$ have completed by $t$. The scheduler of $T_{i,j}$ maintains its release guard $g_{i,j}$ and updates the release guard according to the following rules.

1. $g_{i,j}$ is initially equal to 0.
2. When an instance of subtask $T_{i,j}$ is released, update the $g_{i,j}$ to the current time plus the period $p_i$ of $T_{i,j}$.
3. Update $g_{i,j}$ to the current time if the current time is a *processor idle point* on the processor where $T_{i,j}$ executes.

The rule to release an instance of a subtask which is not the first subtask becomes very simple:

> Release an instance of $T_{i,j}$ at the time equal to $g_{i,j}$ or when the immediate predecessor of the instance completes, whichever is later.

By Rule (2) alone, the inter-release time of any two consecutive instances of a subtask can never be smaller than its period. Rule (3) allows us to reset the release guards to a possibly earlier time instant. The only system parameters that the RG protocol needs to know are task periods, which makes the RG protocol easily adaptable to workload changes.

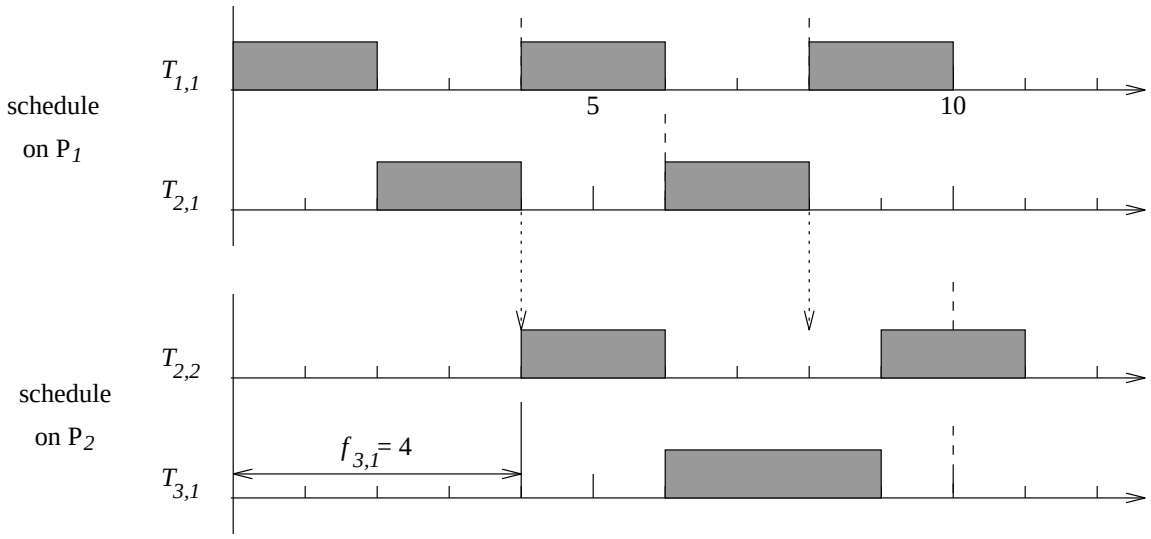



**Figure 4.5**: The Schedule of the System in Figure 4.2 According to the RG Protocol

Let us apply the RG protocol to the example in Figure 4.2, and the schedule on processor $P_1$ and $P_2$ is shown in Figure 4.5. We notice that at time 8 in the schedule the second instance of $T_{2,1}$ completes but the second instance of $T_{2,2}$ is not released on $P_2$. This is because at time 8, the release guard $g_{2,2}$ is 10, which is the release time of the first instance of $T_{2,2}$ plus the period of $T_{2,2}$. The release guard $g_{2,2}$ being equal to 10 at time 8 means that the second instance of $T_{2,2}$ should not be released until time 10. As a result, the first instance of $T_{3,1}$ gets a chance to execute to completion at time 9, and the first instance of $T_3$ thus meets its deadline. At time 9 when the first instance of $T_{3,1}$ completes, the processor $P_2$ becomes idle, i.e., time 9 is an idle point of processor $P_2$. By Rule (3), the release guard of $T_{2,2}$ is updated to 9, and accordingly the second instance of $T_{2,2}$ is released and ready to execute. In general, the RG protocol yields better schedulability for every system than the DS protocol. In addition, the RG protocol leads to shorter average EER times than the PM and MPM protocols due to its aggressiveness in releasing subtask instances. We will return to prove these statements.

According to the RG protocol, each subtask instance is associated with two interrupts in the worst case: one is the synchronization interrupt for signaling the completion of its immediate predecessor, and the other is the timer interrupt when the release guard expires. At most two context switches are associated with each subtask instance.

### 4.2.5 Sporadic Server (SS) Protocol

The sporadic server was initially proposed by Sprunt et al. [52] to handle sporadic requests in a periodic environment. From the schedulability analysis point of view, a sporadic server is just another periodic task with its own period and execution time. At run-time, however, the execution of the sporadic server may exhibit no periodicity. Instead, its execution depends on the arrivals of the sporadic requests and a set of rules regarding its eligibility for execution at any time instant. The original sporadic server algorithm proposed in [52] is not correct. In Appendix A, we give an example to illustrate the mistake in the original algorithm. A closer look at this mistake reveals that the idea of the sporadic server encompasses a family of algorithms which are differentiated by their complexity and aggressiveness in competing for the processor. A detailed description of this family of sporadic server algorithms is included in Appendix A.

The sporadic server algorithms can be used for the end-to-end scheduling of periodic tasks. The resultant protocol is called the *Sporadic Server* protocol or the SS protocol. According to the SS protocol, we use a sporadic server to execute each subtask $T_{i,j}$. The period of the server for subtask $T_{i,j}$ is equal to the period of $T_i$, and the server execution time is equal to $\tau_{i,j}$ (or $\tau_{i,j}^+$). Unlike the PM, MPM or RG protocols, the SS protocol does not control the releases of subtasks. Instead, it controls the execution of a subtask, namely, it determines when to execute a subtask instance and for how long. Whenever a subtask instance completes, its immediate successor is released right away. The instance executes whenever its own sporadic server is scheduled.

Because the sporadic servers can be treated as periodic tasks in schedulability analysis, we will obtain the same upper bounds on the response times of subtasks and the same upper bounds on task EER times as the PM, MPM and RG protocols. In addition, the SS protocol will yield shorter average EER times than the RG protocol for more "urgent" tasks[2] because the execution time of a subtask instance is often smaller than the maximum execution time, and the sporadic server can reclaim some of the "unused quota" to serve the next instance of the same subtask.

According to the description given in Appendix A, the sporadic server algorithms are fairly complicated. For the purpose of implementing the SS protocol, it is possible to simplify the

[2]Since a task has many subtasks and each subtask has its own priority, we cannot say one task has higher priority than another anymore. However, according to the priority assignment methods proposed in this thesis, subtasks with shorter periods are generally assigned higher priorities. Loosely speaking, more "urgent" tasks are tasks with shorter periods.

algorithms because each server only serves one subtask. (For example, we do not need a queue to hold the sporadic requests.) In fact, we may totally eliminate the sporadic servers in implementation. For each subtask, its scheduler only needs to maintain a budget account and update the account according to the budget replenishment rules used in the sporadic server algorithms. An subtask instance is eligible to execute only if its budget account is non-zero. Nevertheless, the simplified implementation is still significantly more complex than the previous protocols.

The execution of a subtask instance can be suspended before the instance completes. This situation happens when its server has no more execution budget. The execution is resumed when the budget is replenished (through a timer interrupt). Theoretically, the number of suspensions and resumptions can be arbitrarily large for a subtask instance. This implies that the number of context switches and interrupts associated with a subtask instance can be unbounded as well. This shortcoming, plus its complexity, make the SS protocol less appealing when it is compared with the previous protocols, especially the RG protocol.

## 4.3 Complexity and Run-Time Overhead

This section summarizes the various issues related to the implementation and run-time overhead of the synchronization protocols discussed in the previous section. Table 4.1 lists the comparison of the five protocols with respect to the algorithm complexity, number of context switches and number of interrupts associated with each subtask instance.

| Protocol | Complexity | # of context switches | # of interrupts |
|---|---|---|---|
| Direct Synchronization | simple | 2 | 1 |
| Phase Modification | intermediate | 2 | 1 |
| Modified Phase Modification | intermediate | 2 | 2 |
| Release Guard | intermediate | 2 | 2 |
| Sporadic Server | complex | unbounded | unbounded |

**Table 4.1**: Comparison of Protocols with Respect to Complexity and Overhead

As for complexity, the DS protocol requires the least extra support (synchronization interrupt) from the operating system and is clearly the simplest. The PM protocol needs only timer interrupt support, but it requires synchronized clocks on different processors. The MPM and RG protocols

require both synchronization interrupt and timer interrupt support. Furthermore, the PM and MPM protocols rely on the results of schedulability analysis, which makes them not suitable for systems where tasks are frequently deleted and added. For these reasons, the PM, MPM and RG protocols are classified as "intermediate" in terms of their complexity. The SS protocol is the most complicated due to the complexity of the Sporadic Server algorithms.

As for the number of context switches associated with each subtask instance, it is bounded by 2 for the DS, PM, MPM and RG protocols. The number follows directly from the following rules which "charge" the context switch cost to appropriate subtask instances. A context switch can occur in two cases: (1) a subtask instance is released and it has higher priority than the currently executing one; or (2) the currently executing subtask instance completes. In the first case, we charge the context switch cost to the preempting subtask instance, and in the second case we charge the cost to the completed subtask instance. By doing so, we find that every context switch cost is charged to some subtask instance, and obviously no subtask instance is charged more than twice. If the context switch overhead is not negligible in practice, we simply add the overhead of two context switches to the maximum execution time of each subtask, and we can then perform the schedulability analysis as if the context switch overhead is zero.

In the case of the SS protocol, an instance can be switched out due to the server running out of time budget. Theoretically, there can be an unbounded number of context switches for any single subtask instance. Consequently, in schedulability analysis we will have difficulty in computing the context switch overhead if the overhead is not negligible.

As for the number of interrupts associated with each subtask instance, it is equal to one for the DS protocol because the release of each subtask instance is triggered by a synchronization interrupt (except the release of the first subtask, which is triggered by a timer interrupt or an external event). According to the PM protocol, the release of each subtask is controlled by the timer, and each instance requires one timer interrupt. For the MPM and RG protocols, each subtask instance is associated with no more than two interrupts, one for the timer interrupt and one for the synchronization interrupt. For the similar reason discussed in the previous paragraph, each subtask instance in SS protocol can have an unbounded number of interrupts.

## 4.4 Execution Control

We say a subtask instance is available for release when its immediate predecessor has completed, and it is ready to execute after it is released. A synchronization protocol is said to have *execution control* if a subtask instance can be intentionally withheld from release when it is available for release or it can be withheld from execution after it is released. In other words, the protocol is not greedy. On the other hand, a greedy synchronization protocol does not have execution control: it never allows the delay in releasing or executing a subtask instance.

Clearly, only the DS protocol is greedy and does not have execution control. The PM protocol releases subtask instances depending on the clock rather than the completion of the immediate predecessors of subtask instances. Delay in releasing a subtask instance happens when its immediate predecessor completes earlier than its release time. According to the MPM or RG protocol, we need to check conditions other than the completion of its immediate predecessor to release a subtask instance, and an intentional delay in releasing a subtask instance can happen. According to the SS protocol, the release of a subtask instance is never delayed. However, the execution of a subtask instance may be suspended before it completes. Consequently, the SS protocol is a synchronization protocol with execution control. Figure 4.6 shows the classification. In Chapter 5, we will see that similar schedulability analysis methods can be used for the protocols with execution control.

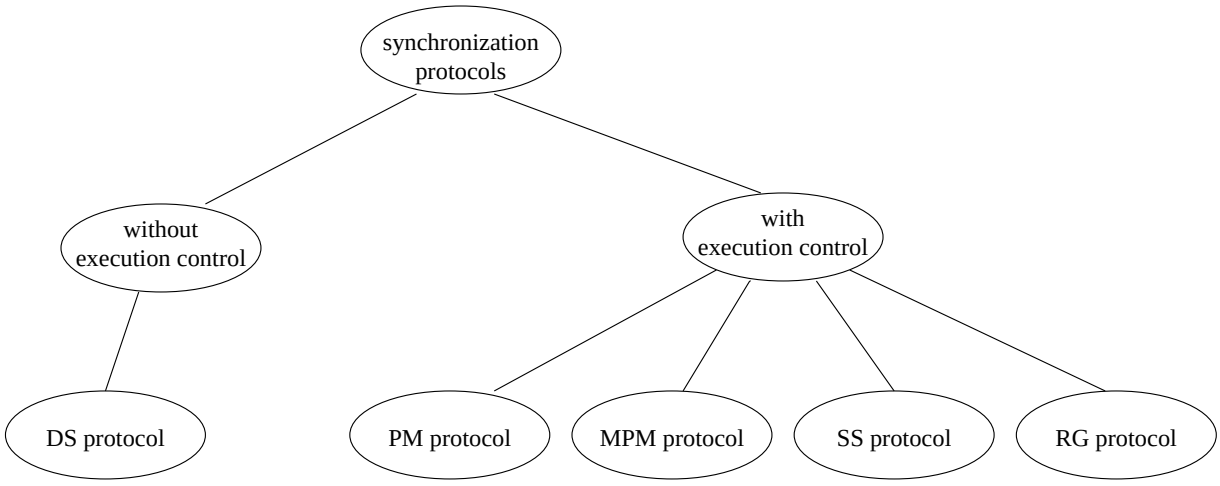



**Figure 4.6**: Classification of Scheduling Protocols

# Chapter 5

# Schedulability Analysis

In this chapter, we focus on schedulability analysis algorithms for the synchronization protocols proposed in Chapter 4. In most cases, to verify the schedulability of each task and hence the schedulability of a system, we bound the end-to-end response (EER) time of each task and compare the bound with the end-to-end relative deadline of the task.

We first present the schedulability analysis algorithms for the Phase Modification (PM) and Modified Phase Modification (MPM) protocols. These protocols are the easiest to analyze because according to these protocols subtasks fit in the periodic task model. We then present the algorithms for the Release Guard (RG) and Sporadic Server (SS) protocols. These algorithms are largely the same as the algorithms for the PM and MPM protocols. The similarity between these schedulability analysis algorithms is not a coincidence. As protocols with execution control, the PM, MPM, SS and RG protocols are designed to ensure that the behavior of each subtask is no worse than a periodic subtask, and one schedulability analysis algorithm for periodic systems can be applied to all of them. On the other hand, the Direct Synchronization (DS) protocol does not have execution control and is much more difficult to analyze. In the second half of this chapter, we present an algorithm that bounds the EER times of tasks in a system using the DS protocol.

All these algorithms are based on a technique called *busy period analysis*, which was first proposed by Lehoczky [5, 6] and later extended by Audsley [7], Tindell [8, 19, 53], and Burns [17]. Before we present the schedulability analysis algorithms, we introduce the notation used in the busy period analysis and describe this technique under the context of the end-to-end scheduling.

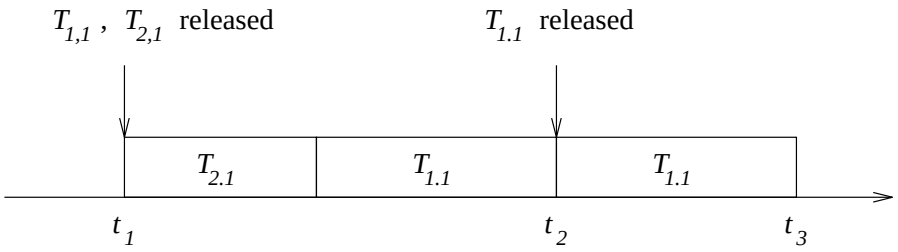



**Figure 5.1**: Two $\phi_{1,1}$-Level Busy Periods

## 5.1 Busy Period Analysis

The busy period analysis method is also called *time demand analysis*. In this thesis, we will use these two terms interchangeably. We now introduce the concept of a *$\phi$-level idle point* and a *$\phi$-level busy period* in an arbitrary schedule on a processor.

**Definition 1** *In the schedule of a processor $P$, a time instant $t$ is a $\phi$-level idle point if and only if every subtask instance that is released on $P$ before time $t$ and has priority higher than or equal to $\phi$ has completed by time $t$.*

**Definition 2** *A $\phi$-level busy period is a time interval of non-zero length between two consecutive $\phi$-level idle points in a schedule.*

Intuitively, a $\phi$-level busy period is simply a continuous time interval during which only subtasks with priorities higher than or equal to $\phi$ execute. Although intuitive, this description of a busy period can be misleading. For example, in Figure 5.1, subtask $T_{2,1}$ has higher priority than $T_{1,1}$. In interval $(t_1, t_3)$ subtasks with priorities higher than or equal to $\phi_{1,1}$ execute continuously. However, $(t_1, t_3)$ is not a single $\phi_{1,1}$-level busy period. Instead, according to Definition (2), it consists of two $\phi_{1,1}$-level busy periods, $(t_1, t_2)$ and $(t_2, t_3)$, because $t_2$ is a $\phi_{1,1}$-level idle point.

By definition, a $\phi_{i,j}$-level busy period can exist in the schedule of any processor, where $\phi_{i,j}$ is the priority of subtask $T_{i,j}$. By convention, however, when we say a $\phi_{i,j}$-level busy period we always mean a $\phi_{i,j}$-level busy period in the schedule of the processor where $T_{i,j}$ executes.

Obviously, the execution of any instance $T_{i,j}(m)$ of $T_{i,j}$ must be contained in a $\phi_{i,j}$-level busy period. Let $T_{i,j}(m)$ denote the $m$th instance of $T_{i,j}$ in an arbitrary schedule. Figure 5.2 shows that release and completion of $T_{i,j}(m)$. Without loss of generality, we assume that the $\phi_{i,j}$-level


**Figure 5.2**: A $\phi_{i,j}$-Level Busy Period

busy period within which $T_{i,j}(m)$ executes starts from time zero. In the figure, $r_{i,j}(m)$ is the release time of $T_{i,j}(m)$, and $C_{i,j}(m)$ is the completion time of $T_{i,j}(m)$.

To compute the completion time $C_{i,j}(m)$, we need to analyze *time demand* generated by many other subtask instances, especially those which must be scheduled ahead of $T_{i,j}(m)$. We say a piece of time demand is generated at time $t$ on a processor if an instance of a subtask is released (and ready for execution) at time $t$ on the processor. The amount of the time demand is equal to the execution time of the instance, and the priority level of the time demand is equal to the priority of the subtask. The *time demand function with respect to* $T_{i,j}(m)$, denoted by $W(t)$, is the total amount of time demand generated in interval $[0, t)$ $(t \leq C_{i,j}(m))$ by $T_{i,j}(m)$ and subtask instances that are released before t and must be scheduled before $T_{i,j}(m)$. Time demand function $W(t)$ has two attributes:[1] (1) at any time $t$ after time zero and before the completion time $C_{i,j}(m)$, the total time demand $W(t)$ is strictly larger than the time supply $t$; and (2) at the completion time $C_{i,j}(m)$, the time demand $W(C_{i,j}(m))$ is equal to the time supply $C_{i,j}(m)$. Put into mathematic equations, these two attributes can be expressed as follows.

$$W(t) > t, \quad \text{for } 0 < t < C_{i,j}(m)$$
$$W(C_{i,j}(m)) = C_{i,j}(m)$$

Combining these two, we can express $C_{i,j}(m)$ as

$$C_{i,j}(m) = \min\{t > 0 \mid t = W(t)\} \tag{5.1}$$

which implies that the completion time $C_{i,j}(m)$ is the earliest time instance when the time demand is met by the time supply.

[1]Please note that these two attributes only hold when the $\phi_{i,j}$-level busy period starts from time 0. It may not hold otherwise.

If we know the time demand function $W(t)$, we can use an iterative process to obtain a solution for $C_{i,j}(m)$. We let $S_0 = W(0^+)$, where $0^+$ stands for a time instant immediate after time 0, and $S_k = W(S_{k-1})$ for $k = 1, 2, \ldots$. If $T_{i,j}(m)$ completes, $C_{i,j}(m)$ is finite, and the series $S_k$ converges to $C_{i,j}(m)$. The response time of $T_{i,j}(m)$ can thus be computed.

In general, we may not know $r_{i,j}$ and $W(t)$ exactly. However, we may know an upper bound on the time demand that can be generated by $T_{i,j}$ and other subtask instances which can delay the completion of $T_{i,j}(m)$. In other words, we may know a function $W'(t)$ which is such that $W'(t) \geq W(t)$ for $0 < t < C_{i,j}(m)$. Let $C'_{i,j}(m) = \min\{t > 0 | t = W'(t)\}$. It is easy to verify that $C'_{i,j}(m) \geq C_{i,j}(m)$, i.e., $C'_{i,j}(m)$ is an upper bound on the completion time of $T_{i,j}(m)$. If we also know a lower bound on $r_{i,j}(m)$, we can compute an upper bound on the response time of $T_{i,j}(m)$. In the rest of this chapter, we will always work with the upper-bound $W'(t)$ and $C'_{i,j}(m)$ in a busy period analysis. For the sake of convenience, we will say a time demand function $W(t)$ while we mean its upper bound function $W'(t)$, and we may say the completion time $C_{i,j}(m)$ while we mean an upper bound $C'_{i,j}(m)$ on the completion time obtained from $W'(t)$.

The above method can be easily modified to compute the duration of a $\phi_{i,j}$-level busy period as well. We define the *time demand function with respect to a $\phi_{i,j}$-level busy period*, denoted by $W(t)$ as well, to be the total amount of the time demand with priority level higher than or equal to $\phi_{i,j}$ generated in interval $[0, t)$ in a $\phi_{i,j}$-level busy period starting from time zero. The duration of this busy period is

$$D_{i,j} = \min\{t > 0 | t = W(t)\}$$

If $W(t)$ is an upper bound of the total amount of time demand rather than the actual amount of time demand, the solution for $D_{i,j}$ is an upper bound on the duration of the busy period.

## 5.2 Schedulability Analysis for the PM and MPM Protocols

For the purpose of schedulability analysis, we assume that clocks on different processors are synchronized, subtasks do not over-run, and the overhead for interrupts and context switches are negligible. The behavior of the PM protocol is thus identical to that of the MPM protocol. The algorithms presented here are applicable to both protocols.

From Figure 4.1, a natural bound on the EER time of a task is the sum of the bounds on the response times of individual subtasks on the chain. Therefore we will first concentrate on bounding the response times of individual subtasks.

### 5.2.1 Algorithm SA/PM

According to the PM and MPM protocols, subtasks fit in the periodic task model. In the case of the PM protocol, subtasks are strictly periodic. In the case of the MPM protocol, the inter-release time between any two consecutive instances of a subtask is no shorter than the period. Thus, according to both the PM and MPM protocols, we have a set of periodic subtasks on each processor.

In this thesis, we assume that the response time of a subtask may be longer than its period, unless stated otherwise. Given a set of periodic subtasks on each processor, the busy period analysis [6] can be readily applied to obtain an upper bound on the response time of each subtask. In [6], Lehoczky presented an analysis algorithm as a sufficient condition. Below we describe his algorithm in consistency with the busy period analysis method presented in the previous section.

In the following discussion, we focus on $T_{i,j}$, the subtask whose response time is to be bounded. We follow the following four steps to obtain a bound on the response time of $T_{i,j}$ in an end-to-end system.

1. Bound the duration of an arbitrary $\phi_{i,j}$-level busy period.

2. Bound the number of instances of $T_{i,j}$ in a $\phi_{i,j}$-level busy period.

3. For each possible instance of $T_{i,j}$ in a $\phi_{i,j}$-level busy period, we obtain an upper bound on its response time.

4. We take maximum of the bounds on the response times obtained in Step 3 as the bound on the response time of $T_{i,j}$.

Let $H_{i,j}$ denote the set of subtasks, excluding $T_{i,j}$, that (1) are on the same processor as $T_{i,j}$ and (2) have priorities higher than or equal to $T_{i,j}$. Obviously, only the instances of subtasks in $H_{i,j} \cup \{T_{i,j}\}$ can execute in a $\phi_{i,j}$-level busy period. The time demand function $W(t)$ with respect to a $\phi_{i,j}$-level busy period is the total amount of time demand generated by instances of

subtasks in $H_{i,j} \cup \{T_{i,j}\}$ during the interval $[0, t)$. Given the fact that all subtasks are periodic, the time demand function $W(t)$ can be bounded from above by

$$\sum_{T_{k,l} \in H_{i,j} \cup \{T_{i,j}\}} \left\lceil \frac{t}{p_k} \right\rceil \tau_{k,l}$$

According to the busy period analysis presented in the previous section, an upper bound $D_{i,j}$ on the duration of a $\phi_{i,j}$-level busy period can be computed by the following equation :

$$D_{i,j} = \min \left\{ t > 0 \mid t = \sum_{T_{k,l} \in H_{i,j} \cup \{T_{i,j}\}} \left\lceil \frac{t}{p_k} \right\rceil \tau_{k,l} \right\} \tag{5.2}$$

The iterative process described before can be applied to obtain the value of $D_{i,j}$. A transformation shows that $D_{i,j}$ has a finite value if $\sum_{T_{k,l} \in H_{i,j} \cup \{T_{i,j}\}} \tau_{k,l}/p_k$ is no greater than 1.

Given the upper bound $D_{i,j}$ on the duration of any $\phi_{i,j}$-level busy period, we can determine an upper bound $M_{i,j}$ on the number of instances of $T_{i,j}$ in a $\phi_{i,j}$-level busy period. This is due to the fact that $T_{i,j}$ is periodic.

$$M_{i,j} = \left\lceil \frac{D_{i,j}}{p_i} \right\rceil \tag{5.3}$$

The time demand function with respect to the $m$th instance $T_{i,j}(m)$ $(1 \leq m \leq M_{i,j})$ in a $\phi_{i,j}$-level busy period is bounded from above by

$$m\tau_{i,j} + \sum_{T_{k,l} \in H_{i,j}} \left\lceil \frac{t}{p_k} \right\rceil \tau_{k,l}$$

Thus, according to the time demand analysis, an upper bound $C_{i,j}(m)$ on the completion time of $T_{i,j}(m)$ can be computed by

$$C_{i,j}(m) = \min \left\{ t > 0 \mid x = m\tau_{i,j} + \sum_{T_{k,l} \in H_{i,j}} \left\lceil \frac{t}{p_k} \right\rceil \tau_{k,l} \right\} \tag{5.4}$$

$C_{i,j}(m)$ has a solution if $\sum_{T_{k,l} \in H_{i,j}} \tau_{k,l}/p_k$ is less than 1, which is the same condition for the existence of an upper bound on a $\phi_{i,j}$-level busy period. The iterative process described in the previous section can be applied to obtain the solution.

A lower bound of the release time of $T_{i,j}(m)$ is $(m-1)p_i$. Hence an upper bound $R_{i,j}(m)$ on the response time of $T_{i,j}(m)$ is given by

$$R_{i,j}(m) = C_{i,j}(m) - (m-1)p_i \tag{5.5}$$

Again, every instance of $T_{i,j}$ must be released and completed in a $\phi_{i,j}$-level busy period. Its response time is upper-bounded by $R_{i,j}(m)$ if it is the $m$th instance of $T_{i,j}$ released in that busy

| $T_{i,j}$ | host | $p_i$ | $\tau_{i,j}$ | $\phi_{i,j}$ | $R_{i,j}$ |
|---|---|---|---|---|---|
| $T_{1,1}$ | $P_1$ | 70 | 26 | 70 | 26 |
| $T_{2,2}$ | $P_1$ | 100 | 62 | 100 | 118 |
| $T_{2,1}$ | $P_2$ | 100 | 50 | 100 | 50 |

**Table 5.1**: Subtask Parameters of an End-to-End System

period. Hence the maximum of $R_{i,j}(m)$'s ($1 \leq m \leq M_{i,j}$) must be a correct upper bound on the response time of any instance of $T_{i,j}$ and, therefore, an upper bound on the response time of $T_{i,j}$.

Once we obtain upper bounds on the response times of subtasks, we can sum up the bounds on the response times of all its subtasks to obtain an upper bound $R_i$ on the end-to-end response time of a task. We call the algorithm based on the above steps Algorithm SA/PM, standing for the schedulability analysis algorithm for the PM protocol. Figure 5.3 lists the pseudo-code of Algorithm SA/PM.

As an example, let us look at an end-to-end system containing two tasks.[2] The subtask parameters are listed in Table 5.1, together with the bounds on the response times of subtasks computed by Algorithm SA/PM. Take subtask $T_{2,2}$ for example. By Eq.(5.2), we obtain an upper bound on the duration of a $\phi_{2,2}$-level busy period, which is 696. Hence there can be up to 7 instances of $T_{2,2}$ released and completed in a $\phi_{2,2}$-level busy period. It turns out that the 5th instance in the busy period has the largest upper bound on its response time, which is 118. The upper bound on the response time of $T_{2,2}$ is thus 118. The bounds on the EER times of $T_1$ and $T_2$ are 26 and 168, respectively.

### 5.2.2 An Improvement of Algorithm SA/PM

An end-to-end task is said to be *recurrent* if it has two or more subtasks on a same processor and these subtasks are not adjacent. Under the following two additional conditions, we are able to obtain tighter bounds on task EER times by using an improved algorithm, called Algorithm SA/IPM, rather than by using Algorithm SA/PM.

- Some tasks are recurrent.
- The relative deadline of each task is shorter than or equal to its period.

[2]This example is extracted and modified from an example in [6] by Lehoczky.

Algorithm SA/PM

**Input :** Subtask set $\{T_{i,j}\}$ and the subtask parameters.

**Output :** For each task $T_i$, the bound $R_i$ on its end-to-end response time.

**Algorithm :**

1. For each subtask $T_{i,j}$,
   (a) Compute an upper bound $D_{i,j}$ on the duration of a $\phi_{i,j}$-level busy period by
   $$D_{i,j} = \min \left\{ t > 0 \mid t = \sum_{T_{k,l} \in H_{i,j} \cup \{T_{i,j}\}} \left\lceil \frac{t}{p_k} \right\rceil \tau_{k,l} \right\}$$
   (b) Compute an upper bound $M_{i,j}$ on the number of instances of $T_{i,j}$ in a $\phi_{i,j}$-level busy period by
   $$M_{i,j} = \left\lceil \frac{D_{i,j}}{p_i} \right\rceil$$
   (c) For $m = 1, 2, \ldots, M_{i,j}$,
      i. Compute an upper bound $C_{i,j}(m)$ on the completion time of the $m$th instance of $T_{i,j}$ in the busy period by
      $$C_{i,j}(m) = \min \left\{ t > 0 \mid t = m\tau_{i,j} + \sum_{T_{k,l} \in H_{i,j}} \left\lceil \frac{t}{p_k} \right\rceil \tau_{k,l} \right\}$$
      ii. Compute an upper bound $R_{i,j}(m)$ on the response time of the $m$th instance of $T_{i,j}$ in the busy period by
      $$R_{i,j}(m) = C_{i,j}(m) - (m-1)p_i$$
   (d) Compute the bound $R_{i,j}$ on the response time of $T_{i,j}$ by
   $$R_{i,j} = \max\{R_{i,j}(m)\}, \ for \ m = 1, 2, \ldots, M_{i,j}$$
2. For each task $T_i$, compute the bound on the end-to-end response time by
   $$R_i = \sum_{j=1}^{n_i} R_{i,j}$$

**Figure 5.3**: Pseudo-Code of Algorithm SA/PM

### An Example

As an example, Figure 5.4 shows a system that satisfies these conditions. The system has two tasks. Task $T_1$ has four subtasks, and it is recurrent. Task $T_2$ has only one subtask. Their relative deadlines are equal to their periods. The subtask parameters are listed in Table 5.2. Column labeled as $\phi_{i,j}$ lists the priorities of subtasks. The smaller the number, the higher the priority.

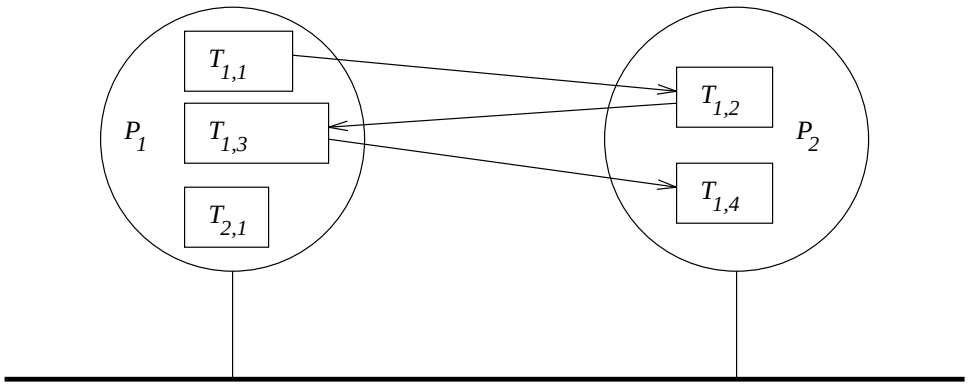



**Figure 5.4**: A Simple Recurrent End-to-End System

| $T_{i,j}$ | host | $p_{i,j}$ | $\tau_{i,j}$ | $\phi_{i,j}$ |
|---|---|---|---|---|
| $T_{1,1}$ | $P_1$ | 15 | 3 | 3 |
| $T_{1,3}$ | $P_1$ | 15 | 4 | 1 |
| $T_{2,1}$ | $P_1$ | 8 | 2 | 5 |
| $T_{1,2}$ | $P_2$ | 15 | 3 | 3 |
| $T_{1,4}$ | $P_2$ | 15 | 3 | 3 |

**Table 5.2**: Subtask Parameters for the Example in Figure 5.4

Suppose that we want to bound the EER time of $T_2$, i.e., the response time of $T_{2,1}$. By applying Algorithm SA/PM, we find that the bound on the response time of $T_{2,1}$ is 9, and hence the bound on the EER time of $T_2$ is 9. Consequently, we cannot guarantee that $T_2$ will always meet its deadline because its relative deadline is 8.

An instance of $T_{2,1}$ can have a response time of 9 time units only if it is preempted by both $T_{1,1}$ and $T_{1,3}$. However, we observe that the phase of $T_{1,1}$ is $f_1$, and the phase of $T_{1,3}$ is $f_1 + 6$. It is impossible for any instance of $T_{2,1}$ to be preempted by both $T_{1,1}$ and $T_{1,3}$. The worst-case response time $T_{2,1}$ never exceeds 6 time units, and $T_2$ is schedulable.

In the previous example, we know exactly the phases of $T_{1,1}$ and $T_{1,3}$, and it is possible for us to bound together their interference to the execution of $T_{2,1}$ and obtain a tighter bound on the response time of $T_{2,1}$. In general, however, we may not be fortunate enough to know the adjusted phases of all interfering subtasks when we want to bound the response time of $T_{i,j}$. Nevertheless, it is safe for us to assume that other tasks are schedulable when we bound the response time of $T_{i,j}$ for the following reason. If at the end of the schedulability analysis all tasks are found schedulable, any upper bound on the response time of a subtask $T_{i,j}$ based on this assumption is indeed correct. Otherwise, the upper bound on the response time of $T_{i,j}$ may not be correct. In the latter case, we will find some task not schedulable and hence the system not schedulable. In other words, as far as system schedulability is concerned, we reach the same conclusion that the system is not schedulable, no matter if the upper bound on the response time of $T_{i,j}$ is indeed correct. In subsequent discussion, we will assume that all tasks other than $T_i$ are schedulable when we bound the response time of a subtask of $T_i$.

We now describe a general method that computes a tighter upper bound on the response time of a subtask $T_{i,j}$, which is the core step in Algorithm SA/IPM. Let us take subtask $T_{2,1}$ as an example. Assuming that $T_1$ is schedulable, the phases of $T_{1,1}$ and $T_{1,3}$ must be adjusted such that $f_{1,1} < f_{1,3} < f_{1,1} + p_i$. Thus an instance of $T_{1,1}$ and an instance of $T_{1,3}$ cannot be released at the same time. Suppose an instance of $T_{2,1}$ is released at the same time as an instance of $T_{1,1}$. Let this time be $t_0$. At time 3 after $t_0$, the instance of $T_{1,1}$ completes, and the corresponding instance of $T_{1,3}$ will not be released earlier than time 6 after $t_0$ because it must wait for at least 3 time units for the corresponding instance of $T_{1,2}$ to execute and complete on $P_2$. Consequently the instance of $T_{2,1}$ released at time $t_0$ gets a chance to execute and completes 5 time units later. On the other hand, suppose that an instance of $T_{2,1}$ is released at the same time as an instance of $T_{1,3}$ at time $t_0$. At time 4 after $t_0$, the instance of $T_{1,3}$ completes, and the corresponding instance of $T_{1,4}$ may start to execute on $P_2$. If $T_1$ is schedulable, the corresponding instance of $T_{1,4}$ should complete before the next instance of $T_{1,1}$ is released on $P_1$. In other words, the next instance of $T_{1,1}$ will not be released before time $t_0 + 7$, which is the earliest possible completion time of the instance of $T_{1,4}$. Consequently the instance of $T_{2,1}$ released at time $t_0$ can execute and complete at time $t_0 + 6$. It is easy to verify that under any other situation, the response time of $T_{2,1}$ never exceeds 6 time units. As a result, we can guarantee that $T_2$ is schedulable.

**Algorithm SA/IPM**

The above example shows us that if we explore the dependency relationship among sibling subtasks, we may obtain a tighter upper bound on the response time of a subtask of another task on the same processor. This is the idea behind our first improvement of Algorithm SA/PM.

To explore the dependency relationship among sibling subtasks, we bound the total time demand that can be generated by these subtasks as a whole. For this purpose, we introduce the *interference function* $M_k^{i,j}(t)$ of a task $T_k$ with respect to a subtask $T_{i,j}$. $M_k^{i,j}(t)$ gives the maximum amount of $\phi_{i,j}$-level time demand generated by subtasks of $T_k$ during the interval $[t_0, t_0 + t)$ in a $\phi_{i,j}$-level busy period starting from time $t_0$. Given the interference function of every task $T_k$ with respect to $T_{i,j}$, we can bound the response time of $T_{i,j}$. The time demand function $W(t)$ at time $t_0 + t$ with respect to a $\phi_{i,j}$-level busy period starting from time $t_0$ can be expressed as

$$W(t) = \left\lceil \frac{t}{p_i} \right\rceil + \sum_{T_k} M_k^{i,j}(t)$$

and the time demand function $W(t)$ with respect to the $m$th instance of $T_{i,j}$ in this $\phi_{i,j}$-level busy period can be expressed as

$$W(t) = m\tau_{i,j} + \sum_{T_k} M_k^{i,j}(t)$$

We can modify Algorithm SA/PM according to these new time demand functions.

Since we focus on tasks with relative deadlines shorter than their periods, we can take a short-cut by only bounding the response time of the first instance of $T_{i,j}$ in a $\phi_{i,j}$-level busy period [1]. If the bound is smaller than the relative deadline, which is no greater than the period of $T_{i,j}$, and all other tasks are schedulable, the bound is correct. If the bound is greater than the period of $T_{i,j}$, we know the bound may not be correct. However, the correct bound is surely greater than the period of $T_{i,j}$, and we cannot guarantee that $T_i$ will be schedulable. We thus draw the correct conclusion about the schedulability of the system in both cases.

As a result, we compute an upper bound $C_{i,j}(1)$ on the completion time only for the first instance of $T_{i,j}$ in a $\phi_{i,j}$-level busy period.

$$C_{i,j}(1) = \min \left\{ t > 0 \mid t = \tau_{i,j} + \sum_{T_k} M_k^{i,j}(t) \right\} \tag{5.6}$$

Algorithm SA/IPM

**Input :** Subtask set $\{T_{i,j}\}$ and subtask parameters.

**Output :** The bound $R_i$ on the end-to-end response time of each task $T_i$.

**Algorithm :**

1. For each subtask $T_{i,j}$

   For each task $T_k$ $(k \neq i)$

   compute the interference function $M_k^{i,j}(t)$.

2. For each subtask $T_{i,j}$,

   determine the upper bound $R_{i,j}$ on the response time of the first instance of $T_{i,j}$ in a $\phi_{i,j}$-level busy period by

$$R_{i,j} = \min \left\{ t > 0 \mid t = \tau_{i,j} + \sum_{T_k} M_k^{i,j}(t) \right\}$$

3. For each task $T_i$, compute the bound on the EER time by

$$R_i = \sum_{j=1}^{n_i} R_{i,j}$$

**Figure 5.5**: Pseudo-Code of Algorithm SA/IPM

Since $T_{i,j}(1)$ is released inside the $\phi_{i,j}$-level busy period, $C_{i,j}(1)$ is also an upper bound on the response time. Eq.(5.6) forms the base of our improved schedulability analysis algorithm, Algorithm SA/IPM. Its pseudo-code is listed in Figure 5.5.

### Compute the Interference Functions

So far, we have not discussed how to compute an interference function. To see how, let us look back at the example in Figure 5.4 and try to compute the interference function $M_1^{2,1}(t)$. We identify that two subtasks, $T_{1,1}$ and $T_{1,3}$, in $T_1$ can generate $\phi_{2,1}$-level time demand on $P_1$. If an instance of $T_{1,1}$ is released at time 0, the corresponding instance of $T_{1,3}$ cannot be released before time 6, as we discussed earlier. The combined time demand from both $T_{1,1}$ and $T_{1,3}$ can be bounded from above by the staircase functions in Figure 5.6(a). On the other hand, if an

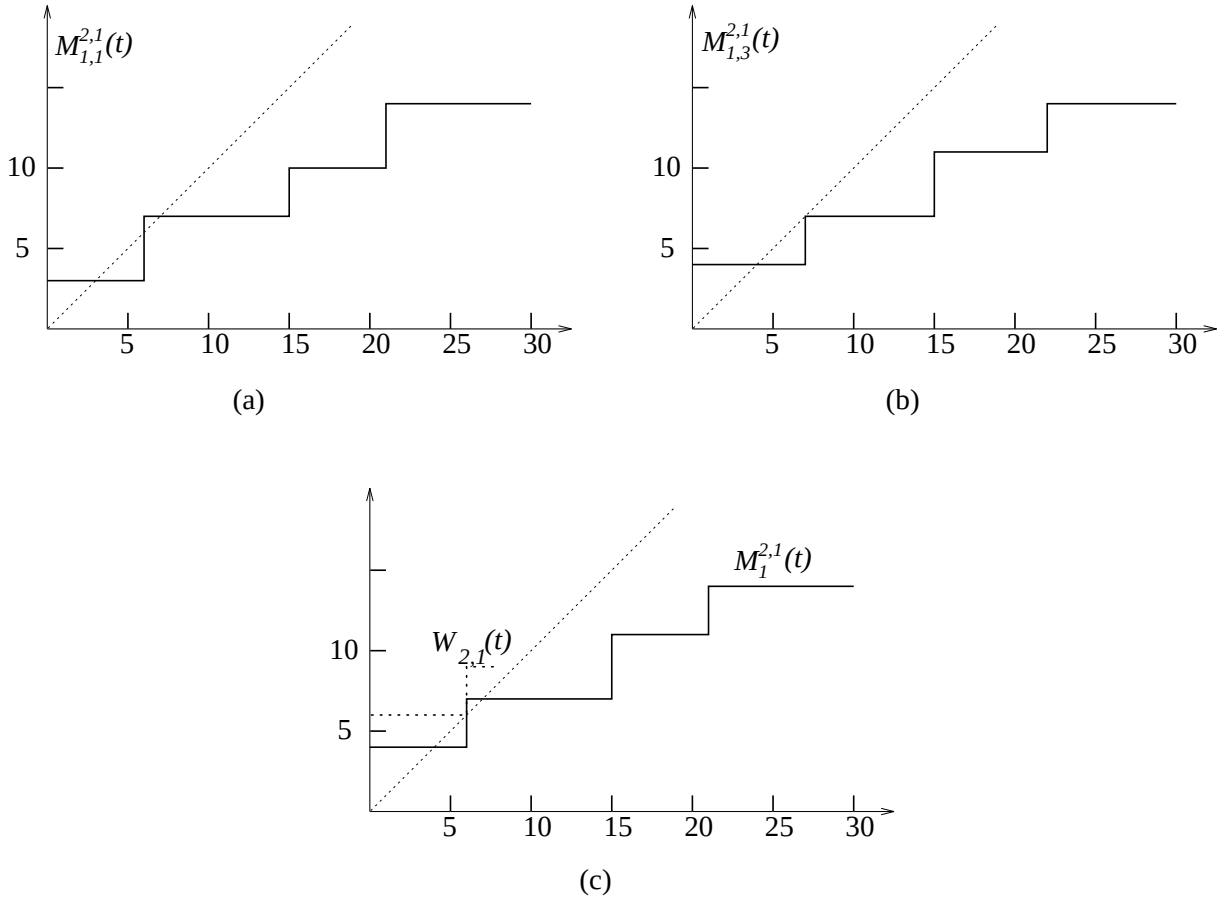



**Figure 5.6**: The Time Demand Functions of $T_1$ in the Example in Figure 5.4

instance of $T_{1,3}$ is released at time 0, the next instance of $T_{1,1}$ cannot be released till after time $R_{1,3} + R_{1,4}$ (at least 7 time units later), assuming that $T_1$ is schedulable. The combined time demand from $T_{1,1}$ and $T_{1,3}$ is no more than the staircase function given in Figure 5.6(b).

We define the *maximum function* $F(x)$ of functions $F_1(x), F_2(x), \ldots, F_n(x)$ to be the function whose value at $x$ is the maximum value of $F_1(x)$, $F_2(x)$, ..., $F_n(x)$ for every $x$. We represent the maximum function as $F(x) = \max\{F_1(x), F_2(x), \ldots, F_n(x)\}$. Let $M_{1,1}^{2,1}(t)$ denote the staircase function in Figure 5.6(a), and $M_{1,3}^{2,1}(t)$ denote the one in Figure 5.6(b). We argue that $\max\{M_{1,1}^{2,1}(t), M_{1,3}^{2,1}(t)\}$, shown in Figure 5.6(c), is an interference function of $T_1$ with respect to $T_{2,1}$. In other words, $M_1^{2,1}(t) = \max\{M_{1,1}^{2,1}(t), M_{1,3}^{2,1}(t)\}$.

To verify this statement, let us examine a $\phi_{2,1}$-level busy period starting from time 0. If an instance of $T_{1,1}$ is released at time $t'$ in the busy period and it is the first subtask instance of $T_1$ released in the busy period, then the amount of time demand from $T_1$ as a whole is no more than

Algorithm SA/IPM/IF1

**Input :**

1. Subtask $T_{i,j}$.
2. Task $T_k$ $(k \neq i)$.

**Output :** $M_k^{i,j}(t)$

**Algorithm :**

1. For every $T_{k,l}$ in $H_{i,j}$
   (a) Set the phase $f_{k,l}$ of $T_{k,l}$ equal to 0.
   (b) Set $f_{k,1}$ equal to $f_{k,n_k} + \tau_{k,n_k}$, if $l \neq 1$.
   (c) Set $f_{k,m}$ equal to $f_{k,m-1} + \tau_{k,m-1}$ if $m \neq l$ and $m \neq 1$.
   (d) Compute the total time demand $M_{k,l}^{i,j}(t)$ generated during the interval $[0, t)$ by all subtasks of $T_k$ that are in $H_{i,j}$, according to the above specified phases.
2. Return $\max\{M_{k,l}^{i,j}(t)\}$ for all $T_{k,l}$ in $H_{i,j}$ as the interference function $M_k^{i,j}(t)$.

**Figure 5.7**: Pseudo-Code of Algorithm SA/IPM/IF1

$\max\{0, M_{1,1}^{2,1}(t-t')\}$, which is smaller than or equal to $M_{1,1}^{2,1}(t)$. On the other hand, if an instance of $T_{1,3}$ is released at time $t'$ in the busy period and it is the first subtask instance of $T_1$ released in the busy period, the amount of time demand at time $t$ from $T_1$ as a whole is no more than $\max\{0, M_{1,3}^{2,1}(t-t')\}$, which is smaller than or equal to $M_{1,3}^{2,1}(t)$. In either case, the total time demand from $T_1$ is no greater than $M_1^{2,1}(t) = \max\{M_{1,1}^{2,1}(t), M_{1,3}^{2,1}\}$. In general, an interference function $M_k^{i,j}(t)$ $(k \neq i)$ can be computed by Algorithm SA/IPM/IF1, whose pseudo-code is listed in Figure 5.7.

We now describe how to compute the interference function $M_i^{i,j}(t)$ of a parent task to its subtask $T_{i,j}$, or more accurately speaking, the interference from the sibling subtasks of $T_{i,j}$. In [13], Bettati argued that if $T_i$ is schedulable, no instances of sibling subtasks of $T_{i,j}$ can be released in the interval between the release and completion of an instance of $T_{i,j}$. Consequently, there is no interference among sibling subtasks, and the interference function of $T_i$ with respect

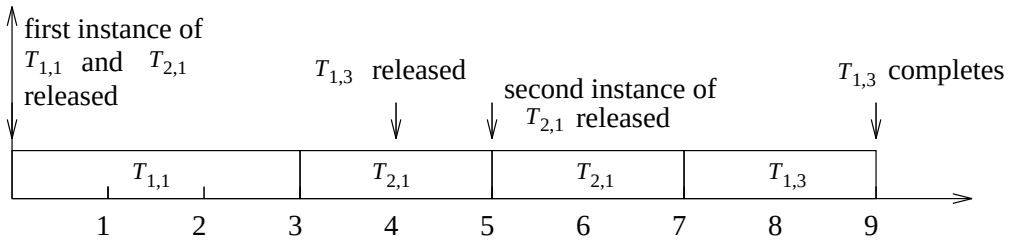



**Figure 5.8**: The Schedule of the System in Table 5.3

to $T_{i,j}$ should be 0. The following example, however, shows that this argument is not correct. In this example, the system has two tasks and two processors. The task parameters are listed in Table 5.3.

| $T_i$ | proc | $\phi_{i,j}$ | $p_{i,j}$ | $\tau_{i,j}$ |
|---|---|---|---|---|
| $T_{1,1}$ | $P_1$ | 1 | 20 | 3 |
| $T_{1,3}$ | $P_1$ | 5 | 20 | 2 |
| $T_{2,1}$ | $P_1$ | 3 | 5 | 2 |
| $T_{1,2}$ | $P_2$ | 2 | 20 | 1 |

**Table 5.3**: An Example to Illustrate the Interference among Sibling Subtasks

If the interference function $M_1^{1,3}(t)$ is equal to 0 for all $t > 0$, an upper bound on the response time of $T_{1,3}$ is 4 according to Algorithm SA/IPM. Suppose that $T_1$ and $T_2$ have zero phases. According to the PM protocol, the phases of $T_{1,1}$ and $T_{2,1}$ are 0, and the phase of $T_{1,3}$ is 4, which is equal to the sum of the upper bound on the response time of $T_{1,1}$ and the upper bound on the response time of $T_{1,2}$. Figure 5.8 shows the schedule of this example for the first 9 time units on $P_1$. The first instance of $T_{1,3}$ has a response time of 5 time units! Clearly, the previous schedulability analysis is wrong. A careful examination shows that the interference from $T_{2,1}$ is worse in the presence of $T_{1,1}$ than when $T_{2,1}$ were the only other subtask on $P_1$. $T_{1,1}$ interferes with the execution of $T_{1,3}$ in an indirect way. Therefore the interference function $M_1^{1,3}(t)$ needs to count the amount of this interference properly.

In general, a simple way to count the interference from sibling subtasks on the same processor as $T_{i,j}$ is to treat them as independent subtasks. The interference function of $T_i$ with respect to

$T_{i,j}$ can thus be computed by

$$M_i^{i,j}(t) = \sum_{T_{i,k} \in H_{i,j}} \left\lceil \frac{t}{p_i} \right\rceil \tau_{i,k} \tag{5.7}$$

Although correct, such a simple treatment can lead to pessimistic upper bounds on the response times of subtasks. For example, if we use Eq.(5.7) to compute the interference functions of $T_1$ to its subtasks in Figure 5.4, the upper bounds on the response times of $T_{1,1}$, $T_{1,2}$ and $T_{1,4}$ are 7, 6, and 6 respectively, which are clearly greater than the actual worst-case response times. How we can improve the schedulability algorithm further by computing tighter interference functions of a parent task to its subtasks is an open question.

### Relation between Algorithm SA/PM and Algorithm SA/IPM

The time demand function in Algorithm SA/PM, which is with respect to the $m$th subtask instance $T_{i,j}(m)$ in a $\phi_{i,j}$-level busy period, can be expressed in terms of interference functions as well. Specifically, we can express the time demand function $W(t)$ by $m\tau_{i,j} + \sum_{T_k} M_k^{i,j}(t)$ and each interference function $M_k^{i,j}(t)$ $(k \neq i)$ by

$$\sum_{T_{k,l} \in H_{i,j}} \left\lceil \frac{t}{p_k} \right\rceil \tau_{k,l}$$

In this way, Algorithm SA/PM can be viewed as a special case of Algorithm SA/IPM where the interference function is computed according to the above expression. (Here, we ignore the fact that Algorithm SA/IPM only computes an upper bound for the first instance of $T_{i,j}$ in a $\phi_{i,j}$-level busy period.) Because the interference function used in Algorithm SA/IPM is tighter than the one used in Algorithm SA/PM, Algorithm SA/IPM computes a tighter upper bound on the response time of the first instance of $T_{i,j}$ in a $\phi_{i,j}$-level busy period than Algorithm SA/PM. As we have discussed before, for a system where the relative deadlines of tasks are shorter than or equal to their respective periods, we only need an upper bound on the response time of the first instance of every subtask $T_{i,j}$ to verify the schedulability of the system. As a result, Algorithm SA/PM may predict a system to be not schedulable while Algorithm SA/IPM may predict it schedulable.

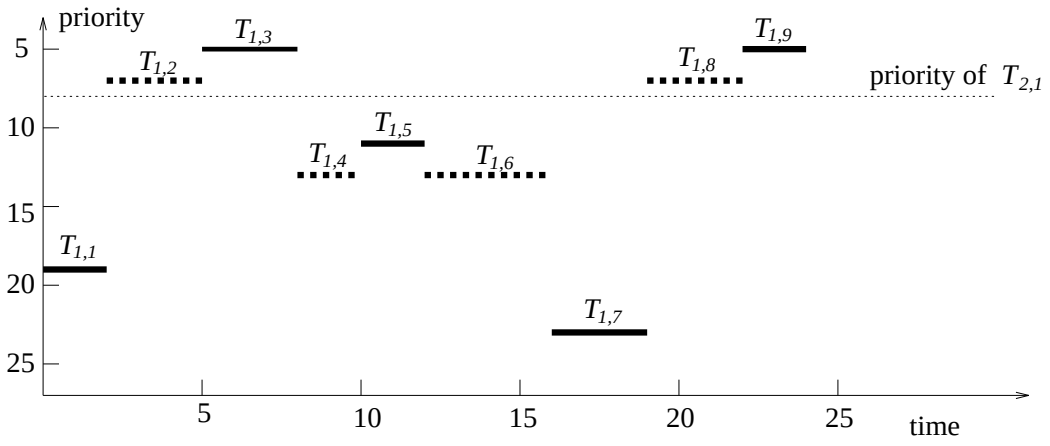



**Figure 5.9**: An Example to Illustrate the Second Improvement of Algorithm SA/PM

### 5.2.3 A Further Improvement of Algorithm SA/PM

Given that tasks may be recurrent and have relative deadlines shorter than or equal to their periods, we can further improve Algorithm SA/PM by a tighter interference function. Let $L_{i,j}$ be the set of subtasks that execute on the same processor as $T_{i,j}$ and have priorities lower than $T_{i,j}$. If a task $T_k$ has a subtask in $L_{i,j}$, the interference function of $T_k$ with respect to $T_{i,j}$ can be smaller than the one computed by Algorithm SA/IPM/IF1. To illustrate, let us look at the task $T_1$ depicted in Figure 5.9. $T_1$ has nine subtasks, among which $T_{1,1}$, $T_{1,3}$, $T_{1,5}$, $T_{1,7}$, and $T_{1,9}$ execute on the same processor as $T_{2,1}$. $T_{1,3}$ and $T_{1,9}$ have higher priorities than $T_{2,1}$, but $T_{1,1}$, $T_{1,5}$ and $T_{1,7}$ have lower priorities than $T_{2,1}$. In the picture, the subtasks of $T_1$ that execute on the same processor as $T_{i,j}$ are denoted as the solid line segments, and the other subtasks of $T_1$ are denoted by dashed line segments. The length of each line segment represents the maximum execution time of its corresponding subtask, and the higher a line segment is drawn in the picture, the higher the priority of the corresponding subtask. The period and the relative deadline of $T_1$ is 50.

If we use Algorithm SA/IPM/IF1 to compute the interference function $M_1^{2,1}(t)$, we obtain the time demand function $M_{1,3}^{2,1}(t)$ depicted in Figure 5.10(a) and time demand function $M_{1,9}^{2,1}(t)$ depicted in Figure 5.10(b). The interference function $M_1^{2,1}(t) = \max\{M_{1,3}^{2,1}(t), M_{1,9}^{2,1}(t)\}$ is depicted in Figure 5.10(c). However, because $T_1$ has three subtasks $T_{1,1}$, $T_{1,5}$ and $T_{1,7}$ in $L_{2,1}$, the actual time demand that $T_1$ can generate in a $\phi_{2,1}$-level busy period is much less than the interference function given in Figure 5.10(c).

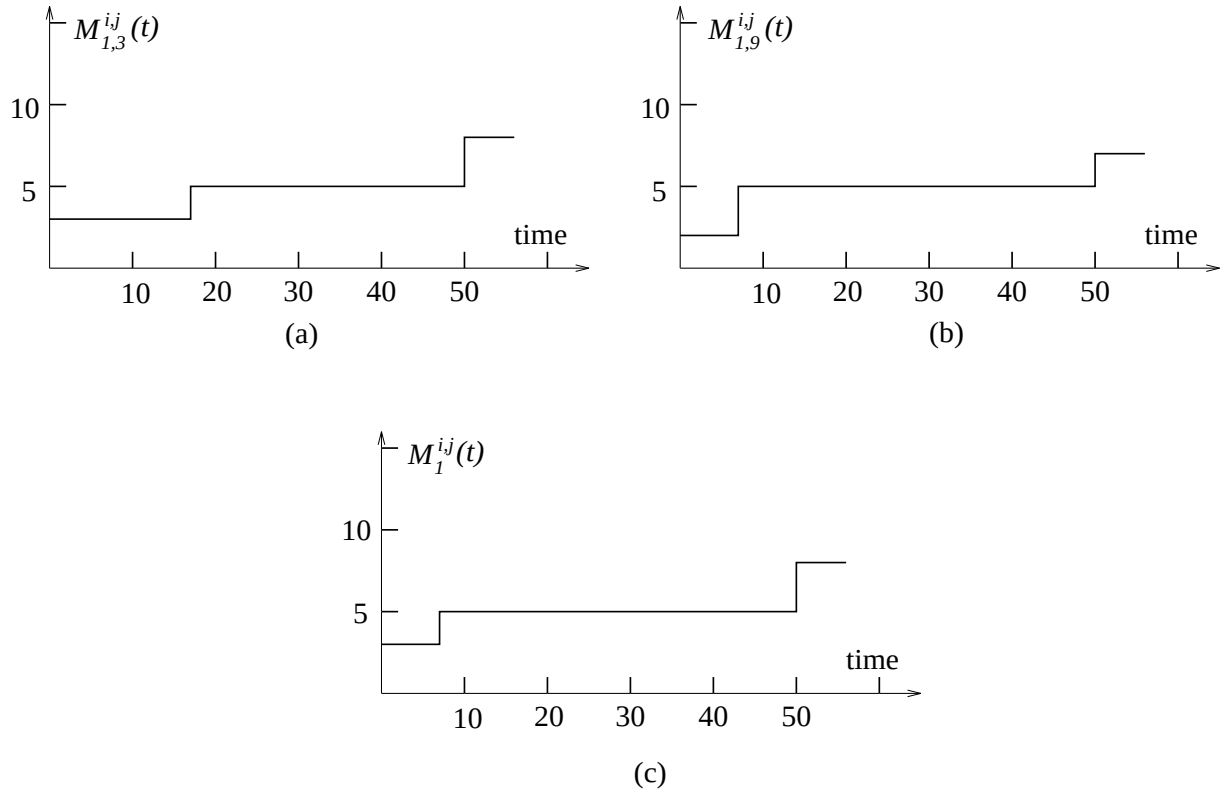



**Figure 5.10**: Time Demand Functions of $T_1$ in Figure 5.9

To see why, we suppose that an instance of $T_{1,3}$ is released and completed in a $\phi_{2,1}$-level busy period. The corresponding instance of $T_{1,5}$ is released sometime later and must be completed after the busy period because $T_{1,5}$ has a lower priority than $\phi_{2,1}$. According to the PM protocol, the corresponding instances of the successors of $T_{1,5}$, including $T_{1,9}$, must be released after this $\phi_{2,1}$-level busy period ends. In this case, the total time demand generated by task $T_1$ is no more than the maximum execution time of $T_{1,3}$, which is 3 time units.

On the other hand, suppose an instance of $T_{1,9}$ is released and completed in a $\phi_{2,1}$-level busy period. By the assumption that $T_1$ is schedulable, the next instance of $T_{1,1}$ must be released later than the completion of the instance of $T_{1,9}$ and complete after the busy period ends. According to the PM protocol, the next instance of $T_{1,3}$ must be released after the busy period ends as well. In this case, we see that the total time demand generated by $T_1$ is no more than the maximum execution time of $T_{1,9}$, which is 2 time units. Overall, we can conclude that the time demand that $T_1$ can generate in a $\phi_{2,1}$-level busy period is no more than 3 time units, much less than what the interference function in Figure 5.10(c) gives.

To take into account the effect of subtasks $T_{k,l}$ in $L_{i,j}$, we devise Algorithm SA/IPM/IF2 to compute the interference function of $T_k$ with respect to $T_{i,j}$. The pseudo-code of Algorithm SA/IPM/IF2 is listed in Figure 5.11. The structure of Algorithm SA/IPM remains the same, except now Algorithm SA/IPM/IF2 is used to compute the interference function instead of Algorithm SA/IPM/IF1. In the rest of this thesis, by Algorithm SA/IPM, we will mean Algorithm SA/IPM that uses Algorithm SA/IPM/IF2 to compute the interference functions, unless specified otherwise.

## 5.3 Schedulability Analysis for the RG Protocol

According to the RG protocol, an instance of a subtask is released when its immediate predecessor completes or when the time is equal to its release guard, whichever is later. There are three rules regarding how to update the value of a release guard, listed in Section 4.2.4. In this section, we show that Algorithm SA/PM presented in the previous section can be applied to a system using the RG protocol to obtain bounds on task EER times. We first establish the following lemma.

**Lemma 2** *In an end-to-end system using the RG protocol, the response time of a subtask is upper-bounded by the bound computed by Algorithm SA/PM.*

Algorithm SA/IPM/IF2

**Input :**

1. Subtask $T_{i,j}$.
2. Task $T_k$ $(k \neq i)$

**Output :** The interference function $M_k^{i,j}(t)$ of $T_k$ with respect to $T_{i,j}$

**Algorithm :**

1. For every $T_{k,l}$ in $H_{i,j}$
   (a) Set the phase $f_{k,l}$ of $T_{k,l}$ equal to 0.
   (b) If $l \neq 1$, set $f_{k,1}$ equal to $f_{k,n_k} + \tau_{k,n_k}$.
   (c) Set $f_{k,m}$ equal to $f_{k,m-1} + \tau_{k,m-1}$ if $m \neq l$ and $m \neq 1$.
   (d) According to the above phase adjustment, let $t'$ be the release time of the first instance of some $T_{k,l}$ in $L_{i,j}$ if $T_k$ has subtasks in $L_{i,j}$. Otherwise let $t'$ be $\infty$.
   (e) $M_{k,l}^{i,j}(t)$ is equal to the time demand generated by all the subtasks of $T_k$ that are in $H_{i,j}$ during the time interval $[0, t)$ for $0 \leq t \leq t'$; $M_{k,l}^{i,j}(t)$ is equal to $M_{k,l}^{i,j}(t')$ for $t > t'$.
2. Return $\max\{M_{k,l}^{i,j}(t)\}$ for all $T_{k,l}$'s in $H_{i,j}$ as the interference function $M_k^{i,j}(t)$.

**Figure 5.11**: Pseudo-Code of Algorithm SA/IPM/IF2

Proof :

In a $\phi_{i,j}$-level busy period, there is no processor idle point. Rule (3) regarding how to update the value of a release guard at a processor idle point is not applicable. By Rule (1) and (2) alone, the inter-release time of every subtask on the processor where $T_{i,j}$ executes is no less than the period of the subtask. As a result, $\lceil t/p_u \rceil \tau_{u,v}$ is an upper bound on the time demand that any subtask $T_{u,v}$ in $H_{i,j} \cup \{T_{i,j}\}$ can generate during an interval $[t_0, t_0 + t)$ in a $\phi_{i,j}$-level busy period starting from time $t_0$. According to the busy period analysis, we will obtain the same bound on the response time of $T_{i,j}$ as Algorithm SA/PM. □

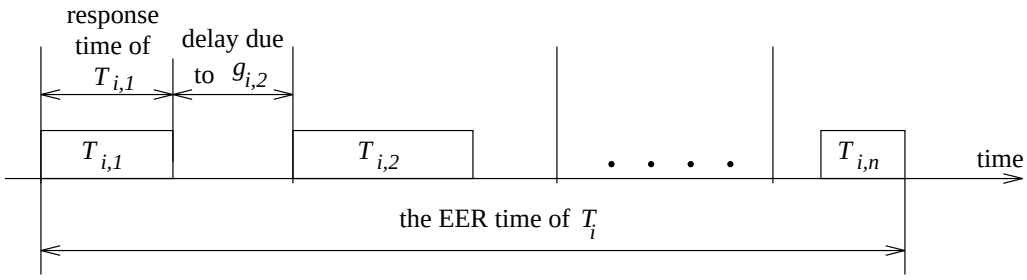



**Figure 5.12**: The EER Time of a Task in a System Using the RG Protocol

Lemma 2 alone cannot support our claim that Algorithm SA/PM is applicable to the RG protocol to find upper bounds on the task EER times. The reason is illustrated by Figure 5.12. Other than the response times of all its subtasks, the EER time of a task also includes delays in releasing instances of subtasks. (In the figure, delay due to $g_{i,2}$ is an example of such delays.) Let $T_{i,j}(m)$ denote the $m$th instance of subtask $T_{i,j}$ in an arbitrary schedule. We define the *intermediate end-to-end response (IEER) time* of $T_{i,j}(m)$ to be the completion time of $T_{i,j}(m)$ minus the release time of $T_{i,1}(m)$, the corresponding instance of the first subtask in $T_i$. The IEER time of the first subtask in a task is simply its response time and the IEER time of the last subtask in a task is the EER time of the task. We now try to establish Lemma 3.

**Lemma 3** *The IEER time of every subtask $T_{i,j}$ is bounded by $\sum_{k=1}^{j} R_{i,k}$, where $R_{i,j}$ is an upper bound on the response time of $T_{i,j}$.*

Proof:

If $T_{i,j}$ is the first subtask in $T_i$, i.e., $j = 1$, its IEER time is its response time, and hence is no greater than $R_{i,1}$. The lemma holds trivially in this case.

We now consider the case when $j > 1$. Let $T_{i,j}(m)$ denote the $m$th instance of $T_{i,j}$ in an arbitrary schedule. Let $r_{i,j}(m)$ denote the release time of $T_{i,j}(m)$ and $C_{i,j}(m)$ denote the completion time of $T_{i,j}(m)$. To prove the lemma, we need to demonstrate that $C_{i,j}(m) - r_{i,1}(m) \leq \sum_{k=1}^{j} R_{i,k}$ for $j = 2, 3, \ldots, n_i$ and $m = 1, 2, \ldots$. In order to do so, it suffices to prove $r_{i,j}(m) \leq r_{i,1}(m) + \sum_{k=1}^{j-1} R_{i,k}$, because $C_{i,j}(m) - R_{i,j} \leq r_{i,j}(m)$. We will prove this by an induction on $m$, the instance index of a subtask instance $T_{i,j}(m)$.

**Induction basis :** The first instance $T_{i,j}(1)$ of each subtask $T_{i,j}$ $(j > 1)$ is released as soon as $T_{i,j-1}(1)$ completes because $g_{i,j}$ is initially set to 0 and delay due to $g_{i,j}$ never occurs. Therefore, the IEER time of $T_{i,j}(1)$ is equal to the sum of response times of itself and all its predecessor. In other words, the IEER time is no greater than $\sum_{k=1}^{j} R_{i,k}$.

**Induction hypothesis :** Suppose that $r_{i,j}(m-1) \leq r_{i,1}(m-1) + \sum_{k=1}^{j-1} R_{i,k}$ $(m \geq 1)$.

**Induction :** We now try to establish that $r_{i,j}(m) \leq r_{i,1}(m) + \sum_{k=1}^{j-1} R_{i,k}$. According to the RG protocol, the release time of a subtask instance $T_{i,j}(m)$ for $j > 1$ and $m > 1$ is when its immediate predecessor completes or when its release guard is due, whichever is later. By Rules (2) and (3) for updating the release guard, we have

$$\begin{aligned} r_{i,j}(m) &\leq \max\{C_{i,j-1}(m), r_{i,j}(m-1) + p_i\} \\ &\leq \max\{r_{i,j-1}(m) + R_{i,j-1}, r_{i,j}(m-1) + p_i\} \end{aligned}$$

By the induction hypothesis, we have $r_{i,j}(m-1) \leq r_{i,1}(m-1) + \sum_{k=1}^{j-1} R_{i,k}$. Because $T_{i,1}$ is a periodic subtask, we have $r_{i,1}(m-1) + p_i \leq r_{i,1}(m)$. The above inequality can then be written as

$$r_{i,j}(m) \leq \max\left\{ r_{i,j-1}(m) + R_{i,j-1}, r_{i,1}(m) + \sum_{k=1}^{j-1} R_{i,k} \right\}$$

We can expand the above inequality recursively and obtain $r_{i,j}(m) \leq r_{i,1}(m) + \sum_{k=1}^{j-1} R_{i,k}$. By induction, we know that this inequality holds for $m = 1, 2, \ldots$. Therefore, we conclude that the lemma is correct when $j > 1$.

□

**Theorem 1** *The task EER times in an end-to-end system using the RG protocol can be bounded by Algorithm SA/PM.*

Proof :

The EER time of a task is equal to the IEER time of its last subtask. The theorem thus follows Lemma 2 and Lemma 3. □

Unfortunately, the improvement to Algorithm SA/PM, Algorithm SA/IPM, is not applicable to a system using the RG protocol. Take the system in Figure 5.4 for example. According to the PM or MPM protocol, the lengths of the interval between the completion time of an instance of $T_{1,1}$ and the release time of the corresponding instance of $T_{1,3}$ and the interval between the completion time of an instance of $T_{1,3}$ and the release time of the next instance of $T_{1,1}$ are at least 2 time units due to the adjusted phases of the subtasks. This gives a chance for an instance of $T_{2,1}$ to execute and complete. We hence obtain a tighter bound on the response time of $T_{2,1}$. According to the RG protocol, however, we do not have a guaranteed minimum length. For example, if subtask instance $T_{1,1}(1)$ and $T_{2,1}(1)$ are released at time 0 and every subtask instance has the maximum execution time except that $T_{1,2}(1)$ has the execution time of 1 time unit, the interval between the completion time of $T_{1,1}(1)$ and the release time of $T_{1,3}(1)$ becomes 1 time unit. As a consequence, $T_{2,1}(1)$ will be preempted by both $T_{1,1}(1)$ and $T_{1,3}(1)$, resulting in a response time of 9 time units and causing the first instance of $T_2$ to miss its deadline.

## 5.4 Schedulability Analysis for the SS Protocol

The task behavior in a system using the SS protocol is much more complicated than in a system using other protocols. According to the PM, MPM, and RG protocols, only the releases of subtasks are controlled. Once a subtask instance is released, it executes to completion according to its own fixed priority. In other words, these synchronization protocols only govern the release but not the execution of a subtask instance. According to the SS protocol, however, a subtask instance is released as soon as its immediate predecessor completes, but its execution can be suspended at any point if the server runs out its budget. In a sense, the SS protocol governs the execution rather than the release of a subtask instance.

To simplify our discussion, let us first consider the case where every subtask instance has the maximum execution time. It can be easily shown that in this case the amount of budget replenished every time is always equal to the server execution time, i.e., there is no partial replenishment. This further implies that at any time the server either has just enough budget to

execute a whole number of waiting subtask instances to completion or does not have any budget at all. In this case, we argue that the schedule obtained by the SS protocol is exactly the same as the schedule obtained by RG protocol. Rather than proving this statement, we list the one-to-one mapping between events in these two protocols in Table 5.4. This mapping gives an intuitive explanation of why this argument is true.

| Events in the RG protocol | Events in the SS protocol |
|---|---|
| $T_{i,j}(m)$ is released as soon as $T_{i,j-1}(m)$ completes, | $T_{i,j}(m)$ is released. and its server has enough budget to complete it. |
| $T_{i,j-1}(m)$ completes, but $T_{i,j}(m)$ is not released. | $T_{i,j}(m)$ is released, but its server does not have budget for it |
| $T_{i,j}(m)$ is released when the current time is $g_{i,j}$. | the budget of the server of $T_{i,j}$ gets replenished, and the server now has enough budget for $T_{i,j}(m)$. |

**Table 5.4**: One-to-One Mapping of Events in the RG Protocol and the SS Protocol

Since we have the same schedules for both the SS protocol and the RG protocol under the assumption that every subtask instance has the maximum execution time, obviously Algorithm SA/PM is equally applicable to bound the EER times of tasks in the SS protocol. For the similar reasons as we discussed for the RG protocol, the improved version of Algorithm SA/PM, Algorithm SA/IPM, is not applicable to the SS protocol either.

In general, the assumption that every subtask instance has the maximum execution time is not true. However, Algorithm SA/PM remains applicable to the SS protocol even after the assumption is removed. If subtask instances have execution times shorter than their maximum execution times, they may not use up all of the budget, and some later instances of the same subtask may get served earlier than they get served otherwise. Subtasks getting served earlier should shorten the overall EER times of their parent tasks. Due to the property of sporadic servers, doing so does not hurt the response times of other subtasks.

## 5.5 Schedulability Analysis for the DS Protocol

In a system that uses the DS protocol, the *time demand analysis* for periodic task systems [1, 5, 6] cannot be used directly to obtain a bound on the response time of each subtask. Several instances

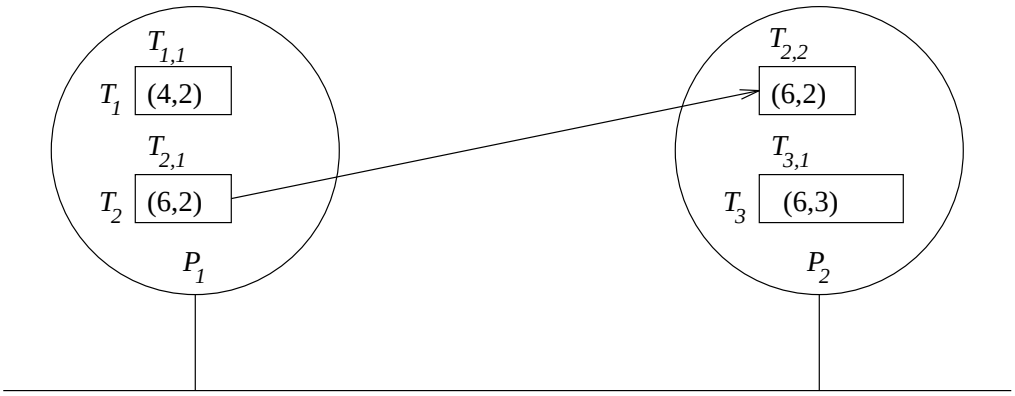



**Figure 5.13**: An Example to Illustrate the Clumping Effect

of a higher-priority subtask may execute back-to-back consecutively or rather closely together in time, a phenomenon called the *clumping effect*. Because of the *clumping effect*, the problem in bounding the task EER times in a system using the DS protocol is quite different from that of the previous four protocols due to a phenomenon called *clumping effect*. In this section, we first demonstrate the clumping effect and then present Algorithm SA/DS that bounds the task EER times in an end-to-end system using the DS protocol.

### 5.5.1 Clumping Effect

The example in Figure 5.13 illustrates the clumping effect. The system is a part of the system shown in Figure 3.3. The end-to-end relative deadline for each task is equal to its period.

On $P_1$ subtask $T_{1,1}$ has higher priority than $T_{2,1}$, and on $P_2$ subtask $T_{2,2}$ has higher priority than $T_{3,1}$. Task $T_3$ has a phase of 4 and others have zero phases. A schedule for the first 10 time units of this system was shown in Figure 3.4. It is copied here as Figure 5.14 for easy reference. In the schedule, we notice that the first two instances of $T_{2,1}$ complete at time 4 and 8, causing two instances of $T_{2,2}$ to be released at those times. As a result, the first two instances of $T_{2,2}$ execute closely together, which causes the first instance of $T_{3,1}$ to be preempted twice by $T_{2,2}$ and $T_3$ misses its deadline at time 10. On the other hand, if $T_{2,2}$ were released periodically at the period of 6, $T_{3,1}$ would never be preempted twice by $T_{2,2}$ because they have the same period.

In general, the completion time of a subtask can vary widely, especially if the subtask is near the tail of a long subtask chain. The variation is not only caused by preemption by higher priority subtasks but also due to variations in the execution times of subtasks. If one instance of

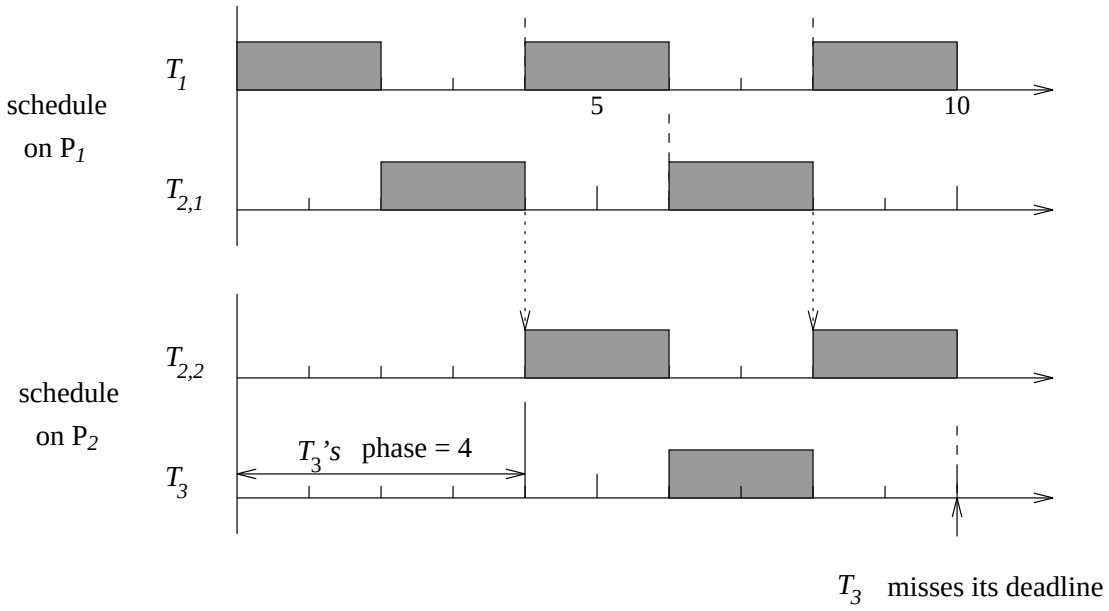



**Figure 5.14**: A Schedule of the System in Figure 5.13

$T_{i,j}$ completes rather late but all subsequent instances complete relatively early, it is conceivable that many instances of $T_{i,j+1}$ can be released and ready for execution rather close together in time. If there is another subtask on the same processor as $T_{i,j+1}$ but with a lower priority, its execution will be delayed much longer than if $T_{i,j+1}$ were released periodically. As a consequence, the schedulability of the whole system is affected negatively. This is what we call the *clumping effect*.

In Algorithm SA/PM and SA/IPM, we bound the response times of all the subtasks of a task and then sum the bounds together as an upper bound on the EER time of the task. The clumping effect, however, causes many difficulties for this approach. We notice, from the previous example, that a subtask whose execution is synchronized according to the DS protocol interferes with the execution of a lower priority subtask on the same processor much more than a periodic subtask. To bound the extra interference precisely, we end up with a mutually dependent system to analyze [19]. Furthermore, since many instances of the same subtask can be released almost simultaneously, they may interfere with each other (because later instances need to wait until previous instances complete before they can start). Under this situation, the bound on the response time of a subtask may be quite pessimistic.

We therefore take a different approach to bounding the EER times of tasks in a system using the DS protocol. We bound the IEER times of subtasks instead of the response times of subtasks, and, naturally, the bound on the IEER time of the last subtask is the bound on the EER time of the parent task.

### 5.5.2 Overall Structure of Algorithm SA/DS

*Algorithm SA/DS* uses the above described strategy to bound the task EER times in a system using the DS protocol. Figure 5.15 shows its structure. The input to Algorithm SA/DS consists of the parameters of a set of end-to-end periodic tasks, $\{T_i\}$, and the output is a set $\{R_i\}$ of correct upper bounds on the task EER times. According to Algorithm SA/DS, we first obtain an initial upper bound $V_{i,j}$ on the IEER time for each subtask $T_{i,j}$. This initial bound may be too optimistic, i.e., the bound may be incorrect because it can be smaller than the actual worst-case IEER time of the subtask. We then feed both the parameters of the tasks and the initial bounds to Algorithm IEERT, which computes a set $\{V'_{i,j}\}$ of new upper bounds on subtask IEER times. If all the new bounds are equal to their corresponding bounds provided at input, the bound $V_{i,n_i}$ on the IEER time of the last subtask of each task is the (correct) upper bound $R_i$ on the task EER time obtained by Algorithm SA/DS. Otherwise, we set $V_{i,j}$ to be equal to $V'_{i,j}$ for every subtask, use the new input and apply Algorithm IEERT again. We repeat this iterative process until all the new bounds are equal to their corresponding input bounds. In the following subsections we first describe Algorithm IEERT and then prove that the bounds obtained by Algorithm SA/DS are correct upper bounds on task IEER times when the iteration terminates. Lastly, we address the termination issue of Algorithm SA/DS and relate our work to existing work.

### 5.5.3 Algorithm IEERT

Algorithm IEERT takes as input the parameters of a set of end-to-end periodic tasks in the system and upper bounds on the IEER times of subtasks. The algorithm computes a set of new upper bounds on the IEER times of subtasks as the output. In the following discussion, we focus on subtask $T_{i,j}$, the *target subtask*, whose IEER time is to be bounded.

The key technique used in Algorithm IEERT is the busy period analysis discussed earlier in this chapter. Similar to Algorithm SA/PM, we first derive a bound $D_{i,j}$ on the duration of a $\phi_{i,j}$-level busy period. Suppose a $\phi_{i,j}$-level busy period starts at time $t_0$. Let $H_{i,j}$ denote the

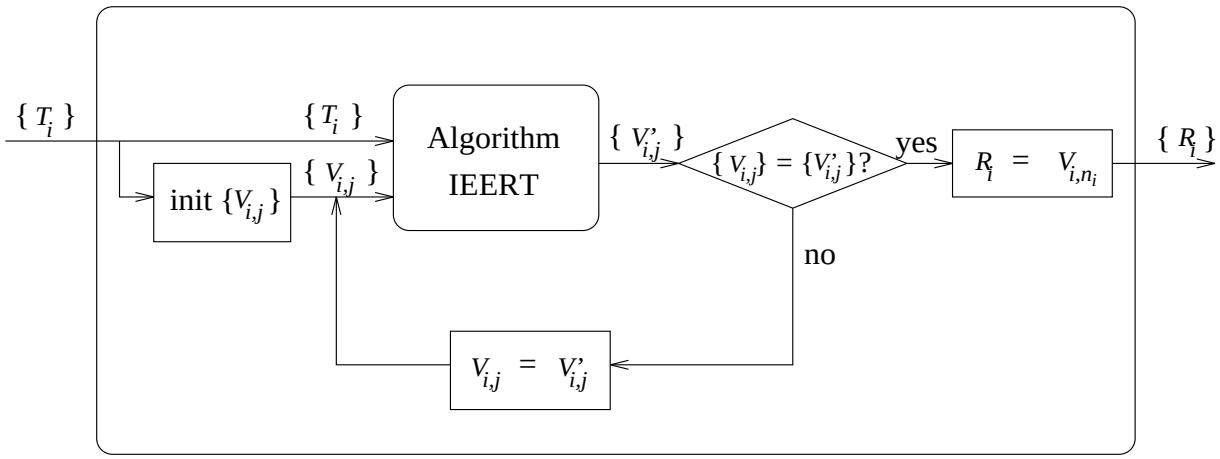



**Figure 5.15**: The Flow Control Diagram of Algorithm SA/DS

set of subtasks, excluding $T_{i,j}$, that are on the same processor as $T_{i,j}$ and have priorities higher than or equal to $T_{i,j}$. The time demand function $W(t)$ with respect to the busy period is equal to the sum of the amount of time demand generated by every subtask in $H_{i,j} \cup \{T_{i,j}\}$ during interval $[t_0, t_0 + t)$. During the interval, the amount of time demand generated by a subtask $T_{u,v}$ in $H_{i,j} \cup \{T_{i,j}\}$ cannot exceed the product of the number of instances of $T_{u,v}$ released in the interval and the maximum execution time $\tau_{u,v}$ of $T_{u,v}$. Unlike Algorithm SA/PM, however, the number of $T_{u,v}$ that are released in interval $[t_0, t_0 + t)$ cannot be easily bounded because $T_{u,v}$ may not be released periodically due to the clumping effect.

To bound the number of instances of $T_{u,v}$ (in $H_{i,j} \cup \{T_{i,j}\}$) that can be released during the interval $[t_0, t_0+t)$ in a $\phi_{i,j}$-level busy period starting from time $t_0$, we examine Figure 5.16 which illustrates the releases of instances of $T_{u,v}$ in the interval, as well as the corresponding instances of the predecessors of $T_{u,v}$. In the figure, $r_{x,y}(z)$ indicates the release time of subtask instance $T_{x,y}(z)$. Suppose there are $k$ instances of $T_{u,v}$ released in interval $[t_0, t_0+t)$. Let $T_{u,v}(1)$ ($T_{u,v}(k)$) denote the first (last) instance of $T_{u,v}$ released in the busy period. $T_{u,1}(1)$ ($T_{u,1}(k)$) denotes the instance of the first subtask $T_{u,1}$ corresponding to $T_{u,v}(1)$ ($T_{u,v}(k)$). Obviously the number of instances of $T_{u,v}$ in the busy period is equal to the number of instances of $T_{u,1}$ released in interval $[r_{u,1}(1), r_{u,1}(k)]$. Since subtask $T_{u,1}$ is a periodic subtask, we can bound the number of instances of $T_{u,1}$ released in interval $[r_{u,1}(1), r_{u,1}(k)]$ if we can bound the duration of the interval. From the figure, we can derive an upper bound on the duration of interval $[r_{k,1}(1), r_{u,1}(k)]$ as follows.

$$r_{u,1}(k) - r_{u,1}(1) \quad < \quad ((t_0 + t) - t_0) + (r_{u,v}(1) - r_{u,1}(1))$$

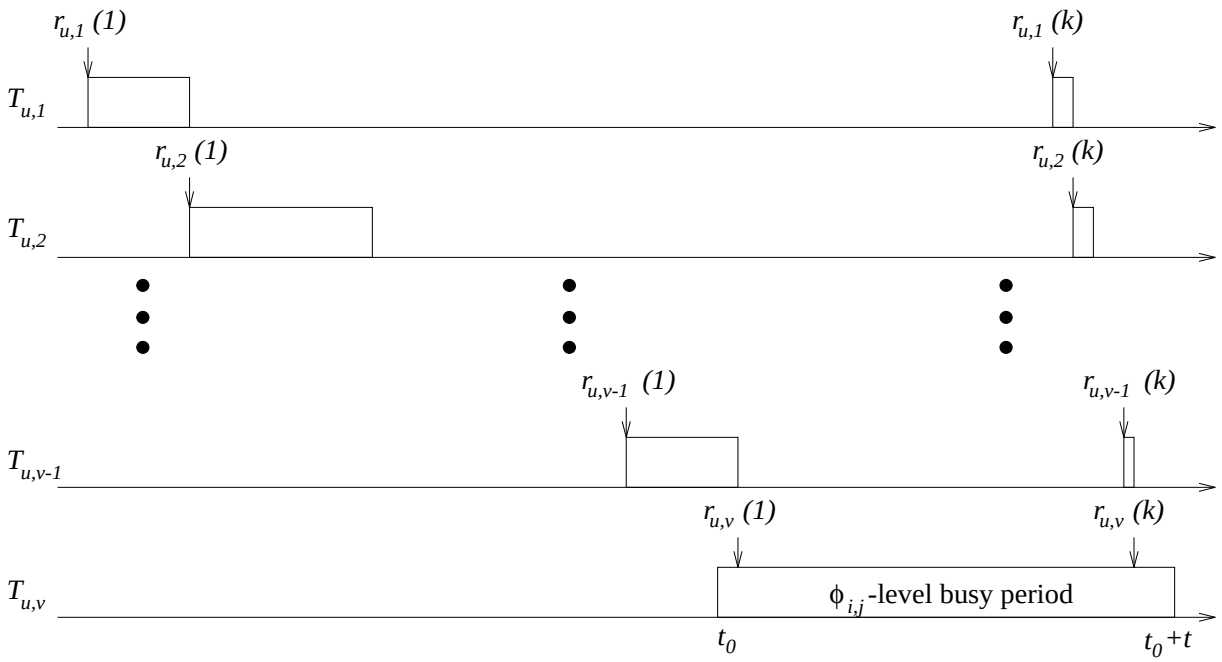



**Figure 5.16**: Maximal Duration of a $\phi_{i,j}$-Level Busy Period (DS Protocol)

$$\leq \quad t + V_{u,v-1}$$

where $V_{u,v-1}$ is the input upper bound on the IEER time of $T_{u,v-1}$. (By assumption, $V_{i,0}$ is equal to 0 for any $i$.) Consequently, the number $k$ of instances of $T_{u,v}$ in interval $[t_0, t_0 + t)$ can be bounded by $\lceil (D + R_{u,v-1})/p_u \rceil$. The time demand function with respect to a $\phi_{i,j}$-level busy period can be upper-bounded by

$$\sum_{T_{u,v} \in H_{i,j} \cup \{T_{i,j}\}} \left\lceil \frac{t + R_{u,v-1}}{p_u} \right\rceil \tau_{u,v}$$

According to the busy period analysis, an upper bound $D_{i,j}$ on the duration of a $\phi_{i,j}$-level busy period can be computed by

$$D_{i,j} = \min \left\{ t > 0 \mid t = \sum_{T_{u,v} \in H_{i,j} \cup \{T_{i,j}\}} \left\lceil \frac{t + R_{u,v-1}}{p_u} \right\rceil \tau_{u,v} \right\} \tag{5.8}$$

There exists a solution for $D_{i,j}$ if $\sum_{T_{u,v} \in H_{i,j} \cup \{T_{i,j}\}} \tau_{u,v}/p_u$ is less than 1. The iterative process described in Section 5.1 can be employed to obtain the solution if it exists.

Given the maximum duration $D_{i,j}$ of any $\phi_{i,j}$-level busy period, the maximal number $M_{i,j}$ of instances of $T_{i,j}$ in the busy period is given by

$$M_{i,j} = \left\lceil \frac{D_{i,j} + V_{i,j-1}}{p_i} \right\rceil \tag{5.9}$$

We now try to obtain a bound on the IEER time of an arbitrary instance of $T_{i,j}$. Since any instance of $T_{i,j}$ must be released and completed in a $\phi_{i,j}$-level busy period, let this arbitrary instance $T_{i,j}(m)$ be the $m$th ($1 \leq m \leq M_{i,j}$) instance of $T_{i,j}$ released in the busy period. By an analysis similar to the one described in the previous paragraph, the time demand function with respect to $T_{i,j}(m)$ is bounded from above by

$$m\tau_{i,j} + \sum_{T_{u,v} \in H_{i,j}} \left\lceil \frac{t + V_{u,v-1}}{p_u} \right\rceil \tau_{u,v}$$

According to the time demand analysis, $t_0 + C_{i,j}(m)$ is an upper bound on the completion time of $T_{i,j}(m)$ where $t_0$ is the time when the busy period starts and $C_{i,j}(m)$ is given by

$$C_{i,j}(m) = \min \left\{ t > 0 \mid t = m\tau_{i,j} + \sum_{T_{u,v} \in H_{i,j}} \left\lceil \frac{t + V_{u,v-1}}{p_u} \right\rceil \tau_{u,v} \right\} \tag{5.10}$$

This equation has a solution for $C_{i,j}(m)$ if $\sum_{T_{u,v} \in H_{i,j}} \tau_{u,v}/p_u$ is less than 1. A lower bound for the release time of $T_{i,1}(m)$ is $t_0 - V_{i,j-1} + (m-1)p_i$. Thus an upper bound $V_{i,j}(m)$ on the IEER time of $T_{i,j}(m)$ can be computed by

$$V_{i,j}(m) = C_{i,j}(m) + V_{i,j-1} - (m-1)p_i \tag{5.11}$$

Since $V_{i,j}(m)$ is an upper bound on the IEER time of the $m$th instance of $T_{i,j}$ in any $\phi_{i,j}$-level busy period, the maximum of $V_{i,j}(m)$ for $m = 1, 2, \ldots, M_{i,j}$ must be an upper bound on the IEER time of $T_{i,j}$ in general. This is the new bound $V'_{i,j}$ computed by Algorithm IEERT. The pseudo-code of the Algorithm IEERT is listed in Figure 5.17. From the above analysis, we can see that the correctness of $V'_{i,j}$ depends on the correctness of the input bounds. As long as all the input bounds are correct, the output bounds computed by Algorithm IEERT are correct.

### 5.5.4 Algorithm SA/DS

Let $\mathbf{V} = \{V_{i,j}\}$, $\mathbf{T} = \{T_{i,j}\}$, and $IEERT(\mathbf{T}, \mathbf{V})$ denote the set $\mathbf{R}'$ ($\{R'_{i,j}\}$) of new upper bounds obtained by Algorithm IEERT, i.e., $\mathbf{R}' = IEERT(\mathbf{T}, \mathbf{R})$. Figure 5.18 lists the pseudo-code of Algorithm SA/DS. In the initialization step, we use the sum of the maximum execution times of $T_{i,j}$ and its predecessors as an initial estimate of the bound on the IEER time of $T_{i,j}$. Obviously, this estimate may be overly optimistic. After the iteration terminates, the bound on the IEER time of $T_{i,n_i}$ computed during the last iteration is the bound on the EER time of $T_i$.

Algorithm IEERT

**Input :**

1. A set $\{T_i\}$ of end-to-end periodic tasks.
2. A set $\{V_{i,j}\}$ of bounds on the IEER times of subtasks.

**Output :** A set $\{V'_{i,j}\}$ of new bounds on the IEER times of subtasks.

**Algorithm :**

For each subtask $T_{i,j}$

1. Compute an upper bound $D_{i,j}$ on the duration of a $\phi_{i,j}$-level busy period

$$D_{i,j} = \min \left\{ t > 0 \mid t = \sum_{T_{u,v} \in H_{i,j} \cup \{T_{i,j}\}} \left\lceil \frac{t + V_{u,v-1}}{p_u} \right\rceil \tau_{u,v} \right\}$$

2. Compute an upper bound $M_{i,j}$ on the number of instances of $T_{i,j}$ in a $\phi_{i,j}$-level busy period

$$M_{i,j} = \left\lceil \frac{D_{i,j} + V_{i,j-1}}{p_i} \right\rceil$$

3. For $m = 1$ to $M_{i,j}$ do

   (a) Compute $C_{i,j}(m)$ by solving the following equation.

$$C_{i,j}(m) = \min \left\{ t > 0 \mid t = m\tau_{i,j} + \sum_{T_{u,v} \in H_{i,j}} \left\lceil \frac{t + V_{u,v-1}}{p_u} \right\rceil \tau_{u,v} \right\}$$

   (b) Compute an upper bound $V_{i,j}(m)$ on the IEER time of the $m$th instance in a $\phi_{i,j}$-level busy period

$$V_{i,j}(m) = C_{i,j}(m) + V_{i,j-1} - (m-1)p_i$$

4. Compute the new bound $V'_{i,j}$ by

$$V'_{i,j} = \max\{V_{i,j}(m)\}, \quad for\ 1 \le m \le M$$

**Figure 5.17**: Pseudo-Code of Algorithm IEERT

Algorithm SA/DS

**Input :** Task set $\mathbf{T}$.

**Output :** The set $\mathbf{R}$ of upper bounds on the EER times of tasks.

**Algorithm :**

1. For each subtask $T_{i,j}$,
$$V'_{i,j} = \sum_{m=1}^{j} \tau_{i,m}$$
$$V_{i,j} = 0$$
2. Repeat until ($V_{i,j} = V'_{i,j}$ for every subtask $T_{i,j}$)
   (a) $\mathbf{V} = \mathbf{V}'$.
   (b) $\mathbf{V}' = IEERT(\mathbf{T}, \mathbf{V})$.
3. For each task $T_i$, $R_i = V_{i,n_i}$.

**Figure 5.18**: Pseudo-Code of Algorithm SA/DS

Theorem 2 states an important attribute of Algorithm IEERT, which can be applied directly to prove that Algorithm SA/DS is correct. The proof of this theorem makes use of the following lemma.

**Lemma 4** *Suppose that a subtask instance $T_{i,j}(m)$ completes at time $t$. If the IEER time of every subtask instance $T_{u,v}(w)$ that completes before $t$ is no greater than some $X_{u,v} > 0$, then the IEER time of $T_{i,j}(m)$ is no greater than $X'_{i,j}$, where $\mathbf{X}' = IEERT(\mathbf{T}, \mathbf{X})$.*

Proof:

The correctness of Lemma 4 follows from the busy period analysis presented in the previous sections. □

**Theorem 2** *Let $\mathbf{X} = \{X_{i,j}\}$ be a set of positive numbers, where there is a one-to-one mapping between $X_{i,j}$ and $T_{i,j}$. If $\mathbf{X} = IEERT(\mathbf{T}, \mathbf{X})$, then $X_{i,j}$ is a correct upper bound on the IEER time of $T_{i,j}$.*

Proof:

Suppose that the system starts to run at time origin ($t = 0$). We prove the theorem by an induction over all the subtask instances in an arbitrary schedule in the order of their completion times.

**Induction basis :** Starting from time 0, the first completed subtask instance in the system must be the first instance of the first subtask of a task. Let this subtask instance be $T_{i,1}(1)$. Suppose that $T_{i,1}(1)$ completes at time $t_0$. Obviously on the processor where $T_{i,1}$ executes, $T_{i,1}(1)$ has the highest priority among all subtask instances that are released before time $t_0$. The response time of $T_{i,1}(1)$ is then equal to its execution time which is less than or equal to $\tau_{i,1}$. On the other hand, the condition of the theorem that $\mathbf{X} = IEERT(\mathbf{T}, \mathbf{X})$ allows us to say that $X_{i,j}$ must be greater than or equal to $\tau_{i,j}$ for every subtask $T_{i,j}$. Therefore, the response time of $T_{i,1}(1)$, or the IEER time of $T_{i,1}(1)$ in this case, must be less than or equal to $X_{i,1}$.

**Induction hypothesis :** The IEER time of every subtask instance $T_{u,v}(w)$ that completes before time $t$ is no longer than the corresponding $X_{u,v}$.

**Induction :** Suppose an arbitrary subtask instance $T_{i,j}(m)$, which is not the first completed subtask instance in the system, completes at time $t$. According to Lemma 4, the IEER time of $T_{i,j}(m)$ is no greater than $X'_{i,j}$, where $\mathbf{X}' = IEERT(\mathbf{T}, \mathbf{X})$. By the condition specified in the theorem, we have $\mathbf{X}' = \mathbf{X}$. Thus the IEER time of $T_{i,j}(m)$ is no greater than $X_{i,j}$. By induction, the IEER time of every subtask instance in this schedule is no greater than $X_{i,j}$. □

**Corollary 1** *When Algorithm SA/DS terminates, it yields correct upper bounds on task EER times.*

### 5.5.5 Termination of Algorithm SA/DS

Unfortunately, Algorithm SA/DS does not always terminate, as shown by an example in Figure 5.19. In this example, there are 6 processors and 2 tasks. The period of each task is 3 time units, and the maximum execution time of each subtask is 1 time unit. On each processor the subtask whose corresponding box is drawn higher in the processor circle has a higher priority.

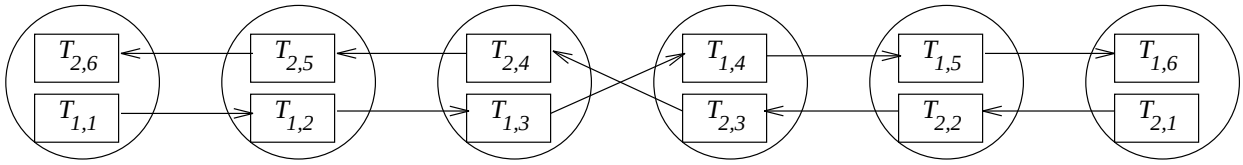



**Figure 5.19**: An Example to Show that Algorithm SA/DS May Never Terminate

We now show that if we apply Algorithm SA/DS to this system, the algorithm will never terminate. To do so, we follow the iteration steps in Algorithm SA/DS. Specifically, we start with $T_{1,1}$, $T_{1,2}$ and $T_{1,3}$.

1. Initially, $V_{1,1}$, $V_{1,2}$ and $V_{1,3}$ are equal to 1, 2, and 3, respectively.

2. After the first iteration, $V_{1,1}$, $V_{1,2}$ and $V_{1,3}$ will increase by at least 1 time unit due to the presence of $T_{2,4}$, $T_{2,5}$ and $T_{2,6}$.

3. After at most 5 iterations, $V_{1,3}$, $V_{1,4}$ and $V_{1,5}$ will increase by at least 3 time units compared with their initial values. This is because the increases of $V_{1,1}$, $V_{1,2}$ and $V_{1,3}$ in Step 2 are propagated to the bounds of later subtasks in the subtask chain. (See Step 3(b) in Algorithm IEERT.)

4. Due to the fact that $V_{1,3}$, $V_{1,4}$ and $V_{1,5}$ have values 3 units larger than their respective initial values, $V_{2,1}$, $V_{2,2}$ and $V_{2,3}$ will increase by at least 1 time unit compared with their initial value in the next iteration step. (See Step 3(a) in Algorithm IEERT.)

5. Similar to Step 3, we see that, at most 5 iterations after step 4, $V_{2,3}$, $V_{2,4}$, and $V_{2,5}$ will increase by at least 3 time units compared with their initial values, due to the propagation of increases of $V_{2,1}$, $V_{2,2}$ and $V_{2,3}$. In the next iteration, $V_{1,1}$, $V_{1,2}$ and $V_{1,3}$ will be further increased by at least 1 time unit. This essentially put us back to the similar situation as when we started from step 2.

From the above description, we can see that the iteration in Algorithm SA/DS essentially forms a positive feedback loop. As a consequence, the bounds on subtask IEER times never converge. In practice, we are not interested in a bound which can be arbitrarily large. In particular, when tasks have relative deadlines, we are not interested in the bounds anymore once we know that the bounds are larger than the deadlines. In this case, we can set up an additional termination condition for Algorithm SA/DS so that it terminates either when a solution is obtained or when the bounds exceed the relative deadlines.

### 5.5.6 Relation with Previous Work

Tindell et al. proposed a solution in [19] for a very similar problem. In their approach, the irregular release times are called *release jitters*, and the extra interference to a target subtask caused by release jitters is counted in their extended busy period analysis method [7, 8]. The release jitters are propagated along the subtask chains. The response time of each subtask is thus bounded and the bound on the EER time of each task is simply the sum of the bounds on the response times of all its subtasks.

However, although the release jitters of the interfering subtasks (i.e., subtasks that are in $H_{i,j}$) are correctly taken into account in their approach, release jitters of the target subtask are not taken into account. Tindell et al. assumed that the target subtask is periodic [8]. This is only true for the first subtask in each end-to-end task. When a target subtask ($T_{i,j}$) is not the first subtask, $T_{i,j}$ itself has release jitter. In an extreme case, several instances of $T_{i,j}$ may be released closely together in time, and the worst-case response time of $T_{i,j}$ is longer than the one when $T_{i,j}$ is periodic, which is obtained by the method in [8]. Since the approach proposed in [19] uses the

method in [8] to obtain an upper bound on the response time of a subtask, we expect that in general Tindell's approach will not yield correct upper bounds on the end-to-end response times of tasks.

## 5.6 Summary

In a system using a synchronization protocol with execution control, (e.g., one of the PM, MPM, SS, and RG protocols), the IEER time of subtask $T_{i,j}$ can be bounded by the sum of the bounds on the response times of $T_{i,j}$ and its predecessors. An induction over the second index of $T_{i,j}$ can show that the bound on IEER time of $T_{i,j}$ yielded by protocols with execution control is smaller than or equal to the bound yielded by the DS protocol, which does not have execution control. This implies that the bound on the EER time of a task yielded by any of the protocols with execution control is also smaller than or equal to the bound yielded by the DS protocol. Smaller bounds on the EER times imply a better schedulability of the system. This fact motivated us to devise protocols with execution control.

For all protocols with execution control, Algorithm SA/PM can be applied to obtain the bounds on the EER times of tasks. Additionally, if the relative end-to-end deadlines are no longer than the periods, Algorithm SA/IPM, which is an improvement of Algorithm SA/PM, can be applied to obtained tighter bounds for systems using the PM and MPM protocols. Algorithm SA/DS bounds the EER times of tasks in a system using the DS protocol.

# Chapter 6

# Priority Assignment

In the previous two chapters, we assumed that we are given priorities for all subtasks. In this chapter we look into the priority assignment problem. We first show that the priority assignment problem is in fact a family of sub-problems, and unfortunately these problems are all NP-hard. Several heuristic methods are described, including some existing ones.

## 6.1 Priority Assignment Problem

We say that a priority assignment is *feasible* if the resultant system is schedulable. Loosely speaking, the priority assignment problem is to find a feasible priority assignment for a given system. Because the schedulability of an end-to-end system also depends on the synchronization protocol used in the system, a priority assignment may be feasible for a system using the PM protocol but not for the same system using the DS protocol. Consequently the priority assignment problem branches into several sub-problems corresponding to all existing synchronization protocols. In other words, each sub-problem corresponding to a specific synchronization protocol is to find a feasible priority assignment for a given system using the particular synchronization protocol.

To complicate the matter further, an end-to-end system may be indeed schedulable with a given priority assignment but cannot be verified schedulable by any existing schedulability analysis algorithm, because schedulability analysis algorithms are in general not optimal. In practice, if a system cannot be verified schedulable by any existing schedulability analysis algorithm, we really do not have any other effective means to determine whether it is schedulable. (Exhaustive simulation is in general not feasible for all but the smallest systems because the solution space is

usually too huge to search.) For this reason, we are only interested in finding a feasible priority assignment whose feasibility can be verified by existing schedulability analysis algorithms. In the rest of this chapter, when we say a feasible priority assignment, we specifically mean an assignment whose feasibility can be verified by some existing schedulability analysis algorithm.

In summary, the priority assignment problem is a family of subproblems, where each subproblem is: Given an end-to-end system using a specific synchronization protocol, find a priority assignment which can be verified feasible by an existing schedulability analysis algorithm. Since there exist several synchronization protocols and at least one schedulability analysis algorithm for each protocol, there are many such subproblems. Unfortunately, Theorem 3 states that all these subproblems for different synchronization protocols and schedulability analysis algorithms proposed in this thesis are NP-hard.

**Theorem 3** *Every subproblem of the priority assignment problem for the case of any synchronization protocol and schedulability analysis algorithm proposed in this thesis is NP-hard.*

Proof :

We prove this theorem by reducing a SET-PARTITION problem [54] to a priority assignment problem.[1] The SET-PARTITION problem can be stated as follows: for a given set $\{S_i\}$ of numbers, partition them into two subsets such that sums of the numbers in both subsets are equal. We first transform a SET-PARTITION problem to a priority assignment problem by a polynomial algorithm. We then illustrate that a SET-PARTITION problem has a solution if and only if the corresponding priority assignment subproblem has a solution as well.

Given a set $\{S_i\}$ of $n$ numbers, we construct an end-to-end system with $n$ processors. There are two identical tasks in the system, $T_1$ and $T_2$. Each task has $n$ subtasks. The $j$th subtask in each task executes on processor $P_j$ and has a maximum execution time $S_i$. Let $S = \sum S_i$. The periods of both tasks are $4S$, and relative deadlines of both tasks are $1.5S$.

Suppose that there exists a solution for the SET-PARTITION problem. Let $\mathbf{A}$ be one of the subset of numbers. We have $\sum_{S_j \in \mathbf{A}} S_i = S/2$. On each processor $P_i$, if $S_i \in \mathbf{A}$, then we assign $T_{1,i}$ a higher priority than $T_{2,i}$. Otherwise, we assign $T_{1,i}$ a lower priority than $T_{2,i}$. It is easy to verify that according to every synchronization protocol and applicable schedulability analysis algorithm proposed in this thesis, we obtain the same upper bound on the EER time of $T_1$, which

[1] This proof was inspired by a similar proof given by Bettati in [13].

is $S + \sum_{S_i \in \{S_i\} - \mathbf{A}} S_i$, and the same upper bound on the EER time of $T_2$, which is $S + \sum_{S_i \in \mathbf{A}} S_i$. Given that $\sum_{S_j \in \mathbf{A}} S_i = S/2$, both bounds are equal $1.5S$, and both tasks will be schedulable. The priority assignment is thus a feasible assignment.

On the other hand, suppose that we obtain a feasible priority assignment with respect to a synchronization protocol and a specific schedulability analysis algorithm. On each processor, the two subtasks cannot have the same priority because, otherwise, the sum of the bounds on the EER times of both tasks would exceed $3S$, implying that they cannot both be guaranteed schedulable. We let $S_i$ belong to set $\mathbf{A}$ if and only if subtask $T_{2,i}$ has a higher priority than $T_{1,i}$ on processor $T_{1,i}$. Similar to the above case, we have $\sum_{S_j \in \mathbf{A}} S_i = S/2$. Thus we obtain a solution for the SET-PARTITION problem.

We see that there exists a solution for a SET-PARTITION problem if and only if there also exists a solution for a corresponding priority assignment subproblem. Since the transformation between these two problems is polynomial time, there exists a polynomial solution for each priority assignment subproblem if and only there exists a polynomial solution for the SET-PARTITION problem. Since the SET-PARTITION problem is proven NP-hard, each subproblem in the priority assignment problem is thus NP-hard. □

## 6.2 Heuristic Priority Assignment Methods

Since no efficient optimal algorithms have been found for any NP-hard problem and exhaustive solutions are impractical for all but extremely small systems, we focus now on heuristic solutions to the priority assignment problem. Heuristic priority assignment methods can be classified into three categories: rate-based methods, deadline-based methods, and optimization-based methods.

### Rate-Based Methods

The rate-monotonic (RM) priority assignment for single-processor systems can be straightforwardly applied to end-to-end systems. We assign every subtask a priority inversely proportional to the period of its parent task. Obviously, this assignment is not optimal. As a matter of fact, as we will see in Chapter 7, the performance of this assignment is in general inferior to other heuristic methods.

### Deadline-based methods

Kao and Garcia-Molina [11] studied the deadline-based priority assignment in the context of a soft real-time system. According to the deadline-based method, we assign a relative deadline to each subtask and then assign to a subtask a priority inversely proportional to its relative deadline. A shorter relative deadline implies a higher priority. We compute the relative deadline of each subtask only for the purpose of assigning priority. It is all right for an instance of a subtask to miss the computed relative deadline, as long as the end-to-end deadline of its parent task is met. Difference priority assignment methods compute the relative deadlines in different ways. Below are several examples.

**Global-deadline-monotonic (GDM) assignment :** The global deadline of a subtask $T_{i,j}$ in task $T_i$, denoted by $GD_{i,j}$, is the relative deadline of $T_i$, i.e., $GD_{i,j} = D_i$.

**Effective-deadline-monotonic (EDM) assignment:** The effective deadline ($ED_{i,j}$) of a subtask $T_{i,j}$ is defined as follows:

$$ED_{i,j} = D_i - \sum_{k=j+1}^{n_i} \tau_{i,k}$$

In other words, the effective deadline of a subtask $T_{i,j}$ is equal to its global deadline minus the total execution time of the successors of $T_{i,j}$.

**Proportional-deadline-monotonic (PDM) assignment:** The proportional deadline ($PD_{i,j}$) of a subtask $T_{i,j}$ is obtained by dividing the relative deadline of its parent task $T_i$ among its subtasks proportionally to their execution times. In other words,

$$PD_{i,j} = \frac{\tau_{i,j}}{\tau_i} D_i$$

where $\tau_i$ is the total maximum execution time of task $T_i$.

**Normalized-proportional-deadline-monotonic (NPDM) assignment :** This assignment is similar to the PDM assignment except that the processor utilization is taken into account. In the PDM assignment, if two sibling subtasks have equal execution times, they have equal proportional deadlines. In the NPDM assignment, however, if a subtask executes on a more heavily utilized processor than the other, it is assigned a relatively longer deadline. We use $\mu_1, \mu_2, \ldots, \mu_{n_i}$ to denote the utilization

| $T_{i,j}$ | host | $p_{i,j}$ | $\tau_{i,j}$ | $GD_{i,j}$ | $ED_{i,j}$ | $PD_{i,j}$ | $NPD_{i,j}$ |
|---|---|---|---|---|---|---|---|
| $T_{1,1}$ | $P_1$ | 80 | 30 | 80 | 80 | 80 | 80 |
| $T_{2,1}$ | $P_1$ | 100 | 50 | 100 | 75 | 66.7 | 82.4 |
| $T_{2,2}$ | $P_2$ | 100 | 25 | 100 | 100 | 33.3 | 17.6 |
| $T_{3,1}$ | $P_2$ | 40 | 5 | 40 | 40 | 40 | 40 |

**Table 6.1**: Subtasks with Calculated Relative Deadlines

factors of processors where $T_{i,1}, T_{i,2}, \ldots, T_{i,n_i}$ execute. Let $\Lambda = \sum_{k=1}^{n_i} \tau_{i,k}\mu_k$. The normalized proportional relative deadline $NPD_{i,j}$ of $T_{i,j}$ is given by

$$NPD_{i,j} = \frac{\tau_{i,j}\mu_j}{\sum_{k=1}^{n_i} \tau_{i,k}\mu_k} D_i$$

Based on the normalized proportional deadlines, priorities are assigned to subtasks in a deadline-monotonic manner.

As an example, the left half of Table 6.1 shows the subtask parameters of a simple end-to-end system. The relative deadline of each task is equal to its period. The right half of the table shows the global deadline, effective deadline, proportional deadline and normalized proportional deadline of each subtask. For this particular system, we see that each deadline-based priority assignment method leads to a different priority assignment. In this example, the EDM assignment and the PDM assignment are feasible assignments for all synchronization protocols and schedulability analysis algorithms while the other two assignments are not feasible.

### Optimization-Based Methods

In recent years, we have seen several research efforts on the priority assignment problem in an end-to-end system. Tindell et al. [9] used the simulated annealing technique to find a feasible priority assignment and a feasible load partition at the same time. An improved algorithm, in terms of efficiency and the likelihood of finding a feasible solution, was proposed by Garcia and Harbour [10]. Their algorithm is called Algorithm HOPA. Both approaches start with a random priority assignment or a priority assignment according to a basic assignment method, and then continually adjust the assignment until a feasible solution is found. A certain criterion is used in directing the adjustment of priorities. A good criterion more likely leads to a feasible

solution than a bad one. The performance of optimization-based methods is in general better than rate-based or deadline-based methods although they run at higher costs.

## 6.3 Local Assignment Methods vs. Global Assignment Methods

All heuristic priority assignment methods can be classified along another dimension: *local assignment methods* vs. *global assignment methods*. A local assignment method assigns a priority to a subtask based solely on the information of the subtask and its parent task, including the execution times and the period. A global assignment method, on the other hand, requires information of other tasks in the system. Clearly, the RM, GDM, EDM and PDM assignments are local assignment methods, and the NPDM assignment and optimization-based methods are global assignment methods.

One advantage of local assignment methods is that the priorities of existing subtasks need not change when a task is added or removed. In other words, they are more adaptive to workload change than global assignment methods. Another advantage of local assignment methods come from the simplicity in computing priorities. Because the priority of a subtask is only dependent on the parameters of the subtask and its parent task and the computation is simple, it is possible to determine the priorities of subtasks by an on-line algorithm, i.e., when a task is added to the system. By contrast, for global assignment methods, it is more appropriate to compute the priorities off-line due to the complexity in the computation and dependency on the global system information.

The disadvantage of local assignment methods is their performance. In general, we expect that global assignment methods will lead to better system schedulability because they take the whole system into consideration when assigning priorities.

## 6.4 Summary

We have shown that in the context of the end-to-end scheduling, the priority assignment problem is NP-hard. Heuristic methods are thus developed. These methods can be classified into three categories: rate-based method, deadline-based methods and optimization-based methods. The performance of deadline-based methods were studied by Kao and Garcia-Molina [11] in the context of soft real-time scheduling. In Chapter 7, we will compare their performance in the

context of end-to-end scheduling. In [10], Garcia and Harbour provided some insight into the performance of optimization-based methods.

# Chapter 7

# Performance of End-to-End Scheduling Algorithms

From the previous three chapters, we see that end-to-end scheduling is a family of scheduling algorithms. We described four deadline-based heuristic methods to assign priorities to subtasks. In fact we can devise a fifth "meta-method" by simply trying all four methods and looking for one which makes a system schedulable. We also have five choices to synchronize the execution of subtasks on different processors. They require different system support and yield different performance. For each synchronization protocol, we developed a schedulability analysis algorithm. In the case of the PM and MPM protocol, we have an improved but restricted version of the schedulability analysis algorithm, in addition to the basic one.

Hereafter, when we say an end-to-end scheduling algorithm, we mean a combination of a priority assignment method, a synchronization protocol and the schedulability analysis algorithm for the synchronization protocol. For example, if a system is scheduled according to an end-to-end scheduling algorithm that consists of the PDM priority assignment method, the PM synchronization protocol and Algorithm SA/IPM, then we assign the subtask priorities according to the PDM method, synchronize the execution of subtasks according to the PM protocol, and analyze the schedulability according to Algorithm SA/IPM. Hence, there are many different combinations, i.e., many end-to-end scheduling algorithms proposed in this thesis.

It is necessary for us to understand the strength and weakness of these scheduling algorithms. Some of the performance issues have already been discussed in previous chapters. These issues

are the complexity and run-time overhead of synchronization protocols, qualitative comparison of different synchronization protocols, qualitative comparison of Algorithm SA/DS and SA/PM, etc.. In this chapter, we focus on quantitative comparisons of these protocols and algorithms. Specifically, we performed four experiments for this purpose.

**Experiment I :** Comparison of the deadline-based priority assignment methods.

**Experiment II :** Comparison of Algorithm SA/DS and Algorithm SA/PM.

**Experiment III :** Comparison of Algorithm SA/PM and Algorithm SA/IPM.

**Experiment IV :** Comparison of average task EER times yielded by different synchronization protocols.

We describe below the work generation, performance criteria and the experiment results for each of these experiments.

## 7.1 Experiment I - Deadline-Based Priority Assignment Methods

In Chapter 6, we gave an example of an end-to-end system, whose parameters are listed in Table 6.1. For this example, we see that the EDM method and the PDM method yield feasible solutions while others do not. In general, we can always devise a pathological example for which one particular assignment yields a feasible solution, and others do not. We want to understand how well they perform relative to each other in general.

In practice, a system designer should try all deadline-based methods till a feasible solution is found because these methods are quick to apply. This is the idea behind the fifth method we introduce in this simulation, the *meta-method*. We will give a precise definition of the meta-method later.

### 7.1.1 Workload Generation

Through preliminary testing, we found that the relative performance of different priority assignment methods is stable and insensitive to changes of many system parameters. Consequently, when generating a synthetic system, we fixed these system parameters and randomized others

to obtain representative samples. Specifically, the parameters that are kept invariant are the number of processors and the number of tasks. In our experiment, each system has 4 processors and 12 tasks. We first generated the period for each task, which is exponentially distributed between 100 and 10000. The number of subtasks in each task is randomly distributed between 1 and 8. Subtasks are randomly assigned to processors as long as two consecutive subtasks in a same task are not assigned to the same processor. The processor utilization is randomly distributed between 0.5 and 0.8. On each processor, all the subtasks randomly divide the processor utilization. Specifically, to determine the utilizations of individual subtasks on a processor, we generated a random number between 0.001 and 1 for each subtask and called it the *utilization factor* of the subtask. We then divided the utilization factor of each subtask by the sum of the utilization factors of all subtasks on the same processor and obtained the *normalized utilization factor* of the subtask. The utilization of a subtask is the product of the processor utilization and its normalized utilization factor. The execution time of each subtask is simply its utilization multiplied by the period of its parent task.

We focused our attention on systems in which the relative deadline of every task is a constant factor of its period, and the constant factors of all tasks are the same. For such a system, the priority assignment yielded by all deadline-based methods is independent of the actual value of the constant factor. Therefore we do not need to specify the exact value of this constant factor. Moreover, the RM method and the GDM method produce the same priority assignment for any given system. Hence, the performance of the GDM method presented in the rest of this section represents the performance of the RM method as well.

### 7.1.2 Performance Criteria

When the relative deadlines of tasks are a constant factor of their respective periods, an important value to indicate the schedulability of a task is the ratio between the upper bound on its EER time and its period. We call this ratio the *schedulability index* of the task. The *worst-case schedulability index* of a system is equal to the maximum of the schedulability indices of all tasks, and the *average schedulability index* is the average of the schedulability indices of all tasks.

The worst-case schedulability index of a system indicates the most "stringent" timing constraint that the system can satisfy. For example, a worst-case schedulability index of 1.5 indicates that the system is schedulable as long as the relative deadline for each task in the system is no

shorter than 1.5 times of its period. Suppose that when we change the priority assignment for this system, we reduce its worst-case schedulability index from 1.5 to 1. The system can then be schedulable even if we set the relative deadline for every task to be equal to its period. By comparing the worst-case schedulability indices yielded by different priority assignment methods for each system generated in our experiment, we compare the relative performance of these methods. A smaller index indicates a better performance.

On the other hand, the average schedulability index of a system is an indication of the schedulability of tasks other than the most "troubled" one in the system. A system having a relatively high worst-case schedulability index but with a relatively low average schedulability index indicates that a few tasks in the system are causing the "trouble" while most of the tasks are fine.

In the simulation, we introduce a new priority assignment method, the *meta-method*. For a given system, we assign subtask priorities according to all the four deadline-based priority assignment methods and perform schedulability analysis. In the end, we choose the assignment that yields the smallest worst-case schedulability index of the system. Obviously, the meta-method should always be used to achieve better schedulability when the priorities are assigned off-line. In the experiment, we included the simulation results for the meta-method so that we could measure how well a particular assignment method performs by comparing its performance with the performance of the meta-method.

### 7.1.3 Simulation Results

In the simulation, we used a synchronization protocol with execution control and applied Algorithm SA/PM to analyze the schedulability of the system. We chose Algorithm SA/PM because it has two merits that other schedulability analysis algorithms do not have: Algorithm SA/PM always terminates while Algorithm SA/DS may not, and Algorithm SA/IPM requires the relative deadlines of tasks to be no longer than their periods while Algorithm SA/PM does not have this restriction. These attributes make Algorithm SA/PM an ideal choice for Experiment I. We found that the results obtained according to other algorithms are similar to the ones we present here.

We generated 1000 systems according to the method described earlier. Table 7.1 lists the average of the worst-case schedulability indices and the average schedulability indices over these

systems for each of the four deadline-based priority assignment methods and the meta-method. In terms of the worst-case schedulability index, the PDM and NPDM methods are much better than the other two deadline-based methods. In fact, for every system tested in the experiment, the worst-case schedulability indices yielded by the PDM and NPDM methods are always smaller than those yielded by the GDM and EDM methods. The difference between the PDM and NPDM methods are negligibly small. As a consequence, the performance of the meta-method is entirely determined by these two methods.

| | GDM | EDM | PDM | NPDM | meta-method |
|---|---|---|---|---|---|
| worst-case index | 2.495 | 2.005 | 1.514 | 1.51 | 1.494 |
| average index | 0.9793 | 0.8762 | 0.9437 | 0.9478 | 0.9432 |

**Table 7.1**: Schedulability Indices Yielded by Different Priority Assignment Methods

We notice that the EDM method yields noticeably smaller average schedulability indices than other methods. A closer look reveals that in a system where the EDM method is used, tasks with shorter periods are strongly favored and thus have much smaller schedulability indices. As a consequence, the average schedulability index of the system is small as well. The same situation occurs when the subtask priorities are assigned according to the GDM method. However, according to the GDM method, sibling subtasks have the same priorities. The bounds on the response times of the subtasks are negatively affected when sibling subtasks happen to be on the same processor. The net effect is that the bound on the end-to-end response time of a task is much longer than the one produced by the EDM method. As a result, the average schedulability indices yielded by the GDM method are no smaller than other methods.

## 7.2 Experiment II - Algorithm SA/DS vs. Algorithm SA/PM

In Chapter 5, we demonstrated that the bound yielded by Algorithm DS is always greater than or equal to the bound yielded by Algorithm SA/PM for the same task in a system. The purpose of this experiment is to determine (1) under what conditions the performance of Algorithm SA/DS is close to the performance to Algorithm SA/PM; (2) under what conditions Algorithm SA/DS has much worse performance than Algorithm SA/PM; and (3) under what conditions SA/DS is likely not to terminate.

### 7.2.1 Workload Generation

The difference in performance between Algorithm SA/DS and SA/PM depends mostly on two system parameters: the processor utilization and the number of subtasks in each task. To simplify the analysis, we let every processor have the same utilization and every task have the same number of subtasks. Each *configuration*, denoted by a pair $(\kappa, \mu)$, is a unique combination of the number of subtasks in each task and the processor utilization. For example, configuration (5,60) represents systems where each task has 5 subtasks and the utilization of each processor is 60%. In the configurations evaluated, the number of subtasks in each task ranges from 1 to 8 and the utilization of each processor is 50%, 60%, 70%, 80%, or 90%. Consequently, we have a total of 28 configurations.

For each configuration $(\kappa, \mu)$, we generated 1000 systems in a way similar to Experiment I. The only difference is that, instead of a random number of subtasks in each task and a random processor utilization on each processor, each task has $\kappa$ subtasks and each processor has a utilization of $\mu$ when the system is generated according to the configuration $(\kappa, \mu)$.

### 7.2.2 Performance Criteria

As we saw in Chapter 5, Algorithm SA/DS may not terminate for certain systems. Given a finite amount of time, however, we cannot tell if Algorithm SA/DS is failing to terminate or if it will terminate with very large upper bounds for a system. For practical purposes, we say that Algorithm SA/DS fails for a system if the worst-case schedulability index is greater than 100. Such a situation can be detected when Algorithm SA/DS fails to find an upper bound on the EER time of a task which is no greater than 100 times the period of the task. We measure the likelihood for Algorithm SA/DS to fail by the *failure rate*. The failure rate of a specific configuration is the total number of systems for which Algorithm SA/DS fails divided by the total number of systems generated according to the configuration. The failure rate is an approximate indication of the likelihood that Algorithm SA/DS will not terminate.

For systems where Algorithm SA/DS can obtain upper bounds on task EER times, we use the *worst-case schedulability index ratio*, or simply *index ratio*, to measure the relative performance of Algorithm SA/DS and SA/PM. Specifically, the index ratio of a system is the worst-case schedulability index of the system computed according to Algorithm SA/DS divided by the

worst-case schedulability index of the system computed according to Algorithm SA/PM. The index ratio is always greater than or equal to 1. We define the index ratio of a configuration to be the average index ratio of all systems generated according to this configuration.

### 7.2.3 Simulation Results

Table 7.2 lists the failure rates for all the configurations we simulated. Each entry in the first column gives the processor utilization for the configurations in the corresponding row. Each entry in the first row gives the number of subtasks in a task. We notice that the failure rates are zeros for most configurations, indicating that Algorithm SA/DS is very likely to terminate for a system generated according to these configurations. However, for configurations at the bottom-right of the table, the failure rates are greater than 0.1, indicating that there is a good chance for Algorithm SA/DS to fail for systems generated according to one of these configurations. Configuration (8, 80%) is an extreme case, where Algorithm SA/DS failed for 90% of the time. Overall, Algorithm SA/DS rarely fails for systems with low processor utilizations or short subtask chains, while it can fail for systems with high processor utilizations and long subtask chains.

| | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 50% | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 60% | 0 | 0 | 0 | 0 | 0 | 0.04 | 0.028 |
| 70% | 0 | 0 | 0 | 0 | 0.019 | 0.121 | 0.369 |
| 80% | 0 | 0 | 0.005 | 0.074 | 0.31 | 0.683 | 0.9 |

**Table 7.2**: Failure Rates for Algorithm SA/DS

Table 7.3 lists the index ratios of the systems for which Algorithm SA/DS does not fail. We observe that when the length of subtask chains approaches 7 and 8 and the processor utilization approaches 70% and 80%, the index ratio becomes larger than 2, making the DS protocol much less desirable for systems having these characteristics. In the case of configuration (8, 80%), the index ratio is 17.05. It is very unlikely for a system with this configuration to be schedulable according to the DS protocol.

| | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 50% | 1.014 | 1.032 | 1.068 | 1.149 | 1.26 | 1.49 | 1.848 |
| 60% | 1.034 | 1.183 | 1.327 | 1.575 | 1.933 | 2.613 | 3.768 |
| 70% | 1.129 | 1.364 | 1.692 | 2.364 | 3.652 | 6.425 | 9.174 |
| 80% | 1.211 | 1.692 | 2.55 | 5.196 | 8.403 | 12.67 | 17.05 |

**Table 7.3**: Worst-Case Schedulability Index Ratios of Algorithm SA/DS over Algorithm SA/PM

## 7.3 Experiment III - Algorithm SA/PM vs. Algorithm SA/IPM

Algorithm SA/IPM is an improved version of Algorithm SA/PM. It takes advantage of recurrent tasks but is applicable only when tasks have relative end-to-end deadlines shorter than their respective periods. We studied the performance difference between these two algorithms and describe the results in this section.

### 7.3.1 Workload Generation

Different from the previous experiments, we need to generate workloads with recurrent end-to-end tasks in this experiment. We used the *recurrence degree of a task* as a control parameter, where the recurrence degree of a task is the maximum number of its subtasks that execute on a same processor. If a task $T_i$ has a recurrence degree of 3, it means that $T_i$ has 3 subtasks executing on the same processor. We let the system still have 4 processors and 12 tasks. However, we assigned 3 tasks to each processor, called their *host processor*. If the recurrence degree is $n$, we generated $2n$ subtasks for each task $T_i$. We let all odd-numbered subtasks execute on the host processor of $T_i$ and the even-numbered subtasks execute randomly on the other three processors. By doing this, we can guarantee that every task in the system has the same recurrence degree.

The other control parameter is the processor utilization. Again, every processor was given the same utilization. Similar to the previous experiments, we used a pair of control parameters to specify a particular configuration. If an end-to-end system is generated according to configuration $(\mu, \alpha)$, every processor has a utilization of $\mu$ and every task has a recurrence degree of $\alpha$. In the experiment, we vary the processor utilization from 0.3 to 0.8 at 0.1 increments. (We will soon see why we included low processor utilizations, such as 0.3, in this experiment.) The recurrence degree is between 2 and 8. We thus have 42 configurations.

The other system parameters were generated in a similar way as in Experiment I, i.e., the task periods are exponentially distributed between 100 and 10000 and subtasks on the same processor randomly divide the processor utilization. Since Algorithm SA/IPM requires that tasks have relative deadlines shorter than or equal to their periods, we let the relative deadline of every task be equal to its period in this experiment.

### 7.3.2 Performance Criteria

We generated 1000 systems according to each configuration. The *success rate* is the percentage of systems generated according to a configuration that was found schedulable according to Algorithm SA/IPM. A low success rate of a configuration indicates that Algorithm SA/IPM is not suitable for systems with that configuration.

For systems that are schedulable according to Algorithm SA/IPM, we also applied Algorithm SA/PM to analyze their schedulability. Similar to the previous experiments, we used the worst-case schedulability index ratio to measure the relative performance of these two algorithms: The index ratio of a system is the worst-case schedulability index computed by Algorithm SA/PM divided by the one computed by Algorithm SA/IPM. Clearly, the index ratio is always greater than 1. Similarly, the worst-case schedulability index ratio of a configuration is the average of the index ratios of all systems that are generated according to the configuration and are schedulable according to Algorithm SA/IPM. In addition to the worst-case schedulability index ratios, we also use average schedulability index ratios to measure the improvement achieved by Algorithm SA/IPM for the overall system. The definition of the average schedulability index ratio is similar to the one of the worst-case schedulability index ratio.

### 7.3.3 Simulation Results

Table 7.4 lists the success rates of Algorithm SA/IPM. Each entry in the first row gives the recurrence degree while each entry in the first column gives the processor utilization. We notice that very few systems with a recurrence degree above 4 can be schedulable according to Algorithm SA/IPM, unless the processor utilization drops to 0.3. This indicates that, for schedulability concerns, the relative deadline of a task being no greater than the period is a stringent requirement for systems with a non-trivial recurrence degree. This conclusion is not surprising. To see why, let us suppose that every task has its relative deadline equal to its period and has a recurrence

| | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 0.3 | 100 | 100 | 100 | 96.5 | 67.9 | 21.5 | 2.4 |
| 0.4 | 100 | 98.9 | 69.7 | 10 | 0 | 0 | 0 |
| 0.5 | 98.7 | 49.3 | 0.8 | 0 | 0 | 0 | 0 |
| 0.6 | 63.7 | 0.4 | 0 | 0 | 0 | 0 | 0 |
| 0.7 | 1.3 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0.8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**Table 7.4**: Success Rates for Algorithm SA/IPM

degree of $\alpha$. Every subtask in a task should have a response time roughly within a factor of $1/2\alpha$ of the period of the task in order for the task to be schedulable. This requirement is rather demanding and brings down the usable processor utilization in order satisfy it. Lehoczky found similar phenomena in [6]. For example, in his study, if tasks have deadlines equal to 1/4 of their periods, the schedulable utilization bound drops to 25%.

For configurations with nonzero success rates, we list the worst-case schedulability index ratios in Table 7.5 and the average schedulability index ratios in Table 7.6. Algorithm SA/IPM is more effective for configurations with large recurrence degrees. Nevertheless, the overall performance difference between Algorithm SA/IPM and Algorithm SA/PM is small. Most of the time, improvement is within 1% in terms of scheduling index. When the worst-case schedulability indices approach one, i.e., when systems becomes difficult to schedule, such a small improvement may result in a more obvious increase in the success rate. We will see such an effect of this improvement in the next chapter.

## 7.4 Experiment IV - Average Task EER Times

The purpose of Experiment IV is to study the average task EER times achievable by different synchronization protocols. In Chapter 4, we introduced five synchronization protocols, the Direct Synchronization (DS), Phase Modification (PM), Modified Phase Modification (MPM), Sporadic Server (SS), and Release Guard (RG) protocols. As we have discussed there, the PM and MPM protocols yield the same schedule under the ideal situation, which is the case in our simulation.

| | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 0.3 | 1.000 | 1.001 | 1.005 | 1.011 | 1.019 | 1.026 | 1.042 |
| 0.4 | 1.001 | 1.001 | 1.005 | 1.014 | na | na | na |
| 0.5 | 1.001 | 1.002 | 1.008 | na | na | na | na |
| 0.6 | 1.001 | 1 | na | na | na | na | na |
| 0.7 | 1.001 | na | na | na | na | na | na |
| 0.8 | na | na | na | na | na | na | na |

**Table 7.5**: Worst-Case Schedulability Index Ratios of Algorithm SA/PM over Algorithm SA/IPM

| | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 0.3 | 1.001 | 1.003 | 1.009 | 1.017 | 1.027 | 1.036 | 1.047 |
| 0.4 | 1.001 | 1.003 | 1.009 | 1.017 | na | na | na |
| 0.5 | 1.001 | 1.003 | 1.008 | na | na | na | na |
| 0.6 | 1.002 | 1.003 | na | na | na | na | na |
| 0.7 | 1.002 | na | na | na | na | na | na |
| 0.8 | na | na | na | na | na | na | na |

**Table 7.6**: Average Schedulability Index Ratios of Algorithm SA/PM over Algorithm SA/IPM

Thus we only need to compare the performance of the PM protocol with others. We also did not evaluate the SS protocol for the following reasons.

1. The original sporadic server algorithm proposed in [52] contains a mistake. In Appendix A, we point out the mistake and give the corrections. We also illustrate that the idea of a sporadic server encompasses a family of algorithms. However, it is not clear which implementation is the most efficient one. Choosing a less efficient implementation of the sporadic server algorithm in this experiment may result in less representative figures and causes unfair comparison against the SS protocol.

2. According to the SS protocol, a task instance may have an unbounded number of context switches due to budget consumption and replenishment. This attribute makes the SS protocol undesirable for a hard real-time system in practice.

3. In our simulation, the task execution time is not variable, and the run-time overhead caused by each protocol is not counted either. Thus the RG and SS protocol will yield the same schedule for any system. It suffices to consider only the performance of the RG protocol.

As a result, in this experiment, we compared the average task EER times yielded by the DS, PM, and RG protocols.

### 7.4.1 Workload Generation

Like Experiment II, we found that the relative performance primarily depends on the processor utilization and the number of subtasks in each task. Consequently, we generated the workload in the same way as in Experiment II. In other words, we also had 28 configurations and generated 1000 systems for each configuration. Additionally, for each task we generated a phase which is randomly distributed between 0 and the task period.

### 7.4.2 Performance Criterion

Here, we use the *PM/DS ratio* to measure the difference in performance between the PM and DS protocol. For each task, the PM/DS ratio is the average EER time of the task when the PM protocol is used over the average EER time of the task when DS protocol is used. For each system, the *PM/DS ratio* is the average PM/DS ratio of all tasks in the system, and for each

configuration the *PM/DS ratio* is the average PM/DS ratio of all systems generated according to the configuration. A higher PM/DS ratio ($> 1$) indicates that the PM protocol yields a longer average task EER time than the DS protocol. Similarly, we also use the *PM/RG ratios* and *RG/DS ratios* to compare the performance of the corresponding two protocols.

### 7.4.3 Simulation Results

For each system generated in the experiment, we first assigned priorities to subtasks according to the PDM method. In the previous sections, we saw that the performance of the PDM and NPDM methods are better than other methods. Since each processor has the same utilization in this experiment, the PDM method and the NPDM method are equivalent. We then simulated the actual run of the system according to the task periods, task phases, subtask execution times, the assigned priorities and each of three synchronization protocols. Each simulated run was terminated when every task in the system had at least 1000 instances released and completed. The average EER time of a task is the average of the EER times of all its instances completed during the simulated run. Based on the average EER times of tasks, we computed the PM/DS ratio, RG/DS ratio, and PM/RG ratio for each task, each system, and each configuration. The PM/DS, RG/DS ratios as functions of configurations are plotted in Figures 7.1 and 7.2. The 90% confidence intervals are negligibly small for all configurations.

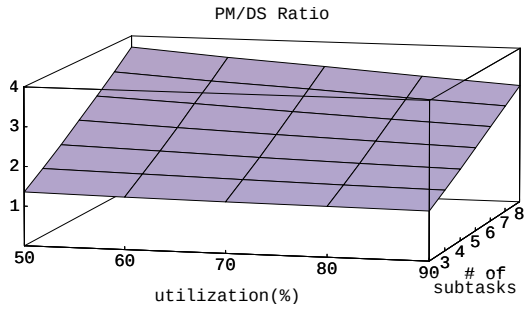



**Figure 7.1**: The PM/DS Ratios

As we expected, the PM/DS ratios are all greater than 1 in Figure 7.1, indicating that the average task EER times are shorter when the DS protocol is used than when the PM protocol is used. Specifically, given a fixed number of subtasks in each task, the PM/DS ratio goes down

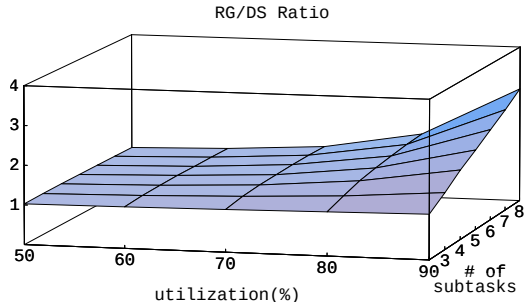



**Figure 7.2**: The RG/DS Ratios

when the utilization of each processor goes up. This is because with low processor utilizations there are many idle intervals in the schedules caused when the releases of subtasks are postponed according to the PM protocol. Therefore the average EER times are much larger than those yielded by the DS protocol with no such delays. The PM/DS ratio increases as the number of subtasks in each task increases. For configurations with 5 or more subtasks in each task, the values of the ratio are all greater than 2, indicating that for these kinds of configurations, the average EER times of tasks synchronized according to the PM protocol are more than twice than when tasks are synchronized according to the DS protocol. For configurations with 8 subtasks in each task, the ratio is around 3 or 4.

The RG/DS ratio varies mostly from 1 to 2 for all configurations, indicating that the performance of the RG protocol in terms of the average task EER time is close to the performance of the DS protocol. As shown in Figure 7.2 the RG/DS ratio increases when the processor utilization increases. When the processor utilization is high and the processors are busy almost all the time, the releases of subtasks according to the RG protocol become almost periodic. The delays in releasing subtasks caused by release guards can lead to longer average EER times. Overall, however, we see that the RG protocol performs much better than the PM protocol with respect to the average task EER times. This is further confirmed by the PM/RG ratio shown in Figure 7.3; the ratio is consistently higher than 1. For configurations with 6, 7 or 8 subtasks in each task, the PM/RG ratio even reaches 2 or 3.

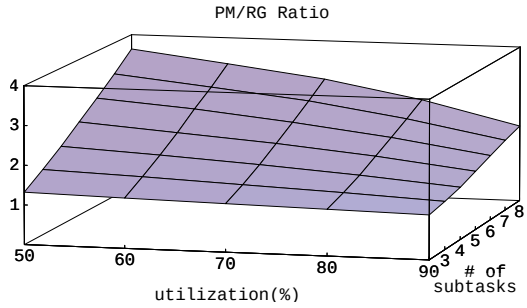



**Figure 7.3**: The PM/RG Ratios

## 7.5 Summary

In this chapter, we described four experiments and presented the experiment results. These experiments systematically compare the performance of all aspects of end-to-end scheduling algorithms proposed in this thesis. In Experiment I, we compared the performance of the four deadline-based priority assignment methods. According to the experiment result, the PDM and NPDM methods are superior to the other two methods. However, when priority assignment is done off-line or when run-time overhead is not a serious consideration, a system designer should try all four methods and choose the assignment which gives the smallest worst-case schedulability index, i.e., the meta-method.

Experiment II is a study on the performance of Algorithm SA/PM and SA/DS. Since Algorithm SA/PM is applicable to all synchronization protocols with execution control, in this experiment we essentially studied the relative performance in terms of system schedulability of the DS protocol and other protocols. The results show that the DS protocol has an acceptable performance when the processor utilization is low or tasks have short chains. Since the DS protocol has a low overhead and is simple to implement, it is an attractive choice for this kind of systems. However, when the processor utilization is high and tasks have long chains, the DS protocol yields poor performance and even fails to provide a solution when Algorithm SA/DS fails to terminate.

In Experiment III we saw that requiring tasks to have short relative deadlines (e.g., being shorter than their respective periods) is a stringent requirement for systems with a non-trivial recurrence degree. This requirement leads to low schedulable processor utilizations. When this

requirement is satisfied, the performance of Algorithm SA/IPM is slightly better than Algorithm SA/PM.

In Experiment IV, we studied the impact of different synchronization protocols to the average task EER times. As we expected, the DS protocol yields the shortest average task EER times, while the PM and MPM protocols yield the longest ones. The performance of the RG protocol is fairly close to the DS protocol in most cases.

In conclusion, the PDM and NPDM methods have better performance among deadline-based priority assignment methods. The RG protocol delivers satisfactory performance with respect to both the worst-case EER times and the average EER times of tasks. Under some circumstances, other protocols may be more suitable than the RG protocol. As an example, if a system has low processor utilizations and tasks have short subtask chains, the DS protocol can be a better choice because it is simpler and yields shorter average EER times of tasks. As another example, if every task has a relative deadline no longer than its period, we may choose the PM or MPM protocol because they allow us to use Algorithm SA/IPM for better system schedulability.

# Chapter 8

# End-to-End Scheduling Approach to the Resource Contention Problem

In the previous chapters, we presented an integrated end-to-end scheduling framework, which encompasses three problems: priority assignment, execution synchronization and schedulability analysis. Solutions to these problems form the building blocks of the framework. If an end-to-end system fits the system model described in Chapter 3, we can apply the framework to schedule such a system and analyze its schedulability. In this chapter, we investigate the application of this framework to the resource contention problem in a multi-processor real-time system. The system model assumed in the resource contention problem is formulated by Rajkumar et al.[21] and is different from the end-to-end system model that we assumed in the previous chapters. In Rajkumar's model, the multiprocessor system can be viewed as a collection of single-processor systems. Each task has a host (local) processor and has a local relative deadline. The complication arises from the fact that a task may request a resource which is on a different processor from its host processor. Scheduling and schedulability analysis of tasks in such a system become more complicated than the case of single-processor systems.

In this chapter, we first lay out the background on the resource contention problem, followed by a formal description of the system model used in this chapter. We then introduce the Multiprocessor Priority Ceiling Protocol (MPCP), the only known solution to the resource contention problem in a multi-processor environment. We will also point out a couple of mistakes in the original MPCP and propose a few straightforward improvements over the original MPCP. Sec-

tion 8.4 presents in detail the end-to-end scheduling approach to the same problem addressed by the MPCP. Finally, we conclude this chapter by comparing the performance of these two approaches.

## 8.1 Background

Semaphore-like operations are typically used to control the access to a shared resource by tasks in order to guarantee mutually-exclusive accesses to critical sections. Sha et al. [15] have shown that careless use of semaphore operations can cause *uncontrolled priority inversion*, which occurs when a high-priority task is blocked by some low-priority tasks for an unpredictable amount of time. On a single-processor system, we refer to the total length of time during which a task is blocked by lower-priority tasks due to resource contention as its *blocking time*. To ensure predictability, it is imperative to keep the blocking time of each task bounded from above [20]. Several effective solutions have been proposed for single-processor systems; two well-known examples are the *Priority Ceiling Protocol* (PCP) [15] and the *Stack Based Protocol* (SBP) [16]. By using either one of the protocols, the blocking time of each task is guaranteed to be no more than the duration of one critical section of some other tasks.

In multiprocessor or distributed systems, concurrency and distribution complicate the resource contention problem. A task $T_i$ can be blocked not only by local tasks on the same processor but also by remote tasks that need the same resources that $T_i$ needs as well. Accordingly, in a multiprocessor or distributed system the blocking time of a task includes the time when the task is blocked by both local lower-priority tasks and remote tasks. Rajkumar et al. [21] extended the PCP for single-processor systems to multiprocessor systems and provided a solution for this problem. The extended protocol is known as the *Multiprocessor Priority Ceiling Protocol* (MPCP). According to the MPCP, the blocking time of a task due to both local resource contention and global resource contention is bounded, and the schedulability of the system can be determined by applying a schedulability analysis method for single-processor systems [1, 5] on each individual processor.

## 8.2 System Model

In Rajkumar's system model, every resource and every task has a *host processor*. (In [21], the host processor of a global resource is called a synchronization processor of the global resource.) Later in this chapter, when we say a task $T_i$ (or a resource $R_i$ ) is on a processor $P_j$, we mean that $P_j$ is the host processor of task $T_i$ (or resource $R_i$). According to the MPCP, resources are classified into two categories, *local resources* and *global resources*. A local resource is needed only by tasks on its host processor. A global resource is needed by tasks on processors different from its host processor. Relative to a task, a global resource is a *remote global resource*, or simply a *remote resource*, if the task and the resource are on different processors. A global resource is a *local global resource* to a task if they are on the same host processor.

Figure 8.1 gives an illustrative example. This example is referred to as Example 1 later on in this chapter. In this example, there are two processors, $P_1$ and $P_2$, and two resources, $Printer$ (PR) and $Database$ (DB). The host processor of $PR$ is $P_1$; the host processor of $DB$ is $P_2$. $T_1$ and $T_2$ are on $P_1$. The host processor of $T_3$ and $T_4$ is $P_2$. The resource access relations are denoted by dotted arrows in the figure. Since $PR$ is accessed only by $T_1$ and $T_2$ which are on its host processor, $PR$ is a local resource. $DB$ is accessed by $T_1$ and $T_4$. Since $T_1$ is on a different processor from the host processor of $DB$, $DB$ is a global resource. Specifically, $DB$ is a remote global resource to $T_1$ and a local global resource to $T_4$.

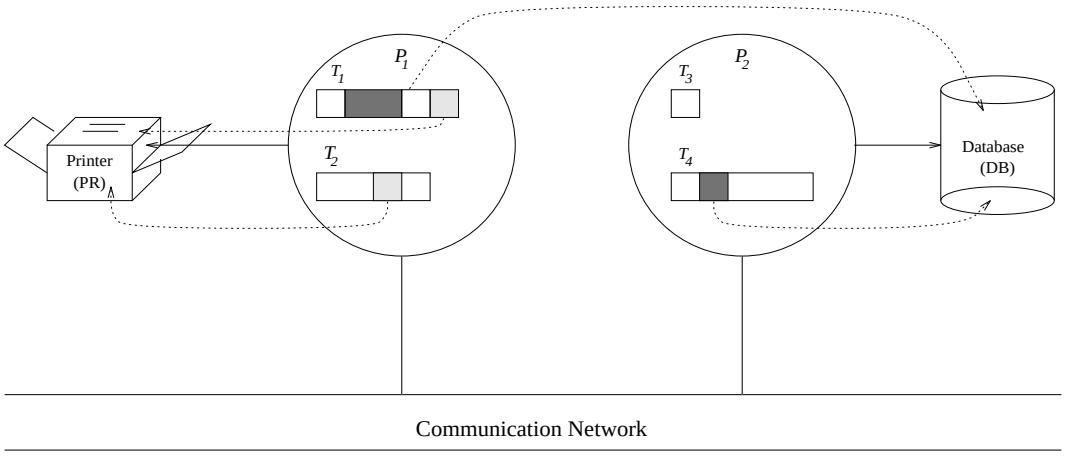



**Figure 8.1**: The Simple System in Example 1

A resource can only be accessed from its host processor. When a task requests a remote global resource, the execution thread of the task is suspended on the host processor of the task, and another execution thread is initiated on the host processor of the global resource on behalf of the task. According to the MPCP, the execution thread on the host processor of the global resource is called a *global critical section* (GCS) server [21]. The GCS server accesses the resource according to the given resource access protocol. Once the GCS server completes on the host processor of the global resource, it terminates and sends back the results; its corresponding task resumes the execution on its host processor. Figure 8.2 illustrates this situation for task $T_1$ in Example 1. The migration of the execution thread is depicted by the dashed lines in the figure. The execution of $T_1$ is suspended when it requests $DB$; $GCS_1$ starts execution and accesses $DB$ on behalf of $T_1$ on $P_2$, following the guidance of the given resource access protocol on $P_2$. When $GCS_1$ completes, it returns the result to $T_1$, and $T_1$ resumes execution on $P_1$.

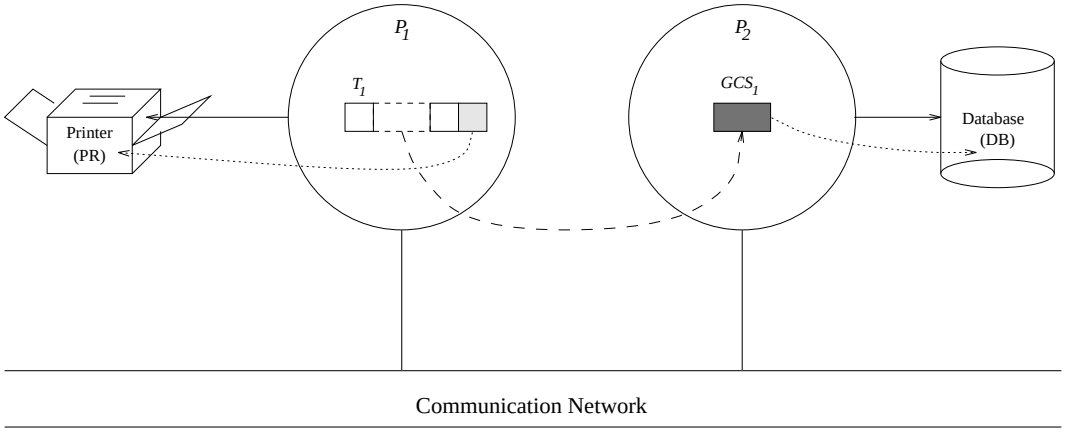



**Figure 8.2**: The GCS Server of $T_1$ in Example 1

According to the MPCP, a similar scenario happens even when a task requests a local global resource. The only difference is that both the task and the GCS server execute on the same processor. As we shall see in the next section, the task and its GCS server have different priorities. Figure 8.3 illustrates the scenario when task $T_4$ accesses the local global resource $DB$ in Example 1. Later, we will see that the classification of resources and the notion of GCS servers are only relevant to the MPCP. Such a distinction is not necessary in the end-to-end scheduling approach,

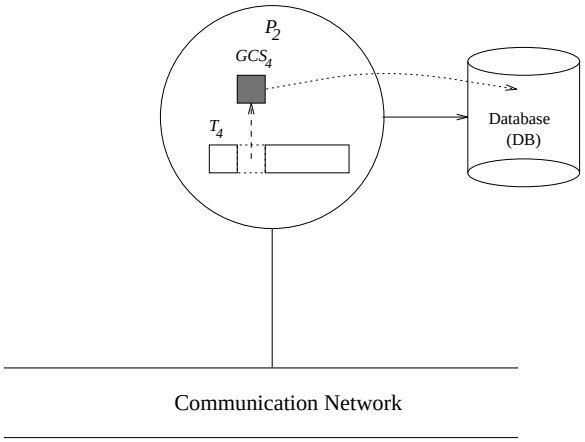



**Figure 8.3**: The GCS Server of $T_4$ in Example 1

According to the MPCP, resource accesses within a task must be properly nested. Furthermore the resources accessed in one nested critical section must reside on the same processor. In other words, accesses to resources on different processors cannot be nested within each other. Hereafter, by a critical section, we mean the outermost of a group of properly nested critical sections.

Additionally, we make the following assumptions about other aspects of the system.

1. Each task $T_i$ is a periodic task, with period $p_i$ and maximum execution time $\tau_i$.
2. The relative deadline $D_i$ of each task $T_i$ is no longer than its period.
3. Tasks are scheduled on a fixed-priority basis.

Assumption 2 is a restriction of the MPCP. As we will see, this assumption is not necessary in the end-to-end scheduling approach.

## 8.3 MPCP Approach

If a system fits the system model described in the previous section, we say it is an *MPCP system*. The MPCP itself, as a resource access protocol, only specifies the assignment of the GCS server priorities, the protocol for tasks and GCS servers to access shared resources, and the computation of upper bounds on task blocking times. To schedule an MPCP system and analyze its schedulability, we need additionally to specify the task priority assignment and the

schedulability analysis algorithm. Later, we use "the MPCP" in a strict sense to mean the resource access protocol itself. We use the term, *the MPCP approach*, to refer to the overall approach that uses the MPCP to solve the resource contention problem in an MPCP system. As we just suggested, the MPCP approach consists of three important parts: (1) priority assignment to both tasks and GCS servers, (2) the resource access protocol, which dictates when a resource access is granted and yields a way to compute upper bounds on blocking times of tasks, and (3) schedulability analysis of the system. We describe the three parts in detail in the following subsections.

### 8.3.1 Priority Assignment

According to the MPCP, each task is assigned a fixed priority. The optimum priority assignment method for an MPCP system is deadline-monotonic assignment, i.e., the priority of a task is inversely proportional to its relative deadline [2]. In the rest of this chapter, we assume deadline-monotonic priority assignment for tasks in an MPCP system.

According to the MPCP, each global critical section of a task, accessing either local global resources or remote resources, corresponds to a GCS server. The GCS servers have fixed priorities as well. Priorities for GCS servers are such that (1) all GCS servers have higher priorities than normal tasks; and (2) if task $T_1$ has a priority higher than (equal to) task $T_2$, the GCS servers of $T_1$ have priorities higher than (equal to ) those of $T_2$. Assuming that a smaller (or more negative) number represents a higher priority, a practical scheme of assigning GCS server priorities is to give the GCS server of task $T_i$ the priority $\phi_i - \hat{\phi}$, where $\phi_i$ is the priority of $T_i$ and $\hat{\phi}$ ($\hat{\phi} > 0$) is the lowest priority of all tasks.

As an Example, suppose that the task parameters of Example 1 are as shown in Table 8.1. There are two GCS servers in the system, $GCS_1$ for $T_1$ and $GCS_4$ for $T_4$. According to the deadline-monotonic priority assignment, the priority of a task can be set equal to its relative deadline. The task priorities are listed in Table 8.2. The priorities of GCS servers can be computed according the above mentioned method, which are also listed in the table.

### 8.3.2 Upper Bounds of Blocking Times

On each processor in an MPCP system, GCS servers contend for global resources on behalf of their corresponding tasks. From this point of view, tasks on each processor only contend for local

| $T_i$ | host | $p_i$ | $D_i$ | $\tau_i$ | Segments | | | |
|---|---|---|---|---|---|---|---|---|
| $T_1$ | $P_1$ | 15 | 15 | 5 | 1 | $2(DB)$ | 1 | $1(PR)$ |
| $T_2$ | $P_1$ | 20 | 20 | 4 | 2 | $1(PR)$ | 1 | |
| $T_3$ | $P_2$ | 2 | 2 | 1 | 1 | | | |
| $T_4$ | $P_2$ | 20 | 20 | 5 | 1 | $1(DB)$ | 3 | |

**Table 8.1**: Task Parameters of the System in Example 1

| $T_i$ | host | $D_i$ | $\phi_i$ |
|---|---|---|---|
| $T_1$ | $P_1$ | 15 | 15 |
| $T_2$ | $P_2$ | 20 | 20 |
| $T_3$ | $P_2$ | 2 | 2 |
| $T_4$ | $P_2$ | 20 | 20 |
| $GCS_1$ | $P_2$ | na | -5 |
| $GCS_4$ | $P_2$ | na | 0 |

**Table 8.2**: Task and GCS Server Priorities in Example 1

resources. A resource access control protocol for single-processor systems, such as the PCP [15] or the SBP [16], is used to control their accesses to the resources. As a consequence of using such a resource access control protocol, we can bound the blocking time that a GCS server can experience due to global resource contention or that a task can experience due to local resource contention. Suppose task $T_i$ has $n_i^g$ global critical sections. Let $GCS_{i,j}$ $(1 \leq j \leq n_i^g)$ denote the $j$th GCS server of $T_i$. We use $blk(T_i)$ (or $blk(GCS_{i,j})$) to denote the blocking time computed according to the PCP or SBP for $T_i$ $(or GCS_{i,j})$. According to the MPCP, the total blocking time $(B_i)$ that a task can experience is the sum of five factors.

**Local blocking time** $(LB_i)$ The local blocking time of a task is the blocking time a task can experience due to local resource contentions. It is given by

$$LB_i = blk(T_i) \times (n_i^g + 1) \tag{8.1}$$

**Global blocking time** $(GB_i)$ The global blocking time of a task is the blocking time that the GCS servers of the task experience when the servers access the global resources. It is given

by

$$GB_i = \sum_{j=1}^{n_i^g} blk(GCS_{i,j}) \tag{8.2}$$

**Remote waiting time** ($RW_i$) When a GCS server ($GCS_{i,j}$) of a task ($T_i$) is ready to execute on the host processor of the global resource that $T_i$ needs to access, some GCS servers of other tasks with higher priorities may periodically preempt the execution of $GCS_{i,j}$. If a GCS server $GCS_{u,v}$ is such a periodically preempting server, its interference to the execution of $GCS_{i,j}$ is counted in the schedulability analysis if task $T_u$ is on the same processor as task $T_i$. Otherwise the interference from $GCS_{u,v}$ should be counted as the remote waiting time. Let $GP_i$ denote the set of processors on which $T_i$ needs to access global resources, $RH_i$ denote the set of *remote higher-priority tasks*, i.e., the set of tasks that have priorities higher than or equal to $T_i$ and are on different host processors from the host processor of $T_i$. Let $CS_k(P_j)$ denote the sum of the durations of all the global critical sections of $T_k$ that execute on $P_j$. The remote waiting time of a task can then be expressed as

$$RW_i = \sum_{P_j \in GP_i} \left( \sum_{T_k \in RH_i} \left\lceil \frac{p_i}{p_k} \right\rceil CS_k(P_j) \right) \tag{8.3}$$

**Deferred blocking time** ($DF_i$) Let $LH_i$ denote the set of *local higher-priority tasks*, i.e., tasks that have priorities higher than or equal to $T_i$ and are on the same processor as $T_i$. Because a task in $LH_i$ may suspend itself during its access to a global resource and resume its execution later, it delays the completion of $T_i$ further than if the task in $LH_i$ were strictly periodic. The extra amount of interference is counted as the deferred blocking time of $T_i$. Let $\delta_k$ denote the remaining execution time of $T_k$ after it finishes the first access to a global resource. If the response time of a task is never longer than the task period, the deferred blocking time of a task can be computed by

$$DF_i = \sum_{T_k \in LH_i} \delta_k \tag{8.4}$$

**GCS server blocking time** ($GCSB_i$) This factor accounts for the inference from all the GCS servers that execute on the same processor as task $T_i$. Let $Host(T_i)$ denote the host processor of $T_i$ and $Dura(gcs)$ denote the duration of a GCS server $gcs$. The GCS server

blocking time can be computed by

$$GCSB_i = \sum_{GCS_{j,k} \ is \ on \ Host(T_i)} \left\lceil \frac{p_i}{p_j} \right\rceil Dura(GCS_{j,k}) \tag{8.5}$$

The total blocking time $B_i$ that task $T_i$ can experience is the sum of above five factors.

$$B_i = LB_i + GB_i + RW_i + DB_i + GCSB_i \tag{8.6}$$

The deferred blocking time is termed differently in [21]. In [21], for any task $T_i$ and any task $T_k \in LH_i$, the extra interference from $T_k$ due to execution suspension/resumption is $\min\{\delta_k, I_k\}$, where $I_k$ is the maximum amount of time that $T_k$ suspends itself. However, the definition and the computation of $I_k$ are not given in the original paper, and the justification is not given for taking the minimum of these two items. In this chapter, we only take $\delta_k$ as the amount of extra interference from $T_k$.

The computation of the deferred blocking time depends on the condition that the response time of a task is never longer than the period. If a task is verified to be schedulable, this condition is effectively satisfied because of the assumption that the relative deadline of a task is no longer than its period.

As an example, Table 8.3 lists the blocking times, as well as individual break down factors, of tasks in Example 1. Interested readers can verify the results by following the formula given above.

| $T_i$ | host | $B_i$ | $LB_i$ | $GB_i$ | $RW_i$ | $DB_i$ | $GCSB_i$ |
|---|---|---|---|---|---|---|---|
| $T_1$ | $P_1$ | 3 | 2 | 1 | 0 | 0 | 0 |
| $T_2$ | $P_1$ | 2 | 0 | 0 | 0 | 2 | 0 |
| $T_3$ | $P_2$ | 3 | 0 | 0 | 0 | 0 | 3 |
| $T_4$ | $P_2$ | 9 | 0 | 0 | 4 | 0 | 5 |

**Table 8.3**: Blocking Times of Tasks in Example 1

### 8.3.3 Schedulability Analysis

The schedulability analysis of the MPCP approach is based on the single-processor schedulability analysis, which can be the utilization bound analysis [1] or the more accurate time demand

analysis [5, 17]. According to the time demand analysis, the upper bound $R_i$ on the response time of $T_i$ can be obtained from the following equation.

$$R_i = \min \left\{ t > 0 | t = \tau_i + B_i + \sum_{T_j \in LH_i} \left\lceil \frac{t}{p_j} \right\rceil \tau_j \right\} \tag{8.7}$$

It is easy to verify that there exists a solution for $R_i$ if the total utilization of tasks in $LH_i$ is less than 1, i.e., $\sum_{T_j \in LH_i} (\tau_j / p_j) < 1$. As we have discussed in Chapter 5, an iterative procedure can be applied to compute the solution for $R_i$. The schedulability of a task can be verified by comparing the upper bound with its relative deadline, and the schedulability of the system can be verified by verifying the schedulability of every task in the system.

The above method for computing the upper bound on the response time of a task gives the correct answer if the upper bound is less than the period. It may give a wrong answer if the upper bound is greater than the period [6]. For the purpose of verifying the schedulability of a task, however, we obtain the same conclusion as if we would with the correct upper bound, because this incorrect upper bound is greater than the period of the task and hence the relative deadline of the task. Therefore we do not need to modify the above method for the purpose of schedulability analysis.

Under the column $R_i$, Table 8.4 gives the upper bounds on the response times of tasks according to the time demand analysis. For task $T_3$ and $T_4$, the bounds are greater than the corresponding task periods. The bounds may not be correct, but the conclusion about the schedulability of $T_3$ and $T_4$ is still right, i.e., we cannot guarantee that $T_3$ and $T_4$ are schedulable.

| $T_i$ | host | $p_i$ | $\tau_i$ | $B_i$ | $D_i$ | $R_i$ | schedulability |
|---|---|---|---|---|---|---|---|
| $T_1$ | $P_1$ | 15 | 5 | 3 | 15 | 8 | yes |
| $T_2$ | $P_1$ | 20 | 4 | 2 | 20 | 11 | yes |
| $T_3$ | $P_2$ | 2 | 1 | 3 | 2 | 4 | no |
| $T_4$ | $P_2$ | 20 | 5 | 9 | 20 | 28 | no |

**Table 8.4**: The Bounds on the Response Times of Tasks in Example 1 According to the MPCP

### 8.3.4 Corrections of the MPCP

Despite the careful analysis by the authors, there still exist some errors in bounding task blocking times. Table 8.5 lists the parameters of tasks in our second example, Example 2. Resource $R$

is on processor $P_2$, and it is a global resource. According to the MPCP, the blocking times and bounds on the response times of tasks are given in Table 8.6. All tasks are schedulable according to the result of the schedulability analysis.

| $T_i$ | host | phase | $p_i$ | $D_i$ | $\tau_i$ | Segments | | |
|---|---|---|---|---|---|---|---|---|
| $T_1$ | $P_1$ | 0 | 10 | 10 | 5 | | | |
| $T_2$ | $P_1$ | 0 | 13 | 13 | 4 | 1 | 2 $(R)$ | 1 |
| $T_3$ | $P_2$ | 6 | 25 | 25 | 20 | | | |

**Table 8.5**: Task Parameters in Example 2

| $T_i$ | $LB_i$ | $GB_i$ | $RW_i$ | $DB_i$ | $GCSB_i$ | $B_i$ | $R_i$ | schedulability |
|---|---|---|---|---|---|---|---|---|
| $T_1$ | 0 | 0 | 0 | 0 | 0 | 0 | 5 | yes |
| $T_2$ | 0 | 0 | 0 | 0 | 0 | 0 | 9 | yes |
| $T_3$ | 0 | 0 | 0 | 0 | 4 | 4 | 24 | yes |

**Table 8.6**: The Upper Bounds on the Blocking Times and Response Times of Tasks in Example 2

However, it is easy to verify that the first instance of $T_3$ misses its deadline at time 31, as shown in Figure 8.4. Clearly, something is wrong in the schedulability analysis. In particular, the GCS server blocking time of $T_3$ is 4, which according to Eq.(8.5) includes the interference of two instances of $GCS_2$. In the schedule, however, the first instance of $T_3$ is preempted by three instances of $GCS_2$ and, hence, misses the deadline. Therefore the calculation of GCS server blocking time is not correct. According to Eq.(8.5), the number of instances of a GCS server $GCS_j$ that can delay the completion of a task $T_i$ is determined by $\lceil p_i/p_j \rceil$. This formula would be correct if $GCS_j$ were a periodic task. However, the execution of $GCS_j$ is jittery, due to the varying completion times of its preceding task segments. Therefore, the Eq.(8.5) must be modified to count this jittery effect of a GCS server. Assuming that the response time of every task is no longer than the period, the following equation corrects the mistake.

$$GCSB_i = \sum_{GCS_{j,k} \in GCS(Host(T_i))} \left\lceil \frac{p_i}{p_j} + 1 \right\rceil Dura(GCS_{j,k}) \tag{8.8}$$

For the same reason, when computing the remote waiting factor, we should also add one more instance of the interfering servers into the remote waiting time.

$$RW_i = \sum_{P_j \in GP_i} \left( \sum_{T_k \in RH_i} \left\lceil \frac{p_i}{p_k} + 1 \right\rceil CS_k(P_j) \right) \tag{8.9}$$

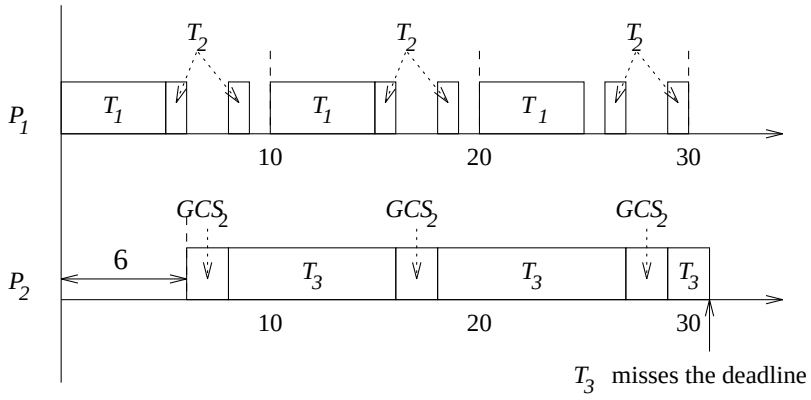



**Figure 8.4**: The Schedule of the System in Example 2

Taking into account the corrections, we compute the blocking times again for the tasks in Example 1. The new blocking times are listed in Table 8.7, which are obviously larger than those in Table 8.3.

| $T_i$ | host | $B_i$ | $LB_i$ | $GB_i$ | $RW_i$ | $DB_i$ | $GCSB_i$ |
|---|---|---|---|---|---|---|---|
| $T_1$ | $P_1$ | 3 | 2 | 1 | 0 | 0 | 0 |
| $T_2$ | $P_1$ | 2 | 0 | 0 | 0 | 2 | 0 |
| $T_3$ | $P_2$ | 6 | 0 | 0 | 0 | 0 | 6 |
| $T_4$ | $P_2$ | 14 | 0 | 0 | 6 | 0 | 8 |

**Table 8.7**: Blocking Times of Tasks in Example 1 According to the Corrected MPCP

### 8.3.5 Improvement to the MPCP

There are many ways to tighten the upper bounds on blocking times of tasks under the MPCP. Here we only discuss a couple of improvements that are obvious and have relatively significant impact on the performance of the MPCP.

In Table 8.7 the interference of $GCS_1$ to $T_4$ is counted twice, as in the remote waiting time and the GCS server blocking time. To eliminate the double counting, we can exclude from the

remote waiting time the interference of GCS servers that access local global resources. Eq.(8.9) can thus be written as

$$RW_i = \sum_{P_j \in GP_i - \{Host(T_i)\}} \left( \sum_{T_k \in RH_i} \left\lceil \frac{p_i}{p_k} + 1 \right\rceil CS_k(P_j) \right) \tag{8.10}$$

Two other straightforward improvements are related to the GCS server blocking time.

1. If a task accesses local global resources, its own GCS servers execute on its own host processor. The blocking times from its own GCS servers should not be counted into the GCS server blocking time of the task.

2. If a task $T_h$ in $LH_i$ accesses a local global resource, $T_h$ has a GCS server on the local processor. The interference from this GCS server of $T_h$ should not be counted in the GCS server blocking time of $T_i$, because this amount of interference is counted in the schedulability analysis as a higher-priority task $T_h$ executing on the same processor.

Based on these two rules, we can modify Eq.(8.8) correspondingly. Table 8.8 lists the blocking times of tasks in Example 1 computed by the improved formula. We notice that the blocking time of $T_4$ is reduced from 14 time units in Table 8.7 to 6 time units.

| $T_i$ | host | $B_i$ | $LB_i$ | $GB_i$ | $RW_i$ | $DB_i$ | $GCSB_i$ |
|---|---|---|---|---|---|---|---|
| $T_1$ | $P_1$ | 3 | 2 | 1 | 0 | 0 | 0 |
| $T_2$ | $P_1$ | 2 | 0 | 0 | 0 | 2 | 0 |
| $T_3$ | $P_2$ | 6 | 0 | 0 | 0 | 0 | 6 |
| $T_4$ | $P_2$ | 6 | 0 | 0 | 0 | 0 | 6 |

**Table 8.8**: Blocking Times of Tasks in Example 1 According to the Improved MPCP

In later discussion, we will refer to the formula introduced in [21] as the *original formula* of computing task blocking times, the formula described in Section 8.3.4 as the *corrected formula* of computing task blocking times, and the formula described in this subsection as the *improved formula* of computing task blocking times. In most cases, however, such a distinction is not necessary, and we simply use "the MPCP" without specifying a specific version of the formula of computing task blocking times.

## 8.4 End-to-End Scheduling Approach

If a task in an MPCP system accesses a remote global resource, it can be naturally viewed as an end-to-end task. The GCS server accessing the remote global resource is a subtask executing on the host processor of the global resource, and the other segments of the task may form other subtasks that execute on the host processor of the task. For example, task $T_1$ in Example 1, which accesses the remote global resource $DB$, can be viewed as an end-to-end task with 3 subtasks, as shown in Figure 8.5. Its GCS server $GCS_1$ in the MPCP approach is a subtask $T_{1,2}$ executing on the host processor $P_2$ of $DB$, and the other segments of $T_1$ form two other subtasks, $T_{1,1}$ and $T_{1,3}$, executing on the host processor $P_1$ of $T_1$. The other tasks in Example 1 do not access remote global resources, and each of them can be viewed as a trivial case of an end-to-end task with only one subtask.

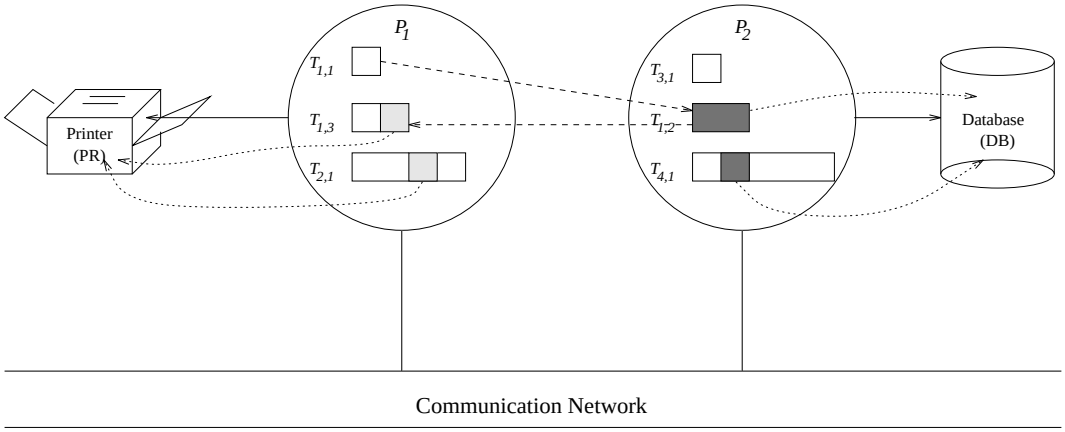



**Figure 8.5**: An End-to-End View of the System in Example 1

Viewing an MPCP system as an end-to-end system has several advantages. The first advantage is that all resource contention becomes local. In such an end-to-end system, a set of subtasks execute on each processor that require no remote resources. Therefore, we can use the PCP or SBP to control their accesses to the local resources and keep their blocking times bounded.

Another advantage is that we can use the rich set of end-to-end scheduling algorithms, described in the previous chapters, to schedule the subtasks and have more flexible control over the scheduling of such a system. For example, according to the MPCP approach, all the segments of a task that do not access global resources have the same priority. According to the end-to-end scheduling approach, however, subtasks in the same task may have different priorities, depending

on the particular priority assignment method used in the system. The other flexibility introduced in the end-to-end scheduling approach is the control over the releases of subtasks. We can choose one of the synchronization protocols described in Chapter 4 to govern the release of subtasks and synchronize the execution of them.

In general, according to the end-to-end scheduling approach, the solution to the resource contention problem in a distributed or multiprocessor real-time system consists of four steps:

**Step 1 :** Map the given MPCP system to an end-to-end system.

**Step 2 :** Assign priorities to subtasks, and find the upper bound on the blocking time of each subtask according to the PCP or SBP.

**Step 3 :** Choose a synchronization protocol.

**Step 4 :** Analyze the schedulability of each task and the system.

### 8.4.1 Mapping an MPCP System to an End-to-End System

We use the following rules to map a task in an MPCP system to a task in an end-to-end system.

1. For each task $T_i$, each outermost global critical section that accesses one or more remote global resources is a subtask. This subtask executes on the host processor of the remote global resource(s).

2. Among the remaining segments of $T_i$, each contiguous group of segments is a subtask that executes on the host processor of $T_i$.

The end-to-end scheduling algorithms does not prevent two consecutive subtasks from executing on the same processor. Theoretically, we could map a local critical section or a local global critical section into a subtask. However, it is better to combine contiguous segments into a single subtask. As a result, in the end-to-end scheduling approach, accesses to a local global resource is treated in the same way as accesses to a local resource. In contrast, in the MPCP approach, accesses to a local global resource are treated in a way similar to accesses to a remote global resource.

Let us look at the system in Example 1 again. The parameters of subtasks after the mapping process are listed in Table 8.9. The second segment in $T_1$ needs to access a remote global resource

$DB$, and therefore it is mapped as a subtask $T_{1,2}$ by itself, executing on $P_2$. The first segment in $T_1$ forms a subtask $T_{1,1}$, and the other two segments form another one $T_{1,3}$. Both of them execute on $P_1$. We notice that $T_{1,3}$ actually contains two segments, including one local critical section. In the case of $T_4$, although it accesses the global resource $DB$, all three segments still form a single subtask $T_{4,1}$, because the global resource is a local global resource.

| $T_i$ | proc | $p_{i,j}$ | $\tau_{i,j}$ | segments | | |
|---|---|---|---|---|---|---|
| $T_{1,1}$ | $P_1$ | 15 | 1 | 1 | | |
| $T_{1,3}$ | $P_1$ | 15 | 2 | 1 | $1(PR)$ | |
| $T_{2,1}$ | $P_1$ | 20 | 4 | 2 | $1(PR)$ | 1 |
| $T_{1,2}$ | $P_2$ | 15 | 2 | $2(DB)$ | | |
| $T_{3,1}$ | $P_2$ | 2 | 1 | 1 | | |
| $T_{4,1}$ | $P_2$ | 20 | 5 | 1 | $1(DB)$ | 3 |

**Table 8.9**: Subtask Parameters of the System in Example 1

### 8.4.2 Priority Assignment and Execution Synchronization

In Chapter 6, we have proposed several deadline-based priority assignment methods. They can be readily applied here. Table 8.10 lists the global deadlines, effective deadlines, proportional deadlines and normalized proportional deadlines of subtasks in Example 1. In order to compute the normalized proportional deadlines of subtasks, we first compute the utilization factors of $P_1$ and $P_2$. They are 0.4 and 0.88 respectively.

| Task | Proc | Period | $\tau_{i,j}$ | $GD_{i,j}$ | $ED_{i,j}$ | $PD_{i,j}$ | $NPD_{i,j}$ |
|---|---|---|---|---|---|---|---|
| $T_{1,1}$ | $P_1$ | 15 | 1 | 15 | 11 | 3 | 2.0 |
| $T_{1,3}$ | $P_1$ | 15 | 2 | 15 | 15 | 6 | 4.1 |
| $T_{2,1}$ | $P_1$ | 20 | 4 | 20 | 20 | 20 | 20 |
| $T_{1,2}$ | $P_2$ | 15 | 2 | 15 | 13 | 6 | 8.9 |
| $T_{3,1}$ | $P_2$ | 2 | 1 | 2 | 2 | 2 | 2 |
| $T_{4,1}$ | $P_2$ | 20 | 5 | 20 | 20 | 20 | 20 |

**Table 8.10**: Different Deadlines for Subtasks in Example 1

Hereafter, we assume that the PDM assignment is used for the system in Example 1. Given the priorities assigned according to the PDM method, the blocking time of each subtask can be bounded according to the PCP or SBP. The maximum blocking times of subtasks in Example 1 are listed in Table 8.11.

| Task | Proc | $PD_{i,j}$ | $\phi_{i,j}$ | $B_{i,j}$ |
|---|---|---|---|---|
| $T_{1,1}$ | $P_1$ | 3 | 3 | 0 |
| $T_{1,3}$ | $P_1$ | 6 | 6 | 1 |
| $T_{2,1}$ | $P_1$ | 20 | 20 | 0 |
| $T_{1,2}$ | $P_2$ | 6 | 6 | 1 |
| $T_{3,1}$ | $P_2$ | 2 | 2 | 0 |
| $T_{4,1}$ | $P_2$ | 20 | 20 | 0 |

**Table 8.11**: Different Deadlines for Subtasks in Example 1

We can choose to use any one of the synchronization protocols described in Chapter 4. The decision usually depends on the constraints of the system.

### 8.4.3 Schedulability Analysis

Depending on the synchronization protocol we choose, we apply an appropriate schedulability analysis algorithm described in Chapter 5 to analyze the schedulability of the system. Here, when we perform the schedulability analysis, we need to take the blocking time of each subtask into account. To do so, we just add the blocking time of $T_{i,j}$ to the appropriate time demand function with respect to an instance of $T_{i,j}$ or with respect to a $\phi_{i,j}$-level busy period. Let $B_{i,j}$ denote the maximum blocking time that a subtask $T_{i,j}$ can experience. The above-mentioned modification to the time demand function is correct because $B_{i,j}$ is viewed as the maximum amount of time demand that can be generated by some subtask with priority lower than $\phi_{i,j}$ in a $\phi_{i,j}$-level busy period.

Take Algorithm SA/PM for example. The time demand function $W(t)$ with respect to a $\phi_{i,j}$-level busy period, counting the blocking time $B_{i,j}$, becomes

$$B_{i,j} + \sum_{T_{k,l} \in H_{i,j} \cup \{T_{i,j}\}} \left\lceil \frac{t}{p_k} \right\rceil \tau_{k,l}$$

and the upper bound on the duration of the busy period is accordingly given by

$$D_{i,j} = \min \left\{ t > 0 \mid t = B_{i,j} + \sum_{T_{k,l} \in H_{i,j} + \{T_{i,j}\}} \left\lceil \frac{t}{p_k} \right\rceil \tau_{k,l} \right\}$$

instead of Eq.(5.2). Similar changes can be made for other time demand functions.

As an example, we apply Algorithm SA/DS, SA/PM and SA/IPM to the system in Example 1 and obtain the same upper bounds on task EER times. The upper bounds are listed in Table 8.12. We see that every task is schedulable and the system is schedulable.

| $T_i$ | host | $D_i$ | $R_i$ | schedulability |
|---|---|---|---|---|
| $T_1$ | $P_1$ | 15 | 11 | yes |
| $T_2$ | $P_1$ | 20 | 7 | yes |
| $T_3$ | $P_2$ | 2 | 1 | yes |
| $T_4$ | $P_2$ | 20 | 14 | yes |

**Table 8.12**: Upper Bounds on Task EER Times in Example 1

## 8.5 Performance Comparison

We see that the system in Example 1 is not schedulable according to the MPCP but schedulable according to the end-to-end scheduling approach. We can also construct systems which are schedulable according to the MPCP approach but not schedulable according to the end-to-end scheduling approach. This motivates us to study the performance of these two approaches. To compare their performance, we generated a set of synthetic systems with shared resources and scheduled these systems according to these two approaches. In this section, we first discuss the workload generation, followed by the criteria we use to evaluate the performance of the two approaches, which in turn is followed by the simulation results.

### 8.5.1 Workload Generation

In order to reveal the strength and weakness of the MPCP approach and the end-to-end scheduling approach, we generated synthetic systems in a controlled fashion. As fixed parameters, each system has five processors, and each processor has five tasks. Task periods are exponentially distributed between 10000 and 100000. We chose these numbers because they represent real-life

applications to a certain extent on one hand. On the other hand, these numbers are small enough to keep the simulation practical. We found that varying these numbers made little impact on the relative performance of both approaches and hence did not affect the conclusions we draw from the simulation.

Other parameters of each generated system are specified by a *configuration*. Each configuration is a 4-tuple $(\mu, \kappa_{cs}, \lambda, \eta)$, where $\mu$ is the utilization on each processor, $\kappa_{cs}$ is the number of critical sections in each task, $\lambda$ is the method to determine the duration of each critical section and $\eta$ is the *critical section duration factor*, or *CSD factor*, which we will explain later. For simplicity, we let every processor have the same utilization and let tasks on each processor evenly divide the processor utilization. In other words, the utilization of each task is $\mu/5$, and the execution time of task is equal to $\mu/5$ times the task period. We also let every task have the same number $\kappa_{cs}$ of critical sections.

The method to determine the duration of a critical section can be either the *fixed duration (fixed)* method or the *variable duration (variable)* method. In the case of the variable duration method, the duration of a critical section is proportional to the execution time of the task, and CSD factor $\eta$ specifies the fraction of the duration over the task execution time. A real-life example might be a critical section which accesses the printer resource; the length of time when the task uses the printer is proportional to the execution time of the task. In the case of fixed duration method, the duration of a critical section is not proportional to the execution time of the task. Instead, it is a fixed amount of time, and CSD factor $\eta$ specifies the fraction of the duration over the minimum possible task execution time. If the processor utilization is $\mu$, the minimum possible task execution time is $10000\mu/5$, and the fixed duration is $1000 * \mu * \eta/5$. An example of a fixed critical section might be an exclusive access to a database. When a task has more than one critical section, we randomly distribute them over the task execution time, as long as no two critical sections overlap with each other.

The last issue is to determine which resource each critical section accesses. We assign a resource to each processor and associate each critical section randomly with a resource in the system. A resource can be a local resource if all the tasks that need to access it happen to be on the same host processor as the resource, but more than likely resources are global resources.

In our simulation, we let the processor utilization be 0.5, 0.6, 0.7 and 0.8, the number of critical sections in each task be 1, 2, ..., 6, and the CSD factor be 0.01, 0.02, ..., 0.08. Since

we have two different methods to generate the duration of a critical section, we thus have a total of 384 configurations. For each configuration, the number of systems generated depends on the performance criteria, which we will explain in the next section.

Let us look at an example. Suppose a system is generated according to the configuration $(0.5, 4, variable, 0.02)$. In the system, there are five processors, on each of which there are one resource and five tasks. The processor utilization of each processor is 0.5, and the utilization for each task is 0.1. The task periods are distributed according to an exponential probability density function which is truncated within the range [10000,100000], and the execution time of a task is one-tenth of its period. Each task has four critical sections, each of which accesses a resource chosen at random and has a duration equal to 2% of the task execution time.

### 8.5.2 Performance Criteria

For each generated system, we first verified its schedulability according to the MPCP approach, using both the corrected formula and the improved formula of computing the task blocking times. Obviously, if a system is schedulable by using the corrected formula, it is also schedulable by using the improved formula. We say a system is *schedulable according to the MPCP approach* if it is verified schedulable by using the improved formula of computing task blocking times.

For the end-to-end scheduling approach, we first assign subtask priorities according to the PDM method or the NPDM method, because in general the PDM and NPDM methods have better performance than other methods. Among the three schedulability analysis algorithms in the end-to-end scheduling approach, Algorithm SA/IPM gives the best prediction on system schedulability. (We notice that Algorithm SA/IPM is applicable here because the relative deadlines of tasks are no longer than the respective periods and many tasks are recurrent.) If a system is schedulable according to the Algorithm SA/DS it must be schedulable according to Algorithm SA/PM, and if a system is schedulable according to Algorithm SA/PM it must be schedulable according to Algorithm SA/IPM. We say a system is *schedulable according to the end-to-end scheduling approach* if it uses the PDM or NPDM priority assignment method and is schedulable according to Algorithm SA/IPM. To compare the overall performance of the MPCP approach and the end-to-end scheduling approach, it suffices for us to verify the schedulability of a system according to Algorithm SA/IPM alone. For other performance comparison, as we

will see later, we also verify the schedulability of a system according to Algorithm SA/DS and Algorithm SA/PM.

For a given set of synthetic systems, a good performance indicator of a certain approach is the *success rate* of the approach, which is the percentage of systems predicted schedulable according to the approach over all the systems which are indeed schedulable according to an optimal approach. However, we do not know such an optimal algorithm and hence do not know which system is indeed schedulable. Consequently, we cannot obtain an absolute success rate. For this reason, we approximate the success rate of an approach by the number of systems schedulable according to this approach over the total number of *valid systems*, where a valid system is one that is schedulable according to either the MPCP approach or the end-to-end scheduling approach. The approximate success rates tell us how well these two approaches perform relative to each other. In later discussion, by success rates, we mean the approximate success rates.

In order to gain statistical significance, we kept generating systems according to a specified configuration till the number of valid systems reached 1000. For a configuration, the *valid rate* is the number of valid systems over the total number of systems generated. When the valid rate is too low (such as in the case of configurations with high utilization, many critical sections in each task and long critical section durations) the time required to generate the sufficient large number valid system becomes too long to be practical. We dropped the simulation for configurations whose valid rates are below 4% and marked such configurations as *invalid configurations*. All other configurations are marked as *valid configurations*.

For some configurations, the valid rate is 1, and the success rates of both the MPCP approach and the end-to-end scheduling approach are 100%. This indicates that systems generated according to these configurations are very easy to schedule. The performance indicators tell little about the performance difference between these two approaches. We mark these configurations as non-effective configurations. In later discussion, we focus on configurations which are both valid and effective.

### 8.5.3 Overall Performance Comparison

Table 8.13(a) lists the success rates of the MPCP approach and the end-to-end scheduling approach when the processor utilization is 0.5 and the duration of each critical section is determined according to the fixed duration method. Each entry in the first row lists the CSD factor and each

entry in the first column indicates the number of critical sections in each task. Table 8.13(b) lists the success rates under the similar situation except that the duration of each critical section is determined according to the variable duration method. In the table, an entry marked "–/–" indicates the corresponding configuration is not effective, and an entry marked "xx/xx" indicates the corresponding configuration is not valid. All other entries correspond to valid and effective configurations. Inside each of such entries, the number before "/" is the success rate (in percentage) of the MPCP approach, and the number after "/" is the success rate of the end-to-end scheduling approach.

| # of CS | 0.01 | 0.02 | 0.03 | 0.04 | 0.05 | 0.06 | 0.07 | 0.08 |
|---|---|---|---|---|---|---|---|---|
| 1 | –/– | –/– | –/– | –/– | –/– | –/– | –/– | –/– |
| 2 | –/– | –/– | –/– | –/– | 99/100 | 99/100 | 96/100 | 93/100 |
| 3 | –/– | 99/100 | 98/100 | 93/100 | 85/100 | 71/ 99 | 53/ 99 | 39/100 |
| 4 | 100/ 90 | 96/ 87 | 88/ 86 | 74/ 82 | 54/ 85 | 32/ 86 | 17/ 90 | 6/ 96 |
| 5 | 99/ 42 | 97/ 36 | 91/ 30 | 84/ 28 | xx/xx | xx/xx | xx/xx | xx/xx |
| 6 | 100/ 5 | 100/ 2 | 100/ 1 | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx |

(a)

| # of CS | 0.01 | 0.02 | 0.03 | 0.04 | 0.05 | 0.06 | 0.07 | 0.08 |
|---|---|---|---|---|---|---|---|---|
| 1 | –/– | –/– | –/– | –/– | 99/100 | 98/100 | 97/100 | 96/100 |
| 2 | –/– | 98/100 | 91/100 | 75/100 | 49/ 99 | 22/100 | 4/100 | 1/100 |
| 3 | 98/ 99 | 79/ 96 | 37/ 92 | 6/ 97 | 0/100 | 0/100 | xx/xx | xx/xx |
| 4 | 91/ 77 | 60/ 56 | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx |
| 5 | 97/ 14 | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx |
| 6 | 100/ 0 | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx |

(b)

**Table 8.13**: The Success Rates When the Processor Utilization is 0.5

If for a particular configuration there is less than 10% difference in success rates of the MPCP approach and the end-to-end approach, we say the two approaches have similar performance. In both tables, each valid and effective configuration falls into three categories, (1) where both approaches have similar performance (the neutral zone), (2) where the MPCP approach has

better performance (the MPCP zone), and (3) where the end-to-end scheduling approach has better performance (the EE zone). Clearly, the configurations in the MPCP zone cluster around left-bottom corner. They have large numbers of critical sections in each task and small CSD factors. On the other hand, configurations in the EE zone cluster around upper right corner, and they have small numbers of critical sections in each task and large CSD factors. This indicates that when each task has a large number of (global) critical sections, the MPCP approach is favored, and when the duration of each critical section is long, the end-to-end scheduling approach is favored.

To see the reason for the difference in the performance, let us examine how the system schedulability is verified according to either of the two approaches. According to the end-to-end scheduling approach, specifically Algorithm SA/PM and SA/IPM, the schedulability of a system is verified by comparing the upper bound on the end-to-end response time of each task with its relative deadline, and the upper bound of each task is equal to the sum of upper bounds on the response times of all its subtasks. When bounding the response time of a subtask $T_{i,j}$, we count the worst-case interference from other tasks (subtasks) to $T_{i,j}$. When we bound the response time of a sibling subtask $T_{i,k}$ which is on the same processor as $T_{i,j}$, part of the worst-case interference counted for $T_{i,j}$ may be counted again. In reality, such an interference may never occur to both $T_{i,j}$ and $T_{i,k}$ during the execution of a single $T_i$ instance. For example, suppose that subtask $T_{1,1}$, $T_{1,3}$ and $T_{2,1}$ are on the same processor and $T_{2,1}$ has the highest priority and a longer period. When bounding the response time of $T_{1,1}$ and $T_{1,3}$ according to Algorithm SA/PM, we count the execution time of $T_{2,1}$ as part of the interference to them in each case. However, since the period of $T_{2,1}$ is longer than the period of $T_1$, for one instance of $T_1$, we know that $T_{1,1}$ and $T_{1,3}$ cannot both be preempted by $T_{2,1}$ when the end-to-end response time of $T_1$ is no longer than its period. Yet in the upper bound on the end-to-end response time of $T_1$, the execution time of $T_{2,1}$ is counted twice. A similar situation happens when Algorithm SA/IPM is used. Unfortunately, for the PM and RG protocols to work correctly, this repetitive counting cannot be eliminated. When a task has a large number of global critical sections, this problem becomes more serious.

On the other hand, since the schedulability analysis in the MPCP approach is based on the schedulability analysis for single-processor systems, the repetitive counting problem does not exist. This is why the MPCP approach is favored when each task has a large number of critical sections. However, complicated blocking situations make the bounds on task blocking times rather

pessimistic under the MPCP approach. When the duration of critical sections becomes longer, this problem is "magnified," and eventually makes the end-to-end approach more favorable under such situations.

In Table 8.14, 8.15 and 8.16, we list the success rates of the MPCP approach and the end-to-end scheduling approach when the processor utilization is 0.6, 0.7, and 0.8 respectively. The systems are more difficult to schedule when the utilization goes up. We notice that although a similar partition of three different zones is still visible, valid configurations gradually shift to those with a small number of critical sections in each task, where the end-to-end scheduling approach is favored. As a consequence, among the valid configurations in these tables, the success rates of the MPCP approach become worse compared with those of the end-to-end scheduling approach. However, from these numbers alone, we cannot draw the conclusion that the end-to-end scheduling approach is better when the processor utilization is high, because we do not know how long the critical section is in reality. It is conceivable that if we create some new configurations with higher processor utilization and large number of critical sections in each task but very short critical section durations, the MPCP approach may perform better than the end-to-end scheduling approach. In this sense, this study only reveals the shape of the performance curve and gives insight to the performance difference. How much actual difference between the performance of these two approaches depends on exactly where the system sits in the curve in terms of the four system parameters. We will return to this issue again in Section 8.5.7.

### 8.5.4 Improvement to the MPCP

In our simulation, we verified the schedulability of each system according to the MPCP approach using both the corrected formula and the improved formula of computing task blocking times. We can determine how much improvement in performance we achieve by the improved formula. We define the *average success rate* of using a particular formula of the MPCP to be the average of the success rates for all valid and effective configurations. Table 8.17 lists the two average success rates of using the corrected formula and the improved formula. We see on average a 27% increase on the average success rate of the improved formula over the corrected formula.

| # of CS | 0.01 | 0.02 | 0.03 | 0.04 | 0.05 | 0.06 | 0.07 | 0.08 |
|---|---|---|---|---|---|---|---|---|
| 1 | 95/100 | 96/100 | 95/100 | 93/100 | 91/100 | 91/100 | 88/100 | 85/100 |
| 2 | 81/ 99 | 73/100 | 58/100 | 48/100 | 36/100 | 26/100 | 17/100 | 11/100 |
| 3 | 67/ 86 | 48/ 89 | 28/ 93 | 13/ 97 | 4/ 99 | 1/100 | 0/100 | 0/100 |
| 4 | 86/ 29 | 70/ 42 | 32/ 72 | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx |
| 5 | 100/ 1 | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx |
| 6 | 100/ 0 | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx |

(a)

| # of CS | 0.01 | 0.02 | 0.03 | 0.04 | 0.05 | 0.06 | 0.07 | 0.08 |
|---|---|---|---|---|---|---|---|---|
| 1 | 95/100 | 89/100 | 84/100 | 75/100 | 62/100 | 49/100 | 36/100 | 24/100 |
| 2 | 63/100 | 33/ 99 | 7/100 | 1/100 | 0/100 | 0/100 | 0/100 | 0/100 |
| 3 | 40/ 88 | 3/ 98 | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx |
| 4 | 73/ 30 | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx |
| 5 | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx |
| 6 | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx |

(b)

**Table 8.14**: The Success Rates When the Processor Utilization is 0.6

| # of CS | 0.01 | 0.02 | 0.03 | 0.04 | 0.05 | 0.06 | 0.07 | 0.08 |
|---|---|---|---|---|---|---|---|---|
| 1 | 31/100 | 25/100 | 21/100 | 20/100 | 17/100 | 14/100 | 9/100 | 8/100 |
| 2 | 8/ 97 | 4/ 98 | 1/100 | 1/100 | 0/100 | 0/100 | 0/100 | 0/100 |
| 3 | 18/ 84 | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx |
| 4 | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx |
| 5 | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx |
| 6 | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx |

(a)

| # of CS | 0.01 | 0.02 | 0.03 | 0.04 | 0.05 | 0.06 | 0.07 | 0.08 |
|---|---|---|---|---|---|---|---|---|
| 1 | 26/100 | 14/100 | 8/100 | 4/100 | 1/100 | 1/100 | 0/100 | 0/100 |
| 2 | 2/100 | 0/100 | 0/100 | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx |
| 3 | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx |
| 4 | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx |
| 5 | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx |
| 6 | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx |

(b)

**Table 8.15**: The Success Rates When the Processor Utilization is 0.7

| # of CS | 0.01 | 0.02 | 0.03 | 0.04 | 0.05 | 0.06 | 0.07 | 0.08 |
|---|---|---|---|---|---|---|---|---|
| 1 | 0/100 | 0/100 | 0/100 | 0/100 | 0/100 | 0/100 | 0/100 | 0/100 |
| 2 | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx |
| 3 | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx |
| 4 | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx |
| 5 | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx |
| 6 | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx |

(a)

| # of CS | 0.01 | 0.02 | 0.03 | 0.04 | 0.05 | 0.06 | 0.07 | 0.08 |
|---|---|---|---|---|---|---|---|---|
| 1 | 0/100 | 0/100 | 0/100 | 0/100 | 0/100 | 0/100 | 0/100 | 0/100 |
| 2 | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx |
| 3 | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx |
| 4 | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx |
| 5 | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx |
| 6 | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx | xx/xx |

(b)

**Table 8.16**: The Success Rates When the Processor Utilization is 0.8

| | the corrected formula | the improved formula |
|---|---|---|
| the average success rate | 0.34 | 0.43 |

**Table 8.17**: The Average Success Rates of the Fixed MPCP and the Improved MPCP

### 8.5.5 Performance of End-to-End Scheduling Algorithms

It is educational to study the performance of different end-to-end scheduling algorithms. Since in each generated system, each processor has the same processor utilization,[1] the PDM and NPDM priority assignment methods give the same priority to each subtask and hence have the same performance. This allows us to focus on the performance difference of Algorithm SA/DS, SA/PM and SA/IPM. Table 8.18 lists the average success rates of these three algorithms for all valid and effective configurations. Compared with Algorithm SA/PM and SA/IPM, Algorithm SA/DS does not yield satisfactory performance. However, it is meaningful to include the DS protocol and Algorithm SA/DS in the end-to-end scheduling approach, as we will see shortly.

| | Algorithm SA/DS | Algorithm SA/PM | Algorithm SA/IPM |
|---|---|---|---|
| the average success rate | 0.36 | 0.86 | 0.90 |

**Table 8.18**: The Average Success Rates of Algorithms SA/DS, SA/PM and SA/IPM

In Chapter 7, we saw a tiny reduction ( 0.2%) in schedulability index of Algorithm SA/IPM over Algorithm SA/PM. Here we see that the tiny reduction in schedulability index translates to an increase of about 4% in the success rate of Algorithm SA/IPM over Algorithm SA/PM.

The average success rate of Algorithm SA/IPM is essentially the average success rate of the end-to-end scheduling approach. Similarly, the average success rate of the MPCP with the improved formula of computing task blocking times is the average success rate of the MPCP approach. By comparing these two numbers, we see that the end-to-end scheduling approach is more effective to schedule the systems generated in our simulation.

### 8.5.6 The MPCP Approach vs. the DS Synchronization Protocol

Among the end-to-end scheduling algorithms, the DS protocol and Algorithm SA/DS deserve special attention. Like the MPCP approach, the DS protocol does not require any special scheduling support, such as those required by the PM and RG protocol. Both of them employ straightforward fixed priority scheduling. From the scheduling point of view, the MPCP approach can be viewed as a special case of the end-to-end scheduling approach using the DS protocol. The difference is that it has its own priority assignment method and its own schedulability analysis

[1]Due to remote resource accesses, each processor may have slightly different processor utilizations.

algorithm. As a matter of fact, since the DS protocol does not assume any particular priority assignment, we can apply Algorithm SA/DS to analyze the schedulability of an MPCP system. However, we do not expect that this application will yield better performance than the current schedulability analysis algorithm used in the MPCP approach.

Since the underlying scheduling techniques are virtually the same for both the MPCP approach and the DS protocol, a system designer can easily switch from the MPCP approach to the end-to-end scheduling approach using the DS protocol, by adjusting the priorities of subtasks (or the priorities of task segments in the context of an MPCP system). Therefore, a practical approach for a system designer is to try both the MPCP approach and the end-to-end scheduling approach using the DS protocol (combined with either the PDM or NPDM assignment). Table 8.19 lists the average success rates for Algorithm SA/DS, the MPCP approach and the combined approach. We see the obvious increase in the success rate of the combined approach over either the MPCP approach or the end-to-end scheduling approach using the DS protocol, indicating that combined approach is a valid and effective choice.

| | Algorithm SA/DS | the MPCP | the combined approach |
|---|---|---|---|
| the average success rate | 0.36 | 0.43 | 0.55 |

**Table 8.19**: The Average Success Rates of the MPCP Approach, Algorithm SA/DS and the Combined Approach

### 8.5.7 Overall Performance with a Mixed Workload

In section 8.5.3, we have studied the performance of the MPCP approach and the end-to-end scheduling approach with respect to controlled workload. In each synthetic system, we let every processor have the same processor utilization, the same number of tasks, and every task has the same number of critical sections whose duration is a fixed length of time or a fixed fraction of the execution time of the task. We can thus study the performance of these two approaches with respect to different parameters. However, the system parameters can never be so uniform in reality. The schedulability of a system is the result of a complicated interaction between the system parameters. The performance results for controlled workload may give us some idea how these two approaches perform for a mixed workload, where system parameters are randomized,

but it will not provide the whole picture. In this section, we study their performance when the system parameters are randomized.

Similar to the method used in the previous experiment in Section 8.5.3, the workload were generated according to *configurations*, each of which is a doublet, $(\hat{\kappa_{cs}}, \hat{\eta})$. Systems were generated in the way described in Section 8.5.3, except for the following differences:

1. The processor utilization is randomly distributed between 0.5 and 0.7.
2. The number of critical sections in each task is randomly distributed between 1 and $\hat{\kappa_{cs}}$.
3. The duration of each critical section is randomly determined according to the fixed duration method or the variable duration method.
4. The CSD factor is randomly distributed between 0.01 and $\hat{\eta}$.

In the simulation, we let $\hat{\kappa_{cs}}$ be 1, 2, ..., 8, and $\hat{\eta}$ be 0.01, 0.02, ..., 0.06. Thus we have 48 configurations. For configurations with larger $\hat{\kappa_{cs}}$ and $\hat{\eta}$, the system parameters generated are more randomized and the systems are more difficult to schedule.

For each configuration, we kept generating systems until we have obtained 1000 valid systems or the valid rate becomes below 4%. Table 8.20 lists the success rates of the MPCP approach and the end-to-end scheduling approach for all configurations.

| # of CS | 0.01-0.01 | 0.01-0.02 | 0.01-0.03 | 0.01-0.04 | 0.01-0.05 | 0.01-0.06 | 0.01-0.07 | 0.01-0.08 |
|---|---|---|---|---|---|---|---|---|
| 1-1 | 80/100 | 77/100 | 76/100 | 74/100 | 70/100 | 69/100 | 66/100 | 65/100 |
| 1-2 | 66/100 | 62/ 99 | 58/100 | 55/ 99 | 45/100 | 42/100 | 35/100 | 31/100 |
| 1-3 | 58/ 96 | 49/ 98 | 39/ 97 | 34/ 98 | 25/ 99 | 20/ 99 | 17/100 | 12/100 |
| 1-4 | 58/ 88 | 44/ 93 | 35/ 94 | 27/ 95 | 19/ 98 | 15/ 98 | 9/ 99 | 6/ 99 |
| 1-5 | 56/ 78 | 44/ 85 | 35/ 87 | 22/ 93 | 15/ 96 | 10/ 98 | 7/ 99 | 4/ 98 |
| 1-6 | 67/ 59 | 53/ 69 | 37/ 79 | 27/ 83 | 17/ 91 | xx/xx | xx/xx | xx/xx |

**Table 8.20**: Success Rates of the MPCP Approach and the End-to-End Scheduling Approach for Mixed Workload

In the table, we notice that the end-to-end scheduling approach consistently performs much better than the MPCP approach, except for one configuration. Each configuration in mixed workload corresponds to several configurations in controlled workload. Compared with the tables

presented in Section 8.5.3, it is easy to verify that the success rate of the end-to-end scheduling approach for a configuration in mixed workload is better than the the average of the success rates for the corresponding configurations in the controlled workload. This indicates that for a mixed workload the end-to-end scheduling approach is more resilient to negative factors than the MPCP approach.

## 8.6 Summary

In this chapter, we introduced the MPCP, an existing solution to the resource contention problem in a multi-processor or distributed real-time system. We also pointed out two mistakes in the original formula of computing task blocking times in the MPCP and proposed a couple of straightforward improvements. We then described the end-to-end scheduling based approach to the same problem. We showed that the end-to-end scheduling algorithms we proposed earlier in this thesis can be applied to solve this problem. Simulation was performed to compare the performance of these two approaches.

From the simulation results, we see that the end-to-end scheduling approach is more suitable for systems where tasks have long critical sections, and the MPCP approach performs better for systems where each task has many critical sections. Overall, the end-to-end scheduling approach performs better for the systems we tested in this chapter.

Compared with the MPCP approach, the end-to-end scheduling approach is more flexible. It allows sibling subtasks to have different priorities and different protocols to release subtasks and synchronize the execution of subtasks. The system designer can choose one which fits the application best.

Another advantage of the end-to-end scheduling approach is that the relative deadlines of tasks do not have to be shorter than or equal to their periods, except when the PM or MPM protocol and Algorithm SA/IPM are used. This makes the end-to-end scheduling approach applicable to more applications.

We can extend the system model in this chapter to a general end-to-end system model, where each task is a chain of subtasks, and we still allow each subtask to access any resource in the system. In this extended model, the end-to-end scheduling approach can be extended accordingly. The mapping process will transform an end-to-end system with global resource contention to

another (more refined) end-to-end system with only local resource contention among subtasks on the same processor. The schedulability analysis is applicable as before. In this general case, the MPCP approach cannot be applied because its schedulability analysis algorithm is based on single-processor schedulability analysis.

# Chapter 9

# Future Research Directions

In this thesis, we have presented an integrated, unified framework of fixed-priority end-to-end scheduling. By no means is the work on the end-to-end scheduling complete. The future work involves extending the system model, removing certain constraints and dealing with practical situations. In this chapter, we discuss the future work along these three lines.

## 9.1 General Subtask Precedence Constraints

Throughout the thesis, we assume that the precedence constraint among subtasks of a same parent task is linear, and in many cases we suggest that no two adjacent subtasks should execute on the same processor. In practice, these assumptions may not be true. We now discuss what impact it has when these assumptions are removed and how the algorithms presented in this thesis can be modified to deal with these general cases.

### 9.1.1 Multiple Adjacent Subtasks Execute on the Same Processor

The end-to-end scheduling algorithms proposed in this thesis do not prohibit adjacent subtasks from executing on the same processor. However, the performance will not be satisfactory. For example, suppose that task $T_1$ has two subtasks, $T_{1,1}$ and $T_{1,2}$, on processor $P_1$. Task $T_2$ has one subtask $T_{2,1}$ also on $P_1$. $T_1$ and $T_2$ has the same period, but subtask $T_{2,1}$ has a higher priority than $T_{1,1}$ and $T_{1,2}$. According to the end-to-end scheduling algorithms, the bound on the end-to-end response time of $T_1$ is equal to $\tau_{1,1} + \tau_{1,2} + 2 \times \tau_{2,1}$ (assuming that the period of $T_1$ and $T_2$ is far greater than $\tau_{1,1} + \tau_{1,2} + 2 \times \tau_{2,1}$). Obviously, this bound is pessimistic because in one

period $T_{2,1}$ cannot preempt both $T_{1,1}$ and $T_{1,2}$. If we use the DS protocol, the actual worst-case end-to-end response time of $T_1$ is no more than $\tau_{1,1} + \tau_{1,2} + \tau_{2,1}$. The reason that end-to-end scheduling algorithms do not yield a satisfactory upper bound in this case is that the subtasks of $T_1$ are treated separately by the the schedulability analysis algorithms. The interference from subtask $T_{2,1}$ is counted into the upper bounds of both $T_{1,1}$ and $T_{1,2}$, and is then counted multiple times in the final upper bound on the end-to-end response time of $T_1$.

In general, when adjacent subtasks in a task execute on the same processor, we refer to all the subtasks as a *subtask group* of the parent task. The model used in earlier chapters can be viewed as a special case where each subtask group has only one subtask. We first argue that the DS protocol should be used for synchronizing the execution of subtasks within a subtask group. The DS protocol has a low overhead because the synchronization signal is sent within the processor and can be easily achieved by a semaphore operation. In addition, if we use an appropriate schedulability analysis algorithm, we will obtain a tighter bound on the response time of the whole subtask group.

Harbour et al. [49] proposed a schedulability analysis algorithm for such a case. In their model, a single-processor system has a set of periodic tasks, each of which is a chain of subtasks which may have different priorities. The DS protocol is used to synchronize the execution of subtasks. The algorithm computes an upper bound on the response time of each task. The bound is much tighter than the bound computed by Algorithm SA/DS, because Harbour's algorithm treats each task as a whole rather than a collection of individual subtasks.

Harbour's algorithm can be readily used in our new situation where adjacent subtasks execute on the same processor. It can be used to compute upper bounds on the response times of subtask groups, if each subtask group is released periodically. This calls for the synchronization protocols with execution control to synchronize the execution among subtask groups. The PM, MPM and RG protocols can serve for this purpose, while the SS protocol cannot because subtasks in a subtask group have different priorities and cannot be handled by a single sporadic server. For example, if we use the RG protocol to synchronize the execution among subtask groups, we have a released guard for each subtask group. The release guard controls the release of the first subtask in the subtask group. Inside the subtask group, we use the DS protocol to synchronize the execution of subtasks. We thus control the release of subtasks hierarchically at two levels. Details of this approach need to be further explored more thoroughly.

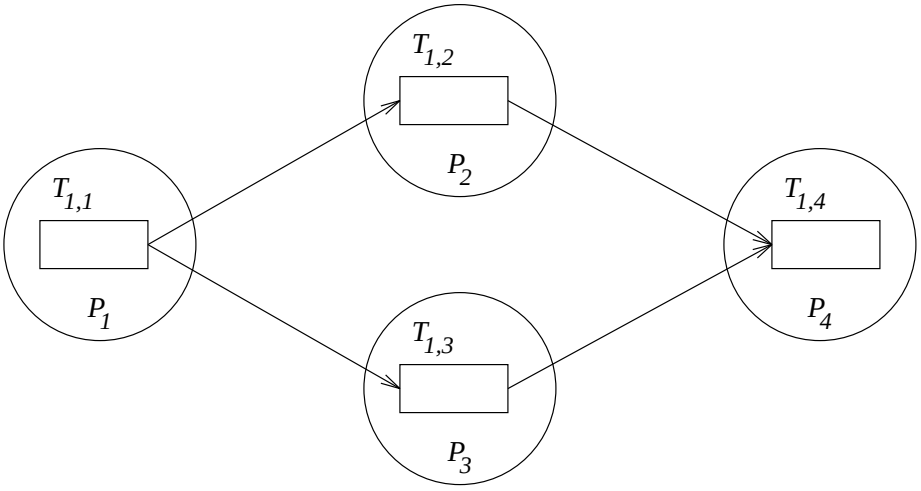



**Figure 9.1**: The Precedence Graph of Subtasks is a DAG

### 9.1.2 The Precedence Graph is a Directed Acyclic Graph (DAG)

We now focus on the case where the precedence constraint among subtasks is not linear. The precedence graph in general is a directed acyclic graph (DAG). An example of such a task is shown in Figure 9.1.

If a subtask has two predecessors, such as subtask $T_{1,4}$ in Figure 9.1, then the subtask can be either an *AND subtask* or an *OR subtask*, a case which was studied by Gillies and Liu [55]. If $T_{1,4}$ is an AND subtask, then an instance of $T_{1,4}$ cannot be released until both corresponding instances of $T_{1,2}$ and $T_{1,3}$ complete. Otherwise, if $T_{1,4}$ is an OR subtask, an instance of $T_{1,4}$ can be released when one of the corresponding instances of $T_{1,2}$ and $T_{1,3}$ completes. When a subtask has more than two predecessors, the precedence constraints between the subtask and its predecessors can be a combination of AND/OR relations, and scheduling such tasks becomes much more complicated.

The generalized DAG precedence graph has impact on all aspects of the end-to-end scheduling. For example, suppose that we want to apply the PM protocol to the system in Figure 9.1. If the precedence relation between $T_{1,2} \rightarrow T_{1,4}$ and $T_{1,3} \rightarrow T_{1,4}$ is an and-relation, the phase of $T_{1,4}$ needs to be adjusted to $R_{1,1} + \max\{R_{1,2}, R_{1,3}\}$, where $R_{i,j}$ is an upper bound on the response time of $T_{i,j}$. Otherwise, the phase of $T_{1,4}$ should be adjusted to $R_{1,1} + \min\{R_{1,2}, R_{1,3}\}$. The schedulability analysis algorithm for the PM protocol should be modified accordingly. How

priorities should be assigned in this case is also a challenging problem. The PDM and NPDM methods cannot be readily applied here anymore.

The most complicated case is when adjacent subtasks execute on the same processor and the precedence graph is a DAG. It is not clear what approach may be effective and how applicable the solutions to the previous two cases can be in this case.

## 9.2 Hybrid End-to-End Scheduling

In this thesis, we assume that fixed priority scheduling of subtasks is used on very processor. In addition, certain scheduling support (such as timer interrupts, synchronization signals, and a scheduler on each processor) is assumed to be available, While these assumptions generally hold for scheduling tasks on CPU's, they may not hold when processors model networks. The FDDI network is a good example. Suppose that a real-time system consists of several computers which are connected through an FDDI network. The network can be modeled as a processor, so that the whole system fits the end-to-end system model. However, a fixed priority scheduling algorithm cannot be used on this "network" processor. Furthermore, the underlying scheduling support, such as timer interrupts, release guard variables, and etc., does not exist on the "network" processor either. Nevertheless, since timing guarantee can be provided by the FDDI network [51], the whole system can be used to schedule tasks with end-to-end deadlines. In general, we call scheduling of such a system, where different scheduling paradigms may be used on different processors, the *hybrid end-to-end scheduling.*

Some end-to-end scheduling algorithms proposed in this thesis can be extended and applied to hybrid end-to-end scheduling. In the previous example with the FDDI network, many scheduling algorithms in this thesis can be modified and become applicable. For example, suppose that we want to implement the Release Guard protocol. We first store the release guard for each "message" subtask on the "network" processor on the CPU where the sender of the message executes. The scheduling of a "message" subtask becomes naturally the job of the scheduler which schedules the sender subtask. In the schedulability analysis, we can obtain an upper bound on the response time of a "message" subtask by the guarantee provided by the FDDI network and an upper bound on the response time of an CPU execution subtask by the busy period analysis. The sum of the upper bounds on the response time of all its subtasks is an upper

bound on the end-to-end response time of a task. Many more issues on how to extend end-to-end scheduling algorithms and apply them to hybrid end-to-end scheduling need to be studied.

## 9.3 End-to-End Scheduling with Dynamic Priorities

In this thesis, we assume that subtasks are scheduled on a fixed-priority basis. Fixed-priority scheduling implies that every instance of a subtask has the same priority and the priority remains fixed from the release of a subtask instance till its completion. An alternative is to use a dynamic-priority scheduling algorithm where instances of a same subtask may have different priorities and the priority of a subtask instance may change during the course of its execution. (In practice, the second case of the dynamic-priority scheduling rarely happens because of the overhead of such an algorithm.) The first case of the dynamic-priority scheduling, such as the *Earliest Deadline First (EDF)* [1], is commonly used on a single-processor system.

In a distributed real-time system, Kao et al. [11, 12] have studied various issues in the context where tasks have soft (end-to-end) deadlines and are scheduled on a dynamic-priority basis. Many problems arise when tasks have hard end-to-end deadlines. Under certain circumstances, the scheduling of end-to-end tasks on a dynamic-priority basis can be simple, and the performance is relatively satisfactory. For example, suppose that every task in a system has $n_i$ subtasks, and its end-to-end deadline is equal to $n_i$ times of the period. We can use one of the synchronization protocols with execution control to schedule such a system. Each subtask is assigned a local deadline equal to the period of its parent task, and subtasks are scheduled on an EDF basis. It is easy to verify that as long as the utilization on each processor is less 1, each subtask will meet its assigned local deadline, and hence each task will meet its end-to-end deadline. More general cases of dynamic-priority, end-to-end scheduling need further investigation.

# Appendix A

# Sporadic Server Algorithms

The notion of Sporadic Server was first introduced by Sprunt et al. [52, 56]. Like a periodic task, a sporadic server is specified by its period and its (maximum) execution time. We say that a periodic task and a sporadic server have the same size if they have the same period and the same execution time. A sporadic server can respond to sporadic requests as promptly as possible while still "behaving" like a periodic task so that the schedulability analysis for periodic tasks can be directly applied to a system with sporadic servers. This concept can lead to many different implementations, each of which we call a *sporadic server algorithm*. A sporadic server algorithm must be correct, i.e., if a system with sporadic servers is verified schedulable when servers are treated as periodic tasks, then under no circumstances will a periodic task miss a deadline.

In this appendix, we will focus on single-processor systems with periodic tasks and sporadic servers. We first show that the sporadic server algorithm proposed by Sprunt in [56] is not correct. A sporadic server implemented according to that algorithm is too aggressive in competing for the processor, and lower priority tasks may miss their deadlines while they would not if the server were indeed a periodic task with the same size. We then introduce the basic structure of a sporadic server, which lays the ground for our further discussion. We proceed to illustrate that the sporadic server consists of a family of algorithms, which are differentiated by their aggressiveness in competing for the processor. Four algorithms are presented here, each of which has different complexity, overhead, and has different aggressiveness in competing for the processor.

## A.1 Background

One common way to control the behavior of sporadic servers is to have an *execution time budget*, or *budget*, for each server. A sporadic request can be served only when the server has (nonzero) budget. The budget gets consumed when the server serves requests and gets replenished at some time later. The following two rules regarding budget replenishment are copied from [56].

1. (a) If the server has execution time available, the replenishment time, $RT$, is set when priority level $P$ becomes active. (b) If, on the other hand, the server capacity has been exhausted, then $RT$ cannot be set until the SS's capacity becomes greater than zero and $P$ is active. In either case, the value of $RT$ is set equal to the current time plus the period of the SS.

2. The amount of execution time to be replenished can be determined when either (a) the priority level of the SS, namely $P$, becomes idle or (b) when the SS's available execution time has been exhausted. The amount to be replenished at $RT$ is equal to the amount of server execution time consumed since the last time at which the status of $P$ changed from idle to active.

In the description, priority level $P$ is said to be active at time $t$ if and only if time $t$ is within a $P$-level busy period, a notion we discussed in Chapter 5. Otherwise, the priority level $P$ is said to be idle.

To show that a sporadic server implemented this way has worse interference to lower priority tasks than a periodic task with the same size, and therefore is not correct, we consider a simple system containing three tasks. $T_1$ is a periodic task; it has the highest priority. $T_s$ is a sporadic server, and it has the second-highest priority. $T_2$ is a periodic task with the lowest priority. Their periods are 3, 4, and 8 respectively. Their execution time (budget) are 1, 1 and 3 respectively. If $T_s$ were a periodic task, the schedule of the first 8 time units is shown in Figure A.1, assuming that all tasks have zero phases. From the figure, we can see that task $T_2$ is schedulable.

In Figure A.1 time 0 is a critical instant for task $T_2$. If $T_s$ is implemented according to a correct sporadic server algorithm, we can conclude that $T_2$ must be schedulable. This is not true with the implementation proposed by Sprunt et al. in [56]. A schedule of the system is shown in Figure A.2, where $T_s$ is implemented according to the rules described above. In this figure,

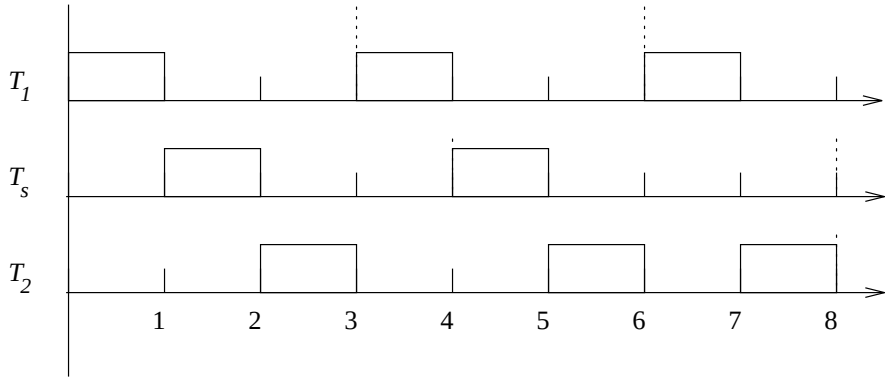



**Figure A.1**: The Schedule if $T_s$ Were a Periodic Task

there is a sporadic request of 0.5 time unit at time 0 and another request of 2 time units at time 4. We use a 2-tuple $(t, a)$ to represent a request, where $t$ is the time the request arrives and $a$ is the amount of execution time the request needs. We use a list of 2-tuples to represent all the requests sorted in the order of their arrival times and call this list the *request list*. In this case, the request list is $[(0, 0.5), (4, 2)]$. In Figure A.2, a solid arrow indicates the arrival of a request, and the number by the arrow is the execution time needed. A dotted arrow represents a budget replenishment, and the number by the arrow is the amount of the budget replenishment.

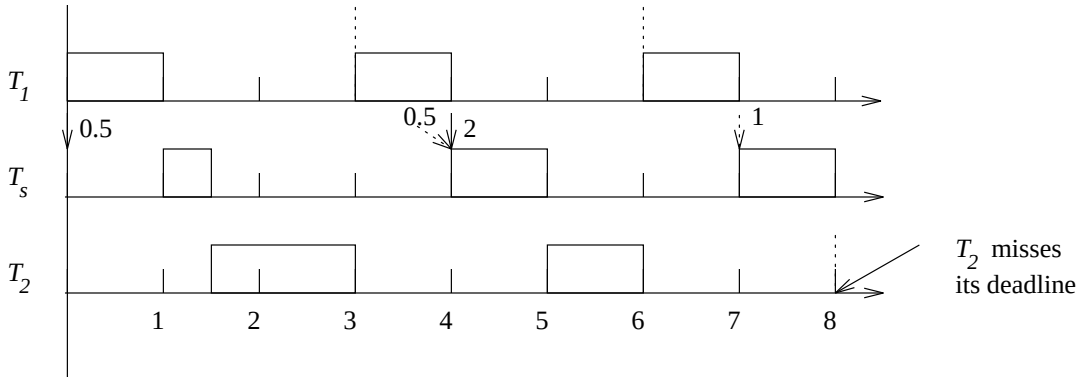



**Figure A.2**: A Schedule of the Example System

1. At time 0, $T_s$ has the full budget and priority level $P$ becomes active. Thus rule 1(a) is satisfied.[1] The next replenishment time is set to time 4.

[1]If this argument is not convincing, we can let $T_2$ be released at time $0 - \varepsilon$ for some very small $\varepsilon > 0$, so that the condition of rule 1(a) is satisfied exactly.

2. At time 1.5, the priority level $P$ becomes idle, and hence we can decide the amount of execution time to be replenished for the next replenishment. The amount is equal to the execution time consumed by $T_s$ from time 0 to time 1.5, which is equal to 0.5 time unit.

3. At time 3, $T_s$ still has a budget of 0.5 time unit, and the priority level $P$ becomes active. Again, rule 1(a) is satisfied. Hence a second replenishment time is set to time 7.

4. At time 4, the budget of $T_s$ is replenished to 1 time unit. Therefore $T_s$ is able to serve the second sporadic request till time 5, when it runs out of budget.

5. At time 5, since $T_s$ runs out of budget and the priority level $P$ becomes idle, both rule 2(a) and 2(b) are satisfied. We determine the amount of budget to be replenished for the replenishment at time 7, which is equal to 1 time unit.

6. At time 6, although priority level $P$ becomes active, $T_s$ does not have any available budget. Therefore no rule is satisfied.

7. At time 7, the execution budget of $T_s$ is replenished by 1 time unit and $T_s$ can serve the second sporadic request till time 8.

8. At time 8, task $T_2$ misses its deadline, since it still has 0.5 time unit to execute.

From this example, we see clearly that the sporadic server algorithm proposed in [56] is not correct. The replenishment rules in this algorithm are too "greedy."

## A.2 The Structure of a Sporadic Server

Again, like a periodic task, each sporadic server has a period $s\_Period$, an execution time $s\_ExecTime$, and a fixed priority $s\_Priority$. A correct implementation of a sporadic server allows us to analyze the system schedulability by treating the sporadic server as a periodic task with the same parameters. By convention, the prefix $s\_$ indicates that the variable is a state variable of a sporadic server. If the following letter is a capital letter, then the state variable is "public," a term borrowed from C++ language which implies that the scheduler or other entity in the system can query this state variable. On the other hand, if the following letter is not a capital letter, then the state variable is "private," meaning that only the sporadic server itself

is aware of its existence. Later on, when we introduce event handlers (functions) of a sporadic server, we follow the same convention.

A sporadic server has a *request queue* that holds sporadic requests it will handle. For now, we assume that when a request arrives, we know the exact amount of execution time the request needs. Later on, we will extend our algorithms to the general case where the execution time is not known when the request arrives. As a consequence of this assumption, we know the total amount of execution time needed by all outstanding requests in the request queue at any time. Let $s\_Req$ denote this amount. A sporadic server also has a budget, which is the maximum amount of time it can execute without causing "troubles" to other tasks. Let $s\_Budget$ denote the budget at the current time. The *scheduling rules of a sporadic server* are straightforward.

1. When $s\_Req$ and $s\_Budget$ are both greater than 0, the sporadic server is put in the ready queue and competes for the processor according to its priority $s\_Priority$.

2. After a sporadic server executes for $\delta$ time units, both $s\_Req$ and $s\_Budget$ are decremented by $\delta$ time units.

3. Whenever $s\_Req$ or $s\_Budget$ becomes 0, the sporadic server is removed from the ready queue.

In this appendix, we assume that the scheduler enforces these rules.

Clearly, to be able to treat a sporadic server as a periodic task in the schedulability analysis, we need to adjust $s\_Budget$ wisely so that the interference of the sporadic server to a lower-priority task is no worse than an equivalent periodic task. The adjustment of $s\_Budget$, and in general the change of any state variable of a sporadic server, are done through *event handlers* or *functions* of the sporadic server, which are invoked by the scheduler when certain events happen. A sporadic server typically includes the following functions. It is possible that a specific algorithm may not implement all the functions, and it is also possible that the algorithm may implement additional functions. We will mention the additional functions when such a situation arises.

$s\_RequestArrive(amount)$ : When a sporadic request arrives, the scheduler notifies the sporadic server by calling this function. The argument $amount$ is the amount of execution time the request needs.

$s\_ReplenishBudget(amount)$ : By calling this function, the scheduler notifies the sporadic server that it should increase its budget by $amount$ time units at the current time.

$s\_RequestFinished()$ : The scheduler calls this function to notify the sporadic server that $s\_Req$ becomes 0.

$s\_NoBudget()$ : The scheduler calls this function to notify the sporadic server that $s\_Budget$ becomes 0.

In order to implement these functions properly, the sporadic server requests services from the scheduler as well. We describe these services in the form of functions. The most important function provided by the scheduler is $sched\_RequestReplenishment(t, a)$, which is called when a sporadic server needs to schedule a budget replenishment with amount $a$ at time $t$. The other service provided the scheduler is $sched\_CurrentTime()$, which returns the current time. For convenience, we use $CurrTime$ as an alias of this function. Some sporadic server algorithms may request additional service from the scheduler. We will mention them when such a situation occurs.

## A.3 Algorithm SS1

In this section and the next two sections, we will describe three sporadic server algorithms. All three algorithms assume that the computation time is known when a request arrives. The difference comes from different implementation of the event handlers and thus different aggressiveness in competing for the processor.

One way to ensure that a sporadic server can be treated as a periodic task is to let the sporadic server mimic the behavior of a periodic task. For a periodic task, the inter-release time between two consecutive instances is no shorter than the task period, and the execution time of each instance is no greater than the maximum execution time of the task. In the case of a sporadic server, its being put in the ready queue (when both $s\_Req$ and $s\_Budget$ are greater than 0) is equivalent to the release of an instance of a periodic task. The server being removed from the ready queue (when $s\_Budget = 0$ or $s\_Req = 0$) is equivalent to the completion of an task instance. To mimic the behavior of a periodic task, we need to ensure that a sporadic server is not put in the ready queue more frequently than specified by its period and that each time

the server executes it never executes longer than its maximum execution time. Assuming that a sporadic server initially has the full budget equal to its execution time, the following two rules make a sporadic server satisfy the above requirement.

- When a server is put in the ready queue, we schedule a budget replenishment at the current time plus the period of the server.
- When a server is removed from the ready queue, we discard all remaining budget, i.e., we set $s\_Budget = 0$.

Algorithm SS1 is based on the idea described in the previous paragraph. The pseudo-code of its event handlers are listed in Figure A.3 in C style. In addition to the event handlers discussed in the previous section, we also include $s\_Init()$, which is invoked only once when the system starts up. It initializes the state variables of a sporadic server.

According to Algorithm SS1, the schedule of the previous simple system for the same request list used in Figure A.2 is shown in Figure A.4. Obviously, task $T_2$ meets its deadline.

1. At time 0, the first request arrives, and $s\_RequestArrive(0.5)$ is called. A budget replenishment with amount equal to 1 is scheduled at time 4 by calling $sched\_RequestReplenishment(4, 1)$. $s\_Req$ becomes 0.5.
2. At time 1.5, the first request is completed; $s\_RequestFinished()$ is called. The budget $s\_Budget$ drops to 0.
3. At time 4, request $(4, 2)$ arrives, and the first replenishment is due. Two functions of the server are invoked, and they can be invoked in any order. As a net effect of these two function calls, a budget replenishment of 1 time unit is scheduled at time 8, $s\_Req = 2$, and $s\_Budget = 1$. By the scheduling rules of a sporadic server, the server is put in the ready queue and starts to compete for the processor again.
4. At time 5, the budget runs out, and the server is removed from the ready queue. The server does nothing.
5. At time 7.5, the first instance of task $T_2$ finishes in time. The processor becomes idle.
6. At time 8, the second replenishment is due, $s\_Budget$ becomes 1, and the server continues to serve the second request.

```
1. s_Init() :
     s_Budget = s_ExecTime;
     s_Req = 0;
2. s_RequestArrive(amount) :
     if (s_Req==0 && s_Budget >0)
       // the server will be put in the ready queue
       sched_RequestReplenishment(CurrTime + s_Period, s_ExecTime);
     s_Req = s_Req + amount;
3. s_ReplenishBudget(amount) :
     s_Budget = s_Budget + amount;
     if (s_Req > 0 && s_Budget==0)
       // the server will be put in the ready queue
       sched_RequestReplenishment(CurrTime + s_Period, s_ExecTime);
4. s_RequestFinished() :
     // the server is removed from the ready queue
     s_Budget = 0;
```

**Figure A.3**: The Pseudo-Code of Event Handlers in Algorithm SS1

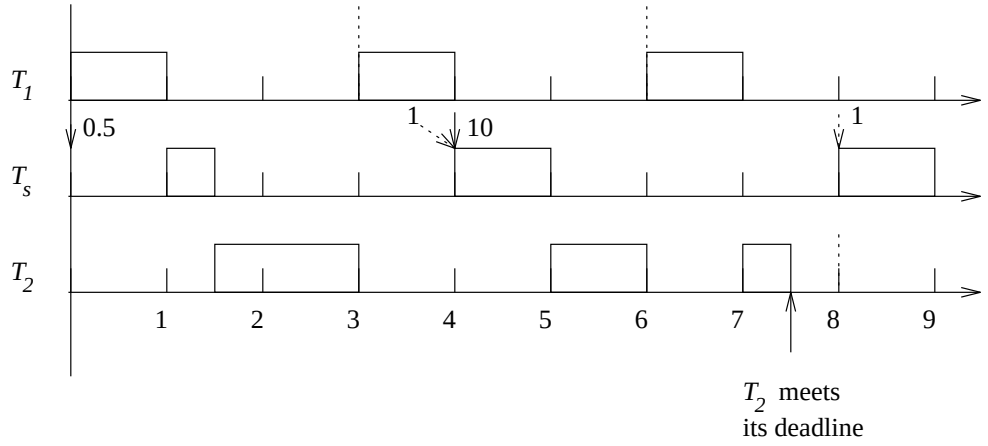



**Figure A.4**: The Schedule of the Simple System According to Algorithm SS1

If a sporadic server is implemented according to Algorithm SS1, we see that the interval between any two budget replenishments is no shorter than $s\_Period$, and the amount of time replenished is equal to $s\_ExecTime$. This limits the execution capability of a sporadic server to be equal to a periodic task with the same size and the same priority. The only visible difference between a sporadic server implemented this way and a periodic task with the same size is when the response time of an instance of the periodic task is longer than the period, multiple instances of the periodic task can appear in the ready queue. In the case of a sporadic server, the server keeps accumulating the budget, through function $s\_ReplenishBudget(amount)$, instead of having multiple instances. The net interference to another periodic task with a lower priority is the same in both cases.

One obvious shortcoming of Algorithm SS1 is that it drops the budget to zero after one request is served. The unused budget could be used to improve the response time of later requests. For example, if the request list is $[(0, 0.5), (2, 0.5), (4, 2)]$ for the simple system, the completion time of the second request will be 4.5, as shown in Figure A.5. When the second request arrives at time 2, the server cannot start to execute because the server has no budget. However, the server "deserves" 1 time unit in its first period while it only uses 0.5 time unit. It should be fine for the server to serve the second request for another 0.5 time unit at time 2, and hence completes the request at time 2.5 - much sooner than time 4.5 shown in Figure A.5. On the other hand, we cannot simply remove function $s\_RequestFinished()$ in Algorithm SS1, because the algorithm may not be correct if we do so. For example, suppose that we remove $s\_RequestFinished()$ in Algorithm SS1 and change the phase of $T_2$ to time 2. We find that the first instance of $T_2$ will miss its deadline. Other cares need to be taken if we want to preserve the unused budget in a sporadic server. This motivates us to investigate the second version of the sporadic server algorithm, Algorithm SS2.

## A.4 Algorithm SS2

Unlike Algorithm SS1 where we let a sporadic server to mimic the behavior of a periodic task, the idea behind Algorithm SS2 is to control the time demand generated by a sporadic server such that it is no more than the time demand generated by a periodic task with the same size. We can thus safely treat a sporadic server as a periodic task in the busy period analysis.

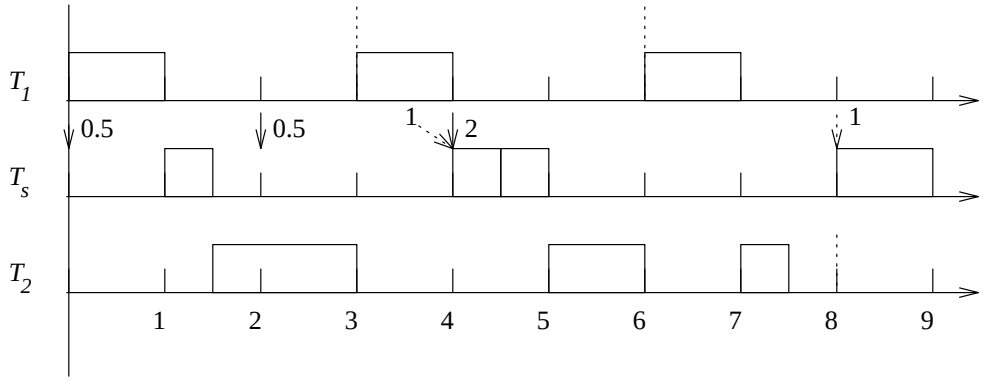



**Figure A.5**: Another Schedule of the Simple System According to Algorithm SS1

By definition, a piece of time demand generated by a task is a certain amount of time the scheduler must let the task execute. In the case of a sporadic server, the exact amount of time the scheduler must let the server execute is equal to the minimum of the total requests and the server budget. Hence this is the time demand generated by a sporadic server. It is generated when both the total request and the server budget become greater than zero. Specifically, the time demand of a sporadic server is generated under one of the following two situations.

**a request arrives and $s\_Budget > s\_Req$ :** If the request needs $a$ time units, the amount of this time demand is equal to $\min\{a, s\_Budget - s\_Req\}$.

**a budget replenishment is due and $s\_Req > s\_Budget$ :** If the amount of budget replenishment is $a$ time units, the amount of this time demand is equal to $\min\{a, s\_Req - s\_Budget\}$.

According to the busy period analysis we discussed in Chapter 5, we can treat a sporadic server as a periodic task if during any interval $[t_0, t_0 + t)$ the time demand generated by the server is no more than $\lceil t/s\_Period \rceil s\_ExecTime$. A simple rule can ensure such a constraint to be satisfied : If a piece of time demand with $a$ time units is generated by the sporadic server at time $t$, the server budget will be replenished at time $t + s\_Period$ for the same amount. Given the fact that the server initially has the budget equal to its execution time, the total amount of time demand a sporadic server can generate in interval $[0, s\_Period)$ is no greater than $s\_ExecTime$. An induction shows that in any interval $[t, t + s\_Period)$ the server cannot generate more than $s\_ExecTime$ units of time demand. A mathematic formation gives that the time demand generated in any interval $[t_0, t_0 + t)$ by the server is no more than $\lceil t/s\_Period \rceil s\_ExecTime$. Hence

```
1. s_Init()
     s_Req = 0;
     s_Budget = s_ExecTime;
2. s_RequestArrive(amount)
     if (s_Budget - s_Req >0)
       { demand = min{amount, s_Budget - s_Req};
         sched_RequestReplenishment(CurrTime + s_Period, demand);
       }
     s_Req = s_Req + amount;
3. s_ReplenishBudget(amount)
     if (s_Req - s_Budget > 0)
       { demand = min{amount, s_Req - s_Budget};
         sched_RequestReplenishment(CurrTime + s_Period, demand);
       }
     s_Budget = s_Budget + amount;
```

**Figure A.6**: The Pseudo-Code of Event Handlers in Algorithm SS2

a sporadic server implemented this way can be treated as a periodic task in the schedulability analysis.

The pseudo-code of Algorithm SS2 is listed in Figure A.6, which implements the idea we discussed in the previous paragraph. Notice that in Algorithm SS2 we do not provide the function $s\_RequestFinished()$, where the unused budget is dropped in Algorithm SS1.

Figure A.7 shows the schedule of the simple system, where the server is implemented according to Algorithm SS2 and the request list is $[(0, 0.5), (2, 0.5), (4, 2)]$. We notice that the second request completes by time 2.5, much earlier than 4.5 obtained in Figure A.5. In the schedule, two pieces of time demand are generated by the sporadic server at time 0 and time 2, each of which has 0.5 time unit. As a consequence, two budget replenishments are set at time 4 and time 6, which in turn trigger the generation of another two pieces of time demand. So on and so forth.

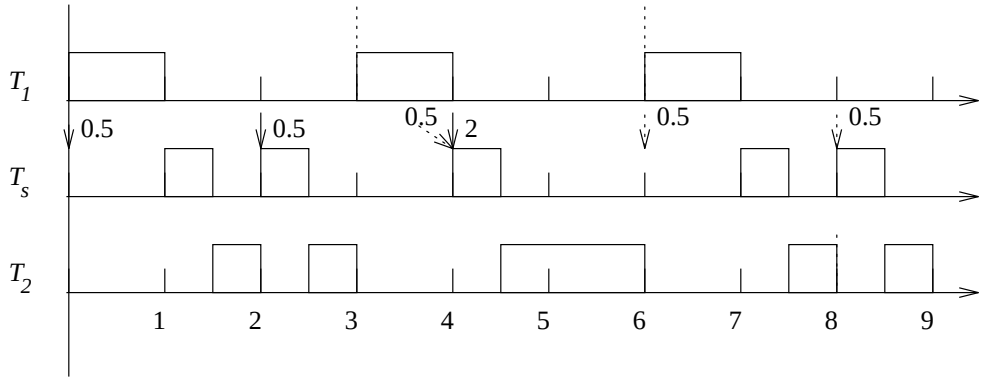
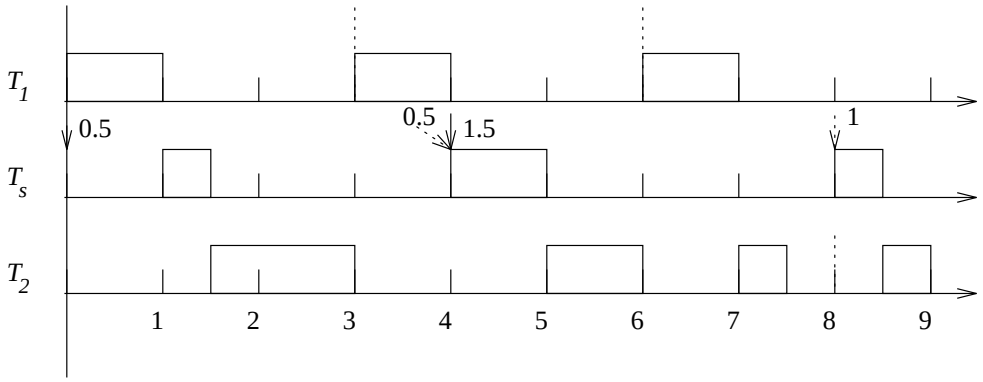



**Figure A.7**: The Schedule of the Simple System According to Algorithm SS2

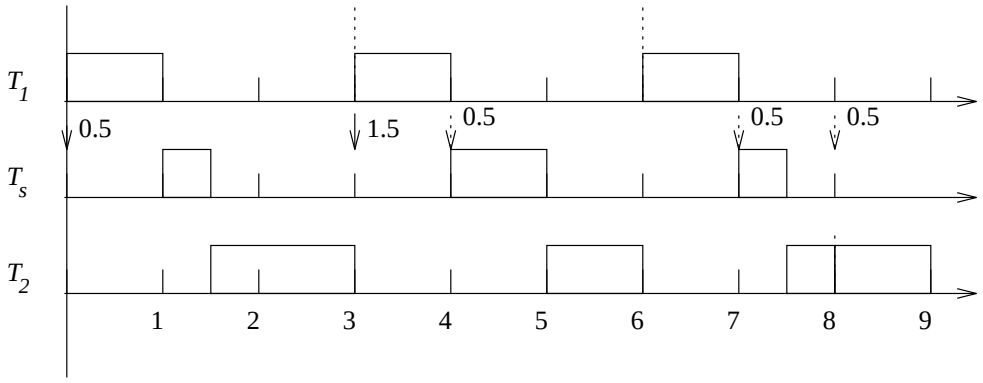
## A.5 Algorithm SS3

Algorithm SS2 can be further improved. Figure A.8 shows two schedules of the simple system with respect to two request lists, $[(0, 0.5), (4, 1.5)]$ and $[(0, 0.5), (3, 1.5)]$. The sporadic server $T_2$ is implemented according to Algorithm SS2. The difference between the two request lists is the arrival time of the second request. The two schedules are identical until time 7, when in Figure A.8(b) the sporadic server gets a budget replenishment of 0.5 time unit and completes the second request at time 7.5, while in Figure A.8(a) the sporadic server does not get a budget replenishment until time 8 and completes the second request at time 8.5. From the viewpoint of $T_2$, the second request arriving at time 3 is no different from its arriving at time 4, because $T_2$ is not able to resume its execution until time 5 in either way. As a consequence, when the second request arrives at time 4 in Figure A.8(a), a more aggressive sporadic server will request two separate budget replenishments, like those in Figure A.8(b), instead of only one at time 8. In general, when a request arrives inside a $\phi_s$-level busy period, where $\phi_s$ is the priority of the sporadic server, the server can schedule more aggressive budget replenishments by treating this request as if it arrives at the beginning of the busy period. This is the idea behind Algorithm SS3.

To be able to "move" a request to the beginning of the busy period and determine the exact amount of time demand generated due to the arrival of this request, we need to know the budget in the past. For this purpose, we keep a history of *free budget* of the server from the beginning of the busy period till the present time, where at any time the free budget of a sporadic server is equal to zero or the server budget minus the total amount of outstanding requests, whichever




(a)




(b)

**Figure A.8**: Two Schedules of the Simple System According to Algorithm SS2

is greater. In other words, the free budget is equal to $\max\{0, s\_Budget - s\_Req\}$. We define the history of the free budget of a sporadic server as a free budget function, $s\_fb(t)$, which returns the amount of the free budget the server has at time $t$ for $t \leq CurrTime$. Take Figure A.8(a) for example. Suppose that the current time is time 4. The current $\phi_s$-level busy period starts from time 3. By definition, we have $s\_fb[3,4) = 0.5$, where $s\_fb[3,4) = 0.5$ is a short notion for $s\_bf(t) = 0.5$ for $t \in [3,4)$. $s\_fb(4)$ is increased to 1 at time 4, due to the first budget replenishment. The free budget function is a non-decreasing staircase function.

Based on the free budget function, we can set the budget replenishments in the following way. Suppose that a $\phi_s$-level busy period starts from time 0, and a request with $a$ time units arrives at the current time. If $t_1$ is the first time instant in $[0, CurrTime)$ such that $s\_fb(t_1)$ is greater than 0, then we declare that a piece of time demand is generated at time $t_1$ with the amount equal to $\min\{s\_fb(t_1), a\}$. Let $d = \min\{s\_fb(t_1), a\}$. A budget replenishment with $d$ time units is scheduled at time $t_1 + s\_Period$. Since $d$ time units of the free budget are used at time $t_1$, the value of $s\_fb(t)$ for $t_1 \leq t \leq CurrTime$ should decrease by $d$ correspondingly. Similarly, the request amount $a$ decreases by $d$ time units as well. If $a$ is still greater than zero, we search for the next time instant $t_2$ where $s\_fb(t_2)$ $(t_1 < t_2 \leq CurrTime)$ is greater than 0, and repeat the above process again.

To facilitate the description of Algorithm SS3, we define a set of operations on the free budget function of a sporadic server. Two of them, $s\_fb(t)$ and $s\_fbFirstNonZeroPoint()$, are queries that do not change the function itself, while other operations may change the free budget function. Figure A.9 defines these operations. For operations that change the free budget function, we use $f(t)$ to denote the free budget function before the mutation and $f'(t)$ to denote the one after the mutation.

Figure A.10 lists the pseudo-code of Algorithm SS3. We define the *priority level of the system* to be equal to the priority of the current executing task or to an arbitrary small value if there is no task currently executing. In Algorithm SS3, we introduce a new service function provided by the scheduler,$sched\_SystemPriority()$, which returns the current priority level of the system. One additional event handler of the sporadic server is also added, $s\_BusyPeriodStarts()$. This function is called by the scheduler when the system priority is changed from one below $s\_Priority$ to a priority that is higher than or equal to $s\_Priority$.

$s\_fb(t)$ :

return the free budget at time $t$.

$s\_fbFirstNonZeroPoint()$ :

return $\min\{t|f(t) > 0\}$.

$s\_fbInit()$ :

$f'(t) = 0$ for all $t$.

$s\_fbIncrease(time, amount)$ :

$f'(t) = f(t)$ for $t < time$, and $f'(t) = f(t) + amount$ for $t \geq time$.

$s\_fbDecrease(time, amount)$ :

$f'(t) = f(t)$ for $t < time$, and $f'(t) = f(t) - amount$ for $t \geq time$.

$s\_fbCollapse(time)$ :

$f'(t) = 0$ for $t < time$, and $f'(t) = f(t)$ for $t \geq time$.

**Figure A.9**: Description of Operations on the Free Budget Function

```
1. s_Init() :
     s_Req = 0; s_Budget = s_ExecTime;
     s_fbInit(); s_fbIncrease(CurrTime,s_ExecTime);
2. s_RequestArrive(amount) :
     s_Req = s_Req + amount;
     if (sched_SystemPriority() < s_Priority)      // a possible starting
       s_fbCollapse(CurrTime);                     // point of a busy period
     for (; amount > 0 && s_fb(CurrTime) > 0; )
       { t = s_fbFirstNonZeroPoint();
         assert ( t <= CurrTime );
         demand = min{s_fb(t), amount};
         if (t + s_Period < CurrTime)
           { s_fbIncrease(t + s_Period, demand);  s_Budget = s_Budget + demand; }
         else
           sched_RequestReplenishment(t + s_Period, demand);
         s_fbDecrease(t,demand);
         amount -= demand;
       }
3. s_ReplenishBudget(amount) :
     if ( s_Req > s_Budget )
       { demand = min{amount, s_Req - s_Budget};
         sched_RequestReplenishment(CurrTime + s_Period, demand);
         amount1 = amount - demand;
       }
     s_fbIncrease(CurrTime,amount1);
     s_Budget = s_Budget + amount;
4. s_BusyPeriodStart() :
     s_fbCollapse(CurrTime);
```

**Figure A.10**: The Pseudo-Code of Algorithm SS3

As an example, suppose that the sporadic server is implemented according to Algorithm SS3 in Figure A.8(a).

1. At time 3, the system priority changes from $\phi_2$ to $\phi_1$, and $s\_BusyPeriodStart()$ is called. As a result, time 3 becomes the first non-zero point in the free budget function, and $s\_fb(3) = 0.5$.

2. At time 4, a replenishment with 0.5 time unit is due, and $s\_ReplenishBudget(0.5)$ is called. $s\_fb(4)$ is increased to 1, while $s\_fb[3,4) = 0.5$.

3. At time 4, the second request with 2 time units arrives, and $s\_RequestArrive(2)$ is called. We first find that time 3 is the first non-zero point in the free budget function, and declare that a piece of time demand is generated at time 3 with amount equal to $\min\{s\_fb(3), 2\} = 0.5$. A budget replenishment is scheduled at time 7 with 0.5 time unit. We then decrease $s\_fb(t)$ by 0.5 for $t \in [3,4]$, and time 4 becomes the first non-zero point in the free budget function. Since we still have 1.5 time units of "undeclared" request, we repeat the above process and eventually schedule another budget replenishment with 0.5 time unit at time 8.

As a result, we will observe the same schedule shown in Figure A.8(b). In the above description, there are two events that happen at time 4, and the server handles the replenishment first. The result will be the same if the server handles the request arrival first.

## A.6 Scheduling Sporadic Requests with Unknown Execution Times (Algorithm SS4)

In the previous discussion, we assume that we know how much execution time a sporadic request needs. In practice, this may not be possible. In this section, we extend the previous sporadic server algorithms to deal with this case. In fact, the extended versions produce exactly the same schedules as the corresponding original sporadic algorithms. Thus the extended versions are generalizations of the previous sporadic server algorithms.

Since we do not know the request size, the sporadic server structure discussed in Section A.2 needs to be modified accordingly. We do not have $s\_Req$ in a sporadic server. Instead, we use a

Boolean variable, $s\_ReqFlag$, to indicate whether there is any outstanding request in the request queue. As a result, a server is put in the ready queue when $s\_ReqFlag$ is true and $s\_Budget$ is greater than 0. When a sporadic server executes, the scheduler decreases $s\_Budget$ based on how much time the sporadic server has executed and set $s\_ReqFlag$ false when there is no outstanding request in the server's request queue. The server is removed from the ready queue if the budget $s\_Budget$ is 0 or the request flag $s\_ReqFlag$ becomes false. The event handler $s\_RequestArrive(amount)$ also needs to be changed to $s\_RequestArrive()$ with no arguments.

Given this modification, we now examine the three algorithms we presented earlier. In the case of Algorithm SS1, we notice that we do not really need the execution time of a request, except when we accumulate the total amount of outstanding requests in $s\_ReqestArrive(amount)$. This accumulation is not necessary anymore according to the modified scheduling rules of a sporadic server. On two other occasions when we refer to $s\_Req$ in Algorithm SS1, we simply check if $s\_Req$ is 0. This can be replaced by checking if the variable $s\_ReqFlag$ is false. Thus Algorithm SS1 needs little change to deal with requests with an unknown computation amount.

We now examine Algorithm SS2 and SS3. It is non-trivial to extend either of them. We only focus on extending Algorithm SS3 and call the new extended version Algorithm SS4.

According to Algorithm SS3, when a request arrives or when a budget replenishment is due, a piece of time demand may be generated. Relying on the free budget function, we can move the time instant when a piece of time demand is generated to an earlier time instant, by treating every request arriving within a $\phi_s$-level busy period as if it arrives at the beginning of the busy period. Since we do not know the request size, we do not know the exact amount of the piece of time demand generated by the server. Thus the server cannot schedule a future budget replenishment on these two occasions as it does in Algorithm SS3. However, the amount of time demand generated is known or partially known only when all outstanding requests are processed or the budget runs out. These are the two occasions when a sporadic server can schedule a future budget replenishment according to Algorithm SS4.

Figure A.11 lists the pseudo-code of Algorithm SS4. In event handler $s\_RequestArrive()$, we simply set the request flag to the truth value. In $s\_ReplenishBudget(amount)$, we do a simple accounting of the free budget and the total budget. If a server runs out of budget, the total amount of time executed by the server from the beginning of the busy period till the current time is equal to $s\_fb(CurrTime)$. This is the total amount of time demand the server generated so

far in the current $\phi_s$-level busy period. Relying on the free budget function, we can properly set the replenishments in a similar way as we did in Algorithm SS3. Similarly, when all requests in a server are completed, the total amount of time demand generated from the beginning of the current $\phi_s$-level busy period is $s\_fb(CurrTime) - s\_Budget$, and we can replenish the same amount of time to the budget properly based on the free budget function. As a matter of fact, the code for these two functions are identical, and we put them together in Figure A.11. Interested readers can apply Algorithm SS4 to the previous example with respect to different request lists. The resultant schedules will be identical to the ones according to Algorithm SS3.

## A.7 Discussion

### Complexity and Overhead of Sporadic Server Algorithms

The complexity of a sporadic server algorithm can be measured by examining the code of its event handlers. In the case of Algorithm SS1 and SS2, we have $O(1)$ memory space and $O(1)$ computation time for each event handler. Algorithm SS3 and SS4 are more complicated, because they not only have iterations in event handlers but also manipulate complex data, the free budget function. Overall Algorithm SS3 and SS4 are more complex than the other two algorithms.

In addition to the algorithmic complexity, the overhead of a sporadic server algorithm also depends on how many event handlers are there and how often they are invoked. Algorithm SS1 has 4 event handlers; Algorithm SS2 has 3; Algorithm SS3 has 4; and Algorithm SS4 has 6 event handlers. In general, $s\_BusyPeriodStart()$ is invoked much more often than other event handlers. Thus, the overhead by the algorithms that use $s\_BusyPeriodStart()$, i.e., Algorithm SS3 and Algorithm SS4, is expected to be much higher than the other algorithms. Overall, we observe an increasing complexity and overhead in the order of Algorithm SS1, SS2, SS3 and SS4. As our discussion suggests, the performance of these algorithms is expected to increase in the same order.

### Processor Idle Point

In Chapter 4, we define that a time instant $t$ is a processor idle point if all task instances that are released by time $t$ have completed by time $t$. For the similar reason we gave for the Release Guard protocol, we can reset the budget of a sporadic server to a full budget at a processor idle

```
1. initialization :
     s_Req = 0;  s_Budget = s_ExecTime;
     s_fbInit();  s_fbIncrease(CurrTime,s_ExecTime);
2. s_RequestArrive() :
     s_ReqFlag = true;
3. s_ReplenishBudget(amount) :
     s_fbIncrease(CurrTime,amount);
     s_Budget = s_Budget + amount;
4-5. s_NoBudget(), s_RequestFinished() :
     demand = s_fb(CurrTime) - s_Budget;
     for (; demand > 0 ; )
       { t = s_fbFirstNonZeroPoint();
         d = min{demand, s_fb(t)};
         if (t+s_Period < CurrTime)
           { s_Budget = s_Budget + d; s_fbIncrease(t+s_Period,d); }
         else
           sched_RequestReplenishment(t+s_Period,d);
         demand = demand - d;
         s_fbDecrease(t,d);
       }
6. s_BusyPeriodStart() :
     s_fbCollapse(CurrTime);
```

**Figure A.11**: The Pseudo-Code of Algorithm SS4

point. Hence a sporadic server can achieve a better performance. For example, in Figure A.4, time 7.5 is an idle point. We can reset the server budget to 1, and the second request can complete 0.5 time unit earlier.

### Multiple Sporadic Servers in a System

Unlike slack stealing algorithms [57], which exploit the capacity of the system, sporadic server algorithms exploit the capacity of a periodic task to serve sporadic requests. As a consequence, multiple sporadic servers can co-exist in the same system without interference. Certain desirable attributes of sporadic servers still hold in this case. For example, if there are sufficient outstanding requests for every sporadic server, each sporadic server will execute for no less amount of time than a periodic task with the same size. When analyzing the schedulability of a periodic task whose priority is lower than multiple sporadic servers, we can treat all the servers as periodic tasks.

One potential usage of multiple sporadic servers is to hierarchically schedule multiple real-time applications and provide timing protection between them. We notice that the requests can be scheduled in an arbitrary order inside a sporadic server. Thus if a real-time system runs multiple real-time applications, each application can be "served" by a sporadic server. In a sense, applications partition the CPU resource based on the fixed-priority periodic task model. Inside each application, different scheduling algorithms can be used to schedule its own tasks. Timing protection between applications is achieved naturally by sporadic server algorithms.

# Bibliography


[1] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.

[2] J. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2:237–250, 1982.

[3] N. C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical Report YCS 164, Dept. of Computer Science, University of York, December 1991.

[4] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal of the British Computer Society*, 29(5):390–395, October 1986.

[5] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *IEEE Real-Time Systems Symposium*, pages 166–171, December 1989.

[6] J. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *11th IEEE Real-Time Systems Symposium*, pages 201–209, December 1990.

[7] N. Audsley, A. Burns, K. Tindell, M. Richardson, and A. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[8] K. W. Tindell, A. Burns, and A. J. Wellings. An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Systems*, 6(2):133–152, March 1994.

[9] K. Tindell, A. Burns, and A. Wellings. Allocating real-time tasks. An NP-hard problem made easy. *Real-Time Systems Journal*, 4(2), May 1992.

[10] J. Garcia and M. Harbour. Optimized priority assignment for tasks and messages in distributed hard real-time systems. In *The Third Workshop on Parallel and Distributed Real-Time Systems*, pages 124–132, April 1995.

[11] B. Kao and H. Garcia-Molina. Deadline assignment in a distributed soft real-time system. In *The 13th International Conference on Distributed Computing Systems*, pages 428–437, May 1993.

[12] B. Kao, H. Garcia-Molina, and B. Adelberg. On building distributed soft real-time systems. In *The Third Workshop on Parallel and Distributed Real-Time Systems*, pages 13–19, April 1995.

[13] R. Bettati. *End-to-End Scheduling to Meet Deadlines in Distributed Systems*. PhD thesis, University of Illinois at Urbana-Champaign, 1994.

[14] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Proceedings: The 10th International Conference on Distributed Computing Systems*, pages 116–123, May 1990.

[15] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[16] T. P. Baker. A stack-based resource allocation policy for real-time processes. In *11th IEEE Real-Time Systems Symposium*, pages 191–200, 1990.

[17] A. Burns, K. Tindell, and A. J. Wellings. Fixed priority scheduling with deadlines prior to completion. In *Sixth Euromicro Workshop on Real-Time Systems*, pages 138–142, June 1994.

[18] S. Chatterjee and J. Strosnider. Distributed pipeline scheduling: End-to-end analysis of heterogeneous, multi-resource real-time systems. In *The 15th International Conference on Distributed Computing Systems*, pages 204–211, May 1995.

[19] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 50(2):117–134, April 1994.

[20] Ragunathan Rajkumar. *Synchronization in Real-Time Systems - A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[21] R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *IEEE Real-Time Systems Symposium*, pages 259–269, December 1988.

[22] H. Zhang and D. Ferrari. Rate-controlled service disciplines. *Journal of High Speed Networks*, 3(4), 1994.

[23] R. Cruz. A calculus for network delay, Part I: Network elements in isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, January 1991.

[24] R. Cruz. A calculus for network delay, Part II: Network analysis. *IEEE Transactions on Information Theory*, 37(1):132–141, January 1991.

[25] D. Ferrari and D. Verma. A scheme for real-time channel establishment in wide-area networks. *IEEE Journal on Selected Areas in Communications*, 8(3):368–379, April 1990.

[26] D. D. Kandlur, K. Shin, and D. Ferrari. Real-time communication in multi-hop networks. In *Proceedings of 11th International Conference on Distributed Computer Systems*, pages 300–307, May 1991.

[27] Q. Zheng and K. Shin. On the ability of establishing real-time channels in point-to-point packet-switching networks. *IEEE Transactions on Communications*, 42(2-4):1096–1105, March 1994.

[28] L. Zhang. Virtual clock: A new traffic control algorithm for packet switching networks. In *Proceedings of ACM SIGCOMM'90*, pages 19–29, Philadelphia, Pennsylvania, September 1990.

[29] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *Journal of Inernetworking Research and Experience*, 1(1):3–26, October 1990.

[30] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, June 1993.

[31] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the multiple node case. *IEEE/ACM Transactions on Networking*, 2(2):137–150, April 1994.

[32] D. Verma, H. Zhang, and D. Ferrari. Guaranteeing delay jitter bounds in packet switching networks. In *Proceedings of Tricomm'91*, pages 35–46, Chapel Hill, North Carolina, April 1991.

[33] S. J. Golestani. A stop-and-go queueing framework for congestion management. In *Proceedings of ACM SIGCOMM'90*, pages 8–18, Philadelphia, Pennsylvania, September 1990.

[34] C. R. Kalmanek, H. Kanakia, and S. Keshav. Rate controlled servers for very high-speed networks. In *IEEE Global Telecommunications Conference*, pages 12–20, San Diego, California, December 1990.

[35] E. L. Lawler, J. K.Lenstra, A. H. G. Rinnooy, and D. B. Shmoys. Sequencing and scheduling: Algorithms and complexity. Technical Report BS-R8909, Center for Mathematics and Computer Science, Erasmus University, Rotterdam, June 1989.

[36] J. Blazewicz. Selected topics in scheduling theory. *Annals of Discrete Mathematics*, 31(1):1–59, 1987.

[37] J. Xu and D. L. Parnas. On satisfying timing constraints in hard-real-time systems. *IEEE Transactions on Software Engineering*, 19(1):70–84, January 1993.

[38] J. A. Stankovic, M. Spuri, M. D. Natale, and G. C. Buttazzo. Implications of classical scheduling results for real-time systems. *Computer*, 28(6):16–25, June 1995.

[39] R. Graham. Bounds on the performance of scheduling algorithms. In E. G. Goffman, editor, *Computer and Job Shop Scheduling Theory*, pages 165–227. John Wiley and Sons, 1976.

[40] R. Ha. *Validating Timing Constraints in Multiprocessor and Distributed Systems.* PhD thesis, University of Illinois, Urbana-Champaign, Department of Computer Science, 1995.

[41] S. M. Johnson. Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly*, 1:61–68, 1954.

[42] J. R. Jackson. An extension of Johnson's results on job lot scheduling. *Naval Research Logistics Quarterly*, 3:201–203, 1956.

[43] N. Hefetz and I. Adiri. An efficient optimal algorithm for the two-machines unit-time jobshop schedule-length problem. *Mathematics of Operations Research*, 7:354–360, 1982.

[44] P. Brucker. Minimizing maximum lateness in a two-machine unit-time job shop. *Computing*, 27:367–370, 1981.

[45] P. Brucker. A linear time algorithm to minimize maximum lateness for the two-machine, unit-time, job-shop, scheduling problem. In R. F. Drenick and F. Kozin, editors, *Lecture Notes in Control and Information Sciences 38*, pages 566–571. Springer, Berlin, 1982.

[46] J. K. Lenstra, A. H. G. Rinnooy Kan, and P. Brucker. Complexity of machine scheduling problem. *Annals of Discrete Mathematics*, 1:343–362, 1977.

[47] T. Gonzalez and S. Sahni. Flowshop and jobshop schedules: Complexity and approximation. *Operations Research*, 26:36–52, 1978.

[48] J. K. Lenstra and A. H. G. Rinnooy Kan. Computational complexity of discrete optimization problems. *Annals of Discrete Mathematics*, 4:121–140, 1979.

[49] M. G. Harbour, M. H. Klein, and J. P. Lehoczky. Timing analysis for fixed-priority scheduling of hard real-time systems. *IEEE Transactions on Software Engineering*, 20(1):13–28, January 1994.

[50] B. Hunting. The solution's in the CAN — Part 1. *Circuit Cellar*, pages 14–20, May 1995.

[51] N. Malcolm and W. Zhao. The timed-token protocol for real-time communications. *IEEE Computer*, 27(1):35–41, January 1994.

[52] B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Journal of Real-Time Systems*, 1:27–60, 1989.

[53] K. W. Tindell. *Fixed Priority Scheduling of Hard Real-Time Systems*. PhD thesis, University of York, Department of Computer Science, 1994.

[54] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. II. Freeman and Company, New York, 1979.

[55] D. Gillies and J. Liu. Scheduling tasks with and/or precedence constraints. *SIAM Journal on Computing*, 24(4):797–810, August 1995.

[56] B. Sprunt. *Aperiodic Task Scheduling for Real-Time Systems*. PhD thesis, Carnegie Mellon University, Department of Electrical and Computer Engineering, Pittsburgh, PA, August 1990.

[57] S. Ramos-Thuel and J. Lehoczky. Algorithms for scheduling hard aperiodic tasks in fixed-priority systems using slack stealing. In *Real-Time System Symposium*, pages 22–33, 1994.

# Vita

Jun Sun was born in Yangzhou, People's Republic of China on June 5, 1968. He attended Shanghai Jiao Tong University in September 1985 and graduated with a B.S. degree in Computer Science in July 1989. In his sophomore year, he received the title of *Distinguished Student at Shanghai Jiao Tong University.* From August 1990 to December 1991 he studied in the Master program in Computer Science at the University of North Dakota. Later, he transferred to the Ph.D. program in Computer Science at the University of Illinois, Urbana-Champaign. In May 1993 he joined the Real-Time Research Lab and received a Research Assistantship in January 1994. He successfully defended his Ph.D. dissertation in July of 1996 and later joined Geoworks, Inc. as a Design Engineer. He received his Ph.D. degree in Computer Science in May of 1997.

Jun Sun is the primary author of the following publications. His paper, *Synchronization Protocols in Distributed Real-Time Systems*, which is co-authored by Professor Jane Liu, received the *Outstanding Paper Award* in the 16th International Conference on Distributed Computing Systems.

- Jun Sun, Riccardo Bettati, and Jane W.-S. Liu. *An end-to-end approach to schedule tasks with shared resources in multiprocessor systems.* In Proceedings of the 11th IEEE Workshop on Real-Time Operating Systems and Software, Seattle, Washington, May 1994.

- Jun Sun and Jane Liu. *Bounding the end-to-end response time in multiprocessor real-time systems.* Published in the Proceedings of Workshop on Parallel and Distributed Real-Time Systems, pages 91-98, Santa Barbara, California, April 1995.

- Jun Sun and Jane Liu. *Synchronization protocols in distributed real-time systems.* Published in the 16th International Conference on Distributed Computing Systems, Hong Kong, May 1996.

- Jun Sun and Jane W.S. Liu. *Bounding completion times of jobs with arbitrary release times and variable execution times.* Published in the Proceedings of IEEE Real-Time Systems Symposium, pages 2-12, Washington, D. C., December 1996.