REAL-TIME SCHEDULING OF PERIODIC AND APERIODIC TASKS ON MULTIPROCESSOR SYSTEMS

A Dissertation Presented
by
SHINPEI KATO

Submitted to
the School of Science for Open and Environmental Systems
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

at Keio University

March 2008
To my parents.
ACKNOWLEDGMENT

A great many people have supported and assisted me in so many ways during my stay here at Keio University. I owe my gratitude to all those people who have made this dissertation possible.

My first, and the most earnest, gratitude must go to my adviser Prof. Nobuyuki Yamasaki. I have been fortunate to have such a great adviser who has outstanding skill and knowledge. He has always indicated me the right road to proceed when I faltered on my research. This dissertation could have been never completed without his professional suggestion and advice.

I would like to give my sincere gratitude to Prof. Kenji Kono and Prof. Hideharu Amano as well. They have dedicated their time and energy far beyond their duty to substitute a technical editor of this dissertation and an adviser of my research respectively instead of Prof. Yamasaki because of his sabbatical leave to the United States of America through this year. I wish to extend my gratitude to Prof. Fumio Teraoka and Prof. Takahiro Yakoh for serving on my dissertation committee. Their constructive criticism and invaluable advice have significantly improved this dissertation. I am also thankful to my first adviser Prof. Kenji Harada who helped me take the very first step to this point.

I appreciate the financial supports from the Research Fellowships of the Japan Society for the Promotion of Science for Young Scientists and the Core Research for Evolutional Science and Technology of Japan Science and Technology Agency. I also appreciate the members of the National Institution of Advanced Industrial Science and Technology for their technical advice to the part of my research.

I am grateful to Takahiro Sasaki, Seiichi Arai, and Nobuhide Kamata who have spent all the time with me for two years of the master’s course. I could not have survived the master’s research and achieved the thesis without their kind cooperation. I am also thankful to Dr. Hidenori Kobayashi and Dr. Tsutomu Ito for their technical assistance of my research. My special thanks go to Naoki Inoue, Nobuyasu Takahashi, and Naru Kudo for their friendship from the undergraduate years.

My final, and the most heartfelt, acknowledgment must go to my parents and Mrs. Fumiko Omi. They have continuously encouraged, supported, and looked after me during all my life. No words can adequately express my gratitude to them.
ABSTRACT

REAL-TIME SCHEDULING OF PERIODIC AND APERIODIC TASKS ON MULTIPROCESSOR SYSTEMS

SHINPEI KATO

Real-time scheduling techniques for multiprocessor systems have attracted considerable attention in recent years. However, the theoretical intricateness of scheduling periodic tasks raises a controversial trade-off that improving a system utilization with guaranteeing timing constraints of periodic tasks leads to more task preemptions and migrations. In addition, the subject of scheduling aperiodic tasks has not been considered very much. The research established herein considers the efficient scheduling of both periodic and aperiodic tasks on multiprocessor systems.

The dissertation first presents the portioned scheduling technique for improving the schedulability to periodic tasks with a small number of task preemptions and migrations. Like the traditional partitioned scheduling, most of tasks are scheduled on dedicated processors with no migrations, however the portioned scheduling differs in that special tasks are permitted to migrate within the restricted range. The RMDP algorithm is developed for the discipline of static-priority scheduling, in which the traditional Rate Monotonic algorithm is combined with the portioned scheduling. The EDDHP algorithm is then devised for the discipline of dynamic-priority scheduling, in which the traditional Earliest Deadline First algorithm is combined. Finally, the EDDP algorithm is invented by extending the prioritization policy of the EDDHP algorithm to improve the worst-case system utilization with a guarantee of timing constraints.

The dissertation also presents the global dispatch technique and the temporal migration technique for enhancing the responsiveness to aperiodic tasks without jeopardizing periodic timing constraints. An aperiodic task is dispatched to a processor on which the response time is estimated to be minimized, and then periodic tasks with higher priorities are temporarily migrated onto different processors to moreover reduce the response time. The design of the traditional Priority Exchange algorithm and the Total Bandwidth Server algorithm is considered, in which the presented techniques are applied.

Beyond the theoretical analysis for the worst case, the effectiveness of the developed scheduling algorithms for general cases is also studied by several sets of simulations.
# Contents

1 INTRODUCTION ........................................ 1  
  1.1 Multiprocessor Systems ........................... 4  
  1.2 Motivation ....................................... 6  
  1.3 Research Overview ............................... 9  
  1.4 Organization .................................. 12  

2 STATE OF THE ART ................................... 13  
  2.1 Classification of Scheduling Principles .......... 13  
  2.2 Scheduling Algorithms ........................... 15  
    2.2.1 Pure RM and EDF ............................ 16  
    2.2.2 Partitioned Scheduling Approach ............ 17  
    2.2.3 Global Scheduling Approach ................ 19  
    2.2.4 Another Scheduling Approach ............... 23  
  2.3 Server Algorithms ................................ 25  
    2.3.1 Uniprocessor Approach ...................... 25  
    2.3.2 Multiprocessor Approach .................... 27  
  2.4 Summary ........................................ 28  

3 SYSTEM MODEL .................................... 30  
  3.1 Processor Model .................................. 30  
  3.2 Task Model ..................................... 31  

4 PERIODIC TASK SCHEDULING ........................ 33  
  4.1 Basic Strategy ................................... 34  
  4.2 The RMDP algorithm ............................... 36  
    4.2.1 Task Assigning Phase ....................... 36  
    4.2.2 Task Scheduling Phase ..................... 40  
    4.2.3 Schedulability Analysis .................... 44  
  4.3 The EDDHP algorithm ............................. 55  
    4.3.1 Task Assigning Phase ....................... 55  
    4.3.2 Task Scheduling Phase ..................... 57  
    4.3.3 Schedulability Analysis .................... 57
## CONTENTS

4.3.4 Improving Schedulable Utilization .............................................. 64  
4.4 The EDDP algorithm ............................................................... 66  
  4.4.1 Task Assigning Phase ......................................................... 67  
  4.4.2 Task Scheduling Phase ........................................................ 69  
  4.4.3 Schedulability Analysis ...................................................... 71  
  4.4.4 Improving Schedulable Utilization ....................................... 75  
4.5 Summary ................................................................................. 75  
5 APERIODIC TASK SCHEDULING ............................................... 77  
  5.1 Basic Strategy ......................................................................... 78  
  5.2 The PE-based Algorithm .......................................................... 80  
    5.2.1 Priority Exchange Review ............................................... 80  
    5.2.2 Global Dispatch Procedure .............................................. 82  
    5.2.3 Temporal Migration Procedure ........................................ 83  
  5.3 The TBS-based Algorithm ....................................................... 86  
    5.3.1 Total Bandwidth Server Review ....................................... 86  
    5.3.2 Global Dispatch Procedure .............................................. 87  
    5.3.3 Temporal Migration Procedure ........................................ 87  
  5.4 Compatibility to Portioned Scheduling ..................................... 91  
  5.5 Summary ................................................................................. 92  
6 PERFORMANCE EVALUATION .................................................. 94  
  6.1 Simulation Studies for Periodic Task Scheduling ....................... 94  
    6.1.1 Experimental Setup ......................................................... 95  
    6.1.2 Simulation Results of Static-Priority Scheduling ................. 96  
    6.1.3 Simulation Results of Dynamic-Priority Scheduling .......... 105  
  6.2 Simulation Studies for Aperiodic Task Scheduling ...................... 114  
    6.2.1 Experimental Setup ......................................................... 114  
    6.2.2 Simulation Results of Static-Priority Scheduling ............... 116  
    6.2.3 Simulation Results of Dynamic-Priority Scheduling .......... 122  
  6.3 Summary ................................................................................. 131  
7 CONCLUSION .............................................................................. 133  
  7.1 Summary of Contributions ...................................................... 133  
  7.2 Future Directions .................................................................... 136
## List of Figures

1.1 Problematic scheduling on $M$ processors ........................................ 8

2.1 Global scheduling approach .......................................................... 14
2.2 Partitioned scheduling approach ....................................................... 15
2.3 Partitioning problem ........................................................................ 19
2.4 Pfair windows of task with utilization $8/11$ ....................................... 20
2.5 RM-US scheduling example ............................................................... 21
2.6 EDZL scheduling example ................................................................. 22
2.7 LNREF scheduling example ............................................................... 23
2.8 EKG scheduling example ................................................................. 24

3.1 Periodic task model ............................................................................ 31
3.2 Aperiodic task model .......................................................................... 32

4.1 Portioning example ............................................................................ 34
4.2 Portioned scheduling example ............................................................. 35
4.3 RMDP assigning algorithm ................................................................. 37
4.4 $rmdp\_bound$ function ....................................................................... 38
4.5 RMDP assigning example ................................................................. 39
4.6 RMDP scheduling algorithm ............................................................... 41
4.7 Job finishing functions ....................................................................... 42
4.8 RMDP scheduling example ................................................................. 43
4.9 Preemption-cared RMDP scheduling example ...................................... 43
4.10 Case in which $\tau''_{s}$ is executed twice within $T_1$ and $T_n$ .............. 46
4.11 Case in which $\tau''_{s}$ is executed three times within $T_1$ and $T_n$ .......... 48
4.12 Case in which $\tau''_{s}$ is executed twice within $T_1$ and three times within $T_n$ 50
4.13 EDDHP assigning algorithm .............................................................. 56
4.14 $eddhp\_bound$ function .................................................................. 57
4.15 EDDHP scheduling algorithm ........................................................... 58
4.16 Unfeasible EDDHP scheduling ......................................................... 59
4.17 Worst-case phasing of EDDHP scheduling ........................................ 61
4.18 Absolute worst-case phasing of EDDHP scheduling .......................... 63
4.19 MB heuristic .................................................................................... 64
6.29 Mean response time for PE-GD-TM (M).

6.28 Mean response time for PE-GD-TM (M).

6.27 Mean response time for PE-GD-TM (M).

6.26 Case in which \( \tau_i' \) consumes the most time.

5.4 Total Bandwidth Server example.

5.3 Priority Exchange example.

5.2 Concept of temporal migration.

5.1 Concept of global dispatch.

4.24 Latest completion of EDDP scheduling.

4.23 Latest completion of EDF scheduling.

4.22 EDDP scheduling example.

4.21 EDDP assigning algorithm.

4.20 AIB heuristic.

LIST OF FIGURES

4.20 AIB heuristic ................................................. 65
4.21 EDDP assigning algorithm ..................................... 68
4.22 EDDP scheduling example ...................................... 69
4.23 Latest completion of EDF scheduling ................................ 70
4.24 Latest completion of EDDP scheduling ................................ 70
4.25 EDDP scheduling algorithm ...................................... 72
4.26 Case in which \( \tau_i'' \) consumes the most time .................... 73

5.1 Concept of global dispatch ....................................... 78
5.2 Concept of temporal migration .................................... 79
5.3 Priority Exchange example ....................................... 81
5.4 Total Bandwidth Server example ................................... 86
5.5 Total Bandwidth Server with Temporal Migration example ............ 89

6.1 Success ratio \( (M = 4, U_{\text{min}} = 0.01, U_{\text{max}} = 0.5). \) ........................................ 97
6.2 Success ratio \( (M = 8, U_{\text{min}} = 0.01, U_{\text{max}} = 0.5). \) ........................................ 97
6.3 Success ratio \( (M = 16, U_{\text{min}} = 0.01, U_{\text{max}} = 0.5). \) ........................................ 98
6.4 Success ratio \( (M = 4, U_{\text{min}} = 0.01, U_{\text{max}} = 1.0). \) ........................................ 98
6.5 Success ratio \( (M = 8, U_{\text{min}} = 0.01, U_{\text{max}} = 1.0). \) ........................................ 99
6.6 Success ratio \( (M = 16, U_{\text{min}} = 0.01, U_{\text{max}} = 1.0). \) ........................................ 99
6.7 The number of preemptions \( (M = 4, U_{\text{min}} = 0.01, U_{\text{max}} = 0.5). \) ......................... 101
6.8 The number of preemptions \( (M = 8, U_{\text{min}} = 0.01, U_{\text{max}} = 0.5). \) ......................... 101
6.9 The number of preemptions \( (M = 16, U_{\text{min}} = 0.01, U_{\text{max}} = 0.5). \) ......................... 102
6.10 The number of preemptions \( (M = 4, U_{\text{min}} = 0.01, U_{\text{max}} = 1.0). \) ......................... 102
6.11 The number of preemptions \( (M = 8, U_{\text{min}} = 0.01, U_{\text{max}} = 1.0). \) ......................... 103
6.12 The number of preemptions \( (M = 16, U_{\text{min}} = 0.01, U_{\text{max}} = 1.0). \) ......................... 103
6.13 Success ratio \( (M = 4, U_{\text{min}} = 0.01, U_{\text{max}} = 0.5). \) ........................................ 106
6.14 Success ratio \( (M = 8, U_{\text{min}} = 0.01, U_{\text{max}} = 0.5). \) ........................................ 106
6.15 Success ratio \( (M = 16, U_{\text{min}} = 0.01, U_{\text{max}} = 0.5). \) ........................................ 107
6.16 Success ratio \( (M = 4, U_{\text{min}} = 0.01, U_{\text{max}} = 1.0). \) ........................................ 107
6.17 Success ratio \( (M = 8, U_{\text{min}} = 0.01, U_{\text{max}} = 1.0). \) ........................................ 108
6.18 Success ratio \( (M = 16, U_{\text{min}} = 0.01, U_{\text{max}} = 1.0). \) ........................................ 108
6.19 The number of preemptions \( (M = 4, U_{\text{min}} = 0.01, U_{\text{max}} = 0.5). \) ......................... 110
6.20 The number of preemptions \( (M = 8, U_{\text{min}} = 0.01, U_{\text{max}} = 0.5). \) ......................... 110
6.21 The number of preemptions \( (M = 16, U_{\text{min}} = 0.01, U_{\text{max}} = 0.5). \) ......................... 111
6.22 The number of preemptions \( (M = 4, U_{\text{min}} = 0.01, U_{\text{max}} = 1.0). \) ......................... 111
6.23 The number of preemptions \( (M = 8, U_{\text{min}} = 0.01, U_{\text{max}} = 1.0). \) ......................... 112
6.24 The number of preemptions \( (M = 16, U_{\text{min}} = 0.01, U_{\text{max}} = 1.0). \) ......................... 112
6.25 Mean response time for PE-GD \( (\mu = 0.1). \) .............................. 117
6.26 Mean response time for PE-GD \( (\mu = 0.2). \) .............................. 117
6.27 Mean response time for PE-GD-TM \( (M = 4, \mu = 0.1). \) .............................. 119
6.28 Mean response time for PE-GD-TM \( (M = 8, \mu = 0.1). \) .............................. 119
6.29 Mean response time for PE-GD-TM \( (M = 16, \mu = 0.1). \) .............................. 120
6.30 Mean response time for PE-GD-TM ($M = 4, \mu = 0.2$) . . . . . . . . . . . . . 121
6.31 Mean response time for PE-GD-TM ($M = 8, \mu = 0.2$) . . . . . . . . . . . . . 121
6.32 Mean response time for PE-GD-TM ($M = 16, \mu = 0.2$) . . . . . . . . . . . . . 122
6.33 Mean response time for TBS-GD ($\mu = 0.1$) . . . . . . . . . . . . . . . . 123
6.34 Mean response time for TBS-GD ($\mu = 0.2$) . . . . . . . . . . . . . . . . . . 123
6.35 Mean response time for TBS-GD-TM ($M = 4, \mu = 0.1$) . . . . . . . . . . . 125
6.36 Mean response time for TBS-GD-TM ($M = 8, \mu = 0.1$) . . . . . . . . . . . 125
6.37 Mean response time for TBS-GD-TM ($M = 16, \mu = 0.1$) . . . . . . . . . . . 126
6.38 Mean response time for TBS-GD-TM ($M = 4, \mu = 0.2$) . . . . . . . . . . . 126
6.39 Mean response time for TBS-GD-TM ($M = 8, \mu = 0.2$) . . . . . . . . . . . 127
6.40 Mean response time for TBS-GD-TM ($M = 16, \mu = 0.2$) . . . . . . . . . . . 127
6.41 Mean response time for TBS-TM ($M = 4, \mu = 0.1$) . . . . . . . . . . . . . 128
6.42 Mean response time for TBS-TM ($M = 8, \mu = 0.1$) . . . . . . . . . . . . . 128
6.43 Mean response time for TBS-TM ($M = 16, \mu = 0.1$) . . . . . . . . . . . . . 129
6.44 Mean response time for TBS-TM ($M = 4, \mu = 0.2$) . . . . . . . . . . . . . 129
6.45 Mean response time for TBS-TM ($M = 8, \mu = 0.2$) . . . . . . . . . . . . . 130
6.46 Mean response time for TBS-TM ($M = 16, \mu = 0.2$) . . . . . . . . . . . . . 130
Chapter 1

INTRODUCTION

The 21st century is an era of information and intelligence. We human beings expect a safe and secure lifestyle constructed by an intelligent social infrastructure. To this end, the mission imposed to us is to benefit properly from the experiences of the information technology that has been seeing rapid development since the 1990s, and then deserve a new look from the various points of view. The social infrastructure is such a complex system that supports our entire life. All nations around the world have already launched a study of new technologies, including embedded computing, ubiquitous computing, cyber-physical computing, etc. for the sake of the next generation society.

Most of the computing systems that will construct the intelligent social infrastructure, so-called intelligent systems, are at the same time real-time systems. The primary reason why the intelligent systems are real-time systems is that they operate autonomously in our real world, while the computing systems produced in the 20th century, such as personal computers and workstations, are mainly operated by humans. The intelligent systems are for instance that embedded computers and networks monitor and control physical processes, usually with feedback loops where the physical processes affect computations and vice versa. Since the physical processes in the real world occur in the real time, the intelligent systems must react to stimuli with timing constraints.

The real-time systems are technically characterized by the fact that they require temporal correctness as well as logical correctness. In other words, the correctness of the systems depends on not only the computational results but also the time when they are produced. More precisely, every real-time computation must complete in an interval of certain length. The beginning of the interval is called release time and the end is called deadline. There are mainly two types of deadlines depending on what could happen if a deadline is missed. The deadline is defined hard if the contribution to the system suddenly drops to zero due to a deadline miss. On the other hand, the deadline is defined soft if the contribution to the system does not suddenly becomes zero but gradually degrades as the completion time is further delayed from a deadline. For instance, robot systems have hard deadlines for their control. A catastrophe may
occur if actuators cannot complete processing sensor data within a certain feedback period. Meanwhile, multimedia systems often have soft deadlines for image and audio processing, because the systems are able to continue to provide a service to users even if some computations miss their deadlines, though the quality of service may degrade compared to the case in which all computations meet their deadlines. In order to maintain substantial quality, deadlines should not be missed even in soft real-time systems. Most of commercial real-time systems have soft deadlines. In other words, most of commercial real-time systems are not able to guarantee that all deadlines are met, because the theoretical aspect of real-time computing is not likely to be considered. The research established herein targets a theory to accomplish hard real-time systems where all deadlines are guaranteed to be met. Such a theory also supports soft real-time systems in terms of the quality of service.

The question here is, how is real-time computing made possible? A huge amount of research has been conducted for a long time to explore the answer to this question. The first, and probably the greatest, contribution was a development of a real-time computation model by Liu and Layland in 1973 [1]. Most of real-time computing techniques today have their bases on the Liu and Layland model. This classical computation model presumes that a task, a unit of required computations, is invoked at every certain interval called period. A computation of the task in each period is often called a job. Every job is assumed to have the same cost of execution and have the relative deadline equal to the period of the task. In other words, every job of the task must be completed before the next job of the task arrives. Other assumptions are: an execution time of a task is fixed; all jobs are independent; no resource or precedence constraint exists; no computation suspends itself; and a job can be preempted at any moment, which means that a job can be interrupted immediately upon a request to execute another job and then be resumed some time after that.

Liu and Layland also focused on the scheduling of recurrent tasks so that all the jobs meet their deadlines under the assumptions described above. The scheduling with such timing constraints is widely called real-time scheduling nowadays. They then developed two real-time scheduling algorithms: Rate Monotonic (RM) and Earliest Deadline First (EDF). Almost all real-time scheduling algorithms today are formed by either of those initiatives. The RM algorithm schedules a task with the shortest period first among all ready tasks. Since the period of a task is fixed in their computation model, the priority assignment in the RM algorithm is static. Meanwhile the EDF algorithm schedules a task with the earliest absolute deadline first, which implies that the priority assignment is dynamic. Liu and Layland moreover carried out the theoretical analysis for the two algorithms to guarantee that all jobs meet their deadlines. If a task set is determined to be feasible under a scheduling algorithm, it means that no task in the set violates its timing constraints under any circumstance [2]. They proved that any periodic task set can be scheduled with no deadline misses by the RM algorithm if the total utilization of the periodic task set is less than or equal to the following expression
where \( N \) denotes the number of the periodic tasks.

\[
N(2^{1/N} - 1)
\]  (1.1)

They also proved that any periodic task set can be scheduled with no deadline misses by the EDF algorithm if the total utilization of the periodic task set is less than or equal to 1 regardless the number of the periodic tasks, which means that the EDF algorithm is capable of utilizing 100% of processor capacity without any timing violations. Ever since these two algorithms are invented, there have been a considerable number of arguments on which algorithm performs better in what conditions, some of which are summarized by Buttazzo [3]. The broad consensus of the real-time computing community is for instance that the RM algorithm has more predictability than the EDF algorithm due to the characteristic of its static-priority assignment, while the EDF algorithm can achieve higher processor utilization with guaranteeing timing constraints in many cases. Since the two algorithms both have advantages, the superiority of each algorithm depends on the requirement of the system. Even in recent years, the theory of the two algorithms has been widely discussed and improved [4, 5, 6, 7].

Over the years, real-time systems have mainly focused on the periodic computation model advocated by Liu and Layland. In fact, many types of control and multimedia applications are covered by the periodic computation model. The scheduling of such periodic tasks has therefore been the subject of research on real-time systems for a long time. However, intelligent systems require more complex computation model. For instance, robot systems must operate in dynamic environments where human activities or physical processes occur at any moment. In such a case, some tasks may arrive aperiodically, and their arrival times are not known a priori. Most of those aperiodic requests are desired to complete as soon as possible, while they usually have no critical timing constraints. Consider a speech recognition in robot systems. When humans speak to a robot, they expect the robot to respond quickly. To achieve this, a system needs to reduce the response time of aperiodic tasks as much as possible in addition to satisfying timing constraints of periodic tasks.

The most straightforward method for the scheduling of aperiodic tasks is to allocate available time slots that are left unused by periodic tasks. This kind of background scheduling can be simply applied to the classical Liu and Layland model, since periodic tasks are never interfered by aperiodic tasks. However, a good responsiveness is never attained by the background scheduling. In order to overcome the responsiveness issue of real-time scheduling, Lehoczky et al. invented the server technique [8]. The server approach creates a special periodic task called server whose purpose is to service aperiodic requests as soon as possible. Like any periodic task, a server is characterized by a period and a fixed execution time called server capacity. The server is scheduled according to the same algorithm used for the periodic tasks, and, once active, it serves the aperiodic requests within the limit of its server capacity. Having a basis on such a server approach, a large number of efficient server algorithms have been presented for both the discipline of static-priority scheduling and the discipline of dynamic-priority
CHAPTER 1. INTRODUCTION

scheduling [9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]. Compared to background scheduling where aperiodic tasks consumes time slots that are left unused by periodic tasks, the server algorithms reduce the response time of aperiodic tasks dramatically. Controversially, they have pros and cons in terms of performance and complexity.

Adopting the priority-driven scheduling with server approaches, modern real-time systems realize a guarantee of timing constraints as well as an enhancement of responsiveness. Ever since such a basis of the scheduling model was established, the real time computing community has moreover developed a lot of optional techniques, such as synchronization [20, 21], resource reservation [22, 23], adaptive computing [24, 25], dynamic voltage and frequency scaling [26, 27], and so on. Those techniques make systems more reliable, dependable, and flexible. Far too many novel techniques to describe here have been devised. Thus, today’s theoretical aspects of real-time computing for uniprocessor systems are appreciably powerful.

In recent years, seeing performance limitations of uniprocessor platforms, high performance real-time computing has relied on a power of multiprocessor platforms. With the trend towards multicore architectures, kinds of multiprocessor systems are likely to become much more common for the next generation applications. The real-time computing community has been therefore challenging a domain of multiprocessor systems. Unfortunately, the conventional scheduling techniques are of no use due to the property of concurrency on multiprocessor systems. During the last decade, researchers have just begun to understand the problematic scheduling of periodic tasks on multiprocessor systems. This fact implies that the scheduling of aperiodically-arriving tasks on multiprocessor systems is also controversial, because it definitely relies on the scheduling of periodic tasks. The motivation of this research exists here.

1.1 Multiprocessor Systems

Before moving on the problem definitions of multiprocessor real-time scheduling, the dissertation ought to explicit the reason why multiprocessor systems have received considerable attention in recent years, and also describe the background how the need for multiprocessor systems has arisen. The followings are the historical reviews of computer technologies.

Computer technology has made incredible progress for more than forty years following the Moore’s Law [28], advocated in 1965, which the number of transistors that could be integrated into a single silicon chip would approximately double every 18 to 24 months. During that time, increases in transistor density have driven roughly proportional increases in processor performance and price/performance. However, those performance gains did not come about solely because of increasing transistor densities. They have also relied heavily on another physical factor that is closely related to transistor size: processor clock frequency.

In general, as transistors get smaller, they can switch faster between one state and
another. This has allowed designers to continually increase processor clock frequencies at the same time they have increased the total number of transistors. In many ways, increases in clock frequency have been more important than increases in density. It takes a great deal of engineering ingenuity to make efficient use of more transistors. Frequency gains, on the other hand, deliver instant and easily realizable performance benefits. Existing software code runs faster, without requiring software engineers to revise or optimize their code. This has been a central fact of the computing industry for many years, and a great boon to business users.

Given this advantage, it is not surprising that frequency ramp has long been the primary engine behind processor performance gains. The first Intel processor released in 1971 and ran at 400 KHz. Today, some Intel processors support clock speeds that are nearly ten thousand times as fast. Moore’s Law continues today and can be expected to deliver increasing transistor densities for at least several more generations. However, in recent years, the frequency ramp has faced mounting obstacles. Power consumption and heat generation rise exponentially with clock frequency. Especially in the field of embedded computers, those issues are very critical. Even in the field of personal computers and workstations, system integrators devote considerable engineering resources to optimizing power and cooling systems to avoid overheating, which can dramatically reduce component longevity. For this reason, increasing clock frequency is no longer viable as the primary means for boosting processor performance. Clock frequencies will continue to rise, but only incrementally. In consequence, the necessity of new strategies from the viewpoint of microarchitectural techniques have arisen to maintain historic rates of performance and price/performance improvement.

Most of recent computers seeking throughput exploits instruction-level parallelism (ILP). With ILP, the processor dynamically evaluates software code streams to determine which instructions are independent and can be safely processed simultaneously or out of order. If one instruction is waiting for data, the processor can then execute an independent instruction to stay productive. This strategy has been increasingly important as processor speed has outstripped memory speed. Today, a processor can wait through hundreds of clock cycles if it has to retrieve data or instructions from main memory, which decreases the value of rising frequencies.

Over the years, processor designers have invested heavily in optimizing ILP to reduce the impact of memory wait times. However, it takes a lot of complex, high-speed transistor logic to examine software code during runtime, find opportunities for ILP, and reschedule the software code accordingly. For this reason, ILP is very resource intensive, and accounts for considerable energy consumption and heat generation in today’s processors. Like the ramp of processor clock frequency, it has reached a point of diminishing returns. Such a performance limitation of ILP signals a historic switch from relying solely on ILP to thread-level parallelism (TLP) to gain further performance improvement. Whereas ILP tries to find the parallelism within one thread, TLP does from multiple threads. Hence the processor can process more than one thread simultaneously by interleaving the code streams. Since a single thread rarely uses all of
CHAPTER 1. INTRODUCTION

hardware resources, this enables greater processing efficiency and greater total throughput for multi-threaded applications.

ILP and TLP are not exclusive. Simultaneous multithreading (SMT) [29, 30] is a technique that exploits both ILP and TLP. Hyperthreading [31] implemented in the Intel’s processors is one of SMT derivations. SMT processors fetch and issue instructions to functional units simultaneously from multiple threads. This creates horizontal and vertical sharing of resources within a core, which increases throughput and tolerates processor and memory latencies to increase processor efficiency. The issue of concern with SMT is that layout blocks and circuit delays grow faster than linear with issue width. Besides, multiple threads share the same L1 cache, TLB, and branch predictor units, hence contentions occur on hardware resources. The resulting increase in cache misses and branch mispredict rates limits performance. As a result, SMT will not deliver the kind of massive performance increases needed to replace the frequency ramp as a primary strategy for boosting performance.

The industry’s answer to today’s performance challenges is at this moment to take advantage of ongoing increases in transistor density, i.e. Moore’s Law, to integrate more execution cores on a processor chip. Such a technique is now widely called chip multiprocessing (CMP) [32]. Similarly but contrary to SMT, CMP enables multiple cores to share more distantly-positioned resources such as memory controller, off-chip bandwidth, L2 cache, and so on. Other resources are partitioned among cores, namely each core has its own resources. CMP is also capable of increasing layout efficiency, resulting in more functional units within the same silicon area plus faster clock frequencies than SMT due to its simple logic. With multiple cores executing simultaneously, processor designers can turn down clock frequencies to contain power consumption and heat generation, while still delivering increases in total throughput for multithreaded software. Individual threads might be processed slower due to the lower clock frequencies, but total throughput can be dramatically increased. As a matter of practice, multicore processors (chip multiprocessors) are the wave of the future across all computing systems.

1.2 Motivation

Embedded computer systems, computers lodged in other devices where the presence of the computers is not immediately obvious, are the fastest-growing portion of the computer market. These devices range from everyday machines (microwaves, washing machines, printers, etc.) to futuristic devices (humanoid robots, intelligent buildings/automobiles, virtual video game consoles, etc.) in one way or another. Since those futuristic systems are at the same time real-time systems, the demands for real-time computing are still continuously increasing. Meanwhile, embedded computers are on the trend of multiprocessor platforms for their requirement of high processing capability with low power consumption. For example, humanoid robots must (i) recognize
their states through a bunch of internal and external sensors, (ii) process a variety of information based on the determined modeling, (iii) plan many kinds of actions according to circumstances, and (iv) actuate a number of modules. A sequence of those computations must be done in the real time. In addition, the computations range from tasks having feedback intervals of several tens of micro seconds such as current control and PWM control, to ones having feedback intervals of several seconds such as action planning based on environmental recognition. Humanoid robots are also required to operate with a battery for as long as possible, meaning that a processor with fast clock frequency is not preferred in the power consumption point of view, despite the requirement of high processing capability. The multiprocessor technology fulfill the requirements of high processing capability as well as low power consumption by parallelism. Even if the clock frequency of every individual processor is configured to be low for power consumption, the total performance is substantially boosted up by parallel execution. Thus, the dissertation believes that multiprocessor platforms will be natural choices for humanoid robots.

Humanoid robots have been actually appearing recently. Honda Motor Corporation has already released a well-known humanoid robot named ASIMO [33]. Sony Corporation has also released an entertainment robot QRIO [34]. Hitachi Corporation has developed EMIEW [35]. National Institution of Advanced Industrial Science and Technology (AIST) has developed sophisticated humanoid robots HRP-2 [36] and HRP-3 [37], and is going for its next version HRP-4 [38]. More recently, Toyota Motor Corporation announced their Toyota Partner Robot [39]. Academic institutions have also presented fruits of their research. Bischoff et al. developed a wheel-type humanoid robot HERMES [40]. Kagami et al. invented a full-type humanoid robot H7 [41]. Unfortunately, despite thus many requirements, real-time computing and multiprocessor systems are not easily integrated. Especially, a crucial problem resides in scheduling.

For understanding the problem, an example of scheduling $M+1$ periodic tasks on $M$ processors according to the RM algorithm is given. Let $M$ tasks, indexed by $1, 2, 3, ..., M$ respectively, have the same periods of $x$ and the same execution times of $2\epsilon$. Let the remaining one task (this is a problematic task), indexed by $M+1$, have a period of $x+\epsilon$ and an execution time of $x$. Notice that $2\epsilon < x$. Then, suppose that all the tasks are released at time $t = 0$. The resulting schedule is illustrated in Figure 1.1. Each box is indicative of a task execution. The number indicated within a box is the index of the task. The $M$ tasks whose periods are all $x$ are dispatched in advance according to the RM algorithm. Then, all the $M$ tasks consume $2\epsilon$ time units and complete at time $t = 2\epsilon$ at the same time. Hence, the problematic task begins execution at time $t = 2\epsilon$ on some processor. However, the problematic task can never avoid missing its deadline, since its execution time is $x$, which means that its completion time is at the earliest time $t = x + 2\epsilon$, whereas its deadline is $x + \epsilon$. Letting $\epsilon \to 0$, the $M$ tasks have the zero utilization because their execution times become zero, and the remaining one task, indexed by $M+1$, has the 100% utilization because its period and execution time become the same. Therefore, the total utilization of the tasks becomes 100%, namely the utilization of the
whole system becomes $1/M \times 100\%$. So a deadline can be missed even if only the $1/M$ of the system is utilized. Moreover letting $M \rightarrow \infty$, a deadline can be missed even if the system utilization is zero. The EDF algorithm also sees the same phenomenon with the same task set. This phenomenon is often referred to as Dhall’s effect [42]. The Dhall’s effect occurs for the case in which heavy tasks (tasks that have high utilizations) exist. In consequence, the schedulable system utilization for the RM and EDF algorithms are poorly low on multiprocessor systems in the presence of heavy tasks. Hereinafter the schedulable system utilization is abbreviated as schedulable utilization for simplicity of description. The word schedulability is also used to express a degree of the schedulable utilization.

One of the greatest merits of multiprocessor systems is needless to say their parallelism. The performance of computations is expected to be improved by executing multiple tasks simultaneously on more than one processor. However, the traditional scheduling algorithms that have been devised in the uniprocessor era may miss a deadline when one of processors is fully utilized even if there are more processors. Thus, multiprocessor systems offer no benefits in terms of the worst-case schedulable utilization. Overcoming the Dhall’s effect requires inventing a new theory of real-time scheduling. Due to the backgrounds described above, real-time scheduling techniques for multiprocessor systems, which are qualified for achieving high schedulable utilizations, have attracted considerable attention in recent years. As the situation now stands, though a lot of scheduling algorithms specific for multiprocessor systems have been de-

---

**Figure 1.1: Problematic scheduling on $M$ processors**

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>1</th>
<th>$M+1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor 2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Processor 3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Processor $M$</td>
<td>$M$</td>
<td></td>
</tr>
</tbody>
</table>

During the interval $[0, x+2\varepsilon]$, the earliest one cannot be scheduled.
CHAPTER 1. INTRODUCTION

Developed, the theoretical intricateness of scheduling periodic tasks raises a controversial trade-off that improving a system utilization with guaranteeing timing constraints of periodic tasks leads to more task preemptions and migrations. Such a trade-off between schedulability and complexity has aggrieved the developed scheduling algorithms. Solutions are still widely discussed. Since a basis of periodic task scheduling has not been strongly established yet, very little attention has been on the scheduling of aperiodic tasks on multiprocessor systems.

This research is originally motivated by the controversial trade-off between schedulability and complexity. In general, the traditional approaches aim to accomplish either an optimality or a practicality. The optimal approaches are literally able to achieve a system utilization of 100% in theory but may observe performance degradation in practice due to complex computations. On the other hand, the practical approaches are implementable in practice but have potential limits in performance. Solving this trade-off requires establishing a good balancing point that offers both sufficient performance and practical implementation. Thus, the dissertation considers a distinguished approach to establish such a good balancing point for the efficient scheduling of periodic tasks. This research is also motivated by the poverty of aperiodic task scheduling on multiprocessor systems. Exploiting the advantage of parallel execution, the responsiveness to aperiodic tasks must be able to improve. The dissertation hence presents the efficient scheduling of aperiodic tasks on multiprocessor systems.

1.3 Research Overview

The goal of this research is to increase the total utilization of periodic tasks and to reduce the response time of aperiodic tasks as much as possible, in addition to a guarantee of timing constraints, on multiprocessor systems for embedded computers. Attaining this goal allows constructing embedded multiprocessor systems that can provide high-performance real-time and responsive computing with low power consumption. Therefore, it contributes to a part of intelligent systems.

To achieve this goal, the research established herein considers the efficient scheduling of both periodic and aperiodic tasks on multiprocessor systems. For practical use, the approaches are retained elementary. The thesis established by this research is:

The migrative scheduling improves the schedulability to periodic tasks as well as the responsiveness to aperiodic tasks, without causing complex computations and a lot of task preemptions.

In particular, this research makes contributions of theoretical significance as follows.

- A scheduling technique for the scheduling of periodic tasks on multiprocessor systems is proposed to overcome the trade-off between schedulability and complexity. The scheduling policy allows the selected tasks to migrate among processors to improve schedulable utilization but limits a degree of the migrations to
CHAPTER 1. INTRODUCTION

retain acceptable computation complexity. More specifically, each of the selected tasks is permitted to migrate between restrictive two processors and the other tasks are scheduled on dedicated processors with no migrations. This approach gives a breakthrough to the problematic scheduling on multiprocessor systems. Three scheduling algorithms are developed based on the proposed scheduling technique. One is designed for the discipline of static-priority scheduling and the other two are for the discipline of dynamic-priority scheduling. Since both disciplines have pros and cons, system designers can choose which algorithm is suitable for the constructing system.

- The static-priority scheduling algorithm has its basis on the RM algorithm and exploits the notion of the proposed scheduling technique. The algorithm is not capable of improving the worst-case bound on schedulable utilization over 50%, but it can improve schedulable utilization in general cases, compared to the traditional static-priority scheduling algorithms that have been developed in prior work. In addition, the algorithm does not generate as many task preemptions as the traditional ones.

- The dynamic-priority scheduling algorithms have their bases on the EDF algorithm. The first algorithm is designed by applying the dynamic-priority assignments to the static-priority scheduling algorithm stated above. The algorithm is also not capable of improving the worst-case utilization bound over 50%, but it can usually achieve higher schedulable utilization in practice. Several dynamic-priority scheduling algorithms that have been devised in prior work have greater utilization bounds than 50%, however those algorithms generally cause more task preemptions due to complex computations. The second algorithm can even improve the worst-case utilization bound up to 65%. To the best of my knowledge, no traditional algorithms, except for optimal ones that generate a great number of task preemptions and migrations, have ever achieved utilization bounds over 65%.

All the three algorithms are designed with concerning the practicality. They have the almost same computation complexities as the RM and EDF algorithms. The theoretical superiority of the algorithms in the worst case is presented in schedulability analysis. In addition, the performance advancement of the algorithms in general cases are validated by several sets of simulation studies.

- Two techniques for the scheduling of aperiodic tasks on multiprocessor systems are proposed to reduce the mean response time. The first technique dispatches an arriving aperiodic tasks to a processor on which the response time is estimated to be minimized. The second technique makes efficient use of task migrations. More precisely, periodic tasks with higher priorities than the dispatched aperiodic task are temporarily migrated onto different processor to hand over processor
time to the pending aperiodic tasks, as long as the periodic tasks are guaranteed
to meet their deadlines after migrations. Since aperiodic tasks obtain additional
processor times, their response time are expected to be reduced. Two server
algorithms are designed, in which the proposed techniques are exploited under
two priority disciplines.

– The static-priority server algorithm is designed based on the traditional Pri-
ority Exchange algorithm [8] to work with RM scheduling. The PE algo-
rithm is known to be efficient in terms of performance and simplicity in
the domain of uniprocessor systems. Applying the proposed techniques to
the PE algorithm, the response time of aperiodic tasks can be dramatically
reduced, compared to the pure PE algorithm.

– The dynamic-priority server algorithm is designed based on the traditional
Total Bandwidth Server (TBS) algorithm [16, 17] to perform with EDF
scheduling. In addition to the fact that the TBS algorithm has been known
to be the most efficient in the traditional server algorithms developed in the
domain of uniprocessor systems, its characteristics are more suitable for the
proposed techniques. Therefore, the potential performance of the algorithm
is promising when the proposed techniques are combined.

The proposed techniques are able to be combined with the other traditional server
algorithms. However, the dissertation believes that the PE algorithm and the TBS
algorithm are the most suitable for the proposed techniques. The designs of the
PE-based algorithm and the TBS-based algorithm, in which the proposed tech-
niques are exploited, are also compatible to the scheduling algorithms developed
in this research, though the theoretical description of the algorithms is based on
the traditional RM algorithm and EDF algorithm for simplicity. The algorithms
have chances to generate an additional task migration only when an aperiodic
task arrives at the system. Thus, the prospective runtime overhead is not going to
matter as much. The theoretical superiority of the algorithms is presented through
some examples. The practical performance of the algorithms is validated through
several sets of the simulation studies.

The target area of real-time systems is extensive. Even within a humanoid robot,
various types of tasks are submitted. For instance, the required performance of I/O
control tasks is totally different from that of multimedia tasks. In general, the I/O
control tasks have severe restrictions: deadlines are hard in a true sense, periodic jitters
are not tolerated, computations are not preemptive and migrative, and so on. Such
types of tasks are usually scheduled in a simple manner. Thus, the target real-time
computing of this research is limited to less-restricted computations. More specifically,
real-time applications that require high schedulability and high responsiveness, such
as image/audio/speech processing, non-I/O control, action planning, map generation,
etc., are targeted in this research. Extensions for more restricted computing are not considered in this research, but left for the future work.

Finally, the scope of this research is limited to providing a methodology of scheduling periodic and aperiodic tasks on multiprocessor systems. In concrete terms, the research focuses on periodic scheduling algorithms and aperiodic server algorithms. It is not the scope of this research to consider the entire design of the system. It is also not the scope of this research to provide optional techniques such as synchronization, resource reservation, adaptive computing, dynamic voltage and frequency, and so on. In addition, this research does not focus on hardware, language, and compiler support for real-time computing on multiprocessor systems. In particular, extracting appropriate time attributes such as the worst case execution time from programs, or developing a method that facilitates its precise estimation is out of concern in this research.

1.4 Organization

The rest of the dissertation is organized as follows. Chapter 2 summarizes the state of the art techniques related to real-time scheduling on multiprocessor systems. The techniques are summarized in terms of performance and complexity. Chapter 3 defines the system model of this research. Notice that the effectiveness of the scheduling techniques established in this research is not necessarily limited to the defined system model. Chapter 4 presents a new technique for the scheduling of periodic tasks on multiprocessor systems. Three algorithms are developed based on the presented scheduling technique. The schedulability analysis is also provided to derive the schedulable condition and the worst-case theoretical utilization bounds for the algorithms. In addition, several heuristic techniques are introduced to improve the schedulability of the algorithms in general cases. Chapter 5 presents two new techniques for the scheduling of aperiodic tasks on multiprocessor systems. Two server algorithms are considered so that the presented techniques are exploited. The discussion of heuristic techniques is given to improve the responsiveness of the algorithms. Chapter 6 evaluates the effectiveness of the developed algorithms through simulation studies by comparing with the traditional scheduling algorithms and server algorithms. Chapter 7 concludes the dissertation and gives insights to the future directions of this research.
Chapter 2

STATE OF THE ART

This chapter presents a survey on the state of the art techniques for the scheduling of periodic tasks on multiprocessor systems. The survey is conducted from the viewpoint of improving system utilization of periodic tasks with a guarantee of timing constraints. Meanwhile, the scheduling of aperiodic tasks has received little attention compared to the scheduling of periodic tasks in the domain of multiprocessor systems, hence this chapter also presents a survey on the prior work of aperiodic server techniques for uniprocessor systems as well as those for multiprocessor systems. The survey is performed from the viewpoint of reducing the mean response time of aperiodic tasks.

The survey begins with a classification of real-time scheduling principles for multiprocessor systems, which dominates the form of scheduling algorithms. The survey of scheduling algorithms for multiprocessor systems is given next to establish the context of periodic task scheduling. Then, the survey moves on server algorithms for uniprocessor systems to study bases of aperiodic task scheduling. For reference, a few extensions of the server algorithms for multiprocessor systems are also introduced. In the survey, the trade-offs between achievable system utilization with real-time constraints and the complexity of the algorithms are discussed. The survey ends with summarizing the traditional approaches in terms of performance and complexity.

2.1 Classification of Scheduling Principles

Traditionally, there have been two approaches for the real-time scheduling of periodic tasks on multiprocessor systems: global scheduling and partitioned scheduling [43]. The form of scheduling algorithms is deeply dependent on which scheduling approach is based. In global scheduling, all eligible tasks are stored in a single priority-ordered queue; the global scheduler selects for execution the same number of the highest priority tasks as processors from this queue. The relative order of the task priorities varies depending on which tasks are eligible, hence a task may change a processor to run every selection; a task may migrate among processors. Unfortunately, using this approach
with traditional optimal algorithms for uniprocessor systems, such as the RM algorithm and the EDF algorithm, may result in arbitrarily low system utilization in multiprocessor systems [42]. However, recent research has overcome this crisis and even produced optimal methods as stated in the next section.

In partitioned scheduling, on the other hand, each task is assigned to a single processor on which each of its jobs will be executed, and processors are scheduled independently. Therefore, a task is executed on a dedicated processor and never migrates among processors. The main advantage of the partitioned scheduling is that they reduce a multiprocessor scheduling problem to a set of uniprocessor ones. Unfortunately, the partitioned scheduling has two negative consequences. First, finding an optimal assignment of tasks to processors is a bin-packing problem, which is NP-hard in the strong sense. Thus, tasks are usually partitioned using suboptimal heuristics. Second, as shown later, task sets exist that are schedulable if and only if tasks are not partitioned. Still, the partitioned scheduling approaches are widely used in real systems for simplicity of computation.

In addition to the above classification in which the scheduling is categorized whether inter-processor task migration is permitted or not, the conventional classification with respect to the complexity of the priority assignment scheme is also subsistent. Along
CHAPTER 2. STATE OF THE ART

Figure 2.2: Partitioned scheduling approach

Ever since Dhall and Liu exposed the problematic behaviors of the RM algorithm and the EDF algorithm in multiprocessor systems [42], a great number of scheduling algorithms specific for multiprocessor systems have been discussed with the bases of both the partitioned scheduling and the global scheduling. The primary goal of those algorithms is to manifest and improve the worst-case achievable system utilization at which all the jobs are guaranteed to meet their deadlines. This worst-case achievable system utilization with a guarantee that all tasks are schedulable is often called utilization
Note that, in the optimal case, a task set with the total utilization of tasks up to $M$ is schedulable on $M$ processors, which indicates that the utilization bound is always maintained to be 100%.

The following sections explain the well-known scheduling algorithms specific for multiprocessor systems. For understanding the worthiness of those algorithms, the characteristics and the problems of the RM algorithm and the EDF algorithm in multiprocessor systems are first stated. Those pure forms of the RM algorithm and the EDF algorithm are categorized into the global scheduling class. Then, the partitioned scheduling approaches, which have been preferred to the global scheduling approaches for ages in terms of theoretical simplicity, are presented. The explanation finally backs in the sophisticated global scheduling approaches that break through the boundary of the partitioned scheduling approaches.

2.2.1 Pure RM and EDF

The dissertation first of all reviews the Dhall’s effect illustrated in Figure 1.1. It occurs for the case in which a system includes heavy tasks. Andersson et al. hence focused on the schedulability of a system that does not include heavy tasks [45]. According to Andersson et al., a periodic task set is defined to be a light system on $M$ processors, if it satisfies that (i) the utilization of every individual task is less than or equal to $M/(3M - 2)$, and (ii) the total utilization does not exceed $M^2/(3M - 2)$. They then proved that any periodic task set that is a light system on $M$ processors is schedulable by the RM algorithm, if the tasks are all preemptive. Baruah and Goossens claimed the similar result that a set of tasks, all with deadline equal to period, is guaranteed to be schedulable by the RM algorithm, if (i) the utilization of every individual task is less than or equal to $1/3$, and (ii) the total utilization does not exceed $M/3$. Those results were generalized by Baker [46] so that a set of periodic tasks, all with deadline equal to period and utilization less than or equal to one, is guaranteed to be schedulable on $M$ processors using the RM algorithm, if the total utilization of the tasks does not exceed $M(1 - U_{max})/2 + U_{min}$, where $U_{max}$ and $U_{min}$ are respectively the maximum and minimum utilizations of every individual task. Baker also derived tighter (more precise) schedulability tests for the multiprocessor RM algorithm in exchange for higher computation orders.

The schedulability of the EDF algorithm for multiprocessor systems has been precisely clarified in the similar manner to the RM algorithm. Srinivasan and Baruah defined a periodic task set to be light in the EDF-scheduled system on $M$ processors, if it satisfies that (i) the utilization of every individual task is less than or equal to $M/(2M - 1)$, and (ii) the total utilization does not exceed $M^2/(2M - 1)$. They then proved that any periodic task set that is light on $M$ processors is scheduled to meet all deadlines by the EDF algorithm. This analysis was refined by Goossens et al. [47] so that any periodic task set which has the total utilization less than or equal to $M(1 - U_{max}) + U_{max}$, where $U_{max}$ refers to the maximum utilization of every individual task, is guaranteed
to be schedulable by the EDF algorithm. Baker moreover generalized those results and provided more precise schedulability [48].

2.2.2 Partitioned Scheduling Approach

Recall that, in partitioned scheduling, each task is assigned to a processor on which it will be exclusively executed. Finding an optimal assignment of tasks to processors is equivalent to a bin-packing problem, which is known to be NP-hard in the strong sense. Several polynomial-time heuristics have been proposed for solving this problem. The bin-packing problem is to pack a given set of items with different sizes into a minimum number of equal-sized bins. In those heuristics, the utilization of each task, which is the ratio of its execution time and its period, is considered to be the size of each item and the processor capacity, which is the per-processor utilization bound, is considered to be the size of each bin. Examples include the first-fit (FF) and best-fit (BF) heuristics. In the FF heuristic, each task is assigned to the first (i.e., lowest-indexed) processor that can accept the task. On the other hand, in the BF heuristic, each task is assigned to a processor that (i) can accept the task, and (ii) will have minimal remaining spare capacity after its addition. Whether a processor can accept a task or not depends on the feasibility test corresponding to the uniprocessor scheduling algorithm being used. The feasibility test is for instance that a task can be appended to a processor if the total utilization of the tasks is less than or equal to 1 (100%) in the EDF algorithm or \( N(2^{1/N} - 1) \) in the RM algorithm where \( N \) is the number of the tasks.

The two RM-based algorithms based on the partitioned scheduling are often referred to. The first one is the RM with FF (RM-FF) algorithm [42] and the second one is the RM with FF in decreasing utilization (RM-FFDU) algorithm [49]. The FFDU heuristic sorts a task set in decreasing order of task utilization before the FF heuristic is applied. It is known that the RM-FFDU algorithm succeeds in assigning more tasks to processors compared to the RM-FF algorithm [50]. Since the performance of those algorithms highly depends on a feasibility test for each processor assignment, Burchard et al. proposed new sufficient feasibility tests for RM-scheduled uniprocessor systems that perform better when task periods satisfy certain relationships [51]. They also proposed new heuristics that attempt to assign tasks satisfying those relationships to the same processor, thus leading to better overall utilization. The R-BOUND-MP algorithm [52] is another efficient algorithm that takes similar schedulability tests and heuristics, in which tasks are initially sorted in order of increasing periods.

Although the scheduling algorithms stated above contributed to RM-based partitioned scheduling, no work have derived the absolute schedulability. Oh and Baker proved that a task set is guaranteed to be schedulable by the RM-FF algorithm if the system utilization does not exceed \( \sqrt{2} - 1 \approx 41\% \) [53]. They also proved that no partitioned static-priority scheduling algorithms can achieve a utilization bound greater than 50%. More recently, because 41\% was a lower bound and there was still room for improvement, Lopez et al. presented a tighter utilization bound for the RM-FF al-
algorithm by taking into account not only the number of processors but also the number of tasks and their every individual utilization [54]. Using a similar technique, Lopez et al. also clarified the utilization bound for the RM-FFDU algorithm in their later work [55, 56]. Andersson et al. considered using the next-fit-ring (NFR) heuristic for the R-BOUND-MP algorithm instead the FF heuristic [57]. The utilization bound for the improved algorithm, called R-BOUND-MP-NFR, was finally derived to be 50%. Hence, the problem of partitioned static-priority scheduling was closed.

In contrast to a considerable amount of research on the static-priority scheme, there have been less algorithms proposed for the dynamic-priority scheme of the partitioned scheduling. Lopez et al. showed that the EDF algorithm with the FF (or BF) heuristic, so-called EDF-FF and EDF-BF respectively, can successfully schedule any task set with total utilization at most \((\beta M + 1)/(\beta + 1)\) on \(M\) processors, where \(\beta = \lfloor 1/\alpha \rfloor\) and \(\alpha\) is the maximum utilization of every individual task [58, 59]. Letting \(\alpha = 1\) and \(\beta = 1\), the worst-case utilization bound of \((M + 1)/2\) is derived. They also demonstrated that the EDF-BF algorithm offers slightly higher schedulable utilization than the EDF-FF algorithm in most situations.

Baruah and Fisher proposed an algorithm that sorts a task set in order of increasing relative deadlines and combined it to the EDF algorithm [60]. To know what kinds of partitioning algorithms are efficient, Baker compared three partitioning algorithms from the viewpoint of task sorting methods [61]: the first algorithm sorts a task set in order of increasing utilization, the second algorithm does in order of increasing density, and the third algorithm does in order of increasing relative deadline. The density is a ratio of an execution time and the minimum of a period and a relative deadline. The conclusion was that the method of sorting a task set in order of increasing density performs slightly better than the other two methods.

As stated above, a scheduling problem for multiprocessor systems can be reduced to a set of ones for uniprocessor systems in partitioned scheduling, since after a given task is assigned and partitioned to processors, every subset of them is scheduled on each processor independently according to the policy of conventional uniprocessor scheduling. Moreover, the partitioned scheduling often provides good performance, namely it achieves high schedulable utilization, despite its simplicity of algorithm design. Therefore, partitioned scheduling approaches have been preferable for practical use. However, they have a critical disadvantage that the worst-case utilization bound is potentially limited to at most 50% regardless the priority assignment schemes. Figure 2.3 depicts an example of such a partitioning problem. Consider that \(M + 1\) tasks have the same utilizations \(1/2 + \alpha\%\). Since an individual processor cannot be utilized over 100%, there is no spare room for the \(M + 1\)th task in any processor when the \(M\) tasks are assigned. Letting \(\alpha = 0\), a task set with a total utilization over \((M + 1)/2\) is never schedulable. They also have online problems that when a new task is submitted to the system at runtime, the tasks may be required to be sorted again to accept the new task. Such repartitioning can incur significant runtime overhead. Those drawbacks of the partitioned scheduling lead to revival of the global scheduling approaches to realize more sophisticated scheduling of periodic tasks on multiprocessor systems.
2.2.3 Global Scheduling Approach

Baruah et al. proposed the *proportionate fairness* (Pfair) scheduling method that can produce an optimal periodic execution on multiprocessor systems [62]. In Pfair scheduling, each periodic task is divided into quantum-size pieces called subtasks. Let $T$ be a periodic task and the utilization of $T$ is $U$. Let $T_i$ be the $i$th subtask of $T$. In Pfair scheduling, $T_i$ has a pseudo release time of $r(T_i) = \lfloor (i-1)/U \rfloor$ and a pseudo deadline of $d(T_i) = \lceil i/U \rceil - 1$. Then, $T_i$ must be executed in a window of $[r(T_i), d(T_i)]$. Hence, $T$ is guaranteed to be executed proportionately. If all the subtasks meet their pseudo deadline, the original task also meets its deadline. Figure 2.4 shows Pfair windows of a task with utilization $8/11$. Since the task has an execution time of 8 time units, it has 8 subtasks (windows) with every period 11. The three Pfair algorithms, PF [63], PD [62], and PD² [64], are known to be optimal under any circumstances. The difference among the three algorithms is a rule of breaking ties in pseudo deadlines of subtasks. There is another Pfair algorithm called EPDF [65] that is optimal for two-processor systems.

Taking the advancement of the Pfair scheduling method, Ramamurthy et al. proposed the Weight-Monotonic (WM) algorithm that applies a static-priority scheme to the Pfair scheduling [66, 67]. While the original Pfair algorithms schedule subtasks according to the policy of earliest pseudo deadline first, the WM algorithm does according to the policy of highest weight (utilization) first. The utilization bound for the WM algorithm was proved to be 50% in the later work [57]. Since the common for static-priority scheduling on multiprocessor systems is that it can never have a utilization bound greater than 50% [53, 45], the WM algorithm is optimal for static-priority scheduling as well as the R-BOUND-MP-NFR algorithm.
One of the well-known drawbacks against the Pfair algorithm is that they cause a great number of task preemptions due to quantum-based scheduling, which leads to significant run-time overhead, though the practical performance of Pfair scheduling was reported [68, 69]. Seeking the practicality on global scheduling, Andersson et al. invented a static-priority scheduling algorithm called RM with utilization separation at $M/(2M - 1)$ (RM-US[$M/(2M - 1)$]) [45]. Hereinafter the description of $'[M/3M - 2]'$ is omitted for simplicity of description. The primary purpose of the RM-US algorithm is an avoidance of the Dhall’s effect [42]. The Dhall’s effect happens because of heavy (high-utilization) tasks having low priorities. They first showed that any periodic task set in which the utilization of every individual task is at most $M/(2M - 2)$ can be scheduled successfully on $M$ processors if the total utilization of tasks is at most $M^2/(3M - 2)$. Referring to this result, the RM-US algorithm statically places the highest priority to tasks whose utilizations are greater than $M/(2M - 2)$. The utilization bound for the RM-US algorithm was proved to be $M/(2M - 2)$, namely a task set with a total utilization less than $M^2/(3M - 2)$ can be guaranteed to be schedulable by the RM-US algorithm. Since the global RM algorithm cannot guarantee timing constraints once the total utilization of tasks exceeds 1, the effectiveness of the RM-US algorithm can be recognized. Figure 2.5 shows a scheduling example of the RM-US algorithm with respect to the same task set used in Figure 1.1 to show the Dhall’s effect. Since a heavy task is assigned the highest priority, the Dhall’s effect can be avoided. In the later work, the RM-US algorithm was generalized as the RM-US[$\lambda$] algorithm for applying any task set in which the maximum utilization of every individual task is $\lambda$ [46]. Baruah and Goossens proved that the value $\lambda = 1/3$ is optimal with respect to verifying schedulability using the utilization bound test [70]. It is also argued that the true optimum value of $\lambda$ is approximately 0.3748225282 [71]. The basis of the argument is an assertion that a task experiences the maximum competing load when there is block interference from other tasks.

The notion of the RM-US algorithm is also used in dynamic-priority scheduling. Srinivasan and Baruah examined EDF-based scheduling of periodic tasks on multiprocessor systems [72]. Srinivasan and Baruah showed that any periodic task set in which the utilization of every individual task is at most $M/(2M - 1)$ can be scheduled suc-
successfully on $M$ processors if the total utilization of tasks is at most $M^2/(2M - 1)$. They then proposed a hybrid scheduling algorithm, called EDF with utilization separation at $M/(2M - 1)$ (EDF-US[$M/(2M - 1)$]), that gives the highest priority to tasks with utilizations above $M/(2M - 1)$, and proved that the algorithm is able to successfully schedule any periodic task set with total utilization of tasks is up to $m^2/(2m - 1)$. These results were generalized in [47], where it is shown that any periodic task set can be scheduled successfully by the EDF algorithm if the total utilization of tasks is at most $M(1 - U_{\text{max}}) + U_{\text{max}}$, where $U_{\text{max}}$ is the maximum utilization of every individual task. Baker analyzed this general case of the EDF-US[$\lambda$] algorithm, in which tasks with utilizations higher than $\lambda$ are always placed highest priority, and gave the conclusion that the EDF-US[$1/2$] algorithm is optimal with a guaranteed worst-case schedulable system utilization of $(M + 1)/2$ [48].

In a broad sense, the approach of placing the highest priority to heavy tasks is effectual to improve schedulability. A similar idea was used in the Earliest Deadline Zero Laxity (EDZL) algorithm [73]. The EDZL algorithm places the highest priority to tasks which have zero laxity. The laxity of a task indicates the remaining time amount to the deadline from the current time in which the remaining execution time of the task is subtracted. Letting $d$ be the deadline, $t$ be the current time, and $e$ be the remaining execution time of a task respectively, the laxity is expressed by $(d - t) - e$. Since the laxity of a task varies at every moment, the priority of the task becomes the highest in mid-flow of execution when the task has zero laxity. The authors in [73] proved that the EDZL algorithm is at least as effective as the EDF algorithm. In other words, any task sets that are schedulable under EDF scheduling is always schedulable under EDZL.
CHAPTER 2. STATE OF THE ART

scheduling. Figure 2.6 shows a scheduling example of the EDZL algorithm with respect to the same task set used in Figure 1.1 to show the Dhall’s effect. At time $\varepsilon$, the $M + 1$th task reaches the zero laxity, and the first task is preempted. As a result, all the deadlines are met. It has been shown that any task sets can be scheduled successfully by the EDZL algorithm on $M$ processors if the total utilization of tasks does not exceed $(M + 1)/2$ [74]. Cirinei and Baker gave more precise schedulability tests for the EDZL algorithm [75], using similar techniques presented in their previous work [76, 77]. They also examined experimental evaluations to compare the performance of the EDZL algorithm with other EDF-based global scheduling algorithms. Wei et al. moreover investigated the schedulability of the EDZL algorithm [78]. They derived that the utilization bound for the EDZL algorithm on two processors is $3/2 + |U_{\text{max}} - 1/2|$ where $U_{\text{max}}$ is the maximum utilization of individual tasks. The common opinion in the work on EDZL scheduling is that there is room for improvement in the utilization bound.

Cho et al. backed in the optimal scheduling problem again, with considering less task preemptions [79]. They proposed an optimal algorithm that schedules a task with the largest nodal remaining execution time first (LNREF). Unlike the Pfair algorithms, the LNREF algorithm does not rely on the quantum-based scheduling approach but on the original notion called T-L plane solution. The algorithm makes a fluid schedule path every interval of job arrivals. When a task sees a time at which it cannot meet the fluid schedule path at the end of the interval, the task is assigned the highest priority. A scheduling example of four tasks by the LNREF algorithm is illustrated in Figure 2.7. The authors claimed that the LNREF algorithm may reduce the number of task preemptions compared to the Pfair algorithms, since it does not need to invoke a scheduler.
at every quanta. Besides, the number of task preemptions in LNREF scheduling can be bounded, while that in Pfair scheduling cannot be precisely bounded. However, the T-L plane solution is still expensive especially when jobs arrive in short intervals.

### 2.2.4 Another Scheduling Approach

The pros and cons of the partitioned scheduling and the global scheduling have been argued for a long time. The common agreements is that the partitioned scheduling generates merits of less complexity and no task migrations but has a barrier of the utilization bound, while the global scheduling can improve schedulable utilization in exchange for more task preemptions and migrations.

Anderson et al. devised a middle approach [80]. In their approach, most of tasks are partitioned but the selected tasks can migrate to different processors. They proposed the EDF-based algorithm called EDF-fm (fm denotes that each task is either fixed or migrating). The EDF-fm algorithm first assigns each task to a processor, like partitioning, as long as the processor capacity is not exhausted. If some task cannot receive its full share of utilization from the processor, namely the task will cause the processor to be overutilized if assigned, then a part of the processor capacity that it requires is allocated on the neighbor processor such that the sum of the shares allocated to the task on the two processors equals the utilization of the task. More concretely, let {$u$} be the utilization of a task that is about to be assign to a processor with the remaining spare
branch capacity of \( s \) that is less than \( u \). In this case, the task cannot fit in the processor because of \( s < u \). The EDF-fm algorithm assigns \( u' = s/u \) share of its utilization to the processor and the remaining \( u'' = (u - s)/u \) share of its utilization to the neighbor processor. Hence, each task is assigned to either one or two processors in the EDF-fm algorithm. The tasks assigned to two processors are called migrating tasks, while the task assigned to one processor are called fixed tasks. The jobs of migrating tasks are assigned to processors using static rules that are independent of runtime dynamics so that the jobs that form the migrating task are not executed in parallel on two processors. The rest of the jobs are scheduled according to the EDF algorithm. This priority scheme ensures that migrating tasks never miss their deadlines. However, the scheduling methodology of the EDF-fm algorithm cannot guarantee fixed tasks to meet their deadlines, instead it can guarantee their tardiness which is a delay from the deadline. Thus, the target domains of the EDF-fm algorithm are restricted to soft real-time systems. The EDF-fm algorithm also has a limitation on the utilization of every individual task at most \( 1/2 \).

The advantage of this approach is that it can allocate the full processor capacity to tasks, since a task inducing overutilization of a processor is always split into two portions and the share of its first portion being assigned to the processor is same as the remaining spare capacity of the processor. In addition, the number of migrating tasks is bounded at most \( M - 1 \) on \( M \) processors. As a result, the number of task preemptions and migrations is limited, while a system can be fully utilized. The matter of the EDF-fm algorithm in dispute is that fixed tasks cannot have hard deadlines.

Andersson and Tavor settled the matter of the EDF-fm algorithm by partially introducing a notion of fluid scheduling [81]. The algorithm they proposed is called EDF with task splitting and \( K \) processors in a Group (EKG). The task assignment rule of the EKG algorithm is almost same as that of the EDF-fm algorithm, which means that at most \( M - 1 \) tasks are split on \( M \) processors. While the EDF-fm algorithm prioritized the split tasks statically over the rest of the tasks, the EKG algorithm schedules all the
tasks based on dynamic priorities so that every split task is not executed in parallel on two processors. Let \( u' \) and \( u'' \) be the utilizations of a split task on Processor 1 and Processor 2 respectively. Also let \( t_1 \) be the release time of any job on any processor, and let \( t_1 \) be the release time of the next job of any task on any processor. The split task begins execution at time \( t_1 \) and consumes the time units of \( u'(t_2 - t_1) \) without any preemptions on Processor 1, then consumes the time units of \( u''(t_2 - t_1) \) and ends execution at time \( t_2 \) without any preemptions on Processor 2 after migrating from Processor 1. Figure 2.8 depicts a scheduling example of the EKG algorithm on two processors with respect to three tasks, Task 1, Task 2, and Task 3, where Task 2 is split into Task 2' and Task 2''. According to the EKG algorithm, all the tasks can be guaranteed to meet their deadlines and a system can be fully utilized. That is, the EKG algorithm is optimal. The problem described there is that a split task may cause many preemptions if jobs are released in short intervals. The authors hence gave a trade-off solution. They introduced a parameter \( k \) that indicates the number of processors in each group. Tasks can be split only within a group. When the task assignment moves to a processor in the next group, a task cannot be split. In the case of \( k = 2 \), at most \( M/2 \) tasks are split, while at most \( M - 1 \) tasks are split in the case of \( k = M \) on \( M \) processors. Higher utilization bound is obtained with larger value of \( k \) that incurs more task preemptions. The utilization bound becomes \( 2/3 \approx 66\% \) for \( k = 2 \) and 100% for \( k = M \), which means that the algorithm is optimal for \( k = M \). The EKG algorithm however generates an unacceptable number of task preemptions for a large value of \( k \). Another drawback of the EKG algorithm is that the scheduling of split tasks requires fine-grained timers, since it relies on the fluid scheduling model.

### 2.3 Server Algorithms

This section looks back the traditional techniques for the scheduling of aperiodic tasks on uniprocessor systems to study basic approaches. Then, the extensions for multiprocessor systems are explored.

The survey focuses on improving the response time of aperiodic tasks without causing timing violations of periodic tasks. This can be usually achieved by the use of a server. A server is a special periodic task whose purpose is to service aperiodic requests as soon as possible. Like any periodic task, a server is characterized by a period and a fixed execution time, called server capacity. The server is scheduled with the same algorithm used for the periodic tasks, and, once active, it serves the aperiodic requests within the limit of its server capacity.

#### 2.3.1 Uniprocessor Approach

The most simple server algorithm is Polling Server (PS) [8, 10]. In the PS algorithm, at regular intervals equal to its period, the server becomes active and services any pending
aperiodic requests within the limit of its capacity. As the server services aperiodic requests, the capacity is consumed. When the capacity becomes zero, the server cannot continue to service until it becomes active again. If no aperiodic requests are pending, the server suspends itself until the beginning of its next period, and the time originally allocated for aperiodic service is not preserved for the next aperiodic service but is used by periodic tasks. Note that if an aperiodic task arrives just after the server has suspended, it must wait until the beginning of the next polling period, when the server capacity is replenished at its full value.

Lehoczky et al. invented Deferrable Server (DS) to improve the response time of aperiodic tasks with respect to polling service [8]. As the PS algorithm, the DS algorithm creates a periodic task, usually having the highest priority, for servicing aperiodic requests. However, unlike polling, the DS algorithm preserves the server capacity if no aperiodic requests are pending upon the invocation of the server. The capacity is maintained until the end of the period, so that aperiodic requests can be serviced at the same priority as the server at anytime, as long as the capacity has not been exhausted. At the beginning of any server period, the capacity is replenished at its full value. Thus, the DS algorithm provides much better aperiodic responsiveness than polling. However, the schedulable utilization somewhat decreases when the DS algorithm is applied to static-priority scheduling, since the server defers its execution, which may cause a violation of the basic assumption that every time the task with the highest priority is executed under static-priority scheduling. The schedulability for the RM algorithm when the DS algorithm is applied was analyzed in detail by Strosnider et al. [15].

Lehoczky et al. also invented Priority Exchange (PE). With respect to the DS algorithm, the PE algorithm has a slightly worse performance in terms of aperiodic responsiveness but provides better schedulable utilization for periodic tasks. Like the DS algorithm, the PE algorithm uses a periodic server, usually at a high priority, for servicing aperiodic tasks. However, it differs from the DS algorithm in the manner in which the capacity is preserved. Unlike the DS algorithm, the PE algorithm preserves its high-priority capacity by exchanging it for the execution of a lower-priority periodic task. At the beginning of each server period, the capacity is replenished at its full value. If aperiodic requests are pending and the server is the ready task with the highest priority, then the requests are serviced using the available capacity; otherwise the remaining capacity is exchanged for the execution of the active periodic task with the highest priority.

Sprunt et al. proposed two server algorithms: Extended Priority Exchange (EPE) [9] and Sporadic Server (SS) [10]. Those algorithms are the extensions of the above two server algorithms in the manner of capacity preservation and capacity replenishment. Unfortunately, despite their computation complexity, the EPE algorithm and the SS algorithm could not offer outstanding performance improvement compared to the DS algorithm and the PE algorithm. Thus, the prior two algorithms are often preferable.

Slack Stealing [11, 12] is another aperiodic service technique, proposed by Lehoczky and Ramos-Thuel, which offers substantial improvements in response time over the
previous service methods, i.e. PS, DS, PE, EPE, and SS. Unlike these methods, the Slack Stealing algorithm does not create a periodic server for servicing aperiodic tasks. Rather it creates a passive task which attempts to make time for servicing aperiodic tasks by stealing the processing time as much as possible from the periodic tasks without causing their deadlines to be missed. Unfortunately, it is widely known that the Slack Stealing algorithm needs a huge memory space to store the slack available at each task arrival, whereas it offers excellent performance. In order to reduce the required memory space, Davis et al. developed the Dynamic Slack Stealing algorithm [13] and the Dual Priority Scheduling algorithm [14], which steal the processor time dynamically. However, those algorithms sacrifice unacceptable runtime overhead in exchange for static memory requirements.

All the techniques described above are basically used with the RM scheduling algorithm. It is well known that the lowest bound of the CPU utilization in the RM algorithm is only 69%, whereas dynamic priority scheduling, such as the EDF algorithm and the Least Laxity First (LLF) algorithm [82], have a schedulable utilization of 100% in any cases. Spuri et al. proposed several aperiodic service techniques performing under dynamic priority scheduling [16, 17]. They firstly proposed the Dynamic Sporadic Server (DSS) algorithm and the Dynamic Priority Exchange (DPE) algorithm that are the extensions from the static-priority server algorithms of DS and PE so that they work under dynamic-priority scheduling. Because those algorithms have restrictions taken over from the original static-priority algorithms, they have certain performance boundaries. Spuri et al. also developed an efficient, probably the most efficient, algorithm called Total Bandwidth Server (TBS), which is more suitable for deadline-based scheduling. The TBS algorithm performs based on the assigned bandwidth, and offer excellent performance with retaining the optimality of dynamic-priority scheduling. Moreover, they invented an optimal algorithm called Earliest Deadline as Late as possible (EDL) server, and a suboptimal algorithm called Improved Priority Exchange (IPE). The EDL algorithm is an dynamic extension of the static-priority Slack Stealing algorithm. The IPE algorithm, on the other hand, is derived from the static-priority DPE algorithm and is motivated to make efficient use of idle times in the EDL algorithm.

### 2.3.2 Multiprocessor Approach

While many researchers have proposed server algorithms performing in uniprocessor scheduling, it is surprising that there have been very few server algorithms specific for multiprocessor systems. Baruah et al. considered a multiprocessor implementation of the TBS algorithm [83]. They derived the schedulable condition of the global EDF scheduling with the TBS algorithm. However, the design and policy of their algorithm are subject to the original TBS algorithm, rather the purpose of their work is to apply the TBS algorithm to multiprocessor systems, but not to improve the performance of the TBS algorithm.

Andersson et al. investigated both the global scheduling and the partitioned schedul-
ing of aperiodic tasks on multiprocessor systems [84, 85]. They designed a global aperiodic scheduling algorithm that meets all deadlines if 50% or less of the capacity is required. They also devised a partitioned aperiodic scheduling algorithm that meets all deadlines if 31% or less of the capacity is required. The contributions are significant with taking into account that no algorithms have ever guaranteed timing constraints of aperiodic tasks with those utilizations. However, the responsiveness to aperiodic tasks is never focused on in their work.

Banus et al. extended the Dual Priority algorithm so that it can be applied on multiprocessor systems [86]. The extended algorithm works in such a way that periodic tasks can be migrated to other processors to reduce the response time of aperiodic tasks, but the periodic tasks must be executed on the dedicated processors after the certain times. The former is called dynamic phase and the latter is called static phase. The concluding remarks was that their algorithm can even offer better performance than the Slack Stealing algorithm when the system is overloaded. Although they succeeded in reducing the response time on multiprocessor systems, the algorithm is based on the Dual Priority algorithm that takes complex computations at runtime. Thus, the practicality of the algorithm must be more discussed.

2.4 Summary

This chapter first gave a survey of the state of the art techniques for the scheduling of periodic tasks on multiprocessor systems. The scheme of scheduling is mainly classified into global scheduling and partitioned scheduling. In global scheduling, all tasks are managed by a unique scheduler with global priorities, and they are able to be executed on any processors. On the other hand, tasks are assigned to particular processors and a local scheduler on each processor manages the assigned tasks independently in partitioned scheduling. Scheduling algorithms are characterized as follows.

- The scheduling algorithms based on the global scheduling scheme are able to achieve optimal schedulable utilizations of 100%. However, they inevitably produce a great number of task preemptions and migrations, which incurs unacceptable overhead at runtime. In addition, implementation of the algorithms is likely to be more complicated.

- The scheduling algorithms based on the partitioned scheduling scheme require elementary computations and generate no migrations. However, they can never transcend a worst-case utilization bound of 50%. They also have online problems when a new task is submitted to the system at runtime.

The global scheduling and the partitioned scheduling have pros and cons. The scheduling algorithms designed based on those scheduling schemes suffer from the trade-off between schedulability and complexity. Recently, there has been propounded
another scheme which takes advantage of both the global scheduling and the partitioned scheduling. The EKG algorithm designed based on this new scheme can achieve an optimal schedulable utilization of 100% with less task preemptions than the global scheduling algorithms. However, the partial scheduling of the EKG algorithm becomes of complicated and the number of task preemptions is still expensive. In addition, implementation issues have not been argued very much.

This chapter also provided a survey on the traditional techniques for the scheduling of aperiodic tasks on uniprocessor systems, since the scheduling on multiprocessor systems has received little attention. The domain of uniprocessor systems have several efficient algorithms that offer good responsiveness with less complexity. However, only a few algorithms have been considered in the domain of multiprocessor systems. Thus, the basic form of aperiodic task scheduling on multiprocessor systems is required.
Chapter 3

SYSTEM MODEL

3.1 Processor Model

The system targeted in this research is a memory-shared multiprocessor composed of $M$ processors: $P_1, P_2, ..., P_M$. The code and data of programs (tasks) are shared among the processors. This research neglects the overhead of inter-processor communications. In other words, the cost of task migrations is not taken into account. Such an assumption has not been preferable in the traditional large scale multiprocessor systems such as on-board symmetric multiprocessors and network-connected processors. However, recent advancement of processor technology has somewhat allowed this kind of assumption. For example, the current multithreading techniques are able to eliminate the overhead of context switches dramatically, though the impact of eliminations depends on processor architectures. When threads are managed by hardware, the code and data are also easily shared and synchronized using particular registers and instructions. In fact, the responsive multithreaded (RMT) processor invented by Yamasaki [87, 88] has hardware functions to support fast context switches. For a good reference, every context switch can be done within four clocks in the RMT processor regardless of the context locations. In addition, the cost of switching contexts between processors can be deemed to be almost equal to that within a processor, when the code and data are shared among the processors. Thus, the overhead of switching and migrating contexts is vanishingly small. Those functions are fairly implementable. In fact, recent commercial processors have similar features that reduce the cost of context switches.

Notice that it still takes a cost to preempt tasks, since an operating system enqueues and dequeues tasks. However, the queuing cost is highly dominated by the processor performance. Thus, there is little point in taking notice of the cost of every task preemption. From the viewpoint of scheduling, it is more variant to focus on the number of task preemptions to assess the runtime overhead of the system. Hence, this research takes the number of task preemptions as a performance metric of runtime overhead.

This research also ignores the behavior of a cache system, because the configuration of caches is varied depending on processor performance and architectures. The perfor-
mance deterioration of computations due to transient degradation of the cache hit ratio caused by task preemptions and migrations is out of focus. In fact, such a deterioration affects estimation of the worst-case execution time (WCET). The real-time scheduling with a guarantee of timing constraints is not achievable in most cases, if the WCET of each task is not known a priori. However, the WCET of each task cannot be easily analyzed in the presence of cache systems, especially on multiprocessor systems. Thus, when the WCET analysis is concerned in the scheduling established in this research, using scratch-pad memories rather than cache systems is more suitable. Otherwise, the experimental WCET analysis [89] is also effective.

### 3.2 Task Model

Figure 3.1 illustrates the periodic task model. The system has a set of $N$ periodic tasks, denoted by $\Gamma = \{\tau_1, \tau_2, ..., \tau_N\}$. The $i$th periodic task is defined by $\tau_i = (C_i, T_i)$ where $C_i$ is its worst-case execution time and $T_i$ is its period ($C_i \leq T_i$). The processor utilization of $\tau_i$ is defined by $U_i = C_i / T_i$. A task generates a sequence of jobs periodically. The $j$th job of $\tau_i$ is denoted by $\tau_{i,j}$ that is released at time $r_{i,j}$ and has a deadline at time $d_{i,j} = r_{i,j} + T_i$. The consumed execution time and the remaining execution time of $\tau_{i,j}$ at time $t$ are denoted by $c_{i,j}(t)$ and $\bar{c}_{i,j}(t)$ respectively. The total utilization of the given task set is defined by $U(\Gamma) = \sum_{\tau_i \in \Gamma} U_i$. Letting $\Lambda$ be any subset of the given task set, the total utilization of the subset is also denoted by $U(\Lambda) = \sum_{\tau_i \in \Lambda} U_i$. In the scheduling algorithms developed in this research, periodic tasks are executed on dedicated processors. In other words, each processor has its own task set. Thus, the set of the tasks executed on processor $P_x$ is denoted by $\Pi_x$.

Figure 3.2 indicates the aperiodic task model. Aperiodic tasks arrive on each processor sequentially. The $k$th aperiodic task arriving on the system is defined by $\alpha_k(a_k, E_k)$ where $a_k$ is its arrival time and $E_k$ is its worst-case estimated execution time. Letting $f_k$ be the finish time of $\alpha_k$, the response time of $\alpha_k$ is denoted by $R_k = f_k - a_k$. The consumed execution time and the remaining execution time of $\alpha_k$ at time $t$ are denoted by...
\( e_k(t) \) and \( \tilde{e}_k(t) \) respectively, where \( e_k(t) + \tilde{e}_k(t) \) is satisfied for any \( t \). For any \( k \), \( a_k < a_{k+1} \) is satisfied. The load of the aperiodic tasks is denoted by \( U(\alpha) = \frac{\lambda}{\mu} \) where \( \mu \) is the average service rate and \( \lambda \) is the average arrival rate.

All the tasks are preemptive and independent. Thus, no tasks are assumed to have critical sections and synchronize with any other tasks. When the critical sections and the synchronizations are concerned in the scheduling established in this research, the schedulability analysis must be extended so that it can accept non-preemptive scheduling model and real-time synchronization protocols such as the priority ceiling protocol (PCP) [20] and the stack resource policy (SRP) [21]. Any jobs of a task cannot be executed in parallel, meaning that \( \tau_{i,j} \) cannot be executed in parallel on more than one processor for any \( i \) and \( j \). Although the previous section stated that the performance deterioration of periodic tasks due to transient degradation of the cache hit ratio is ignored in this research, the performance deterioration of aperiodic tasks must be concerned, since it results in prolongation of aperiodic executions, while the goal of the aperiodic executions is to reduce the response time of aperiodic tasks as much as possible. Therefore, this research establishes the algorithms based on the policy that periodic tasks can be migrated, but no aperiodic tasks can be migrated.
Chapter 4

PERIODIC TASK SCHEDULING

This chapter proposes a novel scheduling technique specific for multiprocessor systems, called *portioned scheduling*, to achieve high schedulable utilizations with a small number of task preemptions and migrations. The scheduling algorithms designed based on the portioned scheduling strategy are presented. The optimal scheduling algorithms, such as Pfair [62, 63, 64], LLREF [79], and EKG\((k = M)\) [81], can by definition achieve optimal schedulable utilizations of 100% literally. No better performances than the optimal algorithms can be obtained in the sense of schedulability. However, as stated already, the optimal algorithms suffer from the computation complexity that may bring unacceptable runtime overhead in the forms of task preemptions and migrations. In fact, non-optimal algorithms, such as EDF-FF [58], EDF-BF [58], EDF-US [47], EDZL [73], and EKG\((k < M)\), are preferred to the optimal algorithms for practical use in real environments. Although those efficient algorithms usually perform well, the schedulable utilizations drop down to 50% in the worst case.

The objective of the proposed scheduling algorithms is to achieve higher schedulable utilizations than the traditional non-optimal efficient algorithms with a small number of task preemptions and migrations. An important point to concern is that the scheduling algorithms must be designed so that they are practically implementable. Even if an algorithm performs well in theory, it does not necessarily perform well in practice. For example, the EDZL and EKG\((k < M)\) algorithms perform efficiently in the theoretical point of view, but they have problematic implementations issues such as handling the zero laxity and using fine-grained timers.

This chapter begins with explaining the basic strategy of the portioned scheduling technique. Then, three scheduling algorithms are presented. The first algorithm is *Rate Monotonic Deferrable Portion* (RMDP). The RMDP algorithm is developed for static-priority scheduling that is usually preferred to dynamic-priority scheduling in terms of predictability, release jitter and runtime overhead. The second algorithm is *Earliest Deadline Deferrable Highest-priority Portion* (EDDHP). The EDDHP algorithm is derived from the notion of the RMDP algorithm but is developed for dynamic-priority scheduling that is superior to static-priority scheduling in terms of schedulability. Those
two algorithms outperform the traditional efficient algorithms in the sense of general schedulable utilizations but never improve the utilization bounds. In other words, the utilization bounds of the RMDP and EDDHP algorithms are 50% in the worst case. The final algorithm, which is the major contribution of this research, is *Earliest Deadline Deferrable Portion* (EDDP). The EDDP algorithm is an extension of the EDDHP algorithm by releasing the restriction of the static-priority assignment derived from the RMDP algorithm and it achieves a utilization bound of 65%.

### 4.1 Basic Strategy

The portioned scheduling technique established in this research is not a brand new insight. In the past work, Anderson *et al.* [80] and Andersson *et al.* [81] studied very similar scheduling techniques. However, the form of those scheduling approaches is never strongly defined, compared to the conventional global scheduling and partitioned scheduling techniques. First of all, the form of the portioned scheduling technique is technically defined in this research.

The portioned scheduling strategy is composed of the two phases: the task assigning phase and the task scheduling phase. In the task assigning phase, each task is assigned to a particular processor like partitioning, as long as the task does not cause the total utilization of the processor to exceed its utilization bound. When the total utilization of the processor exceeds the utilization bound, the task on the assignment is virtually split into two portions in the sense of utilization, whereas in the partitioning strategy the task
CHAPTER 4. PERIODIC TASK SCHEDULING

is assigned to another processor which can receive the full utilization of the task. As for the virtually-split task, one portion is assigned to the processor to which the tasks are being assigned, and the other portion is assigned to the next processor to which the rest of the tasks will be assigned. Notice that "virtually-split" means that the task is not really divided into two blocks, but its utilization is shared on the two processors. In other words, the processor capacity that the virtually-split task will use for execution is reserved on the two processors. From the scheduling point of view, the split portions are pseudo tasks to make a schedule time for the original task.

Figure 4.1 shows an example of splitting tasks, defined portioning in this research. The width of the box in which the task name is indicated is the processor utilization of the task. This example presumes the case in which two tasks, $\tau_i$ and $\tau_j$, are already assigned to processor $P_x$, and another task $\tau_k$ is causing the total utilization of $P_x$ to exceed its utilization bound. In this case, the partitioning approach just assigns $\tau_k$ to another processor $P_y$. The partitioning approach, on the other hand, virtually splits $\tau_k$ into $\tau'_k$ and $\tau''_k$. In the dissertation, $\tau'_k$ is defined the first portion of $\tau_k$ and $\tau''_k$ is defined the second portion of $\tau_k$. Then, $\tau'_k$ is assigned to $P_x$ and $\tau''_k$ is assigned to $P_y$. Hence, the partitioning approach obviously improves the total utilization of $P_x$. The execution times of $\tau'_k$ and $\tau''_k$, which are namely the reserved processor capacities of $\tau_k$ within every period $T_k$ on the two processors, are denoted by $C'_k$ and $C''_k$ respectively. Letting $U^*_x$ be the utilization bound of $P_x$, $C'_k$ and $C''_k$ are calculated as follows.

$$C'_k = T_k(U^*_x - U_i - U_j)$$
$$C''_k = C_k - C'_k$$

This means that $\tau_k$ consumes $C'_k$ time units on $P_x$ and $C''_k$ units on $P_y$ within every
T_k. Notice that \( \tau'_k \) and \( \tau''_k \) form the same job of \( \tau_k \), thereby they are not allowed to be executed in parallel. Apart from that, they are treated as general periodic tasks in terms of theoretical real-time scheduling. Thus, they can be executed in any order. The implementation concern is only that the scheduler needs to track their execution times to recognize the completions of split portions \( \tau'_k \) and \( \tau''_k \). The portion that exhausts its capacity in first must be preempted by the scheduler, because it does not recognize consuming its all capacity by itself. Hence, the scheduler must preempt the split task \( \tau_k \) when it exhausts either \( C'_k \) on \( P_x \) or \( C''_k \) on \( P_y \), so as not to let \( \tau'_k \) and \( \tau''_k \) overrun their capacities. Such a mechanism is similar to the resource reservation technique [22, 23].

The split portions \( \tau'_k \) and \( \tau''_k \) are for instance scheduled as shown in Figure 4.2. They are scheduled on \( P_x \) and \( P_y \) individually but exclusively. When \( \tau'_k \) is scheduled, \( \tau_k \) is actually executed on \( P_x \). When \( \tau''_k \) is scheduled, on the other hand, \( \tau_k \) is actually executed on \( P_y \). The result scheduling of \( \tau_k \) is that it is executed with migrating between two processors \( P_x \) and \( P_y \). As described above, the ordering of \( \tau'_k \) and \( \tau''_k \) does not have any limitations. As long as they are scheduled exclusively, they can be scheduled sequentially, alternately, and randomly.

4.2 The RMDP algorithm

This section presents the RMDP algorithm that integrates the portioned scheduling strategy with the well-known RM algorithm [1]. At first, the task assigning algorithm and the task scheduling algorithm are proposed. Then, the schedulable condition and the utilization bound for the RMDP algorithm are analyzed. The dissertation also proves that the RMDP algorithm performs optimally, i.e. it achieves a utilization bound of 100\%, for the special case of harmonic task sets.

4.2.1 Task Assigning Phase

The task assigning algorithm of RMDP is straightforward. The algorithm assumes that the given task set is sorted so that the period of \( \tau_i \) is smaller than or equal to that of \( \tau_{i+1} \) for any \( i \). Then, the algorithm assigns the tasks to the processors sequentially, which means that it always assigns \( \tau_{i+1} \) after \( \tau_i \), and if some task causes the total utilization of a processor to exceed its utilization bound, its splits the task into two portions. The first portion is assigned to the processor which is caused to exceed its utilization bound, and the second portion is assigned to the next processor to which the following tasks will be sequentially assigned. The algorithm continues this procedure until all the tasks are assigned or no processors remain the spare capacities.

Figure 4.3 shows the pseudo code of the RMDP assigning algorithm. First of all, the indexes of the tasks and the processors, and the number of the assigned tasks are initialized (1st line). The variables related to the calculation of the utilization bound are also initialized (2nd line). Then, all the per-processor task sets are emptied (3rd line).
CHAPTER 4. PERIODIC TASK SCHEDULING

Assumption:
i is the index of the tasks.
x is the index of the processors.
n is the number of the tasks assigned to \( P_x \).
cs1 is the execution time of the first portion of a split task.
cs2 is the execution time of the second portion of a split task.
ts is the period of a split task.
tm is the minimal period of the tasks assigned to \( P_x \).
\( U^\ast(P_x) \) is the utilization bound of \( P_x \).
\( \Gamma \) is sorted so that \( T_1 \leq T_2 \leq ... \leq T_n \).

1. \( i = x = n = 1 \);
2. \( cs1 = cs2 = ts = tm = 0 \);
3. \( \forall \Pi_x = \emptyset \);
4. \( U^\ast_i = \text{rmdp\_bound}(n, T_i, cs1, cs2, ts, tm) \);
5. if \( U(\Pi_x) + U_i \leq U^\ast_i \)
6. \( \Pi_x = \Pi_x \cup \tau_i \);
7. else if \( x < M \)
8. \( C'_i = \{U^\ast_i - U(\Pi_x)\}T_i \);
9. \( C''_i = C_i - C'_i \);
10. split \( \tau_i \) into \( \tau'_i(C'_i, T_i) \) and \( \tau''_i(C''_i, T_i) \);
11. \( \Pi_x = \Pi_x \cup \tau'_i \);
12. \( \Pi_{x+1} = \{\tau''_i\} \);
13. \( x = x + 1 \);
14. \( n = 0 \);
15. if \( i + 1 \leq N \)
16. \( cs1 = C'_i \);
17. \( cs2 = C''_i \);
18. \( ts = T_i \);
19. \( tm = T_{i+1} \);
20. else
21. return FAILURE ;
22. \( i = i + 1 \);
23. \( n = n + 1 \);
24. if \( i \leq N \)
25. go back to step 4 ;
26. return SUCCESS ;

Figure 4.3: RMDP assigning algorithm
CHAPTER 4. PERIODIC TASK SCHEDULING

<table>
<thead>
<tr>
<th>Arguments: ((n, T_i, C'_s, C''<em>s, T_s, T</em>{\text{min}}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. if (C''_s = 0)</td>
</tr>
<tr>
<td>2. return (n(2^{1/n} - 1)) ;</td>
</tr>
<tr>
<td>3. else</td>
</tr>
<tr>
<td>4. (L = [(T_i - T_s + C'_s)/T_s] ;)</td>
</tr>
<tr>
<td>5. (U''_s = C''_s/T_s ;)</td>
</tr>
<tr>
<td>6. (R_s = T_{\text{min}}/T_s ;)</td>
</tr>
<tr>
<td>7. return (U''_s + n[(2 - LU''_s/R_s)^{1/n} - 1] ;)</td>
</tr>
</tbody>
</table>

Figure 4.4: \(\text{rmdp\_bound}\) function

When the initializations are done, the algorithm calls the \(\text{rmdp\_bound}\) function, which is indicated in Figure 4.4, to calculate the utilization bound of processor \(P_x\) to which a task \(\tau_i\) will be assigned (4th line). The content of the \(\text{rmdp\_bound}\) function, i.e. the method of calculating the utilization bound, is particularly described in Section 4.2.3. Once the utilization bound \(U'_s\) is calculated, the algorithm assigns \(\tau_i\) to a processor \(P_x\), i.e. inserts \(\tau_i\) to \(\Pi_x\), as long as the total utilization of \(P_x\), denoted by \(U(\Pi_x) + U_i\), does not exceed the utilization bound (5th and 6th lines). If \(\tau_i\) causes \(P_x\) to exceed the utilization bound and more than one empty processor is still left, \(\tau_i\) is going to be split into two portions according to the following procedure. At first, the execution times of the first portion and the second portion of \(\tau_i\), denoted by \(C'_i\) and \(C''_i\) respectively, are calculated based on the spare schedulable utilization of \(P_x\), denoted by \(U'_s - U(\Pi_x)\) (8th and 9th lines). Then, \(\tau_i\) is split into \(\tau'_i = (C'_i, T_i)\) and \(\tau''_i = (C''_i, T_i)\) (10th line). \(\tau'_i\) is assigned to \(P_x\) (11th line) and \(\tau''_i\) is assigned to \(P_{x+1}\) (12th line). Since the assignment will move on the next processor, the index for the processors is incremented (13th line), and the number of the tasks assigned to the next processor is reset by zero (14th line). Note that \(n\) must be reset by zero, not one, though \(\tau''_i\) is already assigned to the processor, because \(n\) does not include the split task according to the schedulability analysis in Section 4.2.3. The task splitting procedure finally saves the values of \(C'_i, C''_i, T_i\) and \(T_{i+1}\) into the variables \(cs1, cs2, ts,\) and \(tm\) respectively (16th to 19th lines) for carrying out the \(\text{rmdp\_bound}\) function in the next iteration. Note that if no empty processors are left when the algorithm tries to split \(\tau_i\), then the algorithm is failed (21st line). If \(\tau_i\) is successfully assigned to \(P_x\) (or split into \(P_x\) and \(P_{x+1}\)), the index for the tasks and the number of the assigned tasks are incremented (22nd and 23rd lines). Finally, the algorithm goes back to the step 4 as long as there are still tasks remaining to assign to the processors (24th and 25th lines). When all the tasks are successfully assigned, the algorithm is succeeded (26th line).

Figure 4.5 depicts an example of task assignments by the RMDP assigning algorithm. Consider a task set \(\Gamma\) composed of eight tasks: \(T_i \leq T_{i+1}: \tau_1(1, 5), \tau_2(2, 5), \tau_3(1, 8), \tau_4(5, 10), \tau_5(3, 12), \tau_6(2, 12), \tau_7(12, 20),\) and \(\tau_8(4, 20)\). Note that \(\Gamma\) is already sorted so that \(T_i \leq T_{i+1}\) is satisfied for any \(i\) where ties of periods are broken arbi-
As stated above, the algorithm sequentially assigns the tasks. Evaluating the \textit{rmdp\_bound} function with \(n = 1\), the utilization bound of 100\% is obtained, and hence \(\tau_1\) can be assigned to \(P_1\). Next, the algorithm is supposed to evaluate the \textit{rmdp\_bound} function with \(n = 2\). Here, the example applies the notion of harmonic chains for more practical use. According to Kuo and Mok [90], the number of the tasks is reduced to the number of the harmonic chains in RM scheduling. A harmonic chain is a group of the tasks whose periods have the same least common multiple. Taking the harmonic chains into account, the 23rd line of Figure 4.3 is extended so that it is carried out only if the period of the next task is not included in any harmonic chains. Thus, the example evaluates the \textit{rmdp\_bound} function with \(n = 1\) due to \(T_1 = T_2\), and \(\tau_3\) can be of course assigned to \(P_1\). \(\tau_3\) does not have a period with the same least common multiple as \(\tau_1\) and \(\tau_2\), then \(n = 2\) is input to the \textit{rmdp\_bound} function, while \(\tau_4\) has a period with the same least common multiple as them, and hence \(n = 2\) is remained to the \textit{rmdp\_bound} function. The resulting utilization bound of about 83\% is obtained by the \textit{rmdp\_bound} function for \(n = 2\). Since \(U_1 + U_2 + U_3 + U_4 = 1.225\) exceeds the utilization bound of 0.83, \(\tau_4\) is split into \(\tau'_4(1, 10)\) and \(\tau''_4(4, 10)\) so that \(U_1 + U_2 + U_3 + U'_4 = 0.825 \leq 0.83\) and \(U''_4 = U_4 - U'_4\). By the same token, the rest of the tasks are classified into \(\Pi_2 = \{\tau''_4(4, 10), \tau_5, \tau_6(1, 12)\}\) and \(\Pi_3 = \{\tau''_6(1, 12), \tau_7, \tau_8\}\), though the utilization bounds are calculated with taking into account that split tasks are assigned in those sets. Recall that the utilization bound analysis is explained in detail in Section 4.2.3.

Figure 4.5: RMDP assigning example
CHAPTER 4. PERIODIC TASK SCHEDULING

The resulting assignment guarantees that the second portion of a split task has the shortest period on each processor. This characteristic helps to derive the schedulable condition for the algorithm. The details are described in Section 4.2.3. The resulting assignment also has such a characteristic that there are at most $M - 1$ tasks that will migrate between two processors. The degree of task migrations is hence restricted dramatically compared to global scheduling, while those restricted migrations enable the algorithm to improve the schedulability sufficiently compared to partitioned scheduling, as the dissertation explains in the following sections.

4.2.2 Task Scheduling Phase

This section explains the task scheduling algorithm of RMDP. Like partitioned scheduling, each processor has its own scheduler to schedule the assigned tasks on the processor. All the scheduler have the same scheduling policies. However, portioned scheduling differs from partitioned scheduling in that the schedulers are not completely independent from each other, since some two processors may share the same task that is split into those two processors. According to the RMDP assigning algorithm, each processor $P_x$ may have at most two split tasks. One is the second portion of a task that is split into $P_{x-1}$ and $P_x$. The other is the first portion of another task that is split into $P_x$ and $P_{x+1}$. Letting $\tau_i$ be such a task that its second portion $\tau''_i$ is assigned to $P_x$ and $\tau_k$ be such a task that its first portion $\tau'_k$ is assigned to $P_x$. The following descriptions focus on the scheduling on a processor $P_x$ where $\tau_i$ submits its second portion $\tau''_i$ and $\tau_k$ submits its first portion $\tau'_k$. The schedulers on other processors perform in the same manner, since the same scheduling policies are applied.

The basic policy of the scheduling algorithm is that the tasks are scheduled according to the RM algorithm except for the case in which $\tau''_i$ is ready but its corresponding first portion $\tau'_i$ is on execution on a neighbor processor $P_{x-1}$. In this case, the RMDP scheduler dispatches a task with the second highest priority so as not to execute the first portion and the second portion of $\tau_i$ simultaneously. In other words, the second portion is deferrable to the RM scheduling policy. Notice that the RMDP scheduling completely follows the RM scheduling if the processor does not have the second portion of a split task.

Figure 4.6 shows the pseudo code of the RMDP scheduler. Every time any tasks are released on $P_x$, the schedule_$P_x$ function is invoked (1st to 9th lines). Let $t$ be such a time. At time $t$, if the split tasks $\tau_i$ and $\tau_k$ that have their first portion or second portion on $P_x$ are released, the scheduler needs to reset the variables $t_i$, $t_k$, $e_i$ and $e_k$, in order to track the remaining execution times of those tasks. Note that $\tau_i$ and $\tau_k$ are not allowed to consume the processor time of $P_x$ over the capacities $C''_i$ and $C'_k$ respectively, but the tasks cannot recognize voluntarily that they exhaust the capacities. Thus, the scheduler needs to track the remaining execution times of those tasks to preempt their executions using timer functions.

The scheduler function proceeds as follows. First of all, if the currently-running
CHAPTER 4. PERIODIC TASK SCHEDULING

Assumption:
τ_i is a split task between P_{x-1} and P_x.
τ_k is a split task between P_x and P_{x+1}.
t is the current time.
t_i is the last time at which τ_i is dispatched on P_x.
t_k is the last time at which τ_k is dispatched on P_x.
e_i is the remaining execution time of τ_i.
e_k is the remaining execution time of τ_k.

1. function system_tick
2. if any tasks are released on P_x
3. if τ_i is released
4. e_i = C_i''
5. t_i = t
6. if τ_k is released
7. e_k = C_k'
8. t_k = t
9. call schedule_P_x
10. function schedule_P_x
11. if P_x is idling
12. go to step 17
13. if τ_i is currently running
14. e_i = e_i - (t - t_i)
15. else if τ_k is currently running
16. e_k = e_k - (t - t_k)
17. let τ_j be a task with the shortest period
18. if τ_j refers to τ_i
19. if τ_j is in execution on P_{x-1}
20. let τ_j be a task with the second shortest period
21. else
22. update the timer to invoke p2_finished at t + e_i
23. t_i = t
24. else if τ_j refers to τ_k
25. update the timer to invoke p1_finished at t + e_k
26. t_k = t
27. if τ_k is in execution on P_{x+1}
28. invoke schedule_P_{x+1} on P_{x+1}
29. dispatch and execute τ_j on P_x

Figure 4.6: RMDP scheduling algorithm
CHAPTER 4. PERIODIC TASK SCHEDULING

1. function job_end
   2. remove the caller task from a ready set ;
   3. call schedule_{P_x} ;

4. function first_end
   5. remove \( \tau_r \) from a ready set ;
   6. call schedule_{P_x} ;
   7. if \( \tau_k \) is ready on \( P_{x+1} \)
      8. invoke schedule_{P_{x+1}} on \( P_{x+1} \) ;

9. function second_end
10. remove \( \tau_i \) from a ready set ;
11. call schedule_{P_x} ;

Figure 4.7: Job finishing functions

task is either a split portion of \( \tau_i \) or \( \tau_j \), the scheduler saves its remaining execution time (13th to 15th lines). Then, it selects a task with the shortest period according to the RM scheduling policy (17th line). If the selected highest-priority task is \( \tau_i \) that has its second portion on the processor (18th line), then the scheduler firstly examines whether \( \tau_i \) is currently in execution on a neighbor processor \( P_{x+1} \) (19th line). If \( \tau_i \) is in execution on \( P_{x+1} \), then the scheduler re-selects a task with the second shortest period (20th line).

Otherwise, \( \tau_i \) is going to be dispatched and executed, so the scheduler updates the timer to invoke the second_end function at time \( t + e_i \), which interrupts and preempts the execution of \( \tau_i \) so that it does not overrun the capacity \( C''_i \) on \( P_x \) (22nd line). Notice that the second_end function may not be invoked at time \( t + e_i \), since it may be updated if \( \tau_i \) is preempted and later dispatched again within the same period. Time \( t + e_i \) is just the earliest time at which \( \tau_i \) may possibly exhaust \( C''_i \). It is guaranteed that \( \tau_i \) cannot continue to be executed after time \( t + e_i \) if it is never preempted until then. As long as the remaining execution time of \( \tau_i \) is tracked, \( \tau_i \) is guaranteed not to overrun the capacity \( C''_i \) on \( P_x \). After updating the timer, the scheduler must save the current time as the last dispatched time of \( \tau_i \), in order to track its remaining execution time at the 14th line. By the same token, if the selected task is \( \tau_k \) that has its first portion on the processor (24th line), then the scheduler updates another timer to invoke the first_end function at time \( t + e_k \), which interrupts and preempts the execution of \( \tau_k \) so that it does not overrun the capacity \( C'_k \) on \( P_x \), and saves the current time as the last dispatched time of \( \tau_k \) (25th and 26th lines). Unlike the case of \( \tau_i \), if the corresponding second portion of \( \tau_k \) is currently in execution on \( P_{x+1} \) (27th line), the scheduler needs to invoke the scheduler function operating on \( P_{x+1} \) to reschedule the tasks on \( P_{x+1} \), since \( \tau_k \) that has the highest priority on \( P_{x+1} \) must be deferred while its corresponding first portion is executed on \( P_x \) (28th line). Because of this part, the schedulers are not independent from each other. Finally, the scheduler dispatches and executes the selected task on \( P_x \) (29th line).
The job finishing functions are indicated in Figure 4.7. When any jobs except for the jobs of the split tasks are completed, the jobs call the job_end function (1st to 3rd lines). The first_end and second_end functions are called only through the timers. Those finishing functions call the scheduler function to reschedule the tasks (3rd, 6th, and 11th lines). Only when \( \tau_k \) consumes the capacity \( C'_k \) on \( P_x \) but has not consumed the capacity \( C''_k \) on \( P_{x+1} \), the scheduler invokes the scheduler operating on \( P_{x+1} \) to dispatch and execute \( \tau_k \) on \( P_{x+1} \), since \( \tau_k \) has the shortest period on \( P_{x+1} \) but has been deferred.

Figure 4.8 indicates how the three task sets of \( \Pi_1 = \{ \tau_1(1, 5), \tau_2(2, 5), \tau_3(1, 8), \tau'_4(1, 10) \} \), \( \Pi_2 = \{ \tau''_4(4, 10), \tau_5(3, 12), \tau''_6(1, 12) \} \) and \( \Pi_3 = \{ \tau''_7(1, 12), \tau_7(12, 20), \tau_8(4, 20) \} \), are scheduled according to the RMDP scheduling policy. Since \( \Pi_1 \) does not include the second portion of a split task, the task set is scheduled completely according to the RM scheduling policy. Meanwhile, \( \Pi_2 \) include \( \tau''_4 \) that is the second portion of \( \tau_4 \), so it is interfered by its corresponding first portion \( \tau'_4 \) when it is in execution on \( P_1 \). At time
$t = 0$, $\tau'_4$ is scheduled on $P_2$ and then completed at time $t = 4$ without any interference from $\tau'_4$, because $\tau'_4$ is scheduled at time $t = 4$ on $P_1$. As for the second job of $\tau_4$, on the other hand, $\tau''_4$ is scheduled at time $t = 10$ on $P_2$ but is preempted at time $t = 13$, because $\tau'_4$ is scheduled at this time on $P_1$. Hence, $\tau_5$ is scheduled instead of $\tau''_4$ and $\tau'_4$ is later resumed when $\tau'_4$ is completed at time $t = 14$. The third job of $\tau_4$ also has the same situation. In this case, there are no ready tasks on $P_1$ when $\tau''_4$ is preempted at time $t = 23$. Therefore, the time slot is left idle. Although $\Pi_3$ also includes the second portion $\tau''_6$, the tasks on $P_3$ can be scheduled without any restrictions within the example, since the scheduling times of $\tau'_6$ and $\tau''_6$ are never overlapped.

Here, the dissertation provides an idea of reducing the number of task preemptions caused by unnecessary task migrations in the RMDP algorithm, though a specific implementation of this idea is left for the future work. Figure 4.9 shows the idea of reducing preemptions. In the example of Figure 4.8, two split tasks of $\tau_4$ and $\tau_6$ migrate to another processor unnecessarily. The idea is that if $\tau_5$ is migrated to $P_1$ from $P_2$ instead of $\tau'_4$ at time $t = 13$ and if $\tau_4$ stays on $P_2$ at time $t = 23$, unnecessary preemptions can be eliminated as shown in Figure 4.9. Note that this implementation of this idea is out of concern in this research.

The costs for checking the state of a task on a different processor and invoking the scheduler operating on a different processor are not necessarily expensive, since recent memory-shared multiprocessors, especially multicore processors, have shared registers and hardware synchronization mechanisms which reduce the overhead of interprocessor communications dramatically. In addition, making only two fine-grained timers for preventing the split tasks from overrunning their reserved capacities are not so expensive from the implementation point of view. If the practicality of the algorithm is criticized because of those issues, the traditional sophisticated algorithms must be revised. For instance, the Pfair algorithms obviously need a fine-grained timer to realize quantum-based scheduling. The LLREF algorithm may preempt tasks many times and EKG algorithm may preempt tasks twice within each interval of job arrivals, so they also require a fine-grained timer. The EDZL algorithm has less preemption timings but needs to find the event of the zero laxity that can happen at any moment, hence it need a fine-grained timer after all. In consequence, this paper believes that the proposed algorithm is as practical as the traditional efficient scheduling algorithms.

### 4.2.3 Schedulability Analysis

This section provides the schedulability analysis of the RMDP algorithm. In order to guarantee the system to satisfy the required real-time constraints, the schedulable condition must be evinced. Besides, system designers desire that the utilization bound for the scheduling algorithm are clarified so that they can make a dependable system without considering a complex array of factors, and can explain the worst-case performance of the system to users. Now therefore, the dissertation firstly presents the schedulable condition of the RMDP algorithm, and secondly derives the utilization bound for the
worst case. Throughout the analysis, the dissertation has the following assumptions. In the task assigning phase, some task \( \tau_s \) is split and its second portion \( \tau_s'' \) is assigned to processor \( P_s \). Then the following \( n \) tasks, denoted by \( \tau_1, \tau_2, ..., \tau_n \), for simplicity of description, are also assigned to \( P_s \). Note that the first portion of a split task is not a concern, because the RMDP scheduler deals with it in the same manner as the non-split tasks. Only the second portion of a split task is scheduled in a different manner, since it may be different. The analysis considers \( \tau_s'' \) to be an individual task, though it is actually the second portion of \( \tau_s \), since \( \tau_s'' \) behaves as an individual task in theoretical scheduling point of view.

The goal of the analysis is to derive a formula leading to the schedulable condition of processor \( P_s \). Once the schedulable condition of \( P_s \) is obtained, the schedulable conditions of other processors can be also obtained using the same formula, because all the schedulers on those processors have the same scheduling policy in the RMDP algorithm. The approach of the analysis is that the processor utilization is never below \( U_s'' \), because the second portion of \( \tau_s \) is always assigned first, and hence the mission is reduced to obtain the minimum of the remaining schedulable utilization for \( n \) tasks.

According to the RM schedulability analysis [1], the minimal schedulable utilization for \( n \) tasks with respect to RM scheduling occurs for the case in which all the tasks are released at the same time and the following relations exist: \( T_1 < T_n < 2T_1 \), \( C_i = T_{i+1} - T_i \) (1 \( \leq i \leq n - 1 \)) and \( C_n = 2T_1 - T_n \). This fact derives that the minimum of the schedulable utilization for \( n \) tasks in the RMDP algorithm also occurs for this condition. Now therefore, the worst-case phasing of \( \tau_s'' \) is a concern. For a fixed value of \( U_s'' \), the remaining utilization of \( P_s \) is obviously minimized when \( \tau_s'' \) submits as many jobs as possible. It occurs for the case in which an arbitrary job of \( \tau_s'' \) is deferred as much as possible and the following jobs of \( \tau_s'' \) are started without any preemption as soon as they are released. Consequently there are three conceivable worst-case phases as shown in Figure 4.10, Figure 4.11 and Figure 4.12. The difference between the three phases is that \( \tau_s'' \) is executed (i) twice within \( T_1 \) and \( T_n \) in the phase 1, (ii) three times within \( T_1 \) and \( T_n \) in the phase 2, and (iii) twice with \( T_1 \) and three times within \( T_n \) in the phase 3. The analysis does not need to consider the case in which \( \tau_s'' \) is executed more than three times within \( T_1 \) or \( T_n \), since the condition of the worst-case phasing includes \( T_1 < T_n < 2T_1 \).

For simplicity of description, the fraction of \( T_s \) and \( T_1 \) is defined \( R_s = T_1/T_s \) and that of the periods of two consecutive tasks \( \tau_i \) and \( \tau_{i+1} \) is defined \( R_i = T_{i+1}/T_i \) henceforth. Notice that \( \forall k, R_1R_2 \cdots R_{k-1} = T_k/T_1 \). Also it is necessary to have the minimum slack \( S \) in the figures to obtain the schedulable utilization. Since \( \tau_s'' \) always has the highest priority on the processor according to the RMDP algorithm, \( \tau_s'' \) is interfered only by its corresponding first portion \( \tau_s' \) running on the neighbor processor. Therefore, the latest finish time of \( \tau_s'' \) is \( r_{s,k} + C_s' + C_s'' = r_{s,k} + C_s = r_{s,k+1} - (T_s - C_s) \). As a result, the minimal slack can be expressed by \( S = T_s - C_s \). If \( T_s = C_s \), the worst-case phases are completely equal to those of the Deferrable Server (DS) algorithm [15] so-called the back-to-back hits phenomenon.
CHAPTER 4. PERIODIC TASK SCHEDULING

Case of Phase 1

The analysis begins with the simplest case in which \( \tau'_s \) is executed twice within \( T_1 \) and \( T_n \) as shown in Figure 4.10. Under this relation, the execution time of each task is defined as follows.

\[
C_i = T_{i+1} - T_i \quad (1 \leq i \leq n-1)
\]

\[
C_n = T_1 - 2C''_s - \sum_{j=1}^{n-1} C_j = 2T_1 - 2C''_s - T_n
\]

Hence the resulting utilization is written as Equation (4.1).

\[
U = \frac{C''_s}{T_s} + \frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_n}{T_n}
\]

\[
= U'' + \sum_{i=1}^{n-1} \frac{T_{i+1} - T_i}{T_i} + \frac{2T_1 - 2C''_s - T_n}{T_n}
\]

\[
= U'' + \sum_{i=1}^{n-1} \frac{T_{i+1}}{T_i} + 2 \left( 1 - \frac{C''_s}{T_1} \right) \frac{T_1}{T_n} - n
\]

\[
= U'' + \sum_{i=1}^{n-1} R_i + \frac{2 \left( 1 - \frac{U''}{R_s} \right)}{R_1R_2 \cdots R_{n-1}} - n
\]  

(4.1)
In order to minimize $U$ over $R_i$, the above expression is partially differentiated with respect to $R_i$.

$$\frac{\partial U}{\partial R_i} = 1 - \frac{2 \left(1 - \frac{U''}{R_s}\right)}{R_i^2 \left(\prod_{j \neq i} R_j\right)}$$

Now, $U$ is minimized when each $R_i$ satisfies the following equation where $P = R_i R_2 \cdots R_{n-1}$.

$$R_i P = 2 \left(1 - \frac{U''}{R_s}\right) (1 \leq i \leq n - 1)$$

That is, $U$ is minimized when all the $R_i$ have the same value.

$$R_1 = R_2 = \cdots = R_{n-1} = \left\{2 \left(1 - \frac{U''}{R_s}\right)\right\}^{1/n}$$

By substituting the value of each $R_i$ to Equation (4.1), the utilization bound $U_b$ is obtained as follows. Here, let $K = 2(1 - U''/R_s)$ due to limitation of space.

$$U_b = U'' + (n - 1)K^{1/n} + K^{K^{1/n} - K^{1/n} - n}$$

The above expression implies that $U_b$ is moreover minimized by reducing the value of $R_s$. According to Figure 4.10, the condition below must be satisfied.

$$\sum_{i=1}^{n} C_i = T_1 - 2C_i'' \geq S = T_s - C_s$$

Dividing by $T_s$, the range of $R_s$ is acquired as follows.

$$R_s - 2U'' \geq 1 - U_s \quad R_s \geq 2U'' - U_s + 1$$

The condition of $T_s \leq T_1$ leads to $R_s \geq 1$. Hence, $U_b$ is minimized when $R_s = \max\{1, 2U'' - U_s + 1\}$. Finally, $U_b$ is described by Equation (4.2) where $R_s = \max\{1, 2U'' - U_s + 1\}$.

$$U_b = U'' + n \left\{2 \left(1 - \frac{U''}{R_s}\right)\right\}^{1/n} - 1$$

(4.2)

Taking the limit as $n \to \infty$, the worst case is derived.

$$\lim_{n \to \infty} U_b = U'' + \ln \left\{2 \left(1 - \frac{U''}{R_s}\right)\right\}$$

(4.3)
CHAPTER 4. PERIODIC TASK SCHEDULING

Figure 4.11: Case in which $\tau''_s$ is executed three times within $T_1$ and $T_n$

With $R_s = 1$, Equation (4.3) is minimized to Equation (4.4).

$$\hat{U}_b = U''_s + \ln[2(1 - U''_s)]$$

Consequently, the absolute minimum value of the utilization bound becomes $\hat{U}_b = 0.5$ when $U''_s = 0.5$ and $U_s = 1$. Note that this bound is given only for the special situation in which $T_s = T_1 = T_2 = \cdots = T_n$ and $C_1 = C_2 = \cdots = C_n = 0$.

Case of Phase 2

The analysis secondly focuses on the case in which $\tau''_s$ is executed three times within $T_1$ and $T_n$ as shown in Figure 4.11. Under this relation, the execution time of each task is defined as follows.

$$C_i = T_{i+1} - T_i \quad (1 \leq i \leq n - 1)$$
$$C_n = T_1 - 3C''_s - \sum_{j=1}^{n-1} C_j = 2T_1 - 3C''_s - T_n$$

Hence, the resulting utilization is written as Equation (4.4).

$$U = U''_s + \sum_{i=1}^{n-1} \frac{T_{i+1}}{T_i} + \left(2 - \frac{3C''_s}{T_1}\right)\frac{T_1}{T_n} - n$$

$$= U''_s + \sum_{i=1}^{n-1} R_i + \frac{2 - \frac{3U''_s}{R_s}}{R_1 R_2 \cdots R_{n-1}} - n$$

(4.4)
Calculating the value of \( R_i \) that minimizes Equation (4.4) by the same step in Section 4.2.3, the utilization bound is obtained by the following expression.

\[
U_b = U''_s + n \left\{ \left( 2 - \frac{3U''_s}{R_s} \right)^{1/n} - 1 \right\}
\]

The above expression is minimized by reducing the value of \( R_s \). Figure 4.11 derives \( T_1 \) as follows.

\[
T_1 = T_s + 2C''_s + S = 2T_s + 2C''_s - C_s
\]

Dividing by \( T_s \), the value of \( R_s \) is acquired as follows.

\[
R_s = 2U''_s - U_s + 2
\]

Hence, \( U_b \) is now expressed by Equation (4.5).

\[
U_b = U''_s + n \left\{ \left( 2 - \frac{3U''_s}{2U''_s - U_s + 2} \right)^{1/n} - 1 \right\} = U''_s + n \left\{ \frac{U''_s - 2U_s + 4}{2U''_s - U_s + 2} \right\}^{1/n} - 1 \right\} \tag{4.5}
\]

Taking the limit as \( n \to \infty \), the worst case is appeared.

\[
\lim_{n \to \infty} U_b = U''_s + \ln \left( \frac{U''_s - 2U_s + 4}{2U''_s - U_s + 2} \right) \tag{4.6}
\]

In order to find the value of \( U''_s \) minimizing \( U_b \), Equation (4.6) is derived with respect to \( U''_s \):

\[
\frac{\partial U_b}{\partial U''_s} = \frac{2U''^2 + (10 - 5U_s)U''_s + 2U_s^2 - 5U_s + 2}{(U''_s - 2U_s + 4)(2U''_s - U_s + 2)}
\]

Finally, the value of \( U''_s \) that minimizes \( U_b \) is obtained by Equation (4.7).

\[
\hat{U''}_s = \frac{5U_s - 10 + \sqrt{9U_s^2 - 60U_s + 84}}{4}
\]

Since \( U''_s \geq 0 \), the absolute minimum value of the bound becomes \( \hat{U}_b \approx 0.652 \) when \( U''_s \approx 0.186 \) and \( U_s = 1 \).
CHAPTER 4. PERIODIC TASK SCHEDULING

Figure 4.12: Case in which $\tau''_s$ is executed twice within $T_1$ and three times within $T_n$

**Case of Phase 3**

The final case the analysis considers is a combination of the previous two cases. Likewise the previous two cases, the execution time of each task is defined as follows.

\[
\begin{align*}
C_i &= T_{i+1} - T_i \quad (1 \leq i \leq k - 1) \\
C_k &= T_{k+1} - C''_s - T_k \\
C_i &= T_{i+1} - T_i \quad (k + 1 \leq i \leq n - 1) \\
C_n &= T_1 - 2C''_s - \sum_{j=1}^{n-1} C_j = 2T_1 - C''_s - T_n
\end{align*}
\]
Then, the resulting utilization is written as Equation (4.7).

\[
U = U'' + \sum_{i=1}^{k-1} \frac{T_{i+1} - T_i}{T_i} + \frac{C_k}{T_k} + \sum_{i=k+1}^{n-1} \frac{T_{i+1} - T_i}{T_i} + \frac{2T_1 - C'' - T_n}{T_n} \\
= U'' + \sum_{i=1}^{k-1} \frac{T_{i+1} - T_i}{T_i} + U_k + \sum_{i=k+1}^{n-1} \frac{T_{i+1} - T_i}{T_i} + \left(2 - \frac{C''}{T_k}\right) \frac{T_1}{T_n} - (n - 1) \\
= U'' + \sum_{i=1}^{k-1} R_i + U_k + \sum_{i=k+1}^{n-1} R_i + \frac{2 - \frac{U''}{R_i}}{R_1 R_2 R_3 \cdots R_{n-1}} - (n - 1) \tag{4.7}
\]

It is too complicated to obtain the value of \( R_i \) that minimizes Equation (4.7) with respect to two variables of \( n \) and \( k \). Leading \( U_k = R_k - C'' / T_k - 1 \) from \( C_k = T_{k+1} - C'' - T_k \), Equation (4.7) can be transformed as follows.

\[
U = U'' + \sum_{i=1}^{n-1} R_i - \frac{C''}{T_k} + \frac{2 - \frac{U''}{R_k}}{R_1 R_2 R_3 \cdots R_{n-1}} - n
\]

The above expression implies that Equation (4.7) is obviously minimized for \( k = 1 \) due to \( T_1 \leq T_2 \leq \cdots \leq T_n \). Therefore, Equation (4.7) is reduced as follows.

\[
U = U'' + U_1 + \sum_{i=2}^{n-1} R_i + \frac{2R_s - U''}{R_s R_1 R_2 R_3 \cdots R_{n-1}} - (n - 1)
\]

In order to minimize \( U \), the above expression is derived with respect to \( R_i \) as follows where \( 2 \leq i \leq n - 1 \).

\[
\frac{\partial U}{\partial R_i} = 1 - \frac{2R_s - U''}{R_s R_1 R_i^2 \prod_{j \neq i}^{n-1} R_j}
\]

That is, \( U \) is minimized when all the \( R_i \) have the same value.

\[
R_2 = R_3 = \cdots = R_{n-1} = \left( \frac{2R_s - U''}{R_s R_1} \right)^{1/(n-1)}
\]

Now, the minimum value of \( U \) is described as follows.

\[
U = U'' + U_1 + (n - 2) \left( \frac{2R_s - U''}{R_s R_1} \right)^{1/(n-1)} + \frac{2R_s - U''}{R_s R_1} \left( \frac{2R_s - U''}{R_s R_1} \right)^{1/(n-1)} - (n - 1)
\]

\[
= U'' + U_1 + (n - 1) \left\{ \left( \frac{2R_s - U''}{R_s R_1} \right)^{1/(n-1)} - 1 \right\}
\]
\( C_k = T_{k+1} - C''_s - T_k \) and \( k = 1 \) lead to \( R_1 = U_1 + U''_s / R_s + 1 \). Also, the value of \( R_s \) that minimizes the above expression is \( \max \{1, 2U''_s - U_s + 1\} \) as in Section 4.2.3. Finally, the utilization bound \( U_b \) is described by Equation (4.8) where \( R_s = \max \{1, 2U''_s - U_s + 1\} \) and \( m = n - 1 \).

\[
U_b = U''_s + U_1 + m \left\{ \frac{2R_s - U''_s}{R_s(U_1 + 1) + U''_s} \right\}^{1/m} - 1 \quad (4.8)
\]

Taking the limit as \( n \to \infty \), the worst case is appeared.

\[
\lim_{n \to \infty} U_b = U''_s + U_1 + \ln \left( \frac{2R_s - U''_s}{R_s(U_1 + 1) + U''_s} \right) \quad (4.9)
\]

By the same token as the first case, the minimum of Equation (4.9) is expressed by Equation (4.10).

\[
\hat{U}_b = U''_s + U_1 + \ln \left( \frac{2 - U''_s}{U''_s + 1 + U_1} \right)
\]

Seeking the values of \( U''_s \) and \( U_1 \) that minimizes \( \hat{U}_b \), the absolute minimum bound is obtained \( \hat{U}_b \approx 0.5 \) when \( U''_s = 1/2 \) and \( U_1 = 0 \). This bound occurs only for the situation in which \( T_s = T_1 = T_2 = \cdots = T_k, T_{k+1} = T_{k+2} = \cdots = T_n = T_s + C''_s \) and \( C_1 = C_2 = \cdots = C_n = 0 \). Comparing the analyzed three cases, the worst-case utilization bound of each processor is finally derived 50%. That is, the utilization bound of the entire system is also 50%.

**General Case**

Remember that the tasks are sorted so that \( T_i \leq T_{i+1} \). Hence, \( T_1 \) is a known value when \( \tau_s \) is split, which means that \( R_s \) is also a known value. Therefore, the utilization bound can be calculated by either Equation (4.2), Equation (4.5), or (4.8).

The analysis now proceeds to consider the general case. Equation (4.4) can be rewritten as follows.

\[
U = U''_s + \sum_{i=1}^{n-1} \frac{T_{i+1} - 1}{T_i} + \left( 2 - \frac{3C''_s}{T_1} \right) \frac{T_1}{T_n} - n
\]

\[
= U''_s + \sum_{i=1}^{n-1} \frac{T_{i+1}}{T_i} + \left( 2 - \frac{C''_s}{T_1} \right) \frac{T_1}{T_n} - n - \frac{2C''_s}{T_n}
\]

\[
(4.10)
\]

52
Equation (4.7) can be also rewritten as follows.

\[
U = U''_s + \sum_{i=1}^{k-1} \frac{T_{i+1}}{T_i} + U_k + \sum_{i=k+1}^{n-1} \frac{T_{i+1}}{T_i} + \left(2 - \frac{C''_s}{T_1} \right) \frac{T_1}{T_n} - (n - 1)
\]

\[
= U''_s + \sum_{i=1}^{k-1} \frac{T_{i+1}}{T_i} + T_{k+1} - \frac{T_k - C''_s}{T_k} + \sum_{i=k+1}^{n-1} \frac{T_{i+1}}{T_i} + \left(2 - \frac{C''_s}{T_1} \right) \frac{T_1}{T_n} - (n - 1)
\]

\[
= U''_s + \sum_{i=1}^{n-1} \frac{T_{i+1}}{T_i} + \left(2 - \frac{C''_s}{T_1} \right) \frac{T_1}{T_n} - n - \frac{C''_s}{T_k}
\]

\[
= U''_s + \sum_{i=1}^{n-1} \frac{T_{i+1}}{T_i} + \left(2 - \frac{C''_s}{T_1} \right) \frac{T_1}{T_n} - n - \frac{C''_s}{T_k}
\]

(4.11)

Notice that \(T_n/T_k\) in Equation (4.11) never exceeds 2, because \(\tau''_s\) must be executed twice within \(T_k\) and three times within \(T_n\). Hence, Equation (4.4) is always smaller than Equation (4.7) with respect to the same set of \(\{T_s, T_1, T_2, \ldots, T_n\}\). This fact implies that the utilization bound for the case in which \(\tau''_s\) is executed \(L\) times within \(T_1\) and \(T_n\) is always smaller than that for the case in which \(\tau''_s\) is executed \(F < L\) times within \(T_1\) and is executed \(L\) times within \(T_n\). Therefore, the analysis needs to concern only the case in which \(\tau''_s\) is executed \(L\) times within \(T_n\) for the general case. Referring to Figure 4.11, \(L\) can be described by the following expression.

\[
L = 1 + \left[ \frac{T_n - C''_s - S}{T_s} \right] = 1 + \left[ \frac{T_n - T_s + C'}{T_s} \right]
\]

Finally, the utilization utilization for the general case is obtained by Equation (4.12).

\[
U_b = U''_s + n \left\{ 2 - \frac{LU''_s}{R_s} \right\}^{1/n} - 1
\]

(4.12)

The value of Equation (4.12) is 50% when \(L = 2, U_s = 1\) and \(U''_s = 0.5\). Hence, the worst case is contained. Letting \(U''_s = 0\), the analysis also contains the case in which there is no task that has the second portion on \(P_s\). In fact, this case leads to Equation (4.13) which is the well-known utilization bound of the RM algorithm [1].

\[
U_b = n (2^{1/n} - 1)
\]

(4.13)

Harmonic Case

This section introduces the special characteristic of the RMDP algorithm for harmonic task sets. Kuo and Mok showed that considering the number of harmonic chains improves the utilization bound of the RM algorithm [90]. Since the RMDP algorithm is
designed based on the RM algorithm, the value of \( n \) in Equation (4.12) can be obviously deemed as the number of the harmonic chains. However, the fact that \( n \) does not include the second portion of a split task must be concerned. In other words, applying \( n = 1 \) to Equation (4.12) does not refer to the utilization bound for harmonic task sets but indicates the utilization bound for the case in which a split task that submits its second portion and one harmonic chain exist. The objective of this section is to analyze the schedulability of the RMDP algorithm for the harmonic case.

As already mentioned, if there is no task that submits its second portion on \( P_x \), the scheduling of the RMDP algorithm completely follows the scheduling of the RM algorithm, hence the utilization bound obviously becomes 100% for the harmonic case. Now therefore, the existence of the second portion of a split task is taken into account. The two cases are considered here: (i) the case in which all the \( n \) tasks are released at the same time, and (ii) the case in which all the \( n \) tasks released with different phases.

The followings prove that the utilization bound for the first case is 100% and that for the second case is not and is calculated by Equation (4.12) with \( n = 1 \). The harmonic case analysis is helpful for practical use, because the tasks are often released at the same time when the system begins in real environments.

The analysis for the first case is as follows. Assume that \( \tau_i'' \) is suspended at time \( t \) when \( \tau_i' \in \Pi_{i-1} \) is scheduled, and resumed at time \( s > t \) when \( \tau'_i \) is completed. In this case, while \( \tau''_i \) is behind schedule for \( s - t \) temporarily, the other tasks in \( \Pi_i \) are never behind schedule. The reason is as follows. The task set is harmonic and the period of \( \tau''_i \) is the shortest in \( \Pi_i \), so \( \forall \tau_j \in \Pi_i, T_j = KT_i \) is certainly satisfied (\( K \) is an integer constant greater than 0). Then at the time at which any job of \( \tau''_i \) is released, all the rest of the tasks in \( \Pi_i \), i.e. \( \{ \tau_j \in \Pi_i \} \), are also released. Hence the time interval of \( s - t \) can be allocated to \( \{ \tau_j \in \Pi_i \} \). That is, \( \{ \tau_j \in \Pi_i \} \) are never behind schedule, though they can be ahead schedule. Therefore \( \Pi_i \) is schedulable even if the processor utilization is 100% unless \( \tau''_i \) misses the deadline. Since \( \tau''_i \) is behind schedule for at most \( C'_i \), it is guaranteed to meet the deadline if \( C'_i + C''_i \leq T_i \) is satisfied. Because \( C'_i + C''_i = C_i \leq T_i \) is true, the condition is always satisfied. Hence the schedulable utilization of the RMDP algorithm is certainly 100% for the harmonic case.

Unlike the first case, the utilization bound for the second case does not become 100%. This reason is easily validated by the following example. Consider a set of three tasks assigned to a processor \( P_x \): \( \tau''_i(4, 10), \tau_j(8, 20), \) and \( \tau_k(8, 40) \). This task set is harmonic with the total utilization 100%. Suppose that there exists a time at which \( \tau''_i \) arrives and none of the other tasks arrive. Let \( t = 30 \) be such a time. If \( \tau'_i \) is scheduled at this time, there are obviously no tasks in execution on \( P_x \), while there must be no idle times in order to achieve an optimal utilization bound of 100%. Therefore, the optimality is never achieved for the second case, which means that Equation (4.12) with \( n = 1 \) must be used to work out the utilization bound of the RMDP algorithm.
CHAPTER 4. PERIODIC TASK SCHEDULING

4.3 The EDDHP algorithm

This section presents the EDDHP algorithm that extends the notion of the RMDP algorithm to dynamic-priority scheduling. The EDDHP shares in the manner of the RMDP algorithm that the second portion of a split task has the highest priority on a processor. The extension is that the rest of the tasks on the processor is scheduled according to the EDF algorithm. By the same token as Section 4.2, the task assigning algorithm and the task scheduling algorithm are firstly described. Then, the schedulable condition and the utilization bound of the EDDHP algorithm are analyzed. In addition, this section introduces several heuristics to improve the schedulable utilization of the EDDHP algorithm in practice.

4.3.1 Task Assigning Phase

The task assigning algorithm of EDDHP is similar to that of RMDP. The algorithm differs from the RMDP algorithm in the manner of calculation of the utilization bound for each processor. While the RMDP algorithm carries out the calculation every time a new task is assigned to a processor, the EDDHP algorithm does only when the task assignment moves on a new processor. Since the schedulable condition of the EDDHP algorithm does not depend on the number of the assigned tasks, as explained in Section 4.3.3 later, the algorithm can calculate the utilization bound when the properties of a split task that submits its second portion on the target processor are revealed. The RMDP algorithm, on the other hand, requires the information of the number of the assigned tasks to calculate the utilization bound according to Equation (4.12), hence it needs to recalculate the utilization bound every time a new task is assigned to a processor, because the number of the assigned tasks is incremented.

Figure 4.13 shows the pseudo code of the EDDHP task assignment. The assumption in the algorithm is same as that in the RMDP assigning algorithm (Figure 4.3), except that some variables are removed due to its simplicity of the schedulability test. Unlike the RMDP assigning algorithm, the EDDHP assigning algorithm needs not to track the number of the assigned tasks, since the utilization bound is not dominated by the number of the assigned tasks. This simplicity yields another simplicity that the algorithm also needs not to save the properties of a split task every time the task is split, since the utilization bound can be calculated when the task is split.

First of all, the algorithm initializes the variables (1st and 2nd lines). The utilization bound of the first processor is always set 1.0 (3rd line), because the first processor produces the EDF scheduling due to the absence of the second portion of a split task. Like the RMDP assigning algorithm, the EDDHP assigning algorithm assigns a task \( \tau_i \), to a processor \( P_x \), i.e. inserts \( \tau_i \) to \( \Pi_x \), as long as the total utilization of \( P_x \) does not exceed the utilization bound (4th and 5th lines). If the total utilization exceeds the utilization bound and empty processors are still left, \( \tau_i \) is going to be split and assigned to \( P_x \) and \( P_{x+1} \) as follows. Similarly to the RMDP assigning algorithm, the execution
$i$ is the index of the tasks.
$x$ is the index of the processors.
$U^*_x$ is the utilization bound of $P_x$.
$\Gamma$ is sorted so that $T_1 \leq T_2 \leq ... \leq T_n$.

1. $i = x = 1$ ;
2. $\forall \Pi_x = \emptyset$ ;
3. $U^*_x = 1.0$ ;
4. if $U(\Pi_x) + U_i \leq U^*_x$
5. $\Pi_x = \Pi_x \cup \tau_i$ ;
6. else if $x < M$
7. $C'_i = \{U^*_x - U(\Pi_x)\}T_i$ ;
8. $C''_i = C_i - C'_i$ ;
9. split $\tau_i$ into $\tau'_i(C'_i, T_i)$ and $\tau''_i(C''_i, T_i)$ ;
10. $\Pi_x = \Pi_x \cup \tau'_i$ ;
11. $\Pi_{x+1} = \{\tau''_i\}$ ;
12. if $i + 1 \leq N$
13. $U^*_{x+1} = eddhp\_bound(C'_i, C''_i, T_i, T_{i+1})$ ;
14. $x = x + 1$ ;
15. else
16. return FAILURE ;
17. $i = i + 1$ ;
18. if $i \leq N$
19. go back to step 4 ;
20. return SUCCESS ;

Figure 4.13: EDDHP assigning algorithm

times of the first portion and the second portion of $\tau_i$ are calculated based on the spare schedulable utilization of $P_x$ (7th and 8th lines). Then, $\tau_i$ is split into $\tau'_i = (C'_i, T_i)$ and $\tau''_i = (C''_i, T_i)$ (9th line), which are assigned to $P_x$ and $P_{x+1}$ respectively (10th and 11th lines). Unlike the RMDP assigning algorithm, the utilization bound of the next processor is calculated at this point if more than one task is remaining (12th and 13th lines). The procedure of the $eddhp\_bound$ function is illustrated in Figure 4.14, which is in detail presented in Section 4.3.3. Since the function requires only $C'_s$, $C''_s$, $T_s$, and $T_{i+1}$ for its arguments, the calculation of the utilization bound can be done here. The index for the processors is incremented for the task assignment on the next processor (14th line). If $\tau_i$ is successfully assigned to $P_x$ (or split into $P_x$ and $P_{x+1}$), the index for the tasks is incremented (15th line), and then the algorithm goes back to the step 4, as long as it still has the tasks to assign to the processors (18th and 19th lines). When all the tasks are successfully assigned, the algorithm is succeeded (20th line).

The characteristics of the resulting assignment in the EDDHP assigning algorithm are also same as that in the RMDP assigning algorithm. Since the tasks are sorted
in increasing order of periods, the algorithm easily finds the task with the minimum period, which dominates the utilization bound of each processor as presented in Section 4.3.3 later, when the \textit{eddhp\_bound} function is called. If the algorithm did not have such a characteristic, it would take more overhead to find the task with the minimum period to calculate the utilization bound.

### 4.3.2 Task Scheduling Phase

The task scheduling algorithm of EDDHP is also incremental, since it differs from the RMDP scheduling algorithm only in that the tasks but the second portion of a split task are scheduled according to the EDF scheduling policy. The pseudo code of the algorithm is shown in Figure 4.15. The procedures of tracking the remaining execution times of the split tasks perform in the same manner as the RMDP scheduling algorithm. In EDDHP scheduling, the second portion of a split task is statically assigned the highest priority on the processor, so \( \tau_i \) is dispatched whenever it is ready and is not in execution on a neighbor processor \( P_{x-1} \). Then, the timer needs to be set to invoke the \textit{second\_end} function, depicted in Figure 4.7, to preempt \( \tau_i \) when it exhausts the capacity \( C_i'' \). If \( \tau_i \) is in execution on \( P_{x-1} \), it is deferred in spite of the highest priority. Except for \( \tau_i \), the tasks are scheduled according to the EDF scheduling policy. When \( \tau_k \), which submits its first portion on the processor, has the earliest deadline, the scheduler needs to update the timer to invoke the \textit{first\_end} function, depicted in Figure 4.7, to preempt \( \tau_k \) when it exhausts the capacity \( C_k' \).

### 4.3.3 Schedulability Analysis

This section derives the schedulable condition and the utilization bound of the EDDHP algorithm. As in the schedulability analysis of the RMDP algorithm in Section 4.2.3, the analysis assumes that a task \( \tau_s \) is split and its second portion \( \tau_s'' \) is assigned to a processor \( P_x \). Note that the first portion of a split task is not a concern, because the
CHAPTER 4. PERIODIC TASK SCHEDULING

Assumption:
\( \tau_i \) is a split task between \( P_{x-1} \) and \( P_x \).
\( \tau_k \) is a split task between \( P_x \) and \( P_{x+1} \).
\( t \) is the current time.
\( t_i \) is the last time at which \( \tau_i \) is dispatched on \( P_x \).
\( t_k \) is the last time at which \( \tau_k \) is dispatched on \( P_x \).
e_i \) is the remaining execution time of \( \tau_i \).
e_k \) is the remaining execution time of \( \tau_k \).

1. function system_tick
2. if any tasks are released on \( P_x \)
3. if \( \tau_i \) is released
4. \( e_i = C_i'' \);
5. \( t_i = t \);
6. if \( \tau_k \) is released
7. \( e_k = C_k' \);
8. \( t_k = t \);
9. call schedule_\( P_x \);

10. function schedule_\( P_x \)
11. if \( P_x \) is idling
12. go to step 17;
13. if \( \tau_i \) is currently running
14. \( e_i = e_i - (t - t_i) \);
15. else if \( \tau_k \) is currently running
16. \( e_k = e_k - (t - t_k) \);
17. if \( \tau_j \) is ready and is not in execution on \( P_{x-1} \)
18. let \( \tau_j \) refer to \( \tau_i \);
19. update the timer to invoke second_end at \( t + e_i \);
20. \( t_i = t \);
21. else
22. let \( \tau_j \) be a task with the earliest deadline;
23. if \( \tau_j \) refers to \( \tau_k \)
24. update the timer to invoke first_end at \( t + e_k \);
25. \( t_k = t \);
26. if \( \tau_k \) is in execution on \( P_{x+1} \)
27. invoke schedule_\( P_{x+1} \) on \( P_{x+1} \);
28. dispatch and execute \( \tau_j \) on \( P_x \);

Figure 4.15: EDDHP scheduling algorithm
CHAPTER 4. PERIODIC TASK SCHEDULING

execution of the tasks that have a higher priority than \( \tau_s' \) on \( P_{m-1} \)
deferred execution of \( \tau_s'' \) on \( P_m \) due to execution of \( \tau_s' \) on \( P_{m-1} \)

Figure 4.16: Unfeasible EDDHP scheduling

EDDHP scheduler dispatches it in the same manner as the non-split tasks. Only the second portion of a split task is scheduled in a different manner. In the analysis, \( \tau_s'' \) is deemed to be one task that can be deferred. Now therefore, the analysis focuses on the schedulable utilization of \( P_x \).

Figure 4.16 shows an example of unfeasible EDDHP scheduling. Note that the second portion of \( \tau_s \), denoted by \( \tau_s'' \), is always assigned the highest priority on \( P_x \) unless its first portion, denoted by \( \tau_s' \), is in execution on \( P_{x-1} \). Now assume that any job \( \tau_{jc} \) missed its deadline as shown in the figure. Let \( t_1 \) be the last time at which the processor is idle or a job whose deadline is later than the deadline of \( \tau_{jc} \) is executed. Let \( t_2 \) be the time at which \( \tau_{jc} \) missed its deadline, that is, the deadline of \( \tau_{jc} \). In order to have \( \tau_{jc} \) miss its deadline, the following condition needs to be satisfied where \( S(t_1, t_2) \) is the total amount of the time at which \( \tau_s'' \) is not executed within \([t_1, t_2]\), namely the total slack amount with respect to \( \tau_s'' \) within \([t_1, t_2]\).

\[
S(t_1, t_2) < \sum_{\tau_i \in \Lambda_m \setminus \tau_s''} \left\lfloor \frac{t_2 - t_1}{T_i} \right\rfloor C_i
\]

Being as \( \lfloor x \rfloor \leq x \) for any \( x \), the following condition must be satisfied in order that \( \tau_{jc} \)
CHAPTER 4. PERIODIC TASK SCHEDULING

may miss its deadline.

\[ S(t_1, t_2) < \sum_{\tau_i \in \Lambda \setminus \tau''_s} (t_2 - t_1) U_i \]

In other words, any job \( \tau_{j,c} \) is guaranteed to meet its deadline if the following condition is satisfied for any \((t_1, t_2)\).

\[ \sum_{\tau_i \in \Lambda \setminus \tau''_s} U_i \leq \frac{S(t_1, t_2)}{t_2 - t_1} \]

Now the analysis seeks to obtain the minimum value of \( R(t_1, t_2) = S(t_1, t_2)/(t_2 - t_1) \).

Note that we have \( t_2 - t_1 \geq \min\{T_i \mid \tau_i \in \Lambda \setminus \tau''_s\} \), since \( t_1 \) should be before or at the release time of \( \tau_{j,c} \). It is obvious that \( \tau''_s \) consumes the most processor time within \([t_1, t_2]\) when its first job within \([t_1, t_2]\) is deferred for the longest time and its last job within \([t_1, t_2]\) is executed immediately with no preemptions. This phasing results in minimizing \( R(t_1, t_2) \). Taking this worst-case phasing into account, we need to take into account two cases to obtain the minimum value of \( R(t_1, t_2) \). The first one is the case of \( T_{min} \geq FT_s + C''_s - C'_s \) and the second one is the case of \( T_{min} \leq FT_s + C''_s - C'_s \).

Hereinafter the analysis defines \( T_{min} \) and \( F \) as follows for simplicity of description.

\[ T_{min} = \min\{T_i \mid \tau_i \in \Lambda \setminus \tau''_s\} \]

\[ F = \frac{T_{min} + C'_s}{T_s} \]

The two cases are shown in Figure 4.17. The goal of the analysis is to obtain the minimum value of \( R(t_1, t_2) \) for each case.

The dissertation firstly proves that the minimum value of \( R(t_1, t_2) \) in the case of \( T_{min} \geq FT_s + C''_s - C'_s \) is described by Equation (4.14) where \( G \) represents \( G = F + 1 \) for limitation of space.

\[ \min[R(t_1, t_2)] = \min \left\{ 1 - \frac{G C''_s}{T_{min}}, \frac{G(T_s - C''_s) - C'_s}{GT_s + C''_s - C'_s} \right\} \]  

(4.14)

The proof is as follows. At first, the analysis assumes \( t_1 = a \) and \( t_2 = b \) in Figure 4.17(a). Then, \( R(t_1, t_2) \) is described as follows.

\[ R(t_1, t_2) = R(a, b) = \frac{S(a, b)}{b-a} = \frac{T_{min} - G C''_s}{T_{min}} = 1 - \frac{G C''_s}{T_{min}} \]

If \( t_2 \) is \( b < t_2 < c \), \( R(t_1, t_2) \) can be written as follows where \( 0 < \alpha \leq c - b \).

\[ R(t_1, t_2) = R(a, b + \alpha) = \frac{S(a, b) + \alpha}{b-a+\alpha} \]

Since \( \frac{\alpha}{y} < \frac{x}{y} \) is always true for any \( x > 0 \), \( y > 0 \) and \( z > 0 \), we have \( R(a, b) < R(a, b + \alpha) \). Next, the analysis assumes \( t_1 = a \) and \( t_2 = d \). Then, \( R(t_1, t_2) \) is described as follows.

\[ R(t_1, t_2) = R(a, d) = \frac{S(a, d)}{d-a} = \frac{G(T_s - C''_s) - C'_s}{GT_s + C''_s - C'_s} \]
If \( t_2 \) is \( b \leq t_2 < d \), \( R(t_1, t_2) \) can be written as follows where \( 0 < \beta \leq d - c \).

\[
R(t_1, t_2) = R(a, d - \beta) = \frac{S(a, d)}{d - a - \beta}
\]

Since \( \frac{z}{y} < \frac{x}{y} \) is always true for any \( x > 0, y > 0 \) and \( z > 0 \), we have \( R(a, d) < R(a, d - \beta) \).

Also if \( t_2 \) is \( d < t_2 \leq e \), \( R(t_1, t_2) \) can be written as \( R(a, d + \gamma) \) where \( 0 < \gamma \leq e - d \). Then, we have \( R(a, d) < R(a, d + \gamma) \) by the same reason of \( R(a, b) < R(a, b + \alpha) \). At last if \( t_2 \) is \( e < t_2 \leq f \), \( R(t_1, t_2) \) is minimized when \( t_2 = f \), since we have \( R(a, f) < R(a, f - \delta) \) by the same reason of \( R(a, d) < R(a, d - \gamma) \). For any \( C'_s > 0 \) and \( C''_s > 0 \), the following condition is obviously satisfied.

\[
R(a, d) = \frac{S(a, d)}{d - a} < \frac{T_s - C''_s}{T_s}
\]

Meanwhile \( R(a, f) \) can be written as follows.

\[
R(a, f) = \frac{S(a, d) + (e - d)}{(d - a) + (f - d)} = \frac{S(a, d) + (T_s - C''_s)}{(d - a) + T_s}
\]

Hence we have \( R(a, d) < R(a, f) \), which implies \( R(a, d) < R(a, g) \) for any \( d < g \). So the minimum value of \( R(t_1, t_2) \) is either of \( R(a, b) \) or \( R(a, d) \). Assuming \( R(a, b) < R(a, d) \),
then the following condition must be satisfied.

\[
\frac{T_{\text{min}} - GC''_s}{T_{\text{min}}} < \frac{G(T_s - C''_s) - C'_s}{GT_s + C''_s - C'_s}
\]

\[
(T_{\text{min}} - GC''_s)(GT_s + C''_s - C'_s) < T_{\text{min}}(GT_s - GC''_s - C'_s)
\]

\[
T_{\text{min}}(G + 1)C''_s < GC''_s(GT_s + C''_s - C'_s)
\]

\[
T_{\text{min}} < \frac{G(GT_s + C''_s - C'_s)}{G + 1} = \frac{G}{G + 1}(d - a)
\]

Here the above inequation is not always true. It is easy to prove if the analysis considers the case of $C'_s \approx 0$ and $C''_s \approx 0$. In this case, we can approximate $d - a \approx d - c$. Since 1 $\frac{G}{G+1}$ $\frac{G}{G+1}$ is always true, we have $\frac{G}{G+1}(d - a) \approx \frac{b}{b+c}(d - a) < (d - a) \approx c$. Being as $C'_s \approx 0$ and $C''_s \approx 0$, $T_{\text{min}}$ can take any length within $[b, c]$. Hence it depends on the length of $T_{\text{min}}$ whether the above inequation is true or false. Consequently the minimum value of $R(t_1, t_2)$ is described by Equation (4.14).

Next, the dissertation proves that the minimum value of $R(t_1, t_2)$ in the case of $T_{\text{min}} \leq FT_s + C''_s - C'_s$ is described by Equation (4.15).

\[
\min[R(t_1, t_2)] = \frac{F(T_s - C''_s) - C'_s}{FT_s + C''_s - C'_s}
\]

(4.15)

The analysis of the proof takes the same step as the one proved Equation (4.15). The analysis firstly assumes $t_1 = a$, then finds $t_2$ that minimizes $R(t_1, t_2)$. According to the first discussion as for Equation (4.14), we can easily obtain $R(a, b) > R(a, c)$. We can also easily obtain $R(a, c) < R(a, e) < R(a, d)$. Therefore $R(t_1, t_2)$ is minimized when $t_1 = a$ and $t_2 = c$.

\[
R(t_1, t_2) = R(a, c) = \frac{S(a, c)}{c-a} = \frac{F(T_s - C''_s) - C'_s}{FT_s + C''_s - C'_s}
\]

Hence the minimum value of $R(t_1, t_2)$ is described by Equation (4.15).

From the above lemmas, the dissertation now provides the theorem that the utilization bound of processor $P_s$ for the EDDHP algorithm is described by Equation (4.16) if $T_{\text{min}} \geq FT_s + C''_s - C'_s$, otherwise by Equation (4.17).

\[
U_b = \frac{C''_s}{T_s} + \min \left\{ \frac{T_{\text{min}} - GC''_s}{T_{\text{min}}} \cdot \frac{G(T_s - C''_s) - C'_s}{GT_s + C''_s - C'_s} \right\}
\]

(4.16)

\[
U_b = \frac{C''_s}{T_s} + \frac{F(T_s - C''_s) - C'_s}{FT_s + C''_s - C'_s}
\]

(4.17)

The $eddhp$ function in Figure 4.13 corresponds to either Equation (4.16) or Equation (4.17), which is a function of $C'_s$, $C''_s$, $T_s$ and $T_{\text{min}}$. Since the $eddhp$ function is called when task $\tau_s$ is split, $C'_s$, $C''_s$ and $T_s$ are already known. Also the tasks
are sorted so that $T_i \leq T_{i+1}$. Thereby the algorithm can presume $T_{min}$ as $T_{s+1}$. If the
tasks are not sorted in increasing order of period, the algorithm moreover needs to find
which task will have the shortest period in the tasks assigned to $P_x$. This procedure
is not easy since the number of the assigned tasks cannot be known priori before the
utilization bound is computed. That is why the algorithm assigns the tasks in increasing
order of period.

This section finally proves that the worst-case utilization bound for the EDDHP
algorithm is 50%. The proof is as follows. For the case of Equation (4.14), it is obvious
that $R(a, b)$ is monotonically increasing in $T_{min}$ and $F$. Since we have the condition of
$T_{min} \geq FT_s + C''_s - C'_s$ in this case, $R(a, b)$ is minimized when $T_{min} = FT_s + C''_s - C'_s$ and
$F = 1$. Meanwhile $R(a, d)$ can be rewritten as follows.

$$R(a, d) = \frac{G(T_s - C''_s) - C'_s}{GT_s + C''_s - C'_s}$$
$$= \frac{GT_s + C''_s - GC''_s - C'_s}{GT_s + C''_s - C'_s}$$
$$= 1 - \frac{(G + 1)C''_s}{GT_s + C'_s - C'_s}$$
$$= 1 - \frac{2C''_s + FC''_s}{(C''_s - C'_s + T_s) + FT_s}$$

Since $C''_s \leq T_s$ is always true, $R(a, d)$ is monotonically increasing in $F$ and it is mini-
mized when $F = 1$. Taking $T_{min} = FT_s + C''_s - C'_s \geq T_s$ into account, we can derive the
following relations.

$$R(a, b) \geq \frac{T_{min} - 2C''_s}{T_{min}} = 1 - \frac{2C''_s}{T_{min}} \geq 1 - \frac{2C''_s}{T_s}$$
$$R(a, d) \geq 1 - \frac{3C''_s}{2T_s + C''_s - C'_s} = 1 - \frac{2C''_s + C''_s}{T_s + (T_s + C''_s - C'_s)}$$
CHAPTER 4. PERIODIC TASK SCHEDULING

Assumption:
the algorithm is applied only when \( i + 1 \leq N \).

let \( s \) be the index of a task that is going to be split.

let \( \tau_k'' \) be the second portion of a split task that was assigned to \( P_x \) if there exists.

1. \( s = i \);
2. \( U^*_s + 1 = 0 \);
3. for each \( \tau_j \in \Pi_s \setminus \tau_k'' \)
4. \( U_{rem} = U^*_s - \{U(\Pi_s) + U_s - U_j\} \);
5. if \( U_{rem} \geq 0 \)
6. \( C_j = U_{rem}T_j \) and \( C_j'' = C_j - C_j' \);
7. \( b = \text{eddhp_bound}(C_j', C_j'', T_j, T_{i+1}) \);
8. if \( U_{s+1}^* < b \)
9. \( U_{s+1}^* = b \);
10. \( s = j \);
11. \( \Pi_s = \Pi_s \cup \tau_s \setminus \tau_s \);
12. \( C_s' = \{U_s^* - U(\Pi_s)\}T_s \) and \( C_s'' = C_s - C_s' \);
13. split \( \tau_s \) into \( \tau_s' (C_s', T_s) \) and \( \tau_s'' (C_s'', T_s) \);
14. \( \Pi_s = \Pi_s \cup \tau_s' \) and \( \Pi_{s+1} = \{\tau_s''\} \);

Figure 4.19: MB heuristic

\( C_s'' \leq T_s + C_s'' - C_s' \) is always true from \( C_s' \leq T_s \). Also \( \frac{x + y}{y + w} > 0 \) is always true for any \( w > z > 0 \), so \( R(a, b) < R(a, d) \) is derived in the case of minimization. Namely the minimization of \( U_b \) occurs for Figure 4.18. That is, the minimum value of \( U_b \) can be described as following \( U^*_b \).

\[
U^*_b = \frac{C_s''}{T_s} + 1 - \frac{2C_s''}{T_s} = 1 - \frac{C_s''}{T_s}
\]

Because of \( C_s' = C_s'' \) and \( C_s' + C_s'' = C_s \leq T_s \), \( C_s'' \) takes the range of \( 0 \leq C_s'' \leq T_s/2 \). Hence the absolute minimum value of \( U^*_b \) is 50%. As for the case of Equation (4.15), it is monotonically increasing in \( F \) by the same reason as the above \( R(a, d) \), so it is minimized when \( F = 1 \). This is completely the same phasing as the one in Figure 4.18. Namely the minimum value of \( U^*_b \) is also 50% in this case. Since the worst-case utilization bound of the schedulable per-processor utilization is 50%, that of the schedulable whole system utilization is also 50%.

4.3.4 Improving Schedulable Utilization

According to the previous section, the worst-case utilization bound for the EDDHP algorithm is only 50%. However, as the dissertation mentioned, this bound is obtained only in the special case. The values of Equation (4.16) and Equation (4.17) are often
CHAPTER 4. PERIODIC TASK SCHEDULING

Assumption:
the algorithm is applied only when \( i + 1 \leq N \).
let \( s \) be the index of a task that is going to be split.
let \( \tau'' \) be the second portion of a split task that was assigned to \( P_x \) if there exists.

1. if \( \{ U^* - U(\Pi_s) \} + U^*_{s+1} > 1.0 \)
2. \( C'_s = \{ U^* - U(\Pi_s) \} T_s \) and \( C''_s = C_s - C'_s; \)
3. split \( \tau_s \) into \( \tau'_s \) and \( \tau''_s \);
4. \( \Pi_x = \Pi_x \cup \tau'_s \) and \( \Pi_{x+1} = \{ \tau''_s \}; \)
5. else
6. \( \Pi_{x+1} = \{ \tau_s \}; \)
7. \( U^*_{x+1} = 1.0; \)

Figure 4.20: AIB heuristic

higher than 50%. Also it can be moreover improved if the task assigning phase efficiently chooses a task to split, because the utilization bound is a function of the period and the execution time of the split task. The necessity condition for calculating the utilization bound of processor \( P_x \) is described by the following expression.

\[
T_s \leq \min\{ T_i \mid \tau_i \in \Pi_s \setminus \tau''_s \}
\]

Since the EDDHP assigning algorithm always holds the following condition, the above inequation is always satisfied.

\[
\max\{ T_j \mid \tau_j \in \Pi_{s-1} \} \leq \min\{ T_i \mid \tau_i \in \Pi_s \}
\]

Hence, the task assigning phase can choose any task residing on \( P_{s-1} \) to split into \( P_{s-1} \) and \( P_s \).

Based on this foundation, the dissertation presents a heuristic technique called \textit{Maximizing Bound} (MB) to maximize the utilization bound of each processor. The MB heuristic can be combined with the EDDHP assigning algorithm by replacing the procedure in Figure 4.19 with the 7th to the 11th line in Figure 4.13. This procedure seeks a task that can maximize the utilization bound of the next processor if the task is split instead of \( \tau_i \) in Figure 4.13. \( s \) indicates the index of a task that will be split, and is initialized with \( i \) (1st line). The utilization bound of the next processor is initialized with zero (2nd line). Then, the procedure tests each task \( \tau_j \) whether it can be split (4th and 5th lines). If \( U^*_s - \{ U(\Pi_s + U_s - U_j) \} \), which is the remaining schedulable utilization of \( P_s \) under the assumption that the entire portion of \( \tau_s \) is assigned to \( P_s \) instead of \( \tau_j \), is below zero, there is no room for any portion of \( \tau_j \) to be assigned to \( P_x \). For only the tasks that passed this test, the procedure finds the task that maximizes the utilization bound of the next processor (6th to 10th lines). After finding the task maximizing the utilization bound, the procedure removes \( \tau_s \) from \( \Pi_x \) and, appends \( \tau_j \) to \( \Pi_x \) instead (11th
line). This replacement is valid only when \(i \neq s\). Finally, the procedure splits \(\tau_s\) (12th to 14th lines).

There is another approach to improve the schedulable utilization for the EDDHP algorithm. As the dissertation already discussed, the utilization bound for the EDDHP algorithm could be 50% in the worst case, whereas that for the EDF algorithm is always 100%. Therefore, the approach of splitting a task may degrade the schedulable utilization for the algorithm compared to the case in which the task is not split. The dissertation gives an example of this degradation. Consider the case that four tasks, \(\tau_1(2, 5), \tau_2(2, 5), \tau_3(6, 10),\) and \(\tau_4(4, 11)\), are assigned to two processors, \(P_1\) and \(P_2\). Being as \(U_1 + U_2 = 0.8 < 1.0\) and \(U_1 + U_2 + U_3 = 1.4 > 1.0\), the EDDHP assigning algorithm splits \(\tau_3\) into \(P_1\) and \(P_2\). Then, we have \(C'_x = 2, C''_x = 4, T_s = 10,\) and \(T_{\text{min}} = 11\). Because these values meets the condition of \(T_{\text{min}} > FT_s + C''_x - C'_x\), the utilization bound can be calculated using Equation (4.16) and is \(U_b \approx 0.733\). Here, the EDDHP assigning algorithm cannot assign all the tasks successfully due to \(U''_3 + U_4 \approx 0.763 > U_b\). Meanwhile, the algorithm can successfully assign the tasks if \(\tau_3\) is not split but is assigned to \(P_2\) instead due to \(U_3 + U_4 \approx 0.963 < 1.0\). This example indicates that the approach of splitting a task can decrease schedulability.

Taking this situation into account, the dissertation presents another heuristic technique called Absolutely-Increasing Bound (AIB). The AIB heuristic can be moreover combined with the MB heuristic by replacing the procedure shown in Figure 4.20 with the line 12~14 in Figure 4.19. Note that the AIB heuristic can be also combined with the EDDHP assigning algorithm alone by letting \(s = i\) in Figure 4.20. The procedure is straightforward. If task \(\tau_s\) is not split but is entirely assigned to \(P_{x+1}\), the remaining schedulable utilization of \(P_x\), denoted by \(U^*_x - U(\Pi_x)\), is wasted, instead the utilization bound of \(P_{x+1}\), denoted by \(U^*_x+1\), is boosted up to 1.0. Therefore, letting \(U^*_x+1\) be the utilization bound of \(P_{x+1}\) for the case in which \(\tau_s\) is split, the AIB heuristic splits a task only if \(U^*_x - U(\Pi_x) + U^*_x+1 > 1.0\) (1st to 4th lines), otherwise it assigns \(\tau_s\) to the next processor (5th to 7th lines).

### 4.4 The EDDP algorithm

This section proposes the EDDP algorithm, which is the major contribution of this research. The algorithm is incremental to the RMDP and EDDHP algorithm presented in the previous sections. However, the EDDP algorithm has different concepts in both task assigning and task scheduling, which results in the worst-case utilization bound of 65% that is greater than most of the traditional algorithms. At first, the task assigning algorithm and the task scheduling algorithm are presented. Then, the schedulable condition, including the worst-case utilization bound, is derived.
4.4.1 Task Assigning Phase

This section describes the task assigning algorithm of EDDP. First of all, let $U^*$ denote the utilization bound for the EDDP algorithm, whose specific value is demonstrated in Section 4.4.3. Then, $\tau_i$ is defined to be a heavy task in the EDDP algorithm if it holds a condition expressed by Inequation (4.18). Otherwise it is defined to be a light task.

$$U_i > U^* \quad (4.18)$$

Figure 4.21 shows the pseudo code of the EDDP assigning algorithm. The algorithm assumes that the given task set includes $h$ heavy tasks and $N - h$ light tasks, where the heavy tasks are indexed $1 \sim h$ and the light tasks are indexed $h + 1 \sim N$. The set of the light tasks is sorted so that the period of $\tau_i$ is smaller than or equal to that of $\tau_{i+1}$. Basically, the algorithm assigns the tasks to the processors sequentially, which means that it always assigns $\tau_{i+1}$ after $\tau_i$. If a task causes the total utilization of a processor to exceed its utilization bound, it splits the task into two portions. Then it continues to assign the rest of the tasks from the next processor. A set of the tasks assigned to a processor $P_x$ is denoted by $\Pi_x$.

The procedure of the algorithm is as follows. If the number of the heavy tasks is smaller than the number of the processors, the heavy tasks are first assigned to dedicated processors (2nd to 5th lines). Note that each processor $P_x$ ($1 \leq x \leq h$) has only one heavy task. Then, the light tasks are assigned to the rest of the processors in the same manner as the EDDHP assigning algorithm shown in Figure 4.13 that sequentially assigns the tasks in the order of increasing period. The first processor containing light tasks does not have the second portion of a split task, thereby the utilization bound is 100% according to the EDF scheduling policy (7th line). Then, each light task $\tau_i$ is assigned to a processor $P_x$ as long as the total utilization of $P_x$ is less than or equal to its utilization bound, denoted by $U_x^*$ (8th and 9th lines). If the total utilization of $P_x$ exceeds the utilization bound, $\tau_i$ is going to be split into two portions and assigned to $P_x$ and $P_{x+1}$ in the same manner as the RMDP and EDDHP assigning algorithms (12th to 16th lines). Finally, the utilization bound of the next processor is calculated using the formula that is specifically presented in Section 4.4.3 (19th line). Notice that the utilization bound formula does not depend of the relation between the split task $\tau_i$ and the task with the minimum period $\tau_{i+1}$, meanwhile the RMDP and EDDHP algorithms have different formulas depending on the relation as shown in Figure 4.4 and Figure 4.14. Therefore, it is more straightforward. If all the tasks are successfully assigned to the processors, the algorithm is succeeded.

As mentioned above, the EDDP assigning algorithm is similar to the EDDHP assigning algorithm but differs from in that the tasks are defined to be heavy or light, and the heavy tasks are assigned to dedicated processors. Such a policy is rather similar to the task assigning algorithm of EKG [81]. However, the reason of separating heavy tasks and light tasks in the EDDP algorithm is arisen by a different factor. The EDDP assigning algorithm separates the heavy tasks, because the utilization bound, derived in
CHAPTER 4. PERIODIC TASK SCHEDULING

Assumption:

$i$ is the index of the tasks.
$x$ is the index of the processors.
$h$ is the number of the heavy tasks.
$U^*_x$ is the utilization bound of $P_x$.
$\Gamma$ is sorted as follows:
(i) $\{\tau_i \mid 1 \leq i \leq h\}$ are heavy tasks.
(ii) $\{\tau_j \mid h + 1 \leq j \leq N\}$ are light tasks.
The light tasks are also sorted in increasing order of period $s$.

1. $\forall \Pi_x = \emptyset$;
2. if $h > M$
3. return FAILURE;
4. for $1 \leq i \leq h$
5. $\Pi_i = \{\tau_i\}$;
6. $x = h + 1$;
7. $U^*_x = 1.0$;
8. for $h + 1 \leq i \leq N$
9. if $U(\Pi_x) + U_i \leq U^*_x$
10. $\Pi_i = \Pi_x \cup \tau_i$;
11. else if $x < M$
12. $C'_i = \{U^*_x - U(\Pi_x)\}T_i$;
13. $C''_i = C'_i - C'_i$;
14. split $\tau_i$ into $\tau'_i(C'_i, T_i)$ and $\tau''_i(C''_i, T_i)$;
15. $\Pi_i = \Pi_x \cup \tau'_i$;
16. $\Pi_{x+1} = \{\tau''_i\}$;
17. $x = x + 1$;
18. if $i + 1 \leq N$
19. $U^*_x = 1.0 - \frac{C''(T_i + \min(C'_i, C''_i) - C''_i)}{T_i/T_{i+1}}$;
20. else
21. return FAILURE;
22. return SUCCESS;

Figure 4.21: EDDP assigning algorithm
Section 4.4.3, is dominated by the maximal utilization of every individual task. That is, the larger the maximal utilization is, the smaller the utilization bound becomes. Therefore, the EDDP assigning algorithm lets the heavy tasks have their own processors so that the heavy tasks are schedulable on their dedicated processors and the utilization bounds of the remaining processors are increased due to the absence of the heavy tasks.

4.4.2 Task Scheduling Phase

This section describes the task scheduling algorithm of EDDP. A processor containing a heavy task needs no scheduler, since there is only one heavy task assigned. The rest of the processors containing light tasks have their own schedulers that have the same scheduling policies. Assume that $\tau''_i$, $\tau'_k$, and $\{\tau_j \mid i < j < k\}$ ($i < k$) are assigned to $P_x$. More specifically, $\tau_i$ is split into $P_{x-1}$ and $P_x$, and its second portion $\tau''_i$ is assigned to $P_x$. In addition, $\tau_k$ is also split into $P_x$ and $P_{x+1}$, and its first portion is assigned to $P_x$. The EDDP scheduling policy on $P_x$ is as follows.

1. If the task with the earliest deadline is $\tau''_i$ but $\tau'_i$ is currently executed on $P_{x-1}$, then the task with the second earliest deadline is dispatched, since $\tau'_i$ and $\tau''_i$, which form a same job, cannot be executed in parallel.

2. Otherwise the task with the earliest deadline is dispatched.

In other words, the tasks are scheduled according to the EDF except that $\tau''_i$ has the earliest deadline but and its corresponding first portion $\tau'_i$ also has the earliest deadline and
is in execution on the neighbor processor $P_{x-1}$. In this exceptional case, the scheduler defers execution of $\tau''_i$ until $\tau'_i$ completes so that $\tau'_i$ and $\tau''_i$ are executed simultaneously.

Figure 4.22 depicts an example of EDDP scheduling in which $\tau_i$ is a split task migrating between $P_{x-1}$ and $P_x$. Suppose that $\tau_i$ is released at time $r_i$ with deadline $d_i$ that is the earliest on $P_x$ but is not on $P_{x-1}$, and there are two active tasks, $\tau_j$ and $\tau_k$, on $P_x$ with deadlines of $d_j$ and $d_k$ respectively, both of which are later than $d_i$. In such a case, $\tau''_i$ starts execution on $P_x$ at time $r_i$. Assume that $\tau'_i$ receives the earliest deadline on $P_{x-1}$ at time $t_1$. Then, $\tau'_i$ is dispatched on $P_{x-1}$ and $\tau''_i$ is preempted even if it has the earliest deadline on $P_x$. Instead, $\tau_j$ with the second earliest deadline is dispatched on $P_x$. Hence, $\tau''_i$ is deferred, though it has the earliest deadline. Assume that $\tau'_i$ is preempted by a task released with an earlier deadline at time $t_2$ and is later resumed at time $t_3$. Then, $\tau''_i$ can be executed during a time interval $[t_2, t_3)$. Also, assuming $\tau'_i$ completes, namely $\tau_i$ consumes $C'_i$ time units at time $t_2$, $\tau''_i$ is resumed again.

The scheduling policy described above has a potential problem with respect to deadline assignments. Consider a job of $\tau''_i$ with deadline $d$. If a set of the tasks assigned to $P_x$, whose total utilization is less than or equal to 1, is scheduled according to the EDF algorithm without taking into account the exclusion between $\tau'_i$ and $\tau''_i$, the job of $\tau''_i$ never misses the deadline of $d$, though the job may begin at time $d - C''_i$ and complete at time $d$ in the latest case as shown in Figure 4.23. However, applying the EDDP scheduling policy to schedule $\tau'_i$ and $\tau''_i$ exclusively, the completion time of the job may be delayed until time $d + \min(C'_i, C''_i)$, if the job of $\tau'_i$ is executed during $[d - C'_i, d)$.
on \(P_{x-1}\). Figure 4.24 depicts such a problematic scheduling. Due to execution of the corresponding job of \(\tau_i'\), the job of \(\tau_i''\) can be deferred at most \(C_i'\) if \(C_i' < C_i''\), otherwise it can be deferred at most \(C_i''\).

Taking the delay of \(\tau_i''\) into account, the EDDP algorithm takes the following policies for assigning job deadlines.

- For a job of \(\tau_i''\) with deadline \(d\), its deadline is virtually transformed to \(d - \min\{C_i', C_i''\}\).
- For a job of \(\{\tau_j | i < j < k\}\) and \(\tau_k'\) with deadline \(d\), its deadline is remained \(d\).

In other words, \(\tau_i''\) is deemed to have a relative deadline of \(T_i - \min\{C_i', C_i''\}\). Once a job of \(\tau_i''\) has the earliest deadline on \(P_x\), no other jobs have earlier deadlines until the job of \(\tau_i''\) completes, since the period (relative deadline) of \(\tau_i''\) is guaranteed to be the smallest in \(\Pi_x\) by the characteristic of the task assigning algorithm presented in the previous section. Therefore, letting \(d = d - \min\{C_i', C_i''\}\), if a job of \(\tau_i''\) is guaranteed to complete by time \(d\) in EDF scheduling, it is also guaranteed to complete by time \(d + \min\{C_i', C_i''\} = d\). The other jobs are never deferred, so they are guaranteed to complete by their deadlines in EDDP scheduling if they are in EDF scheduling. The detailed schedulable condition for the EDDP algorithm is presented in the next section.

Figure 4.25 shows the pseudo code of the EDDP scheduling algorithm. The algorithm is an extension of the EDDHP scheduling algorithm so that the second portion of a split task also has a dynamic-priority based on the EDF algorithm. In other words, the algorithm is an extension of the RMDP scheduling algorithm so that the prioritization policy is based on the earliest deadline but not the shortest period. Hence, the dissertation skips the explanation of the algorithm.

### 4.4.3 Schedulability Analysis

This section derives the schedulable condition for the EDDP algorithm. No deadline misses occur on a processor containing a heavy task, since 100% of processor time is allocated to every heavy task. Hence, the analysis focuses on the schedulable conditions of processors where the light tasks are assigned. Like the previous section, let \(\Pi_x = \{\tau_{i''}, \{\tau_j | i < j < k\}, \tau_k'\} (i < k)\) be a set of the light tasks assigned to a processor \(P_x\).

The goal of this section is to derive the schedulable condition and the utilization bound of \(P_x\).

Suppose that a job of some non-split task, i.e. \(\tau_s \in \Pi_x \setminus \tau_i''\), misses its deadline at time \(d\). Let \(t (d)\) be the latest time instant at which the processor is either idle or is executing a job whose deadline is after \(d\). The total amount of processor time consumed by \(\tau_i''\) in the interval of \((t, d]\) is at most equal to \(W''\) expressed by Equation (4.19).

\[
W'' = C_i'' + \left[ \frac{d + \min\{C_i', C_i''\} - t - C_i''}{T_i} \right] C_i''
\]  

(4.19)
CHAPTER 4. PERIODIC TASK SCHEDULING

Assumption:
\( \tau_i \) is a split task between \( P_{x-1} \) and \( P_x \).
\( \tau_k \) is a split task between \( P_x \) and \( P_{x+1} \).
t is the current time.
t_i is the last time at which \( \tau_i \) is dispatched on \( P_x \).
t_k is the last time at which \( \tau_k \) is dispatched on \( P_x \).
e_i is the remaining execution time of \( \tau_i \).
e_k is the remaining execution time of \( \tau_k \).

1. \textbf{function} system_tick
2. \hspace{1em} if any tasks are released on \( P_x \)
3. \hspace{2em} if \( \tau_i \) is released
4. \hspace{3em} \( e_i = C''_i \);
5. \hspace{3em} \( t_i = t \);
6. \hspace{2em} if \( \tau_k \) is released
7. \hspace{3em} \( e_k = C'_k \);
8. \hspace{3em} \( t_k = t \);
9. \hspace{1em} call schedule_{P_x} ;

10. \textbf{function} schedule_{P_x}
11. \hspace{1em} if \( P_x \) is idling
12. \hspace{2em} go to step 17 ;
13. \hspace{1em} if \( \tau_i \) is currently running on \( P_x \)
14. \hspace{2em} \( e_i = e_i - (t - t_i) \);
15. \hspace{1em} else if \( \tau_k \) is currently running on \( P_x \)
16. \hspace{2em} \( e_k = e_k - (t - t_k) \);
17. \hspace{1em} let \( \tau_j \) be a task with the earliest ;
18. \hspace{1em} if \( \tau_j \) refers to \( \tau_i \)
19. \hspace{2em} if \( \tau_i \) is in execution on \( P_{x-1} \)
20. \hspace{3em} let \( \tau_j \) be a task with the second earliest deadline:
21. \hspace{2em} else
22. \hspace{3em} update the timer to invoke \textit{second_end} at \( t + e_i \) ;
23. \hspace{3em} \( t_i = t \);
24. \hspace{2em} else if \( \tau_j \) refers to \( \tau_k \)
25. \hspace{3em} update the timer to invoke \textit{first_end} at \( t + e_k \) ;
26. \hspace{3em} \( t_k = t \);
27. \hspace{1em} if \( \tau_k \) is in execution on \( P_{x+1} \)
28. \hspace{2em} invoke schedule_{P_{x+1}} on \( P_{x+1} \) ;
29. \hspace{1em} dispatch and execute \( \tau_j \) on \( P_x \);

Figure 4.25: EDDP scheduling algorithm
Figure 4.26 shows the case in which $\tau''_i$ consumes this amount of time. At time $t$, a deferred job of $\tau''_i$ starts with the laxity of zero and finishes at $t + C'_{i'}$ without being preempted. Since $d$ must be after the pseudo deadline of the last job of $\tau''_i$ in Figure 4.26, $\tau''_i$ is executed at most 

$$\left\lfloor \frac{(d + C'_{i'}) - (t + C''_i)}{T_i} \right\rfloor$$

after $t + C''_i$. Hence the maximum total execution time of $\tau''_i$ in the interval of $(t, d]$ is calculated by Equation (4.19). Since $\lceil a \rceil \leq a$ for any $a$, $W''$ must satisfy the following condition.

$$W'' \leq C''_i + \frac{d + \min(C'_{i'}, C''_i) - t - C''_i}{T_i}$$

$$= U''_i(d - t + T_i + \min(C'_{i'}, C''_i) - C''_i)$$

Here the total amount of processor time consumed by each task $\tau_j \in \Pi \setminus \tau''_i$ in the interval of $(t, d]$ is at most $C_j(d - t)/T_j$, since the tasks except for $\tau''_i$ are scheduled according to the EDF policy. In order to cause the job of $\tau_s$ to miss its deadline at time $d$, the total amount of processor time consumed by all the jobs with deadlines before or at $d$ needs to exceed the interval of $(t, d]$. Hence the following inequality must be satisfied where $L = d - t$.

$$L < \sum_{\tau_j \in \Pi \setminus \tau''_i} \frac{C_j}{T_j}L + U''_i(L + T_i + \min(C'_{i'}, C''_i) - C''_i)$$

If the above inequality is not satisfied, the job never misses its deadline. Dividing both sides of the above inequality and considering $T_s \leq d - t$, we have the schedulable condition for $\tau_s$ as described by Equation (4.20).

$$\sum_{\tau_j \in \Pi \setminus \tau''_i} U_j + U''_i \left(1 + \frac{T_j + \min(C'_{i'}, C''_i) - C''_i}{T_s}\right) \leq 1$$

Equation (4.20) implies that all the tasks must satisfy this condition if the task with the shortest period satisfies this condition. Since the light tasks are sorted in the order of increasing period, $T_{i+1}$ is the shortest in $\{T_j \mid \tau_j \in \Pi \setminus \tau''_i\}$. Hence Equation (4.21) is
the schedulable condition for all the tasks assigned to \( P_x \).

\[
\sum_{\tau_j \in \Pi_x \setminus \tau''_i} U_j + U''_i \left( 1 + \frac{T_i + \min\{C'_i, C''_i\} - C''_i}{T_{i+1}} \right) \leq 1
\]

\[
\Leftrightarrow \sum_{\tau_j \in \Pi_x \setminus \tau''_i} U_j + U''_i \leq 1 - \frac{C''(T_i + \min\{C'_i, C''_i\} - C''_i)}{T_i T_{i+1}} \tag{4.21}
\]

Notice that the left-hand side of Equation (4.21) is equal to the total utilization of \( P_x \). Therefore the right-hand side of Equation (4.21) is the utilization bound of \( P_x \). The task assigning algorithm presented in Section 4.4.1 uses the right-hand side of Equation (4.21) as a formula of obtaining the utilization bound at the line 16 in Figure 4.21.

Now the analysis seeks to obtain the worst-case utilization bound for the EDDP algorithm by minimizing the right-hand side of Equation (4.21). Let \( \bar{U} \) denote the right-hand side of Equation (4.21). Since \( T_i \leq T_{i+1} \), \( \bar{U} \) is minimized when \( T_{i+1} \) is reduced to \( T_i \).

\[
\bar{U} \geq 1 - \frac{C''(T_i + C'_i - C''_i)}{T_i^2}
\]

The analysis firstly considers the case of \( C'_i \geq C''_i \). In this case, the relations of \( U'_i \geq U''_i \) and \( U_i = U'_i + U''_i \) derive \( U''_i \leq U_i / 2 \). Hence \( \bar{U} \) is minimized as follows.

\[
\bar{U} \geq 1 - \frac{C''(T_i + C'_i - C''_i)}{T_i^2} \\
\geq 1 - \frac{U_i}{2}
\]

Remember that the utilization of a light task is at most \( U^* \) according to Equation (4.18). It means that \( \bar{U} \) is minimized when \( U_i = U^* \). Hence the worst-case utilization bound for the case of \( C'_i \geq C''_i \) is \( U^* = 1 - U^*/2 \), that is, \( U^* = 2/3 \approx 66\% \). The analysis secondly considers the case of \( C'_i \leq C''_i \). In this case, \( \bar{U} \) is minimized as follows with taking \( U'_i = U_i - U''_i \) into account.

\[
\bar{U} \geq 1 - \frac{C''(T_i + C'_i - C''_i)}{T_i^2} \\
= 1 - U''_i(1 + U'_i - U''_i) \\
= 1 - U''_i(1 + U_i - 2U''_i) \\
= 2U''_i^2 - (1 + U_i)U''_i + 1 \\
= 2\left(U''_i - \frac{1 + U_i}{4}\right)^2 + 1 - \frac{(1 + U_i)^2}{8} \\
\geq 1 - \frac{(1 + U_i)^2}{8}
\]
By the same token as the case of $C_i' \geq C_i''$, $\bar{U}$ is minimized when $U_i = U^*$. Hence, the worst-case utilization bound for the case of $C_i' \leq C_i''$ is $U^* = 1 - (1 + U^*)^2/8$, that is, $U^* = 4 \sqrt{2} - 5 \approx 65\%$. This is the absolute worst-case utilization bound for the EDDP algorithm. This utilization bound is valid for any processor where the light tasks are assigned. In addition, the utilization of any processor where a heavy task is assigned is equal to the utilization of the heavy task that is higher than $U^* = 4 \sqrt{2} - 5$. In consequence, the schedulable utilization of the whole system is always higher than $4 \sqrt{2} - 5 \approx 65\%$.

### 4.4.4 Improving Schedulable Utilization

In Section 4.3.4, two heuristics were introduced to improve the schedulable utilization for the EDDHP algorithm. Both of the heuristics are applied when a task is split in the task assigning phase. The first heuristic is splitting a task that maximizes the utilization bound. This heuristic can be used in the EDDP algorithm, since the utilization bound defined by Equation (4.21) is a function of $C_i'$, $C_i''$, $T_i$ and $T_{i+1}$ that can be all acquired when a task is split in the task assigning algorithm. The second heuristic is splitting a task only if necessary. This heuristic can be also used in the EDDP algorithm. The schedulable utilization for the EDDP algorithm degrades due to existence of the second portion of a split task according to the schedulability analysis. Consider the case in which $\tau_i$ is about to be split into $\tau_i'$ assigned to $P_x$ and $\tau_i''$ assigned to $P_{x+1}$. Let $U(P_x)$ be the current utilization of $P_x$ before $\tau_i'$ is assigned and let $U^*_{x+1}$ be the utilization bound of $P_{x+1}$ for the case in which $\tau_i''$ is assigned to. If task $\tau_i$ is not split into $P_x$ and $P_{x+1}$ but just assigned to $P_{x+1}$ like the partitioning approach, the schedulable utilization of $P_{x+1}$ is 100%. Therefore it is better not to split $\tau_i$ if $U(P_x) + C_i' + U^*_{x+1} < U(P_x) + 100\%$. This heuristic also has an effectiveness of reducing task preemptions, since it reduces unnecessary split tasks that cause more task preemptions.

This section also considers another heuristic technique specific for the EDDP algorithm. According to the EDDP assigning algorithm, a processor where a heavy task is assigned may leave the schedulable utilization, since the schedulable utilization bound of such processor is always 100%. The the heuristic is partitioning the remaining light tasks to the processors where a heavy task is assigned. This heuristic tries to assign the remaining light tasks to the processors where a heavy task is assigned by the partitioning method such as the first-fit algorithm and the best-fit algorithm when the task assigning algorithm fails. Using this heuristic, the processors are utilized more efficiently.

### 4.5 Summary

This chapter first described the basic strategy of the portioned scheduling technique. It then gave the theoretical descriptions of the three scheduling algorithms: RMDP.
CHAPTER 4. PERIODIC TASK SCHEDULING

EDDHP, and EDDP. It also presented the schedulability analyses of those algorithms. The important characteristics of the developed algorithms are:

- The RMDP algorithm has a worst-case utilization bound of 50% that is equal to the maximum utilization bound of the existing RM-based scheduling algorithms. The utilization bound of the RMDP algorithm is boosted up to 65% for the case in which the second portion of a split task is executed three times within any periods.

- The EDDHP algorithm is a dynamic-priority extension of the RMDP algorithm and has a worst-case utilization bound of 50%. Unlike the RMDP algorithm, the utilization bound does not rely on the number of the tasks, and hence the calculation of the utilization bound is done only once for each processor. This characteristic enables the algorithm to exploit the heuristics to improve a utilization bound.

- The EDDP algorithm is a further extension of the EDDHP algorithm. While the RMDP and EDDHP algorithms have similar manners in task assigning and task scheduling, the EDDP algorithm has different manners. It has a worst-case utilization bound of 65% that is higher than most of the traditional efficient algorithms. Only the EKG algorithm transcends this bound, however in implementation it requires more fine-grained timers than the one used in the EDDP algorithm. The heuristics to improve a utilization bound can be also applied to the EDDP algorithm.

- The developed algorithms are implementable enough for practical use. They require only a single timer invocation for either the first portion or second portion of a split task within each period.
Chapter 5

APERIODIC TASK SCHEDULING

This chapter presents two efficient techniques, called global dispatch and temporal migration, for the scheduling of aperiodic tasks on multiprocessor systems. The primary focus under aperiodic task scheduling is at reducing the response time of aperiodic tasks as much as possible, without violating timing constraints of periodic tasks. Timing constraints of aperiodic tasks are not taken into account, meaning that no aperiodic tasks are assumed to have deadlines. It is also assumed that periodic tasks are partitioned among processors. Though the portioned scheduling algorithms presented in Chapter 4 should be applied for the scheduling of periodic tasks when the total integration of the scheduling techniques established in this research is concerned, the dissertation designs the scheduling of aperiodic tasks with the simple partitioned scheduling algorithms in this chapter, since the theoretical description of the aperiodic task scheduling becomes far complicated if the periodic tasks are portioned. Instead, the dissertation mentions the compatibility of the techniques to the portioned scheduling algorithms in Section 5.4. Concrete integration of the established scheduling techniques and its precise theoretical analysis are left for the future work.

The global dispatch technique and the temporal migration technique are used by combining with traditional server algorithms. Since the dissertation covers both the static-priority scheduling scheme and the dynamic-priority scheduling scheme, the Priority Exchange (PE) algorithm [8] and the Total Bandwidth Server (TBS) algorithm [17], which are known to be efficient algorithms in the domain of uniprocessor systems, are adopted as bases for the two scheduling schemes respectively. Then, the objective of this chapter is to extend the designs of the PE algorithm and the TBS algorithm, in which the global dispatch technique and the temporal migration technique are exploited, to reduce the response time of aperiodic tasks.

This chapter begins with describing the basic strategies of the global dispatch technique and the temporal migration technique. Then, the designs of the PE-based algorithm and the TBS-based algorithm, in which those two techniques are exploited, are presented. The compatibility of the presented algorithms to the portioned scheduling algorithms is also discussed.
5.1 Basic Strategy

First of all, the basic strategies of the global dispatch technique and the temporal migration technique are explained. To the best of my knowledge, no work have strongly defined the scheduling model of aperiodic tasks on multiprocessor systems, especially for the partitioned scheduling. In this research, every processor has its local aperiodic server. The capacities, periods, and utilizations (bandwidths) of the servers are not necessarily symmetric. The arrival processor of each aperiodic task is not determined in advance; that is, aperiodic tasks can arrive on any processors.

In partitioned scheduling, it is fairly easy to partition aperiodic tasks as well as periodic tasks. In other words, the arrival processor is fixed for each aperiodic task. However, this partitioning raises unbalanced loads among the processors, which may result in furiously-low responsiveness on some processors. In order to enhance the responsiveness on every processor, the global dispatch technique submits the unique aperiodic handler implemented in one of the local schedulers, which receives all the arrivals of aperiodic tasks. The handler dispatches an arriving aperiodic task to a processor on which the response time of the aperiodic task is estimated to be minimized. After dispatching, the execution of the aperiodic task is handled by the local server. Thus, every processor is always maintained to be responsive. Figure 5.1 depicts the concept of the global dispatch technique. In the figure, the first processor $P_1$ carries out a global aperiodic handler and it dispatches an aperiodic task to $P_3$. The policy of the dispatch, i.e. the procedure to find the effective processor, depends on a server (scheduling) algorithm. The idea of the global dispatch technique is very straightforward, however no specific algorithms based on such a technique have ever been discussed. The disserta-
CHAPTER 5. APERIODIC TASK SCHEDULING

Task migrations

Figure 5.2: Concept of temporal migration

Figure 5.2: Concept of temporal migration

In addition to the global dispatch technique, the dissertation presents the temporal migration technique. This technique stands on the idea that periodic tasks with higher priorities than an arriving aperiodic task are temporarily migrated onto other processors to hand over processor time to the arriving aperiodic task, as long as the periodic tasks can be guaranteed to meet their deadlines after temporal migrations. Figure 5.2 illustrates the concept of the temporal migration technique. In the figure, the first processor $P_1$ has two periodic tasks stored in a priority-ordered queue in advance of an aperiodic task. Without migrations, the pending aperiodic task has a long wait for completions of the prior two periodic tasks so that the timing constraints of the periodic tasks are guaranteed. Migrating those two periodic tasks to the neighbor processors, on the other hand, the pending aperiodic task can be immediately executed on the processor. As a result, the response time of the aperiodic task is reduced. The issue of concern here is that the migrating periodic tasks must be maintained schedulable on the destination processors, and should not affect the responsiveness to the already-pending aperiodic tasks. The migration policy needs to take those kinds of affairs into account.

As for the concept of the temporal migration technique, there can be an idea that aperiodic tasks are also migrative. However, the task migrations inevitably degrade the cache hit ratios, which leads to the performance deterioration in practice. The requirements of periodic tasks are meeting deadlines and the performance deterioration is not a matter as much, because they are supposed to be scheduled based on the worst-case execution time from the theoretical point of view. In contrast, the performance dete-
prioritization is problematic for aperiodic tasks, because the aperiodic tasks are desired to complete as soon as possible. Therefore, the temporal migration technique is established based on the policy that no aperiodic tasks are migrated onto different processors from the processor on which they are released (dispatched).

5.2 The PE-based Algorithm

This section presents the PE-based algorithm that combines the global dispatch technique and the temporal migration technique with the traditional PE algorithm. It is presumed that periodic tasks are partitioned and are scheduled according to the RM algorithm on every processor.

The PE algorithm is often compared with the Deferrable Server (DS) algorithm [8], since their performances are competitive. With respect to the DS algorithm, the PE algorithm has slightly worse performance in terms of aperiodic responsiveness but achieves better schedulable utilization for periodic tasks. More precisely, the DS algorithm declines the utilization bound of the RM algorithm, where a server is assigned the highest priority, down to either of 50% or 65% depending on the characteristic of a task set [15]. By contrast, the PE algorithm never declines the utilization bound of the RM algorithm, i.e. the bound is never less than 69% for any task set [1], though the aperiodic responsiveness is slightly worse than the DS algorithm.

Considering the integration with the RMDP algorithm presented in Section 4.2, this research adopts the PE algorithm to design the specific procedures of the global dispatch and temporal migration techniques due to the following reasons: (i) the fact that the DS algorithm declines the utilization bound of the RM algorithm down to 50% in the worst case, which is originally 69%, implies that it would moreover decline the utilization bound of the RMDP algorithm, which is originally 50%, and (ii) the analysis of the schedulability for the RMDP algorithm combined with the DS algorithm is more complicated, since there exists two deferrable tasks; that is, the second portion of a split task and a server.

5.2.1 Priority Exchange Review

First of all, the design of the PE algorithm is reviewed. The PE algorithm uses a periodic server, usually at a high priority, for servicing aperiodic requests. While the DS algorithm differs the server capacity when there are no pending aperiodic tasks, the PE algorithm preserves the server capacity by exchanging it for the execution time of a lower-priority periodic task. At the beginning of each server period, the capacity is replenished at its full value. If aperiodic tasks are pending and the server is the ready task with the highest priority, then the aperiodic tasks are serviced using the available capacity. Otherwise, the capacity is exchanged for the execution time of the active periodic task with the highest
priority. When a priority exchange occurs between a periodic task and the server, the periodic task is executed at the priority level of the server, while the server accumulates a capacity at the priority level of the periodic task. Thus, the periodic task advances its execution, and the server capacity is not lost but preserved at a lower priority. If no aperiodic tasks arrive to use the capacity, a priority exchange continues with other lower-priority tasks until either the capacity is used for aperiodic service or it is degraded to the priority level of background processing, i.e., the priority level below the lowest priority. Since the objective of the PE algorithm is to provide high responsiveness to aperiodic requests, priority ties are always broken in favor of aperiodic tasks.

Figure 5.3 illustrates an example of scheduling aperiodic tasks using the PE algorithm on a uniprocessor system. In this example, the system has a server $\tau^{svr}$ with a capacity $C^{svr} = 1$ and a period $T^{svr} = 5$. There are two periodic tasks: $\tau_1(2, 10)$ and $\tau_2(12, 20)$. The server and the two tasks are scheduled according to the RM algorithm. Since the aperiodic time managed by the PE algorithm can be exchanged with all periodic tasks, the capacity accumulated at each priority level as a function of time is represented in overlapping with the schedule of the corresponding periodic task. In particular, the first timeline of Figure 5.3 shows the aperiodic tasks arriving on the system, and the second timeline visualizes the capacity available at the priority level of the server $\tau^{svr}$. The capacity is accumulated at the priority level of the periodic tasks and is preserved at a lower priority level if no aperiodic tasks arrive to use it.
server, whereas the third and the fourth ones refer to the capacities accumulated at the corresponding priority levels as a consequence of the priority exchange mechanism.

In the figure, at time $t = 0$, the server is brought at its full capacity, but no aperiodic tasks are pending, so the capacity is exchanged with the execution time of $\tau_1$. As a result, $\tau_1$ advances its execution and the server accumulates one unit of time at the priority level of $\tau_1$. At time $t = 2$, $\tau_1$ completes and $\tau_2$ begins execution. Since no aperiodic tasks are pending at this time, another priority exchange takes place between $\tau_1$ and $\tau_2$. At time $t = 5$, the capacity is replenished at the server priority, and but no aperiodic tasks are pending again, the server accumulates one unit of time at the priority level of $\tau_2$, since $\tau_1$ is idle. So does at the priority level of $\tau_2$ at time $t = 10$. At time $t = 11$, when the first aperiodic task $\alpha_1(11, 2)$ arrives, it is interesting to observe that the first unit of its computation time is immediately executed by using the capacity accumulated at the priority level of $\tau_1$. Then, since the remaining capacity is available at the lowest-priority level and $\tau_1$ is still active, $\alpha_1$ is preempted by $\tau_1$ and is resumed at time $t = 13$, when $\tau_1$ completes. At time $t = 17$, the capacity accumulated at the priority level of $\tau_2$ is used to execute the second aperiodic task $\alpha_2(17, 1)$. At time $t = 20$, the capacity is replenished at the server priority, and it is used to execute the third aperiodic task $\alpha_3(20, 1)$.

Assume that $n$ periodic tasks, $\tau_1, \tau_2, ..., \tau_n$, are scheduled on the system with a Priority Exchange server $\tau^{srv}$ that has a capacity $C^{srv}$ and a period $T^{srv}$. According to Lehoczky et al. [8], the processor is guaranteed to be schedulable if the server has the highest priority on the system and the following condition is satisfied where $U^{srv} = C^{srv}/T^{srv}$.

$$\sum_{i=1}^{n} U_i \leq n \left[ \left( \frac{2}{U^{srv}} + 1 \right)^{1/n} - 1 \right]$$ (5.1)

### 5.2.2 Global Dispatch Procedure

The global dispatch procedure of the PE algorithm dispatches an arriving aperiodic task to a processor on which the worst-case response time of the aperiodic task is minimized. The procedure expects that if the worst-case response time is low, the real response time would be also low. Hence, the worst-case response time of the aperiodic task in the PE algorithm must be analyzed.

Consider that an aperiodic task $\alpha_k$ is arriving on the system with an execution time of $E_k$ at time $a_k$. Let $\tilde{c}_x^{srv}(t)$ be the remaining capacity of the server, not including the exchanged ones, on a processor $P_x$ at time $t$. With respect to the latest finish time of $\alpha_k$, when $\alpha_k$ arrives at time $t = a_k$, the following three cases can occur, where $W_x(t)$ be the total amount of the remaining execution times of pending aperiodic tasks on $P_x$.

1. $E_k + W_x(t) \leq \tilde{c}_x^{srv}(t)$. Hence, $\alpha_k$ completes at time $t + E_k + W_x(t)$.
2. $E_k + W_x(t) > \bar{c}_x^{srv}(t)$, and $\bar{c}_x^{srv}(t)$ is completely consumed within the current period of the server. In this case, time units of $\Delta_k = \bar{c}_x^{srv}(t)$ are consumed by $\alpha_k$ and the prior pending aperiodic tasks.

3. $E_k + W_x(t) > \bar{c}_x^{srv}(t)$, but the current period of the server ends before $\bar{c}_x^{srv}(t)$ is completely consumed. In this case, time units of $\Delta = GT_x^{srv} - t$, where $G = \lceil t/T_x^{srv} \rceil$, are consumed by $\alpha_k$ and the prior pending periodic tasks.

In the last two cases, the time units consumed by $\alpha_k$ and the prior pending aperiodic tasks within the current period of the server can be computed by the following expression.

$$\Delta = \min\{\bar{c}_x^{srv}(t), GT_x^{srv} - t\}$$

Thus, the latest finish time $\hat{f}_k$ of $\alpha_k$ can be derived as follows.

$$\hat{f}_k = \begin{cases} t + E_k + W_x(t) & \text{if } E_k + W_x(t) \leq \bar{c}_x^{srv}(t) \\ (F + G)T_x^{srv} + H & \text{otherwise} \end{cases}$$

Here, $F$ and $H$ are expressed as follows.

$$\begin{align*}
F &= \frac{E_k + W_x(t) - \Delta}{C_x^{srv}} \\
H &= E_k + W_x(t) - \Delta - FC_x^{srv}
\end{align*}$$

The global aperiodic handler then dispatches $\alpha_k$ to a processor $P_x$ which has the smallest value of $\hat{f}_k$, because it expects that if the latest finish time of $\alpha_k$ is minimized on $P_x$, the real finish time of $\alpha_k$ would be also minimized. Notice that $\alpha_k$ may complete much earlier than $\hat{f}_k$, if idle times unused by periodic tasks are left enough, so $P_x$ is not necessarily a processor on which $\alpha_k$ can have the earliest finish time.

### 5.2.3 Temporal Migration Procedure

The temporal migration procedure of the PE algorithm is straightforward. Periodic tasks are assumed to be sorted so that $T_i \leq T_{i+1}$, namely $\tau_i$ has a higher priority than $\tau_{i+1}$. The $k$th aperiodic task dispatched to a processor $P_x$ by the aperiodic handler, presented in the previous section, is defined by $\alpha_{x,k}(a_{x,k}, E_{x,k})$ where $a_{x,k}$ is its arrival time and $E_{x,k}$ is its worst-case execution time. The finish time of $\alpha_{x,k}$ is denoted by $f_{x,k}$ and the response time of $\alpha_{x,k}$ is denoted by $R_{x,k} = f_{x,k} - a_{x,k}$. The consumed execution time and the remaining execution time of $\alpha_{x,k}$ at time $t$ are denoted by $e_{x,k}(t)$ and $\tilde{e}_{x,k}(t)$ respectively.

Assume that at time $t$ any aperiodic task $\alpha_{x,k}$ is dispatched to $P_x$, which has a local server $\tau_x^{srv}$ with a capacity $C_x^{srv}$ and a period $T_x^{srv}$. The utilization of the server is denoted by $U_x^{srv} = C_x^{srv}/T_x^{srv}$. At this moment, temporal migrations occur according to the procedure described as follows.
CHAPTER 5. APERIODIC TASK SCHEDULING

1. Let $\Lambda_x$ be the set of the periodic tasks whose indexes are lower than the lowest index of the tasks that hold the preserved (exchanged) capacities of the server. The procedure searches for a ready periodic job $\tau_{i,j}$ with the shortest period (the highest priority) in $\Lambda_x$, which is not being temporarily migrated within its period. If such a task is not found, the procedure exits and no temporary migrations occur at time $t$.

2. The procedure estimates the latest finish time of $\tau_{i,j}$ when it is served by the PE server on every processor $P_y$ as follows. Let $W_y(t)$ be the total amount of the remaining execution times of pending aperiodic tasks on $P_y$ and $A_y(t)$ be the total amount of the accumulated capacities at the higher priority levels than $\tau_{i,j}$ on $P_y$. Then, taking the same step described in Section 5.2.2, the worst-case finish time $\hat{f}_{i,j}$ of $\tau_{i,j}$ on $P_y$ is obtained by the following expression where $\tilde{c}_{i,j}(t)$ and $\tilde{c}_{srv_y}(t)$ denote the remaining execution time and the remaining capacity of $\tau_{i,j}$ and $\tau_{srv_y}$ respectively.

$$\hat{f}_{i,j} = \begin{cases} t + \tilde{c}_{i,j}(t) + W_y(t) & \text{if } \tilde{c}_{i,j} + W_y(t) \leq \tilde{c}_{srv_y}(t) \\ (F + G)T^{srv}_y + H & \text{otherwise} \end{cases} \quad (5.2)$$

Here, $G$, $\Delta$, $F$ and $H$ are replaced as follows.

$$\begin{align*}
G &= \left\lfloor \frac{t}{T^{srv}_y} \right\rfloor \\
\Delta &= \min\{\tilde{c}_{srv_y}(t), GT^{srv}_y - t\} \\
F &= \left\lfloor \frac{\tilde{c}_{i,j} + W_y(t) - \Delta - A_y(t)}{C_y} \right\rfloor \\
H &= \tilde{c}_{i,j} + W_y(t) - \Delta - A_y(t) - FC^{srv}_y
\end{align*}$$

3. If the procedure finds more than one processor that satisfies $\hat{f}_{i,j} \leq d_{i,j}$, $\tau_{i,j}$ is migrated onto one of such processors. Otherwise, the procedure removes $\tau_{i,j}$ from $\Lambda_x$, i.e. $\Lambda_x = \Lambda_x \setminus \{\tau_{i,j}\}$, and goes back to Step 1.

4. Let $P_z$ be a processor onto which $\tau_{i,j}$ is migrated and there are $l$ aperiodic tasks which have already arrived before time $t$. Then, $\tau_{i,j}$ is transformed into and is treated as the $l+1$th aperiodic task on $P_z$.

5. The procedure accumulates a capacity of $\tilde{c}_{i,j}(t)$ at the priority level $i$.

6. At time $r_{i,j+1}$, $\tau_i$ is migrated back onto $P_x$. That is, $\tau_{i,j+1}$ is released on $P_x$.

7. The procedure goes back to Step 1.
CHAPTER 5. APERIODIC TASK SCHEDULING

Responsiveness

The response time of an arriving aperiodic task can be reduced by the temporal migration procedure, since the processor time of migrated periodic tasks which were supposed to be executed preferentially is allocated to the pending aperiodic tasks. In addition, the response times of the aperiodic tasks which have been already pending are never affected by the temporal migrations, since the migrated periodic tasks are treated as the latest aperiodic tasks and they are never serviced in advance of the pending aperiodic tasks. However, it is not theoretically guaranteed that the overall response time of aperiodic tasks is reduced, because the response time of aperiodic tasks arriving in the future will be affected by the executions of the migrated periodic tasks. Thus, a theoretical guarantee is needed. Although such a guarantee is left for the future work, the dissertation demonstrates that the temporal migration technique has a capability to reduce the response time of aperiodic tasks dramatically in the simulation studies.

Schedulability

Periodic tasks are temporarily migrated only when their latest finish times, described by Equation (5.2), are guaranteed to be earlier than or equal to their deadlines. In addition, the migrated periodic tasks are serviced by the PE servers, and hence the schedulability of the other periodic tasks is never jeopardized. As a result, the system is obviously maintained schedulable after temporal migrations.

Processor Selection

The concern here is a selection of the processors, if more than one processor is found at Step 2, so that the response time of aperiodic tasks is reduced as much as possible. Note that obtaining the optimal selection is out of concern in this research, because (i) the definition of the optimal selection is vague, and (ii) even if the optimal selection is defined, the selection itself is reduced to a bin-packing problem which is NP-hard in a strong sense. Therefore, this research focuses on the heuristic approaches to select a processor onto which a task is migrated. The dissertation introduces the three heuristics: first-fit (FF), best-fit (BF), and worst-fit (WF). The three heuristics determines a target processor as follows.

- The FF heuristic migrates $\tau_{i,j}$ to the first (i.e., lowest-indexed) processor that satisfies $\hat{f}_{i,j} \leq d_{i,j}$.
- The BF heuristic migrates $\tau_{i,j}$ to a processor that (i) satisfies the condition of $\hat{f}_{i,j} \leq d_{i,j}$ and (ii) minimizes the value of $d_{i,j} - \hat{f}_{i,j}$.
- The WF heuristic, on the other hand, migrates $\tau_{i,j}$ to a processor that (i) satisfies the condition of $\hat{f}_{i,j} \leq d_{i,j}$ and (ii) maximizes the value of $d_{i,j} - \hat{f}_{i,j}$.

85
CHAPTER 5. APERIODIC TASK SCHEDULING

The order of the performance superiority in the three algorithms cannot be analyzed theoretically, hence it is evaluated in the simulation studies in Chapter 6.

5.3 The TBS-based Algorithm

This section presents the TBS-based algorithm that combines the global dispatch technique and the temporal migration technique with the traditional TBS algorithm for the discipline of dynamic-priority scheduling. The TBS algorithm is known to be an efficient dynamic-priority server algorithm in the domain of uniprocessor systems. The assumption herein is that periodic tasks are partitioned and scheduled according to the EDF algorithm on every processor.

5.3.1 Total Bandwidth Server Review

The TBS algorithm is known to be an efficient, probably the most efficient, algorithm in the traditional dynamic-priority server algorithms from the viewpoints of implementation complexity, memory requirement, runtime overhead and so on. Every time an aperiodic task arrives, the server assigns the possible earliest deadline to the aperiodic task, which is of course the virtual one. Once the aperiodic task is assigned the virtual deadline, it is scheduled with other periodic tasks according to the EDF algorithm. The virtual deadline of the aperiodic task is determined based on the bandwidth of the server. Suppose that the system has a server $\tau^{srv}$ with a bandwidth $U^{srv}$. Assuming that the $k$th aperiodic task $\alpha_k$ arrives on the system at time $a_k$ with an execution time $E_k$, then its virtual deadline $v_k$ is calculated by Equation (5.3), where $v_0 = 0$ by definition.

$$v_k = \max\{a_k, v_{k-1}\} + \frac{E_k}{U^{srv}} \quad (5.3)$$
Figure 5.4 shows an example of EDF scheduling produced by two periodic tasks, \( \tau_1(3, 6) \) and \( \tau_2(2, 8) \), and a server with a bandwidth \( U_{srv} = 1 - (U_1 + U_2) = 0.25 \). The first aperiodic task, denoted by \( \alpha_1(2, 2) \), arrives at time \( a_1 = 2 \) with an execution time \( E_1 = 2 \) and is serviced with a virtual deadline \( v_1 = a_1 + E_1/U_{srv} = 2 + 2/0.25 = 10 \). Being \( v_1 \) a deadline later than the deadlines of \( \tau_{1,1} \) and \( \tau_{2,1} \), the aperiodic task cannot begin execution until \( \tau_{1,1} \) and \( \tau_{2,1} \) complete. Similarly, the second aperiodic task \( \alpha_2(7, 1) \), which arrives at time \( a_2 = 7 \) with an execution time \( E_2 = 1 \), receives a virtual deadline \( v_2 = \max\{a_2, v_1\} + E_2/U_{srv} = 10 + 4 = 14 \), but it is not serviced immediately, since at time 7 \( \tau_1 \) is still active with an earlier deadline of 12. Finally, the third aperiodic task \( \alpha_3(17, 2) \) arrives at time \( a_3 = 17 \) with an execution time \( E_3 = 2 \) and gets a virtual deadline \( v_3 = a_3 + E_3/U_{srv} = 25 \). It does not receive immediate service, since at time 17 \( \tau_1 \) and \( \tau_2 \) are active and have deadlines of 24 earlier than \( \alpha_3 \).

### 5.3.2 Global Dispatch Procedure

The global dispatch procedure of the TBS algorithm is much more straightforward than that of the PE algorithm. Unlike the PE algorithm, it is far hard to analyze the worst-case response time of each aperiodic task in the TBS algorithm, since the priority order of tasks is dynamically changed based on deadlines. Therefore, another indicator of the response time is needed.

As described in the previous section, every aperiodic task is assigned the virtual deadline in the TBS algorithm. Then, no aperiodic tasks miss their virtual deadlines as long as the total processor utilization is less than or equal to 1. In other words, every aperiodic task is guaranteed to complete before the assigned virtual deadline. Thus, the virtual deadline can be deemed to be the worst-case response time, though the aperiodic task usually completes much earlier than the virtual deadline. By the same reason as the PE algorithm, it is expected that the response time of the aperiodic task is minimized by dispatching it to a processor on which the earliest virtual deadline is assigned. In consequence, the global aperiodic handler for the TBS algorithm is designed so that it dispatches an arriving aperiodic task to a processor on which the virtual deadline expressed by Equation (5.3) is minimized.

### 5.3.3 Temporal Migration Procedure

This section describes the temporal migration procedure of the TBS algorithm. Like in Section 5.2.3, the \( k \)th aperiodic task dispatched to a processor \( P_s \) by the aperiodic handler is defined by \( \alpha_{s,k}(a_{s,k}, E_{s,k}) \) where \( a_{s,k} \) is its arrival time and \( E_{s,k} \) is its worst-case execution time. The virtual deadline assigned to \( \alpha_{s,k} \) is described by \( v_{s,k} \). Then, the finish time of \( \alpha_{s,k} \) is denoted by \( f_{s,k} \) and the response time of \( \alpha_{s,k} \) is denoted by \( R_{s,k} = f_{s,k} - a_{s,k} \). The consumed execution time and the remaining execution time of \( \alpha_{s,k} \) at time \( t \) are denoted by \( e_{s,k}(t) \) and \( \tilde{e}_{s,k}(t) \) respectively.
Suppose that at time $t$ any aperiodic task $\alpha_{x,k}$ arrives on any processor $P_x$, which has its own server with a bandwidth $U_{srv_x}$, and is assigned a virtual deadline $v_{x,k}$ according to Equation (5.3). At this moment, temporal migrations occur according to the following procedure.

1. The procedure searches for a periodic job $\tau_{i,j}$ with the earliest deadline, which has not been already migrated in its period. If such a job does not exist, the procedure exits.

2. The algorithm seeks any processor $P_y$ that satisfies Inequation (5.4) where $v_{y,l}$ is the virtual deadline assigned to the last aperiodic task $\alpha_{y,l}$ that arrived on $P_y$, and $\tilde{c}_{i,j}(t)$ is the remaining execution time of $\tau_{i,j}$.

$$d_{i,j} \geq \max\{t, v_{y,l}\} + \frac{\tilde{c}_{i,j}(t)}{U_{srv_y}}$$  \hspace{1cm} (5.4)

3. The procedure migrates $\tau_{i,j}$ onto $P_y$ and virtually deals with it as the $l + 1$th aperiodic task on $P_y$ with a virtual deadline $v_{y,l+1}$, expressed by Equation (5.5).

$$v_{y,l+1} = \max\{t, v_{y,l}\} + \frac{\tilde{c}_{i,j}(t)}{U_{srv_y}}$$  \hspace{1cm} (5.5)

4. The algorithm reassigns the improved virtual deadline $v'_{x,k}$, expressed by Equation (5.6), to $\alpha_{x,k}$.

$$v'_{x,k} = \max\{t, v_{x,k-1}\} + \frac{E_{x,k}}{U_{srv} + \frac{\tilde{c}_{x,k}(0)}{T_{i,j}}}$$  \hspace{1cm} (5.6)

5. The procedure then makes sure that its next job $\tau_{i,j+1}$ is migrated back onto $P_x$ at time $r_{i,j+1}$, and it may be temporarily migrated again.

The procedure assumes that the migrated job $\tau_{i,j}$ transforms into the aperiodic task $\alpha_{y,l+1}$. Hence, it must use $v_{y,l+1}$ to calculate the virtual deadline of the next arriving aperiodic task. Note that assigning $v_{y,l+1}$ to $\tau_{i,j}$ is not desirable from the responsiveness point of view, because it is a periodic job and does not required to complete as soon as possible. In order to execute the pending aperiodic tasks earlier, the true deadline $d_{i,j}$ should be assigned. However, such a deadline assignment leads a schedulability analysis to be complicated. In consequence, this research takes a simple approach in which the virtual deadline $v_{y,l+1}$ is assigned to the migrating job $\tau_{i,j}$. More sophisticated deadline assignments are left for the future work. Notice that the Step 1 of the procedure guarantees that any migrated job is never migrated again to another processor in the same period. In other words, two-level migrations are not permitted.
Figure 5.5 depicts an example of the TBS algorithm in which the temporal migration technique is applied. Suppose that there are two processors: \( P_x \) and \( P_y \). The same periodic tasks \( \tau_1(3, 6) \) and \( \tau_2(2, 8) \) in the example of Figure 5.4 are scheduled on \( P_x \). In addition, two more tasks \( \tau_3(1, 4) \) and \( \tau_4(5, 10) \) are scheduled on \( P_y \). The servers on both of the processors have bandwidths of 0.25. The periodic tasks are scheduled by the EDF algorithm. For simplicity of explanation, no aperiodic tasks are expected to arrive on \( P_2 \). At time \( t = 2 \), the first aperiodic task \( \alpha_{x,1}(2, 2) \) arrives on \( P_x \). Without a temporal migration, it receives a virtual deadline of 10. Meanwhile, exploiting the temporal migration technique, \( \tau_1 \) can be temporarily migrated onto \( P_y \), since it satisfies Equation (5.4) and receives a virtual deadline \( v'_{x,1} = t + c_{1,1}(t)/U_{srv} = 2 + 1/0.25 = 6 \) according to Equation (5.5). Then, \( \alpha_{x,1} \) is assigned an improved virtual deadline \( v'_{x,1} = 2 + 2/(0.25 + 1/6) = 6.8 \) according to Equation (5.6). \( \alpha_{x,1} \) gets an virtual deadline earlier than \( \tau_2 \) and \( \tau_1 \) is temporarily absent on \( P_x \). \( \alpha_{x,1} \) can be executed immediately. In contrast, \( \tau_1 \) cannot be temporarily migrated when the second aperiodic task \( \alpha_{x,2} \) arrives at time \( t = 7 \), since it would receive a virtual deadline \( t + c_{1,2}(t)/U_{srv} = 7 + 2/0.25 = 15 \) that is later than the original deadline 12. Hence, \( \alpha_{x,2} \) does not receive an improved virtual deadline.
CHAPTER 5. APERIODIC TASK SCHEDULING

in this case. However, $\alpha_{x,2}$ can be executed one time unit earlier than the example of Figure 5.4, because at time 2 $\tau_1$ consumes one time unit on $P_y$ by a temporal migration, and hence the whole schedule is advanced one time unit. Finally, at time $t = 17$ the third aperiodic task $\alpha_{x,3}$ arrives and $\tau_2$ can be migrated onto $P_y$. $\alpha_{x,3}$ therefore receives an improved virtual deadline $v'_{x,3} = 17 + 2/(0.25 + 1/8) = 22.3$ and is serviced immediately, since it has a deadline earlier than $\tau_1$ and $\tau_2$. Comparing Figure 5.5 with Figure 5.4, the effectiveness of temporal migrations can be clearly observed. The response time of $\alpha_{x,1}$ is reduced to $R_{x,1} = 4 - 2 = 2$ from $7 - 2 = 5$. Similarly, the response times of $\alpha_{x,2}$ and $\alpha_{x,3}$ are reduced to $R_{x,2} = 10 - 7 = 3$ and $R_{x,3} = 19 - 17 = 2$ from $11 - 7 = 4$ and $23 - 17 = 6$ respectively.

Responsiveness

Two benefits for reducing the response time of an aperiodic task are obtained by this migration policy. The first one is that the turn of the aperiodic execution comes faster, since a periodic job with an earlier deadline may be migrated onto another processor, and hence it disappears from the processor. Such a benefit is observed in the response time of $\alpha_{x,2}$ in Figure 5.5. Although it does not have an improved virtual deadline, but it can be executed one time unit earlier, because $\tau_1$ disappears from the processor at time 2. The temporal migration procedure presented in Section 5.3.3 can be recursive to moreover improve the responsiveness to the arriving aperiodic task. More precisely, though the algorithm targets only one job with an earlier deadline at the Step 1 of the procedure, if the algorithm chooses more than one job with an earlier deadline to be temporarily migrated onto different processors, it is expected that the response time of the arriving aperiodic task can be moreover reduced. However, such a recursive procedure requires much more computation cost and is not adopted in the algorithm, because this work aims practical approaches.

The second benefit is that the server bandwidth is temporarily broadened due to temporal absences of migrating periodic jobs, which enables an aperiodic task to have an earlier virtual deadline as expressed by Equation (5.6). Such a benefit is observed in $\alpha_{x,1}$ and $\alpha_{x,3}$ in Figure 5.5. For example, $\tau_{x,1}$ is supposed to have a virtual deadline of 10 without temporal migrations, but it actually receives a virtual deadline of 6.8 with temporal migrations, and the improved virtual deadline skips over the deadline of $\tau_2$ equal to 8. The same deadline skip can be seen for the case of $\alpha_{x,3}$. In addition, the responsiveness to the already-pending aperiodic tasks on $P_y$ is never affected by a migrating job $\tau_{i,j}$, since the migration condition of Inequation (5.4) guarantees that the migrating job is inevitably assigned a virtual deadline later than any virtual deadlines of the pending aperiodic tasks. Therefore, the migrating job is never scheduled in advance of the pending aperiodic tasks, which means that the completion time of the pending aperiodic tasks is never prolonged. Meanwhile, the response time of the aperiodic tasks arriving in the future within $[t, v_{y,l+1}]$ on $P_y$ can be worse unfortunately, because the virtual deadlines assigned to them are shifted for $v_{y,l+1} - v_{y,l}$ time units, compared to
the ones without the temporal migration techniques. However, when there are periodic tasks with deadlines earlier than the assigned virtual deadlines of the arriving aperiodic tasks, the algorithm attempt temporal migrations after all, and the response time can be reduced as much as possible again. In consequence, the expectation is that the total responsiveness to aperiodic tasks is improved after all.

**Schedulability**

In temporal migrations, a migrating job \( \tau_{i,j} \) and the original jobs scheduled on \( P_y \) are guaranteed to be schedulable. The proof is as follows. According to the TBS theory [16, 17], assigning a deadline \( v_{y,l+1} \) to \( \tau_{i,j} \), which is earlier than its true deadline \( d_{i,j} \), on \( P_y \) does not jeopardize the schedulability of the EDF algorithm. This fact implies that the original jobs are still guaranteed to be schedulable for the case of \( \tau_{i,j} \) having a true deadline \( d_{i,j} \) according to the sustainability of the EDF algorithm, which is introduced by Baruah *et al.* [91]. As a result, the real-time constraints of the periodic tasks never collapse due to the temporal migration procedure.

**Processor Selection**

The next step is a selection of the processors, if more than one processor is found at Step 1, so that the response time of aperiodic tasks is reduced as much as possible. Like the case of the PE-based algorithm described in Section 5.2.3, obtaining the optimal selection is not considered here, and instead the three heuristics introduced in Section 5.2.3 are applied again. The three heuristics migrate \( \tau_{i,j} \) as follows.

- The FF heuristic migrates \( \tau_{i,j} \) to the first (i.e., lowest-indexed) processor that satisfies Inequation (5.4).
- The BF heuristic migrates \( \tau_{i,j} \) to a processor that (i) satisfies Inequation (5.4), and (ii) minimizes the difference between \( d_{i,j} \) and \( v_{y,l+1} \).
- The WF heuristic, on the other hand, migrates \( \tau_{i,j} \) to a processor \( P_w \) that (i) satisfies Inequation (5.4), and (ii) maximizes the difference between \( d_{i,j} \) and \( v_{y,l+1} \).

Like the PE-based algorithm again, the order of the performance superiority in the three algorithms cannot be analyzed theoretically, hence it is evaluated in the simulation studies in Chapter 6.

### 5.4 Compatibility to Portioned Scheduling

The previous sections presented the designs of the PE algorithm and the TBS algorithm, in which the global dispatch technique and the temporal migration technique are applied, based on the partitioned scheduling. In this section, the compatibility of
the PE-based algorithm and the TBS-based algorithm to the portioned scheduling algorithms presented in Chapter 4 is discussed.

The following factors derive that the PE-based algorithm can be compatible with the RMDP algorithm.

- Equation (5.1) implies that the server can be objectified as a periodic task with the priority level of the server in terms of schedulability analysis.

- The schedulability analysis of the RMDP algorithm allows any periodic task whose priority is lower than the second portion of a split task on each processor to be deemed as a periodic task in the RM algorithm.

- Like the RM algorithm, the schedulability analysis of the RMDP algorithm also depends on the number of tasks executing on the processor.

Consequently, the schedulability of the RMDP algorithm in which the PE algorithm is combined can be analyzed by considering the server to be $\tau_1$ in Figure 4.10, Figure 4.11, and Figure 4.12 respectively. Taking into account that the schedulable condition of the PE-based algorithm is same as that of the PE algorithm, it holds that the incremental PE-based algorithm presented in this chapter can be also used with the RMDP algorithm with a guarantee of schedulability.

The proof of the TBS-based algorithm to be compatible to the EDDHP and EDDP algorithms is more straightforward. The temporarily-migrating jobs are scheduled according to the EDF algorithm with the virtual deadline of the TBS algorithm. Since all the tasks are still scheduled based on the deadlines, the algorithm is obviously compatible to the EDDHP and EDDP deadline-driven scheduling algorithms.

5.5 Summary

This chapter first described the strategies of the global dispatch technique and the temporal migration technique which can improve the responsiveness to aperiodic tasks dramatically. It then presented the PE-based algorithm and the TBS-based algorithm in which the global dispatch procedure and the temporal migration procedure are implemented. It also discussed the compatibility of the presented algorithms to the portioned scheduling algorithms presented in Chapter 4. The important characteristics of the presented algorithms are:

- The PE-based algorithm is developed to work with the RM scheduling. When an aperiodic task arrives, it is dispatched to a processor on which the worst-case response time is minimized. After dispatching, periodic jobs with priorities higher than the PE server are temporarily migrated onto different processors within their periods and are serviced by the PE servers performing on the destination processors.
• The TBS-based algorithm is invented to work with the EDF scheduling. When an aperiodic task arrives, it is dispatched to a processor on which the assigned virtual deadline is minimized. After dispatching, periodic jobs with deadlines earlier than the virtual deadline of the dispatched aperiodic task are temporarily migrated onto different processors within their periods and are scheduled according to the TBS algorithm on the destination processors.

• Both PE-based algorithm and the TBS-based algorithm guarantee all periodic jobs to be schedulable.

• Both PE-based algorithm and the TBS-based algorithm are or can be compatible to the portioned scheduling algorithms developed in this research.
Chapter 6

PERFORMANCE EVALUATION

This chapter presents the results of simulations conducted to evaluate the performance advancement of the proposed techniques and algorithms. A series of simulations estimates the theoretical capability of the algorithms. Thus, no overheads are accounted for and all processor time units can be used for task workloads. In other words, no time units are consumed for task queuing, context switching, interruption handling, and any other run-time calculations.

The effectiveness of the portioned scheduling technique is first described by the simulation studies for periodic task scheduling. The RMDP algorithm is compared with the traditional static-priority scheduling algorithms. The EDDHP and EDDP algorithms, on the other hand, are compared with the traditional dynamic-priority scheduling algorithms. The simulation results show that the proposed three algorithms outperform most of the traditional algorithms in terms of schedulable utilization, with a small number of task preemptions. Then, the effectiveness of the global dispatch technique and the temporal migration technique is demonstrated by the simulation studies for aperiodic task scheduling. The PE-based algorithm and the TBS-based algorithm which exploit global dispatches and temporal migrations are respectively compared with the PE algorithm and the TBS algorithm designed for the scheduling on uniprocessor systems. The simulation results illustrate that the response time of aperiodic tasks is dramatically reduced by using the developed techniques.

6.1 Simulation Studies for Periodic Task Scheduling

This section shows the superiority of the developed scheduling algorithms for periodic tasks by comparing the traditional scheduling algorithms, including both efficient and optimal ones, in terms of schedulable utilization and the number of task preemptions. The worst-case utilization bounds for the developed scheduling algorithms were already derived by the schedulability analysis, and hence the algorithms can be compared with the traditional algorithms from the theoretical point of view. However, in
order to estimate the performance of an algorithm truly, the sufficient number of task sets with different properties must be submitted to the algorithm, since the utilization bound for an algorithm actually varies depending on the property of a given task set. The performance of the algorithm is also dominated by the number of task preemptions. Even if the algorithm achieves high schedulable utilization in a simulation, the practical performance may be much worse than the simulated result, if the algorithm generates a great number of task preemptions. Thus, the dissertation evaluates the algorithms in terms of both schedulable utilization and the number of task preemptions.

In this section, the performances of the static-priority scheduling algorithms are first examined. The RMDP algorithm is compared with the traditional static-priority scheduling algorithms: RM-FF [42], RM-FFDU [50], R-BOUND-MP-NFR [57], RM-US [45], and WM [67, 66]. Secondly, the performances of the dynamic-priority scheduling algorithms are estimated. The EDDHP and EDDP algorithms are compared with the traditional dynamic-priority scheduling algorithms: EDF-FF [58], EDF-BF [58], EDF-US [47], EDZL [73, 75], and EKG [81]. The EDDP and EDDHP algorithms exploit the heuristic procedures for improving schedulable utilizations, which were presented in Section 4.4.4 and Section 4.3.4 respectively. The EKG algorithm takes two parameters with a consideration of the trade-off. The first one, denoted by EKG-2, takes a parameter \( k = 2 \) with a lower utilization bound of 66% and fewer task preemptions. On the other hand, the second one, denoted by EKG-M, takes a parameter \( k = M \) with the optimal utilization bound of 100% in exchange for more task preemptions. The optimal Pfair algorithms [62, 63, 64] and the LLREF algorithm [79] are not included in the simulations, since they are known to be optimal as well as the EKG-M algorithm but generate much more task preemptions than the EKG-M algorithm.

### 6.1.1 Experimental Setup

The simulations estimate the schedulability of an algorithm as follows. Every system utilization \( U_{sys} \) ranging from 30% to 100%, 1000 task sets with different properties, whose system utilizations are all \( U_{sys} \) equally, are generated and submitted to the algorithm. Then, the success ratio, defined by the following expression, is measured.

\[
\text{Success Ratio} = \frac{\text{the number of successfully scheduled task sets}}{\text{the number of scheduled task sets}}
\]

A successfully scheduled task set is defined as follows.

- For the partitioned scheduling algorithms and the portioned scheduling algorithms, a task set is said to be successfully scheduled, if and only if all the tasks in the task set can be assigned to the processors without breaking the utilization bounds of any processors.

- For the global scheduling algorithms, a task set is said to be successfully scheduled, if and only if the task set is accepted by the offered schedulability test.
CHAPTER 6. PERFORMANCE EVALUATION

Having a high success ratio means that the schedulability at the system utilization is also high. Hereinafter, the schedulable utilization for an algorithm refers to the maximum system utilization at which the algorithm achieves the success ratio of 100%. Meanwhile, the simulations estimate the number of task preemptions for an algorithm by calculating its average number, defined by the following expression.

\[
\frac{\text{the total number of task preemptions}}{\text{the number of scheduled task sets}}
\]

Each simulation is characterized by the four parameters: \( M, U_{\text{max}}, U_{\text{min}} \) and \( U_{\text{sys}} \). \( M \) is the number of the processors. \( U_{\text{max}} \) and \( U_{\text{min}} \) are the maximum and minimum values of the processor utilization of every individual task in a given task set. \( U_{\text{sys}} \) refers to the utilization of the system, which ranges from 0% to 100%. Note that \( U_{\text{sys}} \times M \) is equal to the total utilization of all the generated tasks. Although many combinations of the parameters can be considered, the simulations attempt the following combinations due to the limitation of space. The system utilization is determined within the range of [30%, 100%]. Most of the existing algorithms can successfully schedule a task set with a system utilization below 30%, hence system utilizations below 30% are removed. As for the number of the processors, the simulations prepare the three sets: \( M = 4 \), \( M = 8 \), and \( M = 16 \), since embedded processors hardly offer more than 16 cores due to limitation of the scale and the resource. In a humanoid robot system [92] that has been developed in my laboratory, no tasks seldom utilize more than 50% of an individual processor (core). However, the quality of the activities is required to be enhanced, heavier tasks (tasks with high utilizations) are submitted. Thus, for both of the cases, the simulations prepare the two sets of \( U_{\text{min}} \) and \( U_{\text{max}} \): \((U_{\text{min}}, U_{\text{max}}) = (0.01, 0.5)\) and \((U_{\text{min}}, U_{\text{max}}) = (0.01, 1.0)\).

A task set \( \Gamma \) is generated as follows. A new periodic task is appended to \( \Gamma \) as long as \( U(\Gamma) \leq U_{\text{total}} \) is satisfied. For each task \( \tau_i \), its utilization \( U_i \) is computed based on a uniform distribution within the range of \([U_{\text{min}}, U_{\text{max}}]\). Only the utilization of the task generated at the very end is adjusted so that \( U(\Gamma) \) becomes equal to \( U_{\text{total}} \). \( T_i \) is determined within the range of [100, 3000] randomly, since the feedback periods of control and multimedia tasks in humanoid robots are usually about \( 1ms \sim 30ms \). In other words, a minimum time unit simulates 10\( \mu s \). Then, the execution time of the task \( C_i = U_i T_i \) is calculated.

6.1.2 Simulation Results of Static-Priority Scheduling

Scheduled Utilization

Figure 6.1, Figure 6.2, and Figure 6.3 show the success ratios for the static-priority scheduling algorithms, RMDP, RM-FF, RM-FFDU, R-BOUND-MP, RM-US, with respect to task sets in which the utilization of every individual task ranges from within \([0.01, 0.5]\). The schedulable utilization for the proposed RMDP algorithm is around 70 – 73%, which is much higher than the other efficient algorithms, such as RM-FF,
Figure 6.1: Success ratio \((M = 4, U_{\text{min}} = 0.01, U_{\text{max}} = 0.5)\).

Figure 6.2: Success ratio \((M = 8, U_{\text{min}} = 0.01, U_{\text{max}} = 0.5)\).
CHAPTER 6. PERFORMANCE EVALUATION

Figure 6.3: Success ratio ($M = 16, U_{\text{min}} = 0.01, U_{\text{max}} = 0.5$).

Figure 6.4: Success ratio ($M = 4, U_{\text{min}} = 0.01, U_{\text{max}} = 1.0$).
CHAPTER 6. PERFORMANCE EVALUATION

Figure 6.5: Success ratio ($M = 8, U_{\text{min}} = 0.01, U_{\text{max}} = 1.0$).

Figure 6.6: Success ratio ($M = 16, U_{\text{min}} = 0.01, U_{\text{max}} = 1.0$).
CHAPTER 6. PERFORMANCE EVALUATION

RM-FFDU, R-BOUND-MP, and RM-US. The RM-FF, RM-FFDU, and R-BOUND-MP algorithms are very competitive. The schedulable utilizations for those algorithms are around 57 ∼ 60%. Those algorithms actually differ from each other only in task assigning methods. The RM-FFDU algorithm, which sorts the tasks in order of decreasing utilization before assigning, slightly performs worse than the other two algorithms. Meanwhile, the R-BOUND-MP algorithm, which sorts the tasks in order of increasing period before assigning, slightly performs better than the other two algorithms. However, the performance difference among the three algorithms is only a little, hence sorting the tasks does not dominate a schedulable utilization very much for the case in which the utilization of every individual task is less than 50%. The RM-US algorithm performs quite badly compared to the other algorithms. Note that the schedulability test of the RM-US algorithm is not tight, since the schedulability test presumes the absolute worst-case performance. Therefore, the RM-US algorithm may schedule a task set with a higher system utilization successfully. The WM algorithm achieves schedulable utilizations of around 90%, which are much better than the other algorithms. This superiority is acquired from its characteristic of fairness. Since the WM algorithm has its basis on Pfair scheduling, which is able to make an optimal schedule on multiprocessor systems if the task priorities can be assigned dynamically, it offers an excellent performance, though it does not reach the optimality due to the restriction of the static-priority discipline.

Figure 6.4, Figure 6.5, and Figure 6.6 show the success ratios for the static-priority scheduling algorithms with respect to task sets in which the utilization of every individual task ranges within [0.01, 1.0]. Although this ranging generates several heavy tasks, the order of superiority in the algorithms has no changes compared to the previous cases in which only light tasks reside. The remarkable point is that the schedulable utilization for the WM algorithm decreases as the number of the processors increases. That is because the utilization bound for the WM algorithm depends on the number of the processors. Meanwhile, the schedulable utilization for the proposed RMDP algorithm is stable around 73% regardless of the number of the processors, because the utilization bound for the RMDP algorithm does not depend on the number of the processors. Therefore, the expectation is that the RMDP algorithm may outperform the WM algorithm for the case of many processors.

The Number of Task Preemptions

For calculation of task preemptions, the interval of the simulations is set the smaller of the least common multiple of the task periods in the given task set and $2^{32}$ that is the maximum length on a 32-bit computer. For each algorithm, the average number of task preemptions in the 1000 task sets is calculated every system utilization. Then, the number of task preemptions for each algorithm relative to that for the RMDP algorithm is calculated only only for the case in which both the target algorithm and the RMDP algorithm have a schedulable utilization of 100%. Otherwise, the comparison becomes
CHAPTER 6. PERFORMANCE EVALUATION

Figure 6.7: The number of preemptions \((M = 4, U_{\text{min}} = 0.01, U_{\text{max}} = 0.5)\).

Figure 6.8: The number of preemptions \((M = 8, U_{\text{min}} = 0.01, U_{\text{max}} = 0.5)\).
CHAPTER 6. PERFORMANCE EVALUATION

Figure 6.9: The number of preemptions ($M = 16, U_{\text{min}} = 0.01, U_{\text{max}} = 0.5$).

Figure 6.10: The number of preemptions ($M = 4, U_{\text{min}} = 0.01, U_{\text{max}} = 1.0$).
Figure 6.11: The number of preemptions ($M = 8, U_{min} = 0.01, U_{max} = 1.0$).

Figure 6.12: The number of preemptions ($M = 16, U_{min} = 0.01, U_{max} = 1.0$).
unfair. Note that the WM algorithm consistently causes more than fifty times as many task preemptions as the RMDP algorithm, hence the evaluation of the WM algorithm is not included in this section.

Figure 6.7, Figure 6.8, and Figure 6.9 show the numbers of task preemption for the static-priority scheduling algorithms relative to that for the RMDP algorithm, with respect to task sets in which the utilization of every individual task ranges within [0.01, 0.5]. The RMDP algorithm causes about 1.3 ~ 1.8 times as many task preemptions as the RM-FF, RM-FFDU, and R-BOUND-MP algorithms. Since the RMDP algorithm preempts the second portion of a split task when its corresponding first portion is dispatched on the neighbor processor, it incurs additional preemptions to the RM scheduling. Meanwhile the RM-FF, RM-FFDU, and R-BOUND-MP algorithms never incur such additional preemptions, since those algorithms completely partition the tasks among the processors. However, the RMDP algorithm has fewer task preemptions than the RM-US algorithm. The RM-US algorithm deals with all the tasks globally. Hence, the ordering of the task priorities in the RM-US algorithm is changed more frequently than in the other algorithms. As a result, the RM-US algorithm totally causes more task preemptions than the other algorithms.

Figure 6.10, Figure 6.11, and Figure 6.12 show the numbers of task preemption for the static-priority scheduling algorithms relative to that for the RMDP algorithm, with respect to task sets in which the utilization of every individual task ranges within [0.01, 1.0]. This case expands the performance difference between the RMDP algorithm and the partitioned scheduling algorithms. Since each task is likely to have high utilization due to $U_{\text{max}} = 1.0$, which means that the execution time of the task is also likely to be long, the scheduled timings of the first and second portions of a split are easily overlapped on two processors. According to the RMDP algorithm, the second portion of a split task is preempted when its corresponding first portion is scheduled, and hence the number of task preemptions are increased compared to the previous cases in which only light tasks exist. As a result, the RMDP algorithm causes about 2.0 ~ 2.5 times, which is worse than the previous light task set cases, as many task preemptions as the RM-FF, RM-FFDU, and R-BOUND-MP algorithms, though the RM-US algorithm causes more due to the characteristic of global scheduling.

The impact of the performance difference in the number of task preemptions to the system depends on the ratio of the CPU time of task executions and the CPU time of task preemptions. For example, the RMDP algorithm has a schedulable utilization about 13% higher than the RM-FFDU algorithm in Figure 6.4. However, the number of task preemptions for the RMDP algorithm is about twice as many as that for the RM-FFDU algorithm in this case. This relation means that the RM-FFDU algorithm may be better than the RMDP algorithm, if a scheduler consumes more than 13% of the entire system time for task preemptions. System designers should take this fact into account when they choose a scheduling algorithm. As for this example, since the scheduler hardly consumes 13% of the system time for task preemptions, considering the performance of recent processors, the RMDP algorithm is probably a better choice.
6.1.3 Simulation Results of Dynamic-Priority Scheduling

Schedulable Utilization

Figure 6.13, Figure 6.14, and Figure 6.15 show the success ratios for the dynamic-priority scheduling algorithms, EDDP, EDDHP, EDF-FF, EDF-BF, EDF-US, EDZL, EKG-2\((k = 2)\), and EKG-M\((k = M)\), with respect to task sets in which the utilization of every individual task ranges within \([0.01, 0.5]\). The results show that the EDDP, EDDHP, EDF-FF, and EKG-2 algorithms are very competitive. The EDDHP algorithm slightly offers a better performance than the other algorithms. While the schedulable utilization for the EDDHP algorithm is around 90%, those for the EDDP and EKG-2 algorithms are around 87%, and that for the EDF-BF algorithm is around 85%. It is remarkable that the EDDHP algorithm outperforms the EDDP algorithm, though the worst-case utilization bound for the EDDP algorithm is about 65% that is higher than 50% for the EDDHP algorithm. This fact implies that the approach of assigning the highest static-priority to the second portion of a split task can provide a better schedulability in practice than the approach of assigning the virtual deadline. Note that the success ratio for the EKG-M algorithm drops down below 100% before the system utilization reaches 100%, though the EKG algorithm is supposed to become optimal for \(k = M\). Since the execution time of a task cannot be split less than the minimum time unit of 1 in the simulations, each processor inevitably remains a little room unless its remaining utilization in splitting task \(\tau_i\) is an integer multiple of \(1/T_i\). As a result, a few tasks may fail to be assigned to any processor, if the system utilization is very close to 100%. The EDF-FF algorithm performs relatively well, but its schedulable utilization is around 77% that is 8% behind the EDF-BF algorithm. This inferiority of the EDF-FF algorithm to the EDF-BF algorithm is attributed by the performance difference between the FF heuristic and the BF heuristic. Since the BF heuristic packs items more efficiently than the FF heuristic, the performance of the EDF-BF algorithm is also better than the EDF-FF algorithm. The EDF-US and EDZL algorithms, which have their basis on global scheduling, have quite low schedulable utilisations that are around 50%. Like the RM-US algorithm, since the schedulability tests of those algorithms are not tight very much, the EDF-US and EDZL algorithms may reject many task sets which can be actually scheduled without missing any job deadlines. The EDZL algorithm is slightly worse than the EDF-US algorithm in terms of schedulable utilization, though their success ratios are reversed on the way. The authors claimed in [78] that the schedulability analysis of the EDZL algorithm has a room for improvement. Therefore, the potential schedulability of the EDZL algorithm may be better than the simulated results.

Figure 6.16, Figure 6.17, and Figure 6.18 show the success ratios for the dynamic-priority scheduling algorithms with respect to task sets in which the utilization of every individual task ranges within \([0.01, 1.0]\). The EDDP, EDDHP, and EKG-2 algorithms are still competitive. However, contrary to the previous cases in which only light tasks exists, the EDDP algorithm slightly outperforms the EDDHP algorithm as well as the EKG-2 algorithm in most cases. In general, the schedulability of an algorithm tends to
Figure 6.13: Success ratio \((M = 4, U_{\text{min}} = 0.01, U_{\text{max}} = 0.5)\).

Figure 6.14: Success ratio \((M = 8, U_{\text{min}} = 0.01, U_{\text{max}} = 0.5)\).
Figure 6.15: Success ratio \((M = 16, U_{\min} = 0.01, U_{\max} = 0.5)\).

Figure 6.16: Success ratio \((M = 4, U_{\min} = 0.01, U_{\max} = 1.0)\).
CHAPTER 6. PERFORMANCE EVALUATION

Figure 6.17: Success ratio ($M = 8, U_{\text{min}} = 0.01, U_{\text{max}} = 1.0$).

Figure 6.18: Success ratio ($M = 16, U_{\text{min}} = 0.01, U_{\text{max}} = 1.0$).
decrease under the presence of heavy tasks, since the condition of a system approaches the worst case. Therefore, the EDDP algorithm performs better than the EDDHP algorithm due to its higher worst-case utilization bound of 65%. Although the EKG-2 algorithm moreover has a higher utilization bound of 66%, the heuristic procedures introduced in Section 4.4.4 assist the performance of the EDDP algorithm, and hence the EDDP algorithm outperforms even the EKG-2 algorithm. While the above three algorithms are very competitive and maintain schedulable utilizations as high as those in the previous light task set cases, the EDF-BF algorithm declines a schedulable utilization down to 67% that is about 15% inferior to the ones in the previous light task set cases. The schedulable utilization for the EDF-FF algorithm is also declined down to around 60% that is about 15% inferior to the ones in the previous light task set cases. This inferiority of the EDF-BF and EDF-FF algorithms is involved by the presence of heavy tasks. Even if the total remaining utilization of all the processors is sufficient, a heavy task may fail to be assigned to any processor, since the acceptance of a task does not depend on the total remaining utilization but on the remaining utilization of every individual processor. As a result, the task assignment can return the failure in those algorithms even for a low system utilization. The EDZL algorithm, on the other hand, improves a schedulable utilization up to 60% that is about 15% higher than the previous light task set cases. According to Cirinei et al. [75], the schedulable utilization for the EDZL algorithm is dominated by the opportunity that the number of the tasks which can have the zero laxity at the same time becomes greater than the number of the processors. The case in which there are heavy tasks obviously leads the total number of the tasks to be smaller than the case in which there are only light tasks, which results in a less opportunity that the number of the tasks have can have the zero laxity at the same time is greater than the number of the processors. As a result, the schedulable utilization for the EDZL algorithm is improved compared to the previous light task set cases. The EDF-US algorithm still has a schedulable utilization of around 50%, because its utilization bound is anyway \( M/(2M - 1) \).

The Number of Task Preemptions

By the same token as Section 6.1.2, the interval of the simulations is set the smaller of the least common multiple of the task periods in the given task set and \(2^{32}\) for calculation of task preemptions. For each algorithm, the average number of task preemptions in the 1000 task sets is calculated every system utilization. Then, the number of task preemptions for each algorithm relative to that for the EDDP algorithm is calculated only in the case in which both the target algorithm and the EDDP algorithm have the schedulable utilization of 100%. Otherwise, the comparison becomes unfair.

Figure 6.19, Figure 6.20, and Figure 6.21 show the numbers of task preemptions for the dynamic-priority scheduling algorithms relative to the number of task preemptions for the EDDP algorithm, with respect to task sets in which the utilization of every individual task ranges within \([0.01, 0.5]\). The EDDP algorithm has slightly fewer task
CHAPTER 6. PERFORMANCE EVALUATION

Figure 6.19: The number of preemptions ($M = 4, U_{\text{min}} = 0.01, U_{\text{max}} = 0.5$).

Figure 6.20: The number of preemptions ($M = 8, U_{\text{min}} = 0.01, U_{\text{max}} = 0.5$).
CHAPTER 6. PERFORMANCE EVALUATION

Figure 6.21: The number of preemptions ($M = 16, U_{min} = 0.01, U_{max} = 0.5$).

Figure 6.22: The number of preemptions ($M = 4, U_{min} = 0.01, U_{max} = 1.0$).
Figure 6.23: The number of preemptions ($M = 8, U_{\text{min}} = 0.01, U_{\text{max}} = 1.0$).

Figure 6.24: The number of preemptions ($M = 16, U_{\text{min}} = 0.01, U_{\text{max}} = 1.0$).
preemptions than the EDDHP algorithm. Since the EDDP algorithm makes several processors which have only one heavy task and no light tasks, no preemptions occur on those processors. As a result, the number of task preemptions for the EDDP algorithm is totally reduced compared to the EDDHP algorithm. The EDF-FF and EDF-BF algorithms have around $0.65 \sim 0.85$ times (around $0.80$ times in most cases) as many task preemptions as the EDDP algorithm. The primary reason why the numbers of task preemptions for the EDF-FF and EDF-BF algorithms are smaller than those for the others is that those algorithms simply schedule the tasks according to the EDF algorithm on each processor without any interference from other processors, while the other algorithms have more or less interference among the processors. The EDF-US and EDZL algorithm have slightly more task preemptions, which is about $1.3 \sim 1.5$ times, than the EDDP algorithm. Since those algorithms have only one global scheduler which handles all the tasks, the ordering of the task priority is likely to be changed. A preemption occurs every time the ordering of the task priority is changed, and hence those algorithms cause more task preemptions than the other algorithms except for the EKG algorithms. While the EDDP, EDDHP, EDF-FF, EDF-BF, EDF-US, and EDZL algorithms are competitive, the EKG algorithms cause much more task preemptions. The EKG-2 algorithm causes around $2.0 \sim 2.5$ times as many task preemptions as the EDDP algorithm, though the schedulable utilization is less than or almost equal to the EDDP algorithm. The optimal EKG-M algorithm causes far too many task preemptions. The EKG algorithm preempts the first and second portions of a split task once for every interval of task releases on the related two processors in addition to the EDF scheduling, whereas the EDDP algorithm preempts only the second portion of a split task when its corresponding first portion is dispatched on the neighbor processor in addition to the EDF scheduling. In the case of $M = 16$, as shown in Figure 6.21, the number of task preemptions for the EKG-M algorithm finally reaches nearly $20$ times as many as that for the EDDP algorithm.

Figure 6.22, Figure 6.23, and Figure 6.24 show the numbers of task preemptions for the dynamic-priority scheduling algorithms relative to the number of task preemptions for the EDDP algorithm, with respect to task sets in which the utilization of every individual task ranges within $[0.01, 1.0]$. In this case, the EDDP algorithm has about $1.3 \sim 2.0$ times as many task preemptions as the EDF-FF and EDF-BF algorithms, while the EDDP algorithm is competitive with the EDF-FF and EDF-BF algorithms in the case of $(U_{\min}, U_{\max}) = (0.01, 0.5)$. According to the EDDP algorithm, the second portion of a split task is preempted when its corresponding first portion is dispatched on the neighbor processor, while such an extra preemption never occurs for the EDF-FF and EDF-BF algorithms. Since the execution time of each task tends to be long due to $U_{\max} = 1.0$, the scheduled timings of the first and second portions of a split task are easily overlapped. As a result, the EDDP algorithm causes more extra preemptions to schedule the first and the second portions of a split task exclusively. However, the EDDP algorithm suppresses the number of task preemptions to about $0.5 \sim 0.7$ times as many as the EKG-2 algorithm that takes a similar approach of splitting tasks.
CHAPTER 6. PERFORMANCE EVALUATION

The EKG-M algorithm causes at most about 20 times as many task preemptions as the EDDP algorithm as the previous cases in exchange for its optimality. Meanwhile, the EDF-US and EDZL algorithms cause about 1.3 ~ 1.5 times as many task preemptions as the EDDP algorithm, which is almost the same result as the previous cases.

6.2 Simulation Studies for Aperiodic Task Scheduling

This section studies the performance advancement of the global dispatch and temporal migration techniques for the aperiodic task scheduling on multiprocessor systems. The impact of the proposed techniques for static-priority scheduling in which periodic tasks are scheduled by the RM-FF algorithm and aperiodic tasks are handled by the PE algorithm is evaluated. Meanwhile, the impact for dynamic-priority scheduling in which periodic tasks are scheduled by the EDF-FF algorithm and aperiodic tasks are handled by the TBS algorithm is evaluated. For each evaluation, the global dispatch technique is first compared with the simple partitioned dispatch technique. Then, the effectiveness of the temporal migration technique is estimated for the case in which the global dispatch technique is applied. The purpose of this section is to study the effectiveness of the global dispatch and temporal migration techniques, so the conventional RM-FF and EDF-FF algorithms are used in the simulations for simplicity. As described in Section 5.4, the techniques are also valid for the RMDP, EDDHP, and EDDP algorithms established in this research, as well as the other traditional partitioned scheduling algorithms, thereby the performance advancement shown in this section is estimated to appear in those algorithms. More precise evaluation for the integrated scheduling is left for the future work.

The performance metric of this section is the mean response time of aperiodic tasks. Remember that this research is established under the assumption that the cost of task migrations is covered by that of task preemptions. Thus, the number of task migrations is not measured in the simulations and not discussed in this section.

6.2.1 Experimental Setup

The evaluation environment is setup as follows. The three types of multiprocessor systems are simulated. The first system has 4 processors, the second system has 8 processors, and the last system has 16 processors. Taking into account the scale of the current embedded platforms, the systems with more than 16 processors are not studied in the dissertation. For each system, a periodic task set is randomly generated so that the system utilization of the task set becomes $U_{sys}$, where $U_{sys}$ is set 0.5 for the simulations of static-priority scheduling and is set 0.7 for the simulations of dynamic-priority scheduling. Note that system utilizations higher than 0.5 and 0.7 are not schedulable by the RM-FF and EDF-FF algorithms. Meanwhile, the sophisticated techniques are not required in low system utilizations, since aperiodic tasks can be responded quickly.
The task set is generated as follows. A new periodic task is appended to the task set as long as the total utilization of the tasks is less than or equal to $U_{sys} \times M$. The parameters of each task are determined in the same manner as Section 6.1.1, namely they refer to the characteristics of applications implemented in a humanoid robot [92] that has been developed in my laboratory. No tasks are supposed to utilize more than 50% of the processor time. Thus, the utilization of each task is computed based on the uniform distribution within the range of [0.01, 0.5]. The utilization of the last task generated is only adjusted so that the total utilization is equal to $U_{sys} \times M$. Assuming both control tasks and multimedia tasks, the time units of the period of each task is randomly determined within the range of [100, 3000], simulating the range of [1ms, 30ms]. Then, the execution time calculated by $C_i = U_i T_i$.

The aperiodic workloads refer to the M/M/1 queuing model. The arrival times of aperiodic tasks are computed based on the Poisson distribution, and the execution times are computed based on the exponential distribution. In other words, the arrival times and the execution times are calculated based on the average service rate and the average arrival rate. In general, the response time of aperiodic tasks depends on the system load as well as the relation of the average service rate and the average arrival rate. Hence, two values of the average service rates, $\mu = 0.1$ and $\mu = 0.2$, are prepared in the simulations. The setup of $\mu = 0.1$ generates aperiodic tasks with the average execution time of 10 time units, simulating the average execution time of 100\(\mu\)s. The setup of $\mu = 0.2$, on the other hand, generates aperiodic tasks with the average execution time of 5 time units, simulating the average execution time of 50\(\mu\)s. Notice that, for the global dispatch policy, it does not matter which processors the aperiodic tasks arrive on, since the tasks are automatically dispatched to processors on which the response time can be reduced the most, whereas it matters for the simple partitioned dispatch policy. Thus, the arrival processors are determined randomly for fairness in the simulations of partitioned dispatch scheduling. The periodic load, denoted by $\lambda/\mu$, is varied across the range from the system utilization unused by the periodic tasks to the system utilization overloaded by 10%. In particular, the range is varied (i) from 0.3 to 0.6 for static-priority scheduling, and (ii) from 0.1 to 0.4 for dynamic-priority scheduling. Then, the mean response time of aperiodic tasks, denoted by the following expression, is measured for every aperiodic load.

\[
\text{Mean response time} = \frac{\text{the sum of the response time of all aperiodic tasks}}{\text{the number of aperiodic tasks}}
\]

For the simulations of static-priority scheduling, a periodic task set with a system utilization of 0.5 is scheduled by the RM-FF algorithm. Every processor has a server with a policy of the PE algorithm, whose period is equal to the shortest period in the given periodic task set so that the responsiveness of the server is enhanced as much as possible. The server utilization is set to the maximum value for which the periodic
tasks are guaranteed to be schedulable. That is, letting $U(\Pi_x)$ be the total utilization of a processor $P_x$ and $n_x$ be the number of tasks assigned to $P_x$, the utilization of a server performing on $P_x$ is calculated as follows.

$$U_{srv}^x = \frac{2}{\left(\frac{U(\Pi_x)}{n_x} + 1\right)^{n_x}} - 1$$

The server capacity is then calculated based on the period and the utilization. For the simulations of dynamic-priority scheduling, on the other hand, a periodic task set with a system utilization of 0.7 is scheduled by the EDF-FF algorithm. Every processor has a server with a policy of the TBS algorithm. The bandwidth of a server performing on $P_x$ is set to all the utilization left by the periodic tasks; that is, $U_{srv}^x = 1 - U(\Pi_x)$.

6.2.2 Simulation Results of Static-Priority Scheduling

Figure 6.25 and Figure 6.26 show the mean response times, with different numbers of the processors, for the PE algorithm with exploiting the global dispatch technique, denoted by PE-GD, relative to that of the PE algorithm without using the global dispatch technique but using the alternative simple partitioned dispatch policy.

For the case of the average service rate $\mu = 0.1$, the responsiveness of the system is dramatically improved for any aperiodic load regardless the number of the processors. The mean response time is maximally reduced by about 98% compared to the scheduling in which the global dispatch technique is not exploited, though the reduced amount of the mean response time in each aperiodic load is varied with respect to the number of the processors $M$. When the number of the processors is $M = 4$ and $M = 8$, the reduced amount of the mean response time takes a similar curve; that is, the mean response time relative to the simple PE algorithm is around $0.1 \sim 0.15$ in the aperiodic load of 0.3 and it is gradually reduced as the aperiodic load becomes higher. However, once the aperiodic load exceeds 0.5, the system falls into overload and the performance superiority to the simple PE algorithm begins to decrease as the aperiodic load becomes higher. When the system is overloaded, the processor time that can be allocated to the aperiodic tasks is more limited, and hence such a performance degradation is entailed. Notice that the mean response time is nonetheless reduced by around $60 \sim 70\%$ in the highest aperiodic load of 0.6. In contrast, the relative mean response time is maintained around 0.02, i.e. a reduction of 98%, even in high aperiodic loads when the number of the processors is $M = 16$. The conceivable reason is that the scheduler has more chance to find processors that reduce the response time of aperiodic tasks by global dispatches if the system offers more processors, whereas a few choices of dispatch destinations decrease the chance of reducing the response time.

For the case of the average service rate $\mu = 0.2$, even though the number of the processors is 8, the reduced amount of the mean response time is not decreased in high aperiodic loads, unlike the case of the average service rate $\mu = 0.1$. Furthermore, the mean response time is maintained about 3% of the scheduling in which the global
Figure 6.25: Mean response time for PE-GD ($\mu = 0.1$).

Figure 6.26: Mean response time for PE-GD ($\mu = 0.2$).
dispatch technique is not exploited, which is 7% smaller than the setup in which the number of the processors is \( M = 16 \). Since the setup of \( \mu = 0.2 \) generates aperiodic tasks with shorter execution times than the setup of \( \mu = 0.1 \), the response time of each aperiodic task can be reduced substantially by even a slight increase of the processor time available to the aperiodic task. However, such a decrease of the execution times also helps the simple partitioned dispatch technique to reduce the mean response time, and the reduced amount is obviously greater for more processors. Therefore, the performance improvement by the global dispatch technique is slightly decreased compared to the case of the average service rate \( \mu = 0.1 \) when the number of the processors is \( M = 16 \). Thus, having more processors does not necessarily achieve more performance improvement. Notice that the global dispatch technique still reduces the mean response time by about 90\% when the number of the processors is \( M = 16 \). By comparison with the case of the average service rate \( \mu = 0.1 \), the reduced amount of the mean response time is not differentiated very much when the number of the processors is \( M = 4 \).

In summary, the global dispatch technique is basically more effective for the system with more processors, however it is not necessarily true when the average execution time of aperiodic tasks is likely to be small. The mean response time is especially reduced a lot in middle aperiodic loads. When the system falls into overload, the reduced amount of the mean response time over the simple partitioned dispatch technique is gradually decreased when the number of the processors is relatively small. Nonetheless, the global dispatch technique offers a dramatic improvement of the mean response time, even if the system is overloaded.

Figure 6.27, Figure 6.28, and Figure 6.27 show the mean response times for the PE algorithm using the temporal migration technique in addition to the global dispatch technique, denoted by PE-GD-TM, relative to that of the PE-GD algorithm with respect to the average service rate of \( \mu = 0.1 \). Little performance difference is observed among the three heuristics applied to the temporal migration technique. Since the load of each processor is balanced by global dispatches, the choice of destination processors for temporal migrations is very limited. As a result, the three heuristics exert similar performances. Notice that the mean response time is already reduced dramatically by the global dispatch technique, hence the reduced amount of mean response time by the temporal migration technique is not that much. Nonetheless, the mean response time is reduced by 60\% compared to the scheduling in which only the global dispatch technique is applied. It is interesting that when the system is close to or in overload, the reduced amount begins to decrease drastically as the aperiodic load becomes higher when the number of the processors is \( M = 4 \) and \( M = 8 \), whereas that keeps to increase moderately when the number of the processors is \( M = 16 \). In the aperiodic load of 0.6, though no performance improvements are acquired by the temporal migration technique when the number of the processors is \( M = 4 \) and \( M = 8 \), the relative mean response time is reduced to 0.4 when the number of the processors is \( M = 16 \). In general, the more processors the system has, the more destinations for temporal migrations the scheduler tends to find. Thus, a performance improvement is observed even if the
CHAPTER 6. PERFORMANCE EVALUATION

Figure 6.27: Mean response time for PE-GD-TM \((M = 4, \mu = 0.1)\).

Figure 6.28: Mean response time for PE-GD-TM \((M = 8, \mu = 0.1)\).
Figure 6.29: Mean response time for PE-GD-TM \((M = 16, \mu = 0.1)\).

system is overloaded, when the number of the processors is \(M = 16\).

Figure 6.30, Figure 6.31, and Figure 6.30 show the mean response times for the PE algorithm using the temporal migration technique in addition to the global dispatch technique, denoted by PE-GD-TM, relative to that of the PE-GD algorithm with respect to the average service rate of \(\mu = 0.2\). The maximum amount of the reduced mean response time is slightly greater than the case of the average service rate \(\mu = 0.1\).
The mean response time is maximally reduced by around 65 \(\sim\) 70% compared to the scheduling in which only the global dispatch technique is exploited. When the number of the processors is \(M = 4\), the performance behavior is similar to the case of the average service rate \(\mu = 0.1\). On the other hand, the performance is improved, especially in high aperiodic loads, when the number of the processors is \(M = 8\). As described above, the setup of \(\mu = 0.2\) generates aperiodic tasks with shorter execution times, and the response time of each aperiodic task is more likely to be reduced compared to the setup of \(\mu = 0.1\) as the number of the processors is increased. While little performance difference is observed among the three heuristics for the above cases, the performance difference is appeared when the number of the processors is \(M = 16\). The Worst-Fit heuristic, which migrates a periodic task to the processor on which the periodic task is estimated to complete as late as possible before its deadline, usually performs the best. This means that migrating a periodic task to a processor with the most idle times is efficient for the temporal migration technique. Note that such a principle is not nec-
CHAPTER 6. PERFORMANCE EVALUATION

Figure 6.30: Mean response time for PE-GD-TM ($M = 4, \mu = 0.2$).

Figure 6.31: Mean response time for PE-GD-TM ($M = 8, \mu = 0.2$).
Figure 6.32: Mean response time for PE-GD-TM ($M = 16, \mu = 0.2$).

essarily established, since the First-Fit and Best-Fit heuristics outperform the Worst-Fit heuristic in other situations.

In summary, the temporal migration is basically more effective for the system with more processors and the set of aperiodic tasks with shorter execution times. Even if the mean response time is reduced by the global dispatch technique, it can be moreover reduced by the temporal migration technique. By combining the both techniques, the mean response time can be reduced by more than 99% compared to the scheduling in which neither of the techniques is applied. The comparison of the three heuristics is also worth to discuss. Across the board, little performance difference is observed among the three heuristics, though when the average service rate is $\mu = 0.2$ and the number of the processors is $M = 16$ in particular, the worst-fit heuristic performs the best. The first-fit heuristic and the best-fit heuristic are competitive in most cases. Consequently, a choice of the first-fit heuristic is the most reasonable, since it is the most straightforward and is mostly competitive with the other two heuristics.

6.2.3 Simulation Results of Dynamic-Priority Scheduling

Figure 6.33 and Figure 6.34 show the mean response times, with different numbers of the processors, for the TBS algorithm with exploiting the global dispatch technique, denoted by TBS-GD, relative to that of the TBS algorithm without using the global
Figure 6.33: Mean response time for TBS-GD ($\mu = 0.1$).

Figure 6.34: Mean response time for TBS-GD ($\mu = 0.2$).
dispatch technique but using the alternative simple partitioned dispatch policy.

For the cases of both the average service rates $\mu = 0.1$ and $\mu = 0.2$, the responsiveness of the system is amazingly enhanced, especially when the system is not overloaded. More specifically, in low aperiodic loads below 0.3, the mean response time is reduced to about 0.1% of the scheduling in which the global dispatch technique is not used. Even though the system is overloaded, the mean response time is reduced to around 0.1 $\sim$ 1% when the number of the processors is $M = 8$ or $M = 16$. When the number of the processors is $M = 4$, it is also reduced to about 8 $\sim$ 9% in the highest aperiodic load of 0.4. The reduced amount of the mean response time by the global dispatch technique is much greater than the case of the PE algorithm. Since the server is assigned the fixed highest priority on every processor in the PE algorithm, the aperiodic tasks can be executed at certain rate every time the server becomes active. In contrast, the server is assigned the virtual deadline which can be far late in the TBS algorithm, and hence the aperiodic tasks may be waited for a long time. Therefore, the potential amount of the response time that can be reduced is much greater in the TBS algorithm than the PE algorithm.

In summary, likewise the PE algorithm, the global dispatch technique prefers the system with more processors and the set of aperiodic tasks with shorter execution times in the TBS algorithm. A substantial performance improvement is observed even if the system is overloaded. The priority of aperiodic tasks is determined based on the virtual deadline assigned by the TBS algorithm, hence the aperiodic tasks can be never executed in advance to the periodic tasks with earlier deadlines. However, using the global dispatch technique, the aperiodic tasks are dispatched to the processors so that the aperiodic tasks can have as early virtual deadlines as possible. Therefore, the responsiveness to the aperiodic tasks can be dramatically improved by the global dispatch technique in the TBS algorithm. On the whole, the impact of the global dispatch technique is greater for the dynamic-priority scheduling than the static-priority scheduling.

Figure 6.35, Figure 6.36, Figure 6.37, Figure 6.38, Figure 6.39, and Figure 6.37 depict the mean response times for the TBS algorithm using the temporal migration technique in addition to the global dispatch technique, denoted by TBS-GD-TM, relative to that of the TBS-GD algorithm, with respect to the average service rates of $\mu = 0.1$ and $\mu = 0.2$.

In contrast to the PE algorithm, no performance improvement is obtained by the temporal migration technique. The mean response time is even worse than the scheduling in which only the global dispatch technique is taken as shown by the Worst-Fit heuristic in the aperiodic load of 0.27 of Figure 6.37. In addition, the performance difference among the three heuristics is very little. The conceivable reason is that, as shown in Figure 6.33 and Figure 6.34, the mean response time is ultimately reduced by the global dispatch technique already, and hence there is very little room left for responsiveness improvement. Therefore, the performance is seldom improved by the temporal migration technique. The results also indicate that, despite little room for improvement, abusing temporal migrations may decline the responsiveness.
Figure 6.35: Mean response time for TBS-GD-TM \((M = 4, \mu = 0.1)\).

Figure 6.36: Mean response time for TBS-GD-TM \((M = 8, \mu = 0.1)\).
Figure 6.37: Mean response time for TBS-GD-TM ($M = 16, \mu = 0.1$).

Figure 6.38: Mean response time for TBS-GD-TM ($M = 4, \mu = 0.2$).
CHAPTER 6. PERFORMANCE EVALUATION

Figure 6.39: Mean response time for TBS-GD-TM ($M = 8, \mu = 0.2$).

Figure 6.40: Mean response time for TBS-GD-TM ($M = 16, \mu = 0.2$).
CHAPTER 6. PERFORMANCE EVALUATION

Figure 6.41: Mean response time for TBS-TM ($M = 4, \mu = 0.1$).

Figure 6.42: Mean response time for TBS-TM ($M = 8, \mu = 0.1$).
CHAPTER 6. PERFORMANCE EVALUATION

Figure 6.43: Mean response time for TBS-TM ($M = 16, \mu = 0.1$).

Figure 6.44: Mean response time for TBS-TM ($M = 4, \mu = 0.2$).
Figure 6.45: Mean response time for TBS-TM ($M = 8, \mu = 0.2$).

Figure 6.46: Mean response time for TBS-TM ($M = 16, \mu = 0.2$).
In order to show the effectiveness of the temporal migration technique for the TBS algorithm, the mean response times for the TBS algorithm with using only the temporal migration technique, denoted by TBS-TM, relative to that of the pure TBS algorithm, with respect to the average service rates of \( \mu = 0.1 \) and \( \mu = 0.2 \) in Figure 6.41, Figure 6.42, Figure 6.43, Figure 6.44, Figure 6.45, and Figure 6.46.

Using the temporal migration technique, the mean response time is reduced dramatically, especially when the number of the processors is \( M = 4 \) and \( M = 16 \). In this case, the relative mean response time is maximally reduced to 0.02 in low aperiodic loads. Even though the system is overloaded, it is reduced to around 0.1 \( \sim \) 0.2. It is a remarkable result that the reduce amount of the mean response time when the number of the processors is \( M = 8 \) is small. Likewise the previous results, little performance difference is observed among the three heuristics. As a result, the First-Fit is estimated to be the most reasonable, since it is well performed as well as straightforward. Remember that the performance advancement of the temporal migration technique is assimilated by that of the global dispatch technique in dynamic-priority scheduling as already described.

### 6.3 Summary

This chapter presented the results of simulation studies. For periodic scheduling algorithms, the success ratios and the number of task preemptions with respect to randomly-generated 1000 task sets in different system loads were measured. Meanwhile for aperiodic server algorithms, the mean response times of Poisson-modeled aperiodic task sets in different loads were measured.

The RMDP, EDDHP, and EDDP scheduling algorithms developed in this research provided high success ratios with small numbers of task preemptions. By comparison, most of the traditional algorithms showed lower success ratios. In the traditional static-priority scheduling algorithms, only the WM algorithm showed higher success ratios than the RMDP algorithm. However, the WM algorithm generated a huge number of task preemptions, that is, it is not a choice for practical use. In contrast, the RM-FFDU algorithm offered a smaller number of task preemptions than the RMDP algorithm. Unfortunately, the RM-FFDU occasionally failed to schedule task sets with system utilizations above 60%, while the RMDP algorithm never failed to schedule task sets with system utilizations below 70%. Meanwhile in the traditional dynamic-priority scheduling algorithms apart from the optimal ones, only the EKG algorithm with a parameter of \( k = 2 \), denoted by EKG-2, was competitive with the EDDHP and EDDP algorithms in terms of success ratio. However, the EKG-2 algorithm constantly generated more task preemptions than the EDDHP and EDDP algorithms. The results indicate the advancement of the developed algorithms. The EDDHP and EDDP algorithms were also competitive in the simulations. However, the EDDP algorithm has a higher utilization bound of 65%. The optimal algorithms, represented by the EKG-M algorithm in the
simulations, of course outperform the developed algorithms in terms of success ratio, since they always achieve the utilization bound of 100%, but the implementation costs and the numbers of task preemptions occurred in those algorithms are not acceptable for practical use. Therefore, the developed scheduling algorithms are very effective for practical use, since they are well implementable and offer high schedulable utilizations with small numbers of task preemptions.

The PE-based algorithm and the TBS-based algorithm in which the global dispatch technique and the temporal migration techniques are exploited reduced the mean response times of aperiodic tasks dramatically, compared to the pure PE and TBS algorithms. For both the PE-based algorithm and the TBS-based algorithm, the responsiveness improvements to aperiodic tasks were boosted up as the number of the processors increased. Especially, the impact of the global dispatch technique for the TBS-based algorithm was remarkable. The effectiveness of the temporal migration technique was in contrast more greater for the PE-algorithm. As for the traded overhead, task migrations occur at most once for every aperiodic task arrival, thereby the dissertation believes that the performance improvement of the temporal migration technique is definitely worth exchanging for that overhead. Little performance difference was observed among the FF heuristic, the BF heuristic, and the WF heuristic in temporal migrations.
Chapter 7

CONCLUSION

This dissertation presented research on the real-time scheduling of periodic and aperiodic tasks on multiprocessor systems. The developed scheduling techniques take advantage of task migrations to accomplish high schedulability and high responsiveness with elementary computations. The research explored both the scheme of static-priority scheduling and the scheme of dynamic-priority scheduling. Three scheduling algorithms were developed for increasing the system utilization of periodic tasks with guaranteeing timing constraints as well as restraining the occurrences of task preemptions. Two server algorithms were also designed for reducing the mean response time of aperiodic tasks. The thesis supported by this dissertation is that the migrative scheduling improves the schedulability to periodic tasks as well as the responsiveness to aperiodic tasks, without causing complex computations and a lot of task preemptions.

7.1 Summary of Contributions

This research made contributions of theoretical significance to real-time computing upon multiprocessor systems. The contributions lead to improvements of the periodic schedulability as well as the aperiodic responsiveness and advance the development of high-performance real-time systems for a part of the intelligent social infrastructure.

The first contribution is made in the proposal of the portioned scheduling technique for increasing the total utilization of multiprocessor systems with guaranteeing timing constraints of periodic tasks. The portioned scheduling technique takes advantage of both the traditional global scheduling and partitioned scheduling techniques. The basic strategy of the portioned scheduling is that, like the partitioned scheduling, most of tasks are scheduled on dedicated processors and never migrate to different processors, while special tasks are permitted to migrate between two restrictive processors like the global scheduling. Such a scheduling policy draws the advancement that the schedulable system utilization is increased by exploiting a small number of task migrations and at the same time the scheduling overhead is reduced by restricting the number of...
tasks and the range of processors related to the migrations. The dissertation therefore advocates that the portioned scheduling technique is capable of breaking through the trade-off between schedulability and complexity.

The major contribution of this research is the development of three scheduling algorithms designed based on the portioned scheduling technique. The scheduling algorithms are developed by taking into account the two types of the priority assignment scheme: the static-priority assignment and the dynamic-priority assignment. For the discipline of static-priority scheduling, in which the priority order of tasks is never changed, the RMDP algorithm is developed. In the domain of uniprocessor systems, the RM algorithm is well known to be optimal for static-priority scheduling. Thus, the RMDP algorithm has its basis on the RM algorithm. On every processor, the scheduling policy of the RMDP algorithm is distinguished from the RM algorithm when special tasks that are split into two processors by portioning are scheduled. While the highest-priority task is never deferred in the RM algorithm, its partial portion, i.e. the second portion of the split task, can be deferred so that two portions of the same split task are scheduled exclusively. Such a deferrable scheduling policy is expected to improve the schedulability to periodic tasks, though the worst-case utilization bound for the RMDP algorithm is proved to be 50%, which is equal to the prior known bound. In a sense of the worst case, no performance improvement is obtained by the RMDP algorithm compared to traditional static-priority scheduling algorithms. However, the simulation results show that the performance of the RMDP algorithm in general cases is often better than the traditional algorithms. The EDDHP and EDDP algorithms, on the other hand, are invented to perform for the discipline of dynamic-priority scheduling. In the EDDHP algorithm, the second portion of the split task is statically assigned the highest priority and is deferrable like the RMDP algorithm, then the rest of the tasks is scheduled according to the EDF algorithm, which is well known to be optimal for dynamic-priority scheduling in the domain of uniprocessor systems. Unfortunately, the worst-case utilization bound of every processor for the EDDHP algorithm is no greater than 50% due to restriction of the static-priority assignment to the second portion of the split task. The EDDP algorithm is an extension of the EDDHP algorithm in that all the tasks, including the second portion of the split task, are assigned the priorities dynamically based on the deadlines. If the priority of the second portion of the split task is assigned based on its true deadline, the worst-case schedulability is far declined, and that is a reason why the EDDHP algorithm statically assigns the highest priority to it. In the EDDP algorithm, the schedulability is maintained by assigning the virtual deadline, which is set earlier than the true deadline, to the second portion of the split task. Such a deadline assignment achieves a worst-case utilization bound of 65%. To the best of my knowledge, no traditional algorithms, except for complex optimal ones, have ever achieved a worst-case utilization bound greater than 65%. The optimal algorithms literally achieve an optimal utilization of 100% in any case, however they raise a lot of task preemptions and a run-time overhead. Although the EKG algorithm is able to trade the worst-case utilization bound with the number of task preemptions, the sim-
ulation results showed that the EKG algorithm with the utilization bound of 66% still generates more task preemptions than the EDDP algorithm. The general performance of the developed algorithms also outperform the traditional algorithms in most cases according to the simulation results.

Another contribution of this research is made in the proposals of the global dispatch technique and the temporal migration technique for reducing the response time of aperiodic tasks. For the domain of multiprocessor systems, very few scheduling techniques for improving the responsiveness to aperiodic tasks have been discussed and considered. To the best of my knowledge, this research is the first contribution for improving the responsiveness to aperiodic tasks on multiprocessor systems. The presented techniques target the partitioned scheduling strategy for simplicity, but they are compatible to the portioned scheduling strategy. The global dispatch technique improves the responsiveness by dispatching an arriving aperiodic task to a processor on which the response time of the aperiodic task is estimated to be reduced the most. The temporal migration technique, on the other hand, improves the responsiveness in such a way that if processor time cannot be allocated to an arriving aperiodic task immediately due to executions of periodic tasks, periodic tasks with higher priorities than the aperiodic task are temporarily migrated to different processors to devolve the processor time, as long as the timing constraints of the periodic tasks are still guaranteed. The behavior of the global dispatch and temporal migration procedures is similar to load balancing. Since the global dispatch technique assigns every aperiodic task to an effective processor when it arrives, which is not a migration, the overhead is expect to be vanishingly low. The temporal migration technique, on the other hand, generates additional migrations, however they occur only when aperiodic tasks arrive and cannot be served immediately. Thus, the expectation is that the incurred overhead is not a crisis.

For the disciplines of static-priority scheduling and dynamic-priority scheduling, the designs of the PE and TBS algorithms are respectively considered to integrate the global dispatch and temporal migration techniques. In the PE algorithm, an arriving aperiodic task is dispatched to a processor on which its worst-case response time is minimized. Then, temporal migrations occur in such a way that when an arriving aperiodic task is not served by a server immediately, a periodic task with the highest priority is migrated to a different processor as long as its latest finish time served by the PE algorithm on the new processor is earlier than or at its deadline. The remaining execution time of the migrated periodic task is accumulated as the server capacity at its priority level. On the other hand, in the TBS algorithm, an arriving aperiodic task is dispatched to a processor on which its assigned virtual deadline is minimized. Then, temporal migrations occur in such a way that when an arriving aperiodic task is not served by a server immediately, a periodic task with the earliest deadline is migrated to a different processor as long as its virtual deadline assigned by the TBS algorithm on the new processor is earlier than or at its deadline. Since the server bandwidth can be broaden temporarily due to the absence of the migrated periodic task, a new virtual deadline, which is obviously earlier than the original one, is assigned to the arriving aperiodic
task. According to the simulation results, the mean response time of aperiodic tasks is dramatically reduced by the global dispatch and temporal migration techniques with respect to both the PE and TBS algorithms. Especially, the global dispatch technique is more effective for the TBS algorithm, though the impact of the temporal migration technique is decreased instead, compared to the PE algorithm.

7.2 Future Directions

In the end, the dissertation gives several insights to the future work. It is worth considering an extension of the portioned scheduling technique so as to have applicability to more realistic system model. In this research, periodic tasks are assumed preemptive. However, non-preemptive periodic tasks are often submitted in real-time systems. For instance, motor control tasks in robot systems may use I/O functions which cannot be preempted in general. If the periodic tasks are partially not preemptive, which means that they are also non migrative, the scheduling is not precisely produced according to the theory of the portioned scheduling. It is also worth considering a problem of periodic jitters. In the portioned scheduling, the second portion of a split task can be deferred. However, such a deferrable approach generates periodic jitters which may degrade the accuracy of control. Thus, a theoretical analysis of the portioned scheduling in which some tasks cannot be preempted or cannot be deferred is needed. If the portioned scheduling technique can be extended to be even available for those types of tasks, the value of the technique is boosted.

There is also room for improvement in the presented scheduling model. First, a migration manner of portioned (split) tasks can be moreover improved. As pointed out in Chapter 4, inessential task preemptions and migrations occur for the presented algorithms. A kind of a look-ahead approach is wanted to prevent the algorithm from generating redundant task preemptions and migrations.

The portioned scheduling technique presented in this dissertation focused on the idea that the second portion of a portioned task is deferrable. The resulting utilization bound was 65%, which is actually a significant contribution, because no traditional efficient algorithms have ever transcended a utilization bound of 50%. In addition, the presented approaches are well practical. Hence, the discussion of this idea is closed. However, it is worth considering an idea that the first portion of a portioned task is deferrable. The expectation is that it can reduce the number of task preemptions, since the second portion is always assigned the shortest period or the earliest relative deadline, which means that it is usually a high-priority task, and is not often preempted, that is, the deferred first portion is not unnecessarily preempted. A specific scheduling algorithm and its schedulability analysis are desired.

The global dispatch technique and the temporal migration technique also have room for improvement. First of all, they also exploit task migrations, and hence the extended scheduling model in which some tasks are not preemptive and migrative is required to
be applicable to various types of tasks. The dissertation believes that the global dispatch technique itself is very efficient and there is technically little room for improvement. In contrast, the temporal migration technique is arguable. As stated in Chapter 5, the policy of processor selections for temporal migrations must be discussed, since it dominates the responsiveness to aperiodic tasks very much. The timing of a temporal migration occurrence is also controversial. In the presented approaches, a temporal migration occurs for an arrival of aperiodic tasks. The response time may be reduced more if a temporal migration can also occur for another event. However, an increase of temporal migration results in larger scheduling overhead. Thus, the trade-off must be taken into account.

One of the largest future work is to integrate the portioned scheduling technique with the global dispatch technique and the temporal migration technique. The impact of the integrated scheduling is far promising to next generation real-time systems. As discussed in Section 5.4, it is actually possible to integrate those scheduling techniques, thereby the theoretical analysis of the integrated scheduling is planned as well as the design and implementation.

The final insight to the future work is the development of an operating systems in which the scheduling techniques established in this research are implemented. The total costs of scheduling techniques are then able to be assessed. Since the theoretical superiority of the established techniques is demonstrated by the simulation studies, it is next required to report the practical superiority of the established techniques. To achieve this, the development of an embedded real-time and multicore-oriented operating system in which the established techniques are implemented is on going.
Bibliography


LIST OF PAPERS

Articles on Periodicals


Articles on International Conference Proceedings


