The Design and Analysis of Scheduling Algorithms for Real-Time and Fault-Tolerant Computer Systems

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Yingfeng Oh

欧英锋

May 1994

APPROVAL SHEET

This dissertation is submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy (Computer Science)

Yingfeng Oh 欧英锋

This dissertation has been read and approved by the Examining Committee :

Dissertation Advisor: Sang H. Son

Committee Chairman: Jim Cohoon

Committee Member: Barry W. Johnson

Committee Member: Paul F. Reynolds, Jr.

Committee Member: Jorg Liebeherr

Accepted for the School of Engineering and Applied Science :

Dean, School of Engineering and
Applied Science

May 1994

In memory of my father,

who taught me

少壮不努力，老大徒伤悲。

(If youth only knew! If age only could!)

Acknowledgements

Without the encouragement and support of many people, this thesis would never have been written. These people have contributed to the completion of this thesis and the fulfillment of my education here in their unique ways.

Special thanks go to my advisor Sang H. Son, for his support and guidance during the last three years. When I was going nowhere, it was his patience and encouragement that brought me back to the track. I am indebted to him in many other ways.

I wish to thank the other members of my committee for their advice and wisdom. I thank Jim Cohoon for his willingness to take over reading the thesis at the last moment and providing valuable advice to improve the quality of the thesis afterwards. Jorg Liebeherr, along with Almut Burchard, contributed many wonderful ideas that led to the work presented in Section 2.3 and Section 3.6. Ever since I took his course in fault-tolerance computing, Barry W. Johnson has been an inspiration to me: to be a better person and a better professional. I am surprised at the care Paul F. Reynolds, Jr. took in reading my thesis and the soundness of his advice on scheduling a meeting.

I would like to thank my former advisor Robert P. Cook for his support in my first two years here and for his much valuable advice. I would also like to thank all the professors with whom I have interacted, particularly Anita Jones, Gabriel Robins, Jack Davidson, Bill Wulf, Jim Ortega, Worthy Martin, Jim French, Jeff Salowe, and John Knight.

I thank David Warme, Juhnyoung Lee, Chris Koeritz, Dongwei Liao, and TongTong Zhang for their friendship, and Mike Alexander, Kevin Wika, Lifeng Hsu, Weifeng Zheng, Craig Williams, Youngkuk Kim, Ambar Sarkar, Carmen Pancerella, Mark Bailey, Sally McKee, Tim Strayer, Rich Gossweiler, and Chiang Shi-Ching for their help in various ways.

I am grateful for much help provided by Francis X. Mooney and Ginny Hilton.

The families of Wang Zhao Qing, Long Dou and Sun Eve, Mary and Bob Vickery, Steve and Evelyn Braintwain deserve special thanks for their love, encouragement, and many free meals.

Finally, but not the least, I would like to thank my mother, my brother, and the rest of my family for their unyielding love and unwaving support throughout all these years.

“Where there is a will, there is a way.”
-- Anonymous

“So every defect of the mind may have a special receipt.”
-- Francis Bacon, 1561-1626

Abstract

Many applications are not feasible without the support of real-time and fault-tolerant computer systems. Timeliness and dependability are properties that predominantly distinguish a real-time system and a fault-tolerant system from other computer systems. In this thesis we address the issue of supporting timeliness and dependability by studying four fundamental scheduling problems that are inherent to these systems.

The real-time systems we consider are the ones in which tasks are executed periodically; each task has an infinite number of requests and there are multiple tasks being executed. There arises a problem of scheduling all requests of the tasks on as few processors as possible such that not a single deadline is missed. This problem has optimal solutions for a single processor system when tasks are preemptive; the Rate-Monotonic (RM) algorithm is optimal for fixed-priority assignment and the Earliest Deadline First (EDF) is optimal for dynamic priority assignment. However, the problem is intractable for a multiprocessor system. The main goal of this thesis is to design and analyze algorithms for the problem of scheduling a set of periodic, preemptive tasks on as few processors as possible such that task deadlines are met on each processor by the RM algorithm.

We give the best on-line and off-line scheduling algorithms for this problem to date with regard to worst case performance and average case performance. The worst case performance of the algorithms is shown to have constant tight bounds through complex analysis and the average case performance is assessed by conducting simulation experiments. Several new schedulability conditions are also obtained for the RM scheduling.

The second problem is defined the same way as the first one except that instead of one version, a task has a number of versions that must be executed on different processors for fault-tolerance purposes. We propose a solution to this problem with its worst case performance analyzed and average case performance simulated.

The third problem is defined the same way as the second one except that instead of the RM algorithm, the EDF algorithm is used to guarantee task deadlines on each processor. This problem is equivalent to the problem of packing a list of colorful items into as few bins as possible without violating the constraint that no two items having the same colors are packed into a bin. An algorithm is proposed to solve this problem, with its worst case performance shown to be tightly bounded by 1.7.

Finally we address the fundamental question of scheduling non-preemptive tasks on a multiprocessor system for tolerance of processor failures or task errors. We show that this problem is intractable even for three processors with the tolerance of one arbitrary processor failure. Two heuristic algorithms are then proposed to solve a restricted case of the problem.

By solving these problems, the thesis contributes to the establishment of a firm theoretical foundation for guaranteeing task deadlines in real-time and fault-tolerant computer systems.

Table of Contents

1 Introduction 1

1.1. Overview 1
1.2. Motivations and Objectives 3
1.3. Assumptions and Problem Statements 7
1.4. Related Work 13
1.5. Approaches Taken 18
1.6. Organization 22

2 Rate-Monotonic Scheduling on a Single Processor System 24

2.1. Task Model 24
2.2. Some Important Lemmas 28
2.3. Period-Oriented Schedulability Conditions 32
2.4. Utilization-Oriented Schedulability Conditions 36
2.5. Miscellaneous Schedulability Conditions 44

3 Rate-Monotonic Scheduling on a Multiprocessor System 47

3.1. Fundamental Results of RM Scheduling on Multiprocessor 47
3.2. Scheduling Heuristics and Their Worst Case Performance Analysis 51
3.3. Rate-Monotonic-First-Fit 55
3.4. Rate-Monotonic-Best-Fit 63
3.5. The Refinements of RM-FF and RM-BF 70
3.6. Period-Oriented Heuristic Algorithms 74
3.7. Rate-Monotonic-First-Fit-Decreasing-Utilization 85
3.8. Heuristic Algorithms Using the Necessary and Sufficient Condition 105
3.9. Average Case Performance Evaluation 110
3.9.1. Performance Comparison of New and Existing On-line Algorithms 110
3.9.2. Performance Comparison of New and Existing Off-line Algorithms 114
3.9.3. Yet Another Performance Evaluation of the Algorithms 116
3.10. Summary 122

4 Supporting Fault-Tolerance in Rate-Monotonic Scheduling 123

4.1. Task Model 125
4.2. The Design and Analysis of FT-Rate-Monotonic-First-Fit 127
4.3. Average Case Performance Evaluation 135

5 Supporting Fault-Tolerance in Earliest-Deadline-First Scheduling 138

5.1. The Design and Analysis of FT-EDF-First-Fit 139
5.2. The Average Case Performance Evaluation 149

6 Non-preemptive Scheduling of Periodic Tasks for Fault-Tolerance 156

6.1. Non-overlapping of Backup Copies 158
6.1.1. Complexity of the Scheduling Problem 158
6.1.2. A 1-Timely-Fault-Tolerant Scheduling Algorithm 160
6.1.3. Analysis and Performance Evaluation 166
6.2. Overlapping of Backup Copies 171
6.2.1. Complexity of the Scheduling Problem 171
6.2.2. A 1-Timely-Fault-Tolerant Scheduling Algorithm 174

6.2.3. Analysis and Simulation Results 178

7 Conclusion 185

7.1. Contributions 185

7.2. Future Work 188

List of Figures

Figure 1.1 Problem Structure 12

Figure 2.1 Schedule for Two Tasks........ 26

Figure 2.2 The $f(n, \gamma)$ Function........ 34

Figure 2.3 Relationship between T_1 and T_2. 45

Figure 3.1 Algorithm RM-FF-WC 52

Figure 3.2 RM-NF-WC vs. Optimal Schedules 55

Figure 3.3 Algorithm RM-FF........ 56

Figure 3.4 RM-FF vs. Optimal Schedules........ 61

Figure 3.5 Algorithm RRM-FF 72

Figure 3.6 Algorithm RMST........ 76

Figure 3.7 Algorithm RMGT 79

Figure 3.8 Algorithm RMGT-M 83

Figure 3.9 Algorithm RM-FFDU 86

Figure 3.10 Algorithm RM-FF-IFF........ 106

Figure 3.11 Algorithm RM-FFDU-IFF........ 108

Figure 3.12 Performance of Some On-line Algorithms ($\alpha = 0.2$)........ 111

Figure 3.13 Performance of Some On-line Algorithms ($\alpha = 0.5$)........ 112

Figure 3.14 Performance of Some On-line Algorithms ($\alpha = 0.7$)........ 112

Figure 3.15 Performance of Some On-line Algorithms ($\alpha = 1.0$)........ 113

Figure 3.16 Performance of RM-FF, RM-BF, RRM-FF, and RRM-BF ($\alpha = 0.3$) ... 114

Figure 3.17 Performance of RM-FF, RM-BF, RRM-FF, and RRM-BF ($\alpha = 0.7$) ... 115

Figure 3.18 Performance of RM-FF, RM-BF, RRM-FF, and RRM-BF ($\alpha = 1.0$) ... 115

Figure 3.19 Performance of Off-line Algorithms ($\alpha = 0.2$) 117

Figure 3.20 Performance of Off-line Algorithms ($\alpha = 0.5$) 117

Figure 3.21 Performance of Off-line Algorithms ($\alpha = 0.7$) 118

Figure 3.22 Performance of Off-line Algorithms ($\alpha = 1.0$) 118

Figure 3.23 Performance of Some Algorithms (Tasks/processor = 3) 120
Figure 3.24 Performance of Some Algorithms (Tasks/processor = 6) 120
Figure 3.25 Performance of Some Algorithms (Tasks/processor = 3) 121
Figure 3.26 Performance of Some Algorithms (Tasks/processor = 6) 122
Figure 4.1 Algorithm 0 127
Figure 4.2 Algorithm 1 128
Figure 4.3 Algorithm FT-RM-FF 130
Figure 4.4 Task Configuration when $\kappa < m \leq 2\kappa$ 131
Figure 4.5 Task Configuration when $2\kappa < m$ 132
Figure 4.6 Performance of FT-RM-FF and Algorithm 1 ($\alpha = 1.0$) 136
Figure 4.7 Performance of FT-RM-FF and Algorithm 1 ($\alpha = 0.5$) 137
Figure 5.1 Weighting Function $W(\alpha)$ 141
Figure 5.2 Worst Case Configuration of Zero Coarseness 144
Figure 5.3 Performance Comparison of the Four Algorithm ($\alpha = 1.0$) 150
Figure 5.4 Performance Comparison of the Four Algorithm ($\alpha = 0.5$) 151
Figure 5.5 Performance Comparison of the Four Algorithm ($\alpha = 1.0$) 152
Figure 5.6 Performance of RM-Algorithm 1 154
Figure 5.7 Performance of FT-RM-FF with Sorted Input 155
Figure 5.8 Performance of FT-EDF-FF with Sorted Input 155
Figure 6.1 Mapping from PARTITION to Task Sequencing 160
Figure 6.2 Mapping from Task Sequencing to PARTITION 161
Figure 6.3 Algorithm NOV 162
Figure 6.4 Scheduling Process of NOV 162
Figure 6.5 Schedule Generated by LPT 163
Figure 6.6 Schedule Generated after Swapping and Appending 163
Figure 6.7 Algorithm NOV-Test 166
Figure 6.8 Performance of NOV 168
Figure 6.9 Relationship Between Performance Parameters 169

Figure 6.10 A Schedule Generated by LPT and NOV_1 170
Figure 6.11 Mapping from PARTITION to Task Sequencing................................. 173
Figure 6.12 Mapping from Task Sequencing to PARTITION................................. 174
Figure 6.13 Mapping from Task Sequencing to PARTITION................................. 174
Figure 6.14 Algorithm OV 176
Figure 6.15 Schedule created by LPT 177
Figure 6.16 Scheduling Process of OV 177
Figure 6.17 Relationship Between Schedules 181
Figure 6.18 Performance of OV 184

List of Tables

Table 1.1 Worst Case Performance of Existing Scheduling Algorithms........... 16

Table 3.1 Worst Case Performance Bounds of RM-FF under α 62

Table 3.2 Worst Case Performance Bounds of RRM-FF under α..................... 71

Table 3.3 Performance of RM-FFDU for some values of *x*............................... 87

Table 3.4 Weighting Function for $x \in (1/5, 1/4]$ 90

Table 3.5 Weighting Function for $x \in (2^{1/4} - 1, 1/5]$ 93

Table 3.6 Weighting Function for $x \in (1/6, 2^{1/4} - 1]$ 95

Table 3.7 Weighting Function for $x \in (5(2^{1/5} - 1) - 3/5, 1/6]$ 98

Table 7.1 Performance Bounds of New Algorithms for RMMS 186

List of Theorems

Theorem 2.1 26
Theorem 2.2 27
Theorem 2.3 28
Theorem 2.4 32
Theorem 2.5 34
Theorem 2.6 35
Theorem 2.7 36
Theorem 2.8 42
Theorem 2.9 44
Theorem 2.10 46
Theorem 3.1 47
Theorem 3.2 48
Theorem 3.3 50
Theorem 3.4 53
Theorem 3.5 53
Theorem 3.6 58
Theorem 3.7 60
Theorem 3.8 61
Theorem 3.9 69
Theorem 3.10 70
Theorem 3.11 71
Theorem 3.12 74
Theorem 3.13 76
Theorem 3.14 77
Theorem 3.15 79
Theorem 3.16 83

Theorem 3.17 85
Theorem 3.18 104
Theorem 3.19 106
Theorem 3.20 107
Theorem 3.21 109
Theorem 3.22 109
Theorem 4.1 128
Theorem 4.2 133
Theorem 5.1 140
Theorem 6.1 158
Theorem 6.2 164
Theorem 6.3 169
Theorem 6.4 170
Theorem 6.5 171
Theorem 6.6 177
Theorem 6.7 182

List of Lemmas

Lemma 2.1 28
Lemma 2.2 29
Lemma 2.3 30
Lemma 3.1 56
Lemma 3.2 57
Lemma 3.3 58
Lemma 3.4 59
Lemma 3.5 62
Lemma 3.6 66
Lemma 3.7 66
Lemma 3.8 66
Lemma 3.9 86
Lemma 3.10 87
Lemma 3.11 88
Lemma 3.12 89
Lemma 3.13 90
Lemma 3.14 92
Lemma 3.15 94
Lemma 3.16 97
Lemma 4.1 130
Lemma 4.2 131
Lemma 4.3 134
Lemma 4.4 134
Lemma 5.1 140
Lemma 5.2 141
Lemma 5.3 142

Lemma 5.4 142

Lemma 5.5 143

Lemma 5.6 145

Lemma 5.7 146

Lemma 5.8 147

Lemma 5.9 148

Lemma 6.1 157

Lemma 6.2 157

Lemma 6.3 169

Lemma 6.4 180

Lemma 6.5 181

List of Symbols

α:	The maximum allowable utilization of a task. $\alpha = max_i (C_i / T_i)$.
β:	The difference between two V values of tasks.
γ:	The maximum ratio between any two task periods.
δ:	A small positve number.
Δ:	A time interval.
ε:	Another small positve number.
Z^+ :	The set of natural numbers.
κ:	The maximum redundancy degree of a task, i.e., the maximum number of versions of a task. $\kappa = max_{\{1 \le i \le n\}} \kappa_i$.
κ_i :	The redundancy degree of task τ_i .
Σ :	A set of tasks.
$\sigma(t)$:	The starting time of task t (in a schedule).
τ_i :	The *i*th task. Or the computation time of the *i*th task (in Chapter 6 only).
$\tau_{j,i}$:	The *i*th task assigned on processor P_j .
ϖ:	Total weight of a task set. $\varpi = \sum_{i=1}^{m} W(u_i)$.
B_i :	The *i*th bin.
b_i :	The *i*th item.
C_i :	The computation time of task τ_i .
D:	A time period.
$d(t)$:	The deadline of task t.
$l(t)$:	The length (or computation time) of task t.
N:	The number of processors required to schedule a task set. $N = \sum_{i=1}^{\infty} n_i$.
N_0 :	The minimum number of processors required to schedule a task set.
N_A :	The number of processors required to schedule a task set by a given heuristic A.
n:	The number of tasks in a set or assigned to a processor.

n_i:	The number of processors to each of which exactly $i \geq 1$ tasks are assigned.
P_i :	The *i*th processor (in the processor group P).
Q_i :	The *i*th processor in the processor group Q.
$\Re_A^\infty$:	The worst case performance bound (ratio) for the heuristic A.
$r(t)$:	The release time of task t.
T_i :	The period of task τ_i.
U:	The utilization of a task set. $U = \sum_{i=1}^{n} C_i / T_i = \sum_{i=1}^{n} u_i$.
U_i :	The utilization of processor P_i.
u_i :	The utilization (or load) of the task τ_i. $u_i = C_i / T_i$.
$u_{j,i}$:	The utilization of task $\tau_{j,i}$.
V_i :	The V value of task τ_i. $V_i = \log_2 T_i - \lfloor \log_2 T_i \rfloor$.
$W(u_i)$:	Weighting function for u_i.
x_i :	The ratio between the periods of two adjacent tasks. $x_i = T_{i+1} / T_i$.

Chapter 1 Introduction

“The aim of princes and philosophers is to improve.”
-- Gottfried Wilhelm Leibniz, 1646-1716

1.1. Overview

Scheduling problems occur in a variety of situations in which a set of *resources* is to be used to perform a set of *tasks*. The general problem of scheduling [11, 12] is to allocate resources for the performance of a set of tasks such that specified *objectives* are achieved. Examples of scheduling problems include

(1) In summer Olympic Games, each game must be scheduled to take place at a certain site; some games are scheduled for the same site but for different time slots, and some games must be finished before others begin. For example, no championship game can be carried out until the games which decide the top two teams have been played.

(2) TV programs are interrupted periodically to show commercials. The number of commercials to be shown is subject to the constraint that the total time for commercials cannot exceed the time limit for program interruption.

(3) At an airport with a number of runways, decisions must be made concerning the assignment of aircraft to runways and the order in which aircraft take off or land on these runways. Furthermore, spare runways must be provided, or enough idle times reserved, for emergency situations such as the malfunctioning of some aircraft or the blocking of certain runways.

In these examples, the sites, the TV network, and the runways are resources and the playing of games, the showing of commercials, and the take-off or landing of aircraft are tasks to be performed. Our basic thesis is that regardless of the type of resources available and the character of the particular tasks to be performed, there is a fundamental similarity among all these scheduling problems: given that the following variables are known, our goal is to determine the assignment of resources to tasks and the order in which the tasks will be executed on the assigned resources according to some objectives:

(1) a set of tasks to be performed;

(2) a set of resources or facilities that may be employed in the performance of the tasks; and

(3) the sequence of elemental activities required to perform each of the tasks and any restrictions on the order in which tasks are performed.

While determining the assignment and the ordering, one should satisfy the constraints that are placed on the resources and the tasks. For example, in assigning aircraft to use a runway, the take-off time and the landing time of each aircraft must be observed; leaving insufficient time between a take-off and a landing might put life in jeopardy.

Many scheduling problems obviously get solved quite casually or automatically: students finish their degree requirements, professors teach classes, aircraft land, and we get served in a restaurant. Most of these problems are solved without explicit recognition that a scheduling problem even exists. Sometimes an ordering is determined by chance; more often tasks are performed because their deadlines are close. In many situations, a scheduling problem is worth considering because a proper scheduling decision may result in saving time and resources, or minimizing costs. For example, proper allocation of resources and ordering of tasks to be performed in a car factory may speed up the manufacturing of cars and make the business more profitable. In many real life problems, poor scheduling decisions can lead to excessive costs. Even more critical are cases where it is necessary to complete the tasks before some prescribed deadlines, or else irreparable loss might be incurred.

This thesis concerns itself with one aspect of the general scheduling problem, that of scheduling a set of tasks to meet their deadlines, under the constraint that all task deadlines must be honored. The scheduling problem will be studied in the context of real-time and fault-tolerant computer systems, though many of the results are valid for problems in other fields as well, for instance, a variation of the classical bin-packing problem.

1.2. Motivations and Objectives

Many applications that are mission-critical and life-critical, such as space exploration, the operation of nuclear power plants and defense systems, aircraft avionics, and robotics are not feasible without the support of computer systems. These applications require long duration of reliable operations as well as timeliness of operations. Computer systems that are used to support these applications are mainly parallel or distributed systems that are embedded into complex (or even hazardous) environments. The two most sought-after properties of these systems are *timeliness* and *dependability.*

Timeliness dictates that tasks must be finished within certain timing constraints. Computer systems that support timeliness are referred to as *real-time* systems. The correctness of a computation in a real-time system depends not only upon the results of computation but also upon the time at which results are generated. There are two major types of real-time systems: hard and soft real-time systems. In a hard real-time system, a late answer is a wrong answer. In a soft real-time system, a late answer may have some diminishing value.

Dependability is the quality of service that a particular system provides, which encompasses such concepts as reliability, availability, safety, maintainability, and performability [29]. One type of computer systems that support dependability is the *fault-tolerant* system. A fault-tolerant system can continue to correctly perform its specified tasks even in the presence of hardware failures and software errors [29].

The timeliness of a real-time system is ensured through scheduling algorithms. One major characteristic of real-time tasks is the repeated invocation of tasks at certain time periods, or in the parlance of practitioners, the execution of closed-loop control functions.

A task that arrives at every time interval has an unbounded number of requests, each of which must be executed by its deadline. The periodicity of real-time tasks comes directly from applications. For instance, the task of sensing a physical environment for certain quantities (e.g., the height or speed of a flying space shuttle) must be carried out periodically. Since there are an unbounded number of requests for each task and there are usually multiple tasks in a system, there arises a problem of scheduling all requests of the tasks properly so that their deadlines are met. The development of scheduling algorithms for periodic task systems has been a major focus in the area of real-time scheduling theory [1, 16, 19, 20, 24, 28, 40, 42, 43, 46, 47, 60, 64, 68, 70].

Since current technology is incapable of producing hardware components which never fail or software programs which are free of errors, a task might miss its deadline because of processor failures or task errors. To tolerate hardware failures or software errors, hardware and software redundancy techniques can be used. Examples of redundancy techniques are *N*-Modular-Redundancy (NMR) [29], Data-Diversity [35], Recovery Block [66], and *N*-version Programming (N-VP) [2]. At the level of task scheduling, hardware and software are abstracted as processors and tasks. In general, to tolerate processor failures, redundant processors are used to execute the same software and the final results are obtained through voting on the multiple results. To tolerate task errors, multiple implementations of software in the form of either different versions (*N*-VP) or data diversity are employed. A hybrid approach which combines software and hardware redundancy techniques can be used to tolerate task errors, processor failures, or both. For fault-tolerance purpose, associated with a task is a number of versions, that must be executed on different processors.

A real-time scheduling theory called Periodic Task Scheduling (PTS) has been gradually accepted as a general theory in supporting timeliness and, to some extent, dependability in a real-time system. This theory ensures that for a uniprocessor system, as long as the CPU utilization of all tasks lies below a certain bound and appropriate scheduling algorithms are used, all tasks will meet their deadlines without the programmer knowing exactly when any given request of a task is running. Even if a transient overload occurs, a fixed subset of critical tasks will still meet their deadlines as long as their total CPU utilization lies below a certain bound. This theory puts real-time software development on a sound analytical footing. The major components of PTS are the Rate-Monotonic (RM) algorithm and the Earliest Deadline First (EDF) algorithm. The RM algorithm is optimal for fixed priority assignment for scheduling a set of periodic tasks on a uniprocessor system, while the EDF algorithm is optimal for dynamic priority assignment.

First discovered by Liu and Layland [46] and Serlin [68], the RM and EDF algorithms have been proven to be viable scheduling techniques for real-time systems. Researchers have successfully applied these techniques to tackle a number of practical problems, such as task synchronization [60], bus scheduling [69], joint scheduling of periodic and aperiodic tasks [77, 79], mode change [70, 81], and transient overload [63]. In each of these areas, conventional RM and EDF algorithms have been adapted and extended to produce effective algorithms. Recently, the RM algorithm has been used in a number of applications. For example, it has been specified for use with software on board the Space Station as the means for scheduling multiple independent task execution [22]. The RM algorithm will be built into the on-board operating system, and is directly supported by the Ada compiler in use.

Although the RM scheduling is optimal for uniprocessor systems with fixed priority assignment and the EDF is optimal with dynamic priority assignment, unfortunately, neither is optimal for multiprocessor systems. In fact, the problem of optimally scheduling a set of periodic tasks on a multiprocessor system using either fixed priority or dynamic priority assignment is known to be NP-complete [43]. An optimal algorithm is the one that uses the minimum number of processors to schedule any task set. Given the intractability, any practical solution to the problem of scheduling periodic tasks on multiprocessor systems presents a trade-off between computational complexity and performance. Several efforts have been devoted to the development of heuristic algorithms for the scheduling problem [16, 17, 19, 20]. However, due to the difficulty involved in showing the effectiveness of the algorithms, few results have been obtained. In short, the progress in establishing a firm theoretical foundation for rate-monotonic scheduling on a multiprocessor has been slow.

The satisfactory solution to the problem of scheduling a set of periodic tasks on a multiprocessor system has a number of practical implications. Of these, two are very important: (1) the real-time application domain is becoming increasingly large. As requirements of real-time support for industrial and military applications become more sophisticated, the employment of multiprocessors to meet computational power requirements becomes essential. (2) The state-of-the-art of hardware technology makes multiprocessor support a reality for many more systems. Thus, the scheduling of periodic tasks on a multiprocessor has become an urgent problem that needs to be solved.

In this thesis, we will consider the problem of scheduling a set of periodic tasks on a multiprocessor system such that the task deadlines are guaranteed. Meeting all task deadlines is our first objective in solving the problem. In doing so we primarily employ either the RM or the EDF algorithm. Like so many other multiprocessor scheduling problems, there is an obvious solution for this scheduling problem: if we use as many processors as there are tasks, i.e., one processor for one task, then each task will meet its deadline and hence the scheduling problem will be solved. Aside from the fact that it is likely to be far from optimal, this solution has little practical relevance. Though current technology makes it possible to build computer systems that have hundreds of thousands of processors, it is not cost-effective to solve the scheduling problem in this manner. Requiring more processors in a system affects the cost, weight, size, power consumption, and communication costs of the whole system, the increase of any of which can jeopardize the success of the whole application. Therefore, besides the objective to ensure that all task deadlines are met, our second objective in solving the scheduling problem is to use as few processors as possible to schedule any given set of tasks. This objective will be relentlessly pursued throughout the thesis.

Our third objective is to support the fault-tolerance of real-time systems. The real-time systems under study are strictly hard real-time systems, each of which performs its functions through a set of periodic tasks. To enhance fault-tolerance, we assume that each task has multiple versions, each of which must be executed on different processors. For convenience, we use the word “versions” to mean any of the following: copies of the same implementation (NMR), versions from different implementation strategies (N-VP), or copies of the same implementation with different input and output schemes (Data Diversity). Since tasks are periodic, all versions of a task are periodic and their deadlines are the same. When multiple versions are used for a task, the concept of a task becomes an abstract one, while each of its versions becomes a real entity that is executed periodically. Thus the scheduling problem becomes one to minimize the number of processors used to accommodate a set of multiple-version periodic tasks such that the deadline of each task is met and the versions of each task are executed on different processors. The effectiveness of these redundancy techniques, and the selection of a particular redundancy technique to be used are not considered here. Furthermore, the problems of processor monitoring, failure detection, failure notification, and voting coordination are beyond the scope of this thesis.

This fault-tolerant scheduling problem, though simplified, captures the two most important properties of the systems under study: the *periodicity* of tasks in a real-time system and the *multiple execution* of tasks for fault-tolerance. The solution to this problem will inevitably shed more light on building fault-tolerant real-time systems.

1.3. Assumptions and Problem Statements

Since it is known that any slight modification of the constraints imposed on a scheduling problem may alter its complexity, the meaningful way to approach a scheduling problem is to state the constraints placed upon the problem precisely before any attempt is made to solve it. Besides, we should be aware of how our problems relate to other problems that have been studied. To achieve these two goals, we offer a top-down description of the problems and their relationship to other scheduling problems. A complete description of the general scheduling problem under various constraints [9, 12, 26], though desirable, is beyond the scope of this thesis.

Generally, a scheduling problem is defined by four parameters: (1) the machine environment, (2) the task characteristics, (3) the scheduling environment, and (4) the scheduling objectives. The machine environment specifies the types of the processors and their quantity. The task characteristics specify the timing constraints of the tasks, the relationship among them, and the relative importance of the tasks. The scheduling environment describes the restrictions imposed on the schedule: whether the tasks are preemptive or non-preemptive, and whether the scheduling is on-line or off-line. Finally, the objectives define such scheduling goals as minimizing the total number of processors, maximizing the total value of a system, or guaranteeing task deadlines. Our assumptions are as follows:

(1) For machine environment, we assume that processors are identical in the sense that the run-time of a task remains the same across all processors. Although for the most part, the number of processors available is assumed to be infinite, recall that our dominant objective is to use as few processors as possible.

(2) For the task characteristics, we assume that tasks are periodic and have one or more versions. The release time, computation time, and period for a task can vary. The deadline of each request of a task coincides with the arrival of the next request. Tasks are equally important in the sense that no missing of task deadlines is allowed. The tasks are independent in that the requests of a task do not depend on the initiation or the completion of the requests of other tasks.

The restriction that tasks are independent may seem overly restrictive, because precedence constraints are usually present in conventional task models. Yet a close look reveals that it is not possible to impose any precedence constraint upon periodic tasks, unless all the tasks have the same period. This point can be illustrated by considering the two tasks: $\tau_1 = (C_1, T_1)$ and $\tau_2 = (C_2, T_2)$ with $T_1 = kT_2$ for $k > 1$, where C_i and T_i are the computation time and period of task τ_i, respectively. If the precedence constraint is such that the execution of τ_1 always precedes that of τ_2, then we have the two cases: if the precedence constraint is honored, then for every T_1 time units, $k - 1$ requests of task τ_2 will miss their deadlines. If the deadline constraint is honored, then k requests of task τ_2 must be executed before one request of task τ_1 is executed, resulting in the violation of the precedence constraint. If the precedence constraint is reversed, then it can be similarly shown that both the precedence constraint and the deadline constraint cannot be honored at the same time. If all the tasks share the same period, the presence of precedence constraints might be meaningful. Even for this special case, the tasks that share the same period can be treated as one composite task in the periodic task model. Therefore, the difference in task periods has in fact imposed the characteristic of task independence.

Although it is not sensible to impose any precedence constraint upon periodic tasks, it may occur in some practical applications that the execution of certain requests of the periodic tasks may trigger the execution of some aperiodic tasks. The scheduling of periodic tasks together with aperiodic tasks is beyond the scope of this thesis.

(3) For the scheduling environment, we will consider both preemptive and non-preemptive task scheduling, though we will focus more on the preemptive scheduling. If the tasks are preemptive, then the execution of a task can be interrupted and later be resumed from where it is interrupted; otherwise, the execution of a task must be finished once it is started.

Another parameter in the scheduling environment concerns when the characteristics of the whole set of tasks are available. If the entire task set is known *a priori*, then the problem is referred to as being off-line, otherwise, it is said to be on-line. The algorithm that solves an on-line (off-line) problem is referred to as an on-line (off-line) algorithm.

While off-line algorithms have the advantages of being efficient and invoking minimal run-time scheduling overhead, there are situations where on-line algorithms must be used. For example, a change of mission in an application may require the execution of a totally different task set. Or the failure of some processors may necessitate the re-assignment of tasks. In these scenarios, the entire task set to be scheduled may change dynamically, that is, tasks can be added or deleted from the task set on-the-fly. We will develop both off-line and on-line algorithms for the scheduling problems.

(4) The scheduling objectives are (i) to meet all task deadlines, (ii) to minimize the number of processors required to accommodate a task set such that all task deadlines are met; and (iii) to support fault-tolerance if a task has multiple versions in execution.

The general solution to a multiprocessor problem involves two algorithms: one to assign tasks to individual processors, and the other to schedule tasks assigned on each individual processor. Two major approaches exist for assigning tasks to processors: *partitioning* and *non-partitioning* approaches. In a non-partitioning approach, each occurrence of a task may be executed on a different processor, while a partitioning approach requires that all occurrences of a task be executed on the same processor. The partitioning approach is often preferred because relatively low overhead is involved in the scheduling process.

A scheduling algorithm provides a set of rules that determine the processor(s) to be used and the task(s) to be executed at any particular point in time. One way to specify a scheduling algorithm is to allocate priorities to requests such that a higher-priority request has precedence over a lower-priority request in execution. We will consider only priority-driven scheduling algorithms. Priority-driven scheduling algorithms can be classified into two categories: fixed priority and dynamic priority assignment algorithms. With a fixed priority assignment, the priorities of tasks, once assigned, will not be changed. With a dynamic priority assignment, the priorities of tasks can be changed dynamically in execution. We will consider multiprocessor heuristic algorithms based on both fixed priority and dynamic priority assignment.

Now we are ready to define the scheduling problems more rigorously. A set of n

tasks $\Sigma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ is given to be scheduled on a number of processors. Each task is characterized by the tuple, $\tau_i = ((C_{i1}, C_{i2}, \ldots, C_{i\kappa_i}), R_i, D_i, T_i)$, where $C_{i1}, C_{i2}, \ldots, C_{i\kappa_i}$ are the computation times of the κ_i versions of task τ_i. R_i, D_i, and T_i are the release time, deadline, and period of task τ_i, respectively. The first request of the task τ_i arrives or is released at time R_i, and the subsequent requests are released at times $R_i + j \bullet T_i$ with $j = 1, 2, \ldots$. If T_i is specified as a variable, then the task system is termed an *aperiodic* task system. Otherwise, it is a *periodic* task system. The deadline D_i for a request of task τ_i is defined to be the moment D_i time units away from the release time of the request. In other words, the deadline D_i is relative to the release time. If $D_i < T_i$, then the deadline of a request is shorter than the period of the task, i.e., the current request of a task must be finished before the arrival of the next request. If a task has multiple versions for fault-tolerance, the request of a task constitutes the request of all its versions.

A *k-Timely-Fault-Tolerant* (hereinafter *k*-TFT) schedule is defined as a schedule in which no task deadlines are missed, despite *k* arbitrary processor failures or version errors. Then, given a set Σ of *n* tasks, *m* processors, the scheduling problem (hereinafter referred to as the TFT scheduling problem) can be defined, in terms of a decision problem, as deciding whether there exists a schedule, which is *k*-TFT for the task set Σ on *m* processors. In reality, it is more likely that a task set Σ is given and the scheduling goal is to find the minimum number of processors *m*, such that a *k*-TFT schedule can be constructed for the task set Σ on *m* processors. This problem then becomes an optimization problem.

Since a comprehensive study of the various cases of the TFT problem is beyond the scope of this thesis, we will focus our attention on four problems. The first problem is the development of efficient heuristic algorithms for scheduling a set of periodic tasks on a minimum number of processors such that the task deadlines are met on each processor by the RM algorithm. The second problem is to support fault-tolerance in rate-monotonic scheduling on multiprocessor systems. The third problem is to support fault-tolerance in earliest deadline first scheduling on multiprocessor systems. Finally, the last problem considers the fault-tolerance of the systems where the execution of tasks cannot be interrupted. Because of its relative importance and practical relevance, the first problem is the major focus for this thesis. The relationships among some of the pertinent scheduling problems are given in Figure 1.1.

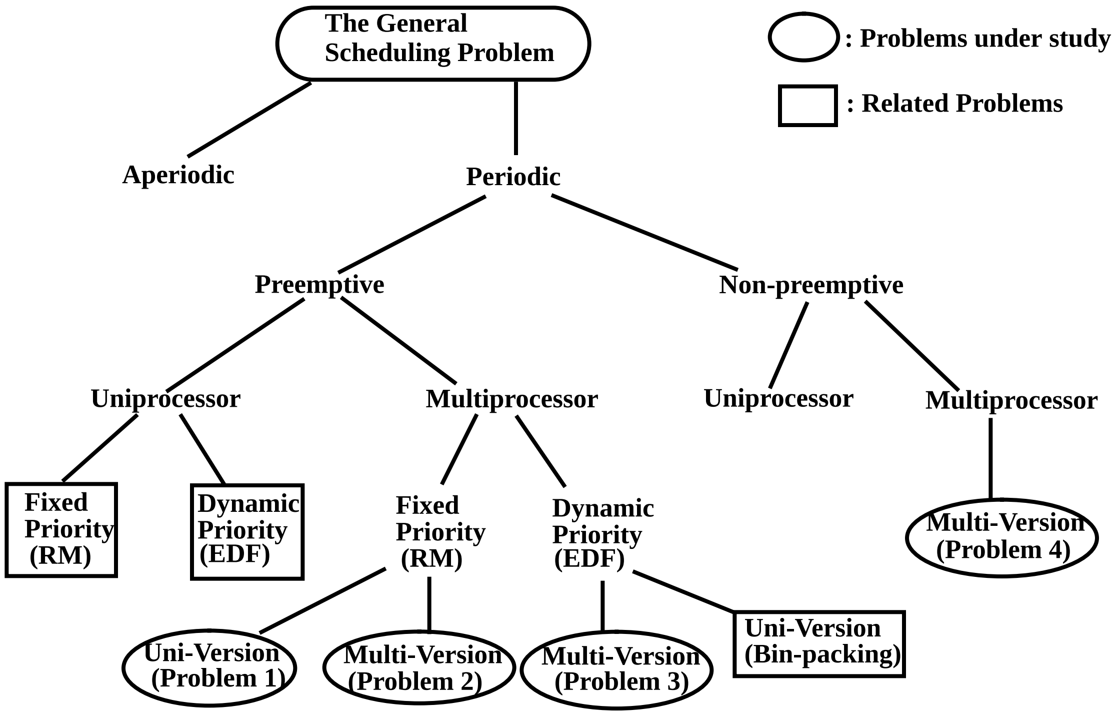


Figure 1.1: Problem Structure

Problem 1: Consider a computer system in which all tasks are periodic. For each periodic task, there are an infinite number of requests, each of which must be executed before the next request arrives. The arrival of a request occurs at a fixed time interval. The deadline of a request is defined as the arrival of the next request. Given a set of such tasks, what is the minimum number of processors required to execute it such that every request of every task finishes at or before its deadline by the RM algorithm? Note that we assume that a processor can only execute one task at a time and once a task is assigned to a processor, all its requests will be executed on that processor. The execution of a task may be interrupted and resumed at another time on the same processor. There is no cost or time loss associated with such an interruption or "preemption". This problem is referred to as the

Rate-Monotonic Multiprocessor Scheduling (RMMS) problem.

Problem 2: Consider a computer system in which tasks are not only periodic, but also have a number of versions. Like a task itself, a version also has an infinite number of requests, that must be executed like the requests of a task. The only difference is that a certain number of versions belongs to a task, and no two of them should be executed on the same processor. The idea of using “version” is to provide an abstraction over various redundancy techniques used to provide fault-tolerance capabilities in a computer system. The number of versions a task can have is called the degree of redundancy. The degree of redundancy may differ from task to task. Given a set of such tasks, what is the minimum number of processors required to execute it such that all requests of all versions finish within their respective deadlines using the RM algorithm? This problem is referred to as the Fault-Tolerant Rate-Monotonic Multiprocessor Scheduling (FT-RMMS) problem. Clearly, RMMS is a special case of FT-RMMS when the number of versions of a task is one for all tasks.

Problem 3: If the task deadlines on each processor are guaranteed by the EDF algorithm instead of by the RM algorithm as in Problem 2, what is the minimum number of processors required to do so? This problem is referred to as the Fault-Tolerant Earliest-Deadline-First Multiprocessor Scheduling (FT-EDFMS) problem.

Problem 4: What is the time complexity of scheduling a set of non-preemptive tasks to a number of processors such that processor failures can be tolerated? Given a set of non-preemptive tasks, each with a primary copy and a backup copy, how should they be scheduled such that task deadlines are met despite one arbitrary processor failure, i.e., how is an 1-TFT schedule generated?

1.4. Related Work

We review the existing work according to the preemption of tasks; work on preemptive scheduling is first reviewed, followed by work on non-preemptive scheduling.

If a set of tasks can be scheduled such that all task deadlines can be met by some algorithms, then we say that the task set is *feasible*. If a set of periodic tasks can be feasibly scheduled on a single processor, then the *Rate-Monotonic* (*RM*) [46] or *Intelligent Fixed Priority* algorithm [68] is optimal for fixed priority assignment, in the sense that no other fixed priority assignment algorithm can schedule a task set which cannot be scheduled by the RM algorithm. The RM algorithm assigns priorities to tasks according to their periods, where the priority of a task is in inverse relationship to its period. In other words, a task with a shorter period is assigned a higher priority. The execution of a low-priority task will be preempted if a high-priority task arrives. Liu and Layland proved that a set of n periodic tasks can be feasibly scheduled by the RM algorithm if the total utilization of the tasks is no more than a threshold number, which is given by $n\left(2^{1/n}-1\right)$. The utilization of a task is defined as the ratio between its computation time and its period, and the total utilization of a set of tasks is the sum of the utilizations of all tasks in the set.

Three other schedulability conditions were later developed by Dhall and Liu [20] in developing heuristic scheduling algorithms for multiprocessor systems. All these conditions are sufficient conditions. Lehoczky, Sha, and Ding recently discovered a schedulability condition that is both necessary and sufficient [40].

For dynamic priority assignment, the EDF algorithm is optimal in the sense that no other dynamic priority assignment algorithm can schedule a task set which cannot be scheduled by the EDF algorithm. The request of a task is assigned the highest priority if its deadline is the closest. Furthermore, a set of periodic tasks can be feasibly scheduled on a single processor system by the EDF algorithm if and only if its total utilization is no more than one.

Since the problem of scheduling a set of periodic tasks on a multiprocessor system, using either fixed priority assignment or dynamic priority assignment, is NP-complete, heuristic algorithms have been sought to solve it. The approach taken by a number of researchers [16, 17, 19, 20, 60] is to partition a given set of tasks into different groups, such that the tasks in each group can be feasibly scheduled on a single processor by a given algorithm.

An optimal algorithm for the scheduling problem is the one that uses the minimum number of processors to schedule any task set. In all studies, the performance of heuristic algorithms is evaluated against that of an optimal algorithm. In particular for real-time heuristics, the performance is measured using the worst case bounds of N_A/N_0, where N_A is the number of processors required to schedule a task set by a given heuristic A, and N_0 is the number of processors required to schedule the same task set by an optimal algorithm. Bounds for the existing heuristics are determined by the following expression (whose meaning will be explained later): $\Re_A^\infty = \lim_{N_0 \to \infty} N_A/N_0$. $\Re_A^\infty$ is often referred to as the *worst case* performance bound or asymptotic bound of the heuristic A.

Since a set of periodic tasks can be feasibly scheduled by the EDF algorithm on a single processor system so long as its total utilization is no more than one, the problem of scheduling a set of periodic tasks on a multiprocessor system where each individual processor runs the EDF algorithm can be reduced to the one-dimensional bin-packing problem. The one-dimensional bin-packing problem is to pack a list of variable-sized items into as few unit-sized bins as possible. The bin-packing problem has been the focus of intensive study for many years. A number of efficient algorithms have been proposed and analyzed. Among the many heuristics, Next-Fit (NF) has a tight bound of 2. First-Fit (FF) and Best-Fit (BF) have a tight bound of 1.7. First-Fit-Decreasing (FFD) has a tight bound of 11/9. Any on-line heuristic cannot have a worst case bound lower than 1.5333 [44].

The existing heuristic algorithms that schedule a set of periodic tasks on a multiprocessor using fixed priority assignment are summarized in Table 1.1. The measure $O(n \log n)$ denotes the computation time complexity for scheduling a set of *n* tasks.

Dhall and Liu were the first to propose two heuristic algorithms for the scheduling problem [20]. The algorithms, *Rate-Monotonic-Next-Fit* (RMNF) and *Rate-Monotonic-First-Fit* (RMFF), were shown to have worst case performance bounds of $2.4 \le \Re_{RMNF}^\infty \le 2.67$, and $2 \le \Re_{RMFF}^\infty \le (4(2^{1/3})/(1 + 2^{1/3}) \approx 2.23$. Unfortunately, the upper bound derived for RMFF was incorrect due to several errors in their proof, which are noted in Appendix A. Furthermore, their RMFF and RMNF are off-line, since they require that tasks must be assigned in the order of increasing period.

Davari and Dhall later considered two other algorithms called *First-Fit-Decreasing-Utilization-Factor* (FFDUF) and *NEXT-FIT-M* (NF-M) [16, 17]. The FFDUF algorithm sorts the set of tasks in non-increasing order of task utilization and assigns tasks to processors in that order. The NEXT-FIT-M algorithm classified tasks into M classes with respect to their utilizations. Processors are also classified into M classes, so that a processor in k-class executes tasks in k-class exclusively. Their worst case performance bounds are $\Re^{\infty}_{FFDUF} \le 2$, and $\Re^{\infty}_{NF-M} \le S_M$ where $S_M = 2.34$ for $M = 4$, and $S_M \rightarrow 2.2837$ when $M \rightarrow \infty$.

The FFDUF algorithm is a static algorithm, since *a priori* knowledge about the tasks is required, i.e., tasks must be in the order of non-decreasing task periods. In the general sense, the NF-M algorithm is an on-line algorithm, but its performance depends on the pre-selection of M and hence S_M, where S_M is a decreasing function of M, e.g., $S_M = 2.34$ for $M = 4$, and $S_M \rightarrow 2.2837$ for $M \rightarrow \infty$.

Table 1.1: Worst Case Performance of Existing Scheduling Algorithms

Algorithm A	$\Re^{\infty}_{A}$	Complexity	Type
RMNF [20]	[2.4, 2.67]	$O(n)$	Off-line
RMFF [20]	[2, 2.23?]	$O(n \log n)$	Off-line
NF-M [16]	$\le S_M \rightarrow 2.2837$	$O(n)$	On-line
FFDUF [17]	≤ 2.0	$O(n \log n)$	Off-line

For non-preemptive scheduling, the problem of minimizing the number of processors is more difficult. Jeffay, Stanat, and Martel [28] have shown that the problem of determining whether a set of non-preemptive, periodic tasks with different release times is schedulable is NP-hard in the strong sense. Furthermore, they have shown that a set of periodic tasks may not be schedulable non-preemptively on a single processor, even if its total utilization is very small, i.e., close to zero.

When the release times of all tasks are the same and the task periods obey a binary distribution, Gonzales and Soh [24] showed that an optimal algorithm exists for it. Let T_i denote the period of the ith task. Then by a *binary distribution* of task periods, they mean that if the tasks are ordered in terms of increasing period, then $T_{i+1} = 2T_i$. The optimal result can be generalized to include conditions in which tasks are related by $T_{i+1} = kT_i$, where k is an integer.

Though there have been several works in the literature [4, 5, 36, 45] which deal with allocation algorithms for fault-tolerant systems, they are developed under vastly different assumptions and are only remotely related to our work. Here we mention several. In order to tolerate processor failures, Balaji et al [4] presented an algorithm to dynamically distribute the workload of a failed processor to other operating processors. The tolerance of some processor failures is achieved under the condition that the task set is fixed, and enough processing power is available to execute it. Krishna and Shin [36] proposed a dynamic programming algorithm that ensures that backup, or contingency, schedules can be efficiently embedded within the original, “primary” schedule to ensure that hard deadlines continue to be met even in the face of processor failures. Unfortunately, their algorithm has the severe drawback that it is premised on the solution to two NP-complete problems.

Perhaps the most closely related work to ours is that of Bannister and Trivedi [5]. They considered the allocation of a set of periodic tasks to a number of processors so that a certain number of processor failures can be sustained. All the tasks have the same number of clones, and for each task, all its clones have the same computation time requirement. An approximation algorithm is proposed, and the ratio of the performance of the algorithm to that of the optimal solution, with respect to the balance of processor utilization, is shown to be bounded by $(9m) / (8(m - r + 1))$, where m is the number of processors to be allocated, and r is the number of clones for each task. However, their allocation algorithm does not consider the problem of minimizing the number of processors used, and the problem of how to guarantee the task deadlines on each processor is not addressed. These are very important considerations which our work addresses.

1.5. Approaches Taken

If we take the timing and fault-tolerant requirements in our scheduling problems as side conditions and the minimization of the number of processors as the objective function, then the problems become optimization problems. The available techniques to solve an optimization problem include graph theory, linear programming, integer programming, dynamic programming, and approximation. Except for the approximation technique, most of the techniques are able to find optimal solutions to the scheduling problems. Unfortunately, it may take a considerable amount of time to find the optimal solutions. The approximation technique tends to trade solution accuracy for computation time, i.e., a reasonably good approximation to the optimal solutions can be obtained by using some simple and fast algorithms.

The first three problems in Section 1.3 have been proven to be NP-complete. Since any solutions to an NP-complete problem for optimal results are typically deemed likely to require exponential time of computation in the worst case, we resort to the approximation techniques. The approximation algorithms, which are heuristic in nature, are called *heuristics*, or *heuristic algorithms* henceforth.

Since there are potentially numerous heuristic algorithms to solve a given problem, we need to find the ones that produce the *best* solutions (i.e., solutions which require the fewest processors). Since a heuristic algorithm cannot be guaranteed to find the optimal solutions for all inputs, we are therefore interested in knowing how close a heuristic solution is to an optimal solution. For our problems, the performance measure of a heuristic algorithm is the number of processors it requires to execute a given task set. Hence a sensible measure is the ratio between the number of processors required by a heuristic and that by an optimal algorithm. In other words, if we let N_A and N_0 denote the number of processors required by heuristic A and that by an optimal algorithm, respectively, then we should develop heuristic algorithms that have a small value of N_A / N_0.

As we soon discover, the ratio N_A / N_0 is not a constant for different sets of input. This is because a heuristic tends to perform well on some input and poorly on others. Accordingly, this performance measure is also problematic if we try to compare the performance of different heuristics. A solution to this problem is to obtain the mean value of N_A / N_0 under different probabilistic assumptions of input data. Another solution is to find the maximum value of N_A / N_0 for any given set of input. The first solution provides us with insight into the average case behavior of a heuristic, while saying nothing about the worst case performance of the heuristic. The second solution provides us with the complementary information. Therefore, in order to effectively evaluate the performance of various heuristics, we will resort to both.

To obtain the average case behavior of the heuristics, one can analyze the algorithms with probabilistic assumptions or conduct simulation experiments. Since a probabilistic analysis of our heuristics is beyond the scope of this study, we employ simulation to gain insight into the average case behavior of the heuristic algorithms.

Our approach to analyze the performance of approximation algorithms for various scheduling problems can be described as follows: we start with a simple but sensible algorithm and analyze its performance, both by proving bounds (or ratios) on what could happen in the worst case and by devising examples to verify that these bounds could not be improved. Then we seek alternative algorithms and analyze them. Our goal is to find algorithms that can provide better performance, i.e., lower worst case performance bounds. In the following, we give a formal definition of the worst case performance bound (ratio). For more details on formal description of such performance criteria, please refer to [23].

Let Π be a scheduling problem and I be any given set of tasks for problem Π. We define $\Re_A(I) = \dfrac{N_A(I)}{N_0(I)}$.

The *worst case performance bound (ratio)* $\Re_A$ for an approximation algorithm A for problem Π is given by $\Re_A = inf\{r \geq 1: \Re_A(I) \leq r$ for all instances $I \in D_{\Pi}\}$.

For our problems, it suffices to establish the following relation:

$N_A(I) \leq c \bullet N_0(I) + d$ for any set I of tasks, where c and d are constants.

Since d becomes insignificant when $N_0(I)$ is large, we let $\Re_A^{\infty} = c$, and use $\Re_A^{\infty}$ to denote the worst case performance bound for convenience.

In the literature, the measurement $\Re_A^{\infty} = \lim_{N_0 \to \infty} N_A / N_0$ is used frequently to evaluate the performance of heuristics [15, 23]. It is used because there are situations where the maximum value of N_A / N_0 is achieved under some pathological cases, mostly when the size of the input data is small. In order to avoid such pathological cases, the limit of N_A / N_0 is used as the worst case performance bound or asymptotic bound instead. For our scheduling algorithms, the worst case performance bound defined as $\Re_A$ is equal to $\Re_A^{\infty}$. It is apparent that $\Re_A^{\infty}$ can never be smaller than one, and the smaller the value of $\Re_A^{\infty}$ is, the better the performance of an algorithm is. We say that an algorithm is a *provably good (or effective)* one if its $\Re_A^{\infty}$ is known to be upper bounded by a number very close to one.

To obtain $\Re_A^{\infty}$, one should presumably know the value of N_0 for every task set. This is obviously impossible since the scheduling problem is NP-complete. The approaches to obtain $\Re_A^{\infty}$ therefore depend on the way a heuristic works. Mostly a lower bound of N_0 is used if it is known. In our analysis, we will rely heavily on a technique that uses mapping functions to relate N_A and N_0 to one another.

There are several strong reasons for obtaining the worst case performance bound for our scheduling heuristics: first, it is a good measurement for comparison of various heuristics. Second, since we are dealing with hard real-time systems, the knowledge about the worst case performance of the heuristics is crucial to guarantee the timeliness of the systems. This is particularly important when the heuristic algorithm is an on-line one.

As we have mentioned previously, the general solution to the multiprocessor scheduling problems involves two algorithms: one to assign tasks to processors and the other to schedule tasks on each individual processor. Since our problems of assigning tasks to processors bear many similarities with the one-dimensional bin-packing problem, we therefore try to adapt some of the best bin-packing heuristics to solve our scheduling problems. Many of the bin-packing heuristics are quite simple, and yet are capable of delivering provably good performance. The major difference between our scheduling problems and the bin-packing problem is, however, that the bins in bin-packing have unitary size, while the "size" or utilization of a processor in our scheduling problems changes dynamically according to some pre-defined functions. This difference makes the analysis of the worst case performance of the scheduling heuristics considerably more complicated than that of bin-packing heuristics. Note that the analysis of bin-packing heuristics is quite complex even when the sizes of bins are unitary [3, 14, 15, 30, 31, 32, 33, 38, 44, 83].

Since we have chosen to use the rate-monotonic algorithm for guaranteeing task deadlines on each processor in the first two problems, it seems that the only thing we need concern ourselves with is the algorithm to allocate tasks to processors, provided that the condition to schedule tasks on a single processor is known. Since the decision whether a task can be assigned on a processor is determined by a schedulability condition, the nature and performance of an allocation algorithm is determined in part by the quality of the condition. For the schedulability conditions reviewed in Section 1.4, the schedulability of a set of tasks depends not only on the utilization of each task, but also on the number of tasks in the set. Furthermore, the schedulability of a task set may depend upon the computation time and period of each individual task, as manifested by the necessary and sufficient condition. The scheduling problems thus become much more complicated.

The worst case condition (or sufficient condition) of rate-monotonic scheduling has been presented by Liu and Layland and the necessary and sufficient condition was recently proven by Lehoczky et al. While the condition given by $n\left(2^{1/n}-1\right)$, where n is the number of tasks assigned to a processor, is too conservative in assigning tasks to a processor, the necessary and sufficient condition is too complicated to be used and analyzed in an allocation scheme. The other conditions also have the weakness of being off-line and ineffective.

Our approach to tackling the problems is to develop schedulability conditions that exhibit good performance while remaining simple enough so that the worst case performance analysis is still possible. We then develop several simple allocation algorithms using the schedulability conditions. In the analysis of the worst case performance, we not only obtain the upper bounds of the algorithms, but also provide examples which show that the upper bounds are either tight or nearly tight. The analysis to determine the worst case performance is non-trivial, since our algorithms are more complex than their bin-packing counterparts, in the sense that the size of a bin is unitary in bin-packing, while the “size” or utilization of a processor is a variable.

For the FT-EDFMS problem, the schedulability condition is as simple as we can hope for and our solution to it is quite natural. To obtain the tight performance bound of the heuristic algorithm is, however, quite involved.

Since most of the non-preemptive scheduling problems are NP-complete, it is reasonable to expect that many cases of our last scheduling problem are also NP-complete. This is indeed the case, as will be shown later. However, this fact does not make the problems impossible to deal with, rather it requires us to develop heuristic algorithms to solve them where the problem instances are NP-complete. We will first prove that a number of problems are NP-complete and then develop two heuristic algorithms to solve two special cases of the scheduling problems. Finally, we evaluate their performance through analysis and simulation.

1.6. Organization

We first present several new schedulability conditions for rate-monotonic scheduling in Chapter 2. Because of the relative importance of the RMMS problem, we devote a major portion of the work to developing heuristic algorithms to solve it; the results are presented in Chapter 3. Since the FT-RMMS problem is more general, some of the solutions presented in Chapter 3 may not apply to it. We present one solution to the problem and analyze the performance of the algorithm in Chapter 4. In Chapter 5, we solve the problem of supporting fault-tolerance in EDF scheduling on a multiprocessor system by proposing a provably good heuristic algorithm. The problem of non-preemptive scheduling of tasks to meet their deadlines even in the presence of processor failures is considered in Chapter 6; also NP-completeness results and heuristic algorithms are presented. Finally, in Chapter 7, we summarize our results and discuss future research directions.

Chapter 2 Rate-Monotonic Scheduling on a Single Processor System

"A journey of a thousand miles begins with a single step."
-- Lao Zi, Dao De Jing

In this chapter, we will first present the task model and review several schedulability conditions that have been developed. Then we show that the time complexity to test the schedulability of a set of fixed-priority, periodic tasks using the necessary and sufficient condition is unbounded with respect to the number of tasks in a set. Accordingly, we present several new schedulability conditions that are sufficient but are linear in time complexity. Though they are all sufficient conditions, these new conditions have some advantages that others do not have: (1) they are more efficient than the previously derived sufficient conditions in the sense that the set of task sets that can be scheduled under the new conditions properly contains the set of task sets that can be scheduled under those previous conditions. (2) They are simpler than the necessary and sufficient condition in that each of the conditions can be expressed in a well-formed mathematical function. (3) They require only linear time complexity for the schedulability testing. Many of the new conditions will be used in the next chapter to develop provably effective scheduling heuristics for the RMMS problem.

2.1. Task Model

The tasks to be scheduled have the following characteristics:

(1) The requests of each task are periodic, with constant interval between requests.

(2) The deadline constraints specify that each request must be completed before the next request of the same task occurs.

(3) The tasks are independent in that the requests of a task do not depend on the initiation or the completion of the requests of other tasks.

(4) Run-time (or computation time) for the request of a task is constant for the task. Run-time here refers to the time a processor takes to execute the request without interruption.

Assumption (1) requires that each request of a task must arrive in the system at fixed interval. This precludes some tasks that need aperiodic processing. Recently, several techniques have been developed to schedule aperiodic tasks together with periodic tasks in a single processor system [18, 65, 79]. The essence of these techniques is either to reserve processor utilization for aperiodic tasks by approximating aperiodic task execution with periodic task execution, or to utilize unused time of periodic tasks for aperiodic task processing. Assumption (2) requires that the deadline of a request coincides with the arrival of the next request. This assumption can be relaxed for uniprocessor scheduling [41]. Assumption (4) basically assumes that all processors are identical.

It follows from the task model that a task is completely defined by two numbers, the run-time of the requests and the request period. The release time of each task does not affect the schedulability of a set of tasks [46]. We shall denote a task τ_i by the ordered pair (C_i, T_i), where C_i is the computation time and T_i is the period of the requests. The ratio C_i / T_i, which is denoted as u_i, is called the utilization (or load) of the task τ_i.

As we have mentioned in Chapter 1, the RM algorithm has been proven to be optimal for scheduling a set of fixed-priority, periodic tasks on a single processor. In the following, we offer a brief review on the RM algorithm. For more details, please refer to the original paper written by Liu and Layland [46].

We define the *response time* of a request for a certain task to be the time span between the request and the end of the response to that request. A *critical instant* for a task is defined to be an instant at which a request for that task will have the largest response time. Then we have the following theorem [46]:

Theorem 2.1: *A critical instant for any task occurs whenever the task is requested simultaneously with requests for all higher priority tasks.*

The implication of this theorem is that if the requests for all tasks at their critical instants are fulfilled before their respective deadlines by a certain scheduling algorithm, then the algorithm is feasible. As an example, consider a set of two tasks $\tau_1 = (C_1, T_1) = (1, 2)$ and $\tau_2 = (C_2, T_2) = (1.5, 5)$. The total utilization of the task set is therefore given by 1/2 + 1.5/5 = 0.8. If we assign τ_1 higher priority, then from Figure 2.1(a) we see that such priority assignment is feasible. Moreover, the value of C_2 can be increased at most to 2 but not further. On the other hand, if we let τ_2 be the higher priority task, then task τ_1 misses its deadline at t = 2. The value of C_1 needs to be decreased to 0.5 to make such priority assignment feasible. Therefore, intuitively, assigning a higher priority to a task with a shorter period (which is what RM does) yields more feasible schedules. The optimality of such priority assignment can be in fact established as Liu and Layland did in [46].

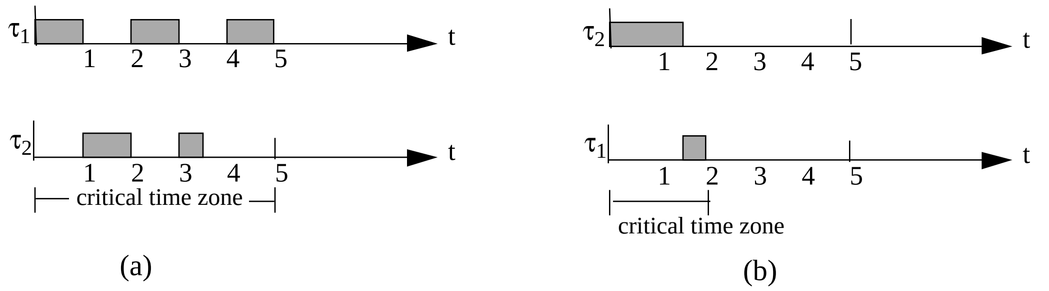


Figure 2.1: Schedule for Two Tasks

The following condition, which was given by Liu and Layland [46] and Serlin [68] and is hereafter referred to as WC (Worst-Case) condition, ensures that a task set can be scheduled to meet their deadlines by the RM algorithm if the total utilization of the tasks is less than or equal to $n\left(2^{1/n} - 1\right)$, where n is the number of tasks in the set.

Condition WC: If a set of n tasks is scheduled according to the RM algorithm, then the minimum achievable CPU utilization is $n\left(2^{1/n}-1\right)$. When $n \to \infty$, $n\left(2^{1/n}-1\right) \to \ln 2$.

The WC condition — $\sum_{i=1}^{n} C_i/T_i \le n\left(2^{1/n}-1\right)$ is a worst case condition, since there are task sets which are feasible, but cannot be determined to be feasible by the WC condition. For example, two tasks as given by $\tau_1 = (C_1, T_1) = (0.4, 1)$ and $\tau_2 = (C_2, T_2) = (0.5, 1)$ are not feasible according to the WC condition, since $\sum_{i=1}^{2} C_i/T_i = 0.9 > 2\left(2^{1/2}-1\right)$. But they are in fact feasible with the RM algorithm. There are task sets, however, that actually meet the worst case condition. For example, a task set consists of $\tau_1 = (C_1, T_1) = (2^{1/2}-1, 1)$ and $\tau_2 = (C_2, T_2) = (2-2^{1/2}, 2^{1/2})$ with $\sum_{i=1}^{2} C_i/T_i = 2\left(2^{1/2}-1\right)$, where any increase in the value of C_1 or C_2 will make the task set infeasible.

Another schedulability condition, which is called *IP* (*Increasing Period*), was given by Dhall and Liu [20] in studying the performance of their multiprocessor scheduling heuristics, RMNF and RMFF.

Condition IP: Let $\{\tau_i = (C_i, T_i) \mid i = 1, 2, \ldots, n\}$ be a set of n tasks with $T_1 \le T_2 \le \ldots \le T_n$ and $u = \sum_{i=1}^{n-1} C_i/T_i$. If $u \le (n-1)\left(2^{1/(n-1)}-1\right)$ and $C_n / T_n \le 2\left(1+u/(n-1)\right)^{-(n-1)}-1$, then the task set can be feasibly scheduled by the RM algorithm. As $n \to \infty$, the minimum utilization of τ_n approaches $2e^{-u}-1$.

This schedulability condition requires that the tasks be sorted in the order of non-decreasing period, thus implying that the task set should be known beforehand. Some of the task sets that cannot be scheduled by using the WC condition can be scheduled by using this condition, since this condition takes advantage of the fact that tasks are ordered against non-decreasing periods.

Dhall [19] also proved the following results:

Theorem 2.2: *Let* $\{\tau_i = (C_i, T_i) \mid i = 1, 2, \ldots, n\}$ *be a set of n tasks with* $T_1 \le T_2 \le \ldots \le T_n$ *and* $u_1 = C_1/T_1$. *If the utilization* $u = \sum_{i=2}^{n} C_i/T_i$ *of the* $(n-1)$ *tasks* $\{\tau_i \mid i = 2, 3, \ldots, n\}$ *is less than or equal to* $(n-1)\{[2/(1+u_1)]^{1/(n-1)}-1\}$, *then* *the given set of n tasks can be feasibly scheduled by the RM algorithm. When* $n \to \infty$, $(n-1)\{[2/(1+u_1)]^{1/(n-1)}-1\} \to \ln(2/(1+u_1))$.

Theorem 2.3: *Let* $\{\tau_i = (C_i, T_i) \mid i = 1, 2, \ldots, n\}$ *be a set of n tasks with* $T_1 \le T_2 \le \ldots \le T_n$. *Let the utilization of the (n – 1) tasks* $\{\tau_i \mid i = 1, 2, \ldots, n-1\}$ *be* $u = \sum_{i=1}^{n-1} C_i/T_i$. *If* $u \le (n-1)\left(2^{1/(n-1)}-1\right)$ *and* $C_n/T_n \le 2[1+u/(n-1)]^{-(n-1)}-1$, *then the given set of n tasks can be feasibly scheduled by the RM algorithm.*

Lehoczky et al recently obtained the following result, which contains a condition that is both necessary and sufficient [40]. This condition, which is called the *IFF* condition, takes into account of both the computation time and period of a task.

Condition IFF: Let $\Sigma = \{\tau_i = (C_i, T_i) \mid i = 1, 2, \ldots, n\}$ be a set of n tasks with $T_1 \le T_2 \le \ldots \le T_n$. τ_i can be feasibly scheduled by the RM algorithm if and only if $L_i = min_{\{t \in S_i\}}((W_i(t))/t) \le 1$. The entire task set Σ can be feasibly scheduled by the RM algorithm if and only if $L = max_{\{1 \le i \le n\}} L_i \le 1$, where $S_i = \{kT_j \mid j = 1, \ldots, i; k = 1, \ldots, \lfloor T_i/T_j \rfloor\}$, $W_i(t) = \sum_{j=1}^{i} C_j \lceil t/T_j \rceil$.

2.2. Some Important Lemmas

While the WC condition may be too conservative in assigning tasks to a processor, the necessary and sufficient condition is too complex to be used and analyzed in an assignment algorithm. In fact, the computational time requirement to test the schedulability of a set of tasks by using the necessary and sufficient condition is unbounded. In the worst cases, the time complexity may be more than exponential. This is shown by the following lemma.

Lemma 2.1: *The time complexity to use the necessary and sufficient condition is unbounded. In some cases, the time complexity may be more than exponential.*

Proof: This lemma can be proven by constructing the following task set. When the necessary and sufficient condition is used to test the schedulability of the task set, the computational time requirement is more than exponential.

A set of n periodic tasks $\Sigma = \{\tau_i = (C_i, T_i) \mid i = 1, 2, \ldots, n\}$ is given with the following characteristics: $n = \lfloor T_{i+1}/T_i \rfloor$, for $i = 1, 2, \ldots, n-1$. Then the power of S_n as defined in the IFF condition is equal to or greater than the following number: $n^{n-1} + n^{n-2} + \ldots + n + 1 = O\left(n^{n-1}\right)$. In other words, assuming that the task set is infeasible, we need to verify an exponential number of equations with respect to *n*, in order to find out whether task τ_n can be feasibly scheduled with the rest of $n-1$ tasks.

Similar examples that require more than exponential time complexity can be constructed. Obviously, the computation time requirement is unbounded and may be more than exponential. ■

One of the implications of the above lemma is that we need to find schedulability conditions that are more time-efficient, though they may not be necessary. Before we present the new conditions, a few lemmas need to be established.

Lemma 2.2: *If a task* (C_0, T_0) *cannot be scheduled together with a set of* $m \geq 1$ *tasks* $\{\tau_i = (C_i, T_i) \mid i = 1, 2, \ldots, m\}$ *by the RM algorithm and* $T_0 \leq T_1 \leq T_2 \leq \ldots \leq T_m$, *then* $(2C_0, 2T_0)$ *cannot be scheduled together with the same set of tasks* $\{\tau_i = (C_i, T_i) \mid i = 1, 2, \ldots, m\}$ *by the RM algorithm.*

Proof: Let us denote the task set of $\{\tau_i = (C_i, T_i) \mid i = 0, 1, \ldots, m\}$ with $T_0 \leq T_1 \leq T_2 \leq \ldots \leq T_m$ as Σ, and the task set of $(2C_0, 2T_0)$, (C_1, T_1), (C_2, T_2), …, (C_m, T_m) as Σ'. Suppose that for task set Σ, the *i*th task with $1 \leq i \leq m$ is the first task to miss its deadline. Then we claim that if $2T_0 \leq T_i$, then the *i*th task misses its deadline in the task set Σ', otherwise, task $(2C_0, 2T_0)$ misses its deadline.

Since (C_i, T_i) misses its deadline, according to the IFF condition, we have

$$f(t) = (C_0\lceil t/T_0 \rceil + C_1\lceil t/T_1 \rceil + C_2\lceil t/T_2 \rceil + \ldots + C_{i-1}\lceil t/T_{i-1} \rceil + C_i) / t > 1 \quad \text{(Eq.2.1)}$$

for $t \in S = \{kT_j \mid j = 0, 1, 2, \ldots, i;\ k = 1, 2, \ldots, \lfloor T_i/T_j \rfloor\} = \{T_0, 2T_0, \ldots, q_0T_0, T_1, 2T_1, \ldots, q_1T_1, T_2, \ldots, T_{i-1}, 2T_{i-1}, \ldots, q_{i-1}T_{i-1}, T_i\}$, where $q_x = \lfloor T_i/T_x \rfloor$ for $x = 0, 1, \ldots, i-1$.

Case 1: $2T_0 \leq T_i$. Let us examine the new function: $f'(t) = (2C_0\lceil t/(2T_0) \rceil + C_1\lceil t/T_1 \rceil + C_2\lceil t/T_2 \rceil + \ldots + C_{i-1}\lceil t/T_{i-1} \rceil + C_i) / t$, for $t \in S' = \{kT_j \mid j = 0', 1, 2, \ldots, i;\ k = 1, 2, \ldots, \lfloor T_i/T_j \rfloor\} = \{2T_0, 4T_0, \ldots, 2q_{0'}T_0, T_1, 2T_1, \ldots, q_1T_1, T_2, \ldots, T_{i-1}, 2T_{i-1},$

$\ldots, q_{i-1}T_{i-1}, T_i\}$, where $C_{o'} = 2C_0$ and $T_{o'} = 2T_0$, and $q_{0'} = \lfloor T_i/(2T_0) \rfloor$.

Since $C_0\lceil t/T \rceil = 2C_0 \lceil t/(2T_0) \rceil$, we have $f'(t) = f(t) > 1$ for $t \in \{2T_0, 4T_0, \ldots, 2q_{0'}T_0\}$.

We then claim that $2C_0\lceil t/(2T) \rceil \geq C_0\lceil t/T \rceil$ for $t \in \{T_1, 2T_1, \ldots, q_1T_1, T_2, \ldots, T_{i-1}, 2T_{i-1}, \ldots, q_{i-1}T_{i-1}, T_i\}$.

Since $t > T_0$, we write $t = wT_0 + r$, where $0 \leq r < T_0$ and $w \geq 1$. If $r = 0$, then $2\lceil t/(2T_0) \rceil = 2\lceil w/2 \rceil \geq w = \lceil t/T_0 \rceil$. If $r > 0$, then $2\lceil t/(2T_0) \rceil = 2\lceil q/2 + r/(2T_0) \rceil \geq \lceil q + r/T_0 \rceil$. Therefore $f'(t) \geq f(t) > 1$. In other words, task (C_i, T_i) misses its deadline.

Case 2: $T_i < 2T_0$. Let us examine the new function: $f'(t) = (C_1\lceil t/T_1 \rceil + C_2\lceil t/T_2 \rceil + \ldots + C_{i-1}\lceil t/T_{i-1} \rceil + C_i\lceil t/T_i \rceil + 2C_0)/t$, for $t \in S' = \{T_1, T_2, \ldots, T_{i-1}, T_i, 2T_0\}$.

Since $\lceil T_j/T_0 \rceil = 2$ and $\lceil T_j/T_i \rceil = 1$ for $j = 1, 2, \ldots, i$, we have $f'(t) = f(t) > 1$ for $t \in \{T_1, T_2, \ldots, T_{i-1}, T_i\}$.

$f'(t) = (2C_1 + 2C_2 + \ldots + 2C_i + 2C_0)/(2T_0)$

$= (C_1 + C_2 + \ldots + C_i + C_0)/T_0 = f(T_0) > 1$ for $t = 2T_0$.

In other words, task $(2C_0, 2T_0)$ misses its deadline.

Therefore, the lemma must be true. ■

Lemma 2.2 is very powerful, since it implies that if a task set is infeasible, then there exists one that is also infeasible with the same task utilizations but with the ratio between any two task periods less than 2. In deriving schedulability condition, we need only to consider task sets where the ratio between any two task periods is less than 2, since this is the worst case scenario.

The following lemma is a reiteration of the fact that was used implicitly by Liu and Layland to derive their worst case condition. Note that since we are considering the uniprocessor scheduling, the total utilization of a task set is equivalent to the utilization of the processor on which the task set is scheduled. Hence we use the two terms interchangeably.

Lemma 2.3: *For a set of n tasks* $\Sigma = \{\tau_i = (C_i, T_i) \mid i = 1, 2, \ldots, n\}$ *scheduled by the RM algorithm on a single processor system, and the restriction that* $T_1 \leq T_2 \leq \ldots \leq$

$T_n < 2T_1$, the least upper bound of the processor utilization is achieved when $C_i = T_{i+1} - T_i$ for $i = 1, 2, \ldots, n-1$ and $C_n = 2T_1 - T_n$.

Proof: Let $\Sigma = \{\tau_i = (C_i, T_i) \mid i = 1, 2, \ldots, n\}$ be a set of n tasks with $T_1 \leq T_2 \leq \ldots \leq T_n < 2T_1$, and $C_1, C_2, \ldots, C_n$ be the computation times of the tasks that fully utilize the processor and minimize the processor utilization, i.e., $U = \sum_{i=1}^{n} C_i / T_i$.

Suppose that

$$C_1 = T_2 - T_1 + \Delta, \Delta > 0.$$

Let

$$C_1' = T_2 - T_1$$

$$C_2' = C_2 + \Delta$$

$$C_3' = C_3$$

$$\ldots$$

$$C_n' = C_n$$

Then $C_1', C_2', C_3', \ldots, C_n'$ also fully utilize the processor. Let $U' = \sum_{i=1}^{n} C_i' / T_i$. Then

$$U - U' = (\Delta / T_1) - (\Delta / T_2) > 0.$$

On the other hand, suppose that

$$C_1 = T_2 - T_1 - \Delta, \Delta > 0.$$

Let

$$C_1' = T_2 - T_1$$

$$C_2' = C_2 - 2\Delta$$

$$C_3' = C_3$$

$$\ldots$$

$$C_n' = C_n$$

Then again, $C_1', C_2', C_3', \ldots, C_n'$ also fully utilize the processor. Let $U' = \sum_{i=1}^{n} C_i' / T_i$. Then

$$U - U' = -(\Delta / T_1) + (2\Delta / T_2) > 0.$$

Therefore, if U is indeed the minimum processor utilization, then

$$C_1 = T_2 - T_1$$

Similarly we can show that

$$C_i = T_{i+1} - T_i, \text{ for } i = 2, \ldots, n-1, \text{ and}$$

$$C_n = T_1 - \sum_{i=1}^{n-1} C_i = 2T_1 - T_n$$

Thus the lemma is proven. ■

2.3. Period-Oriented Schedulability Conditions

In this section, we present several schedulability conditions for the RM algorithm, which are predominantly oriented towards task periods.

Note that in deriving the schedulability conditions, we want to find out a threshold number such that at least one task set is infeasible if its total utilization is greater than the threshold number and feasible if its total utilization is no greater than the threshold number. The threshold number may be determined by functions of the number of the tasks in the set, the relative values of the task periods and computation times. Such a threshold number has been found to be elusive for the necessary and sufficient condition. Yet it is possible for sufficient conditions. Even though we cannot avoid the worst case scenario in all these conditions, we want to find out exactly, or nearly exactly the schedulability conditions that can successfully determine the feasibility of most of the feasible task sets.

Theorem 2.4: *Let* $\{\tau_i = (C_i, T_i) \mid i = 1, 2, \ldots, n\}$ *be a set of n tasks. Let* $\gamma = \left(max_{i,j}\frac{T_i}{T_j}\right)$*. If* $\gamma < 2$ *and*

$$\sum_{i=1}^{n} C_i/T_i \le (n-1)\left(\gamma^{1/(n-1)} - 1\right) + 2/\gamma - 1, \qquad \text{(Eq.2.2)}$$

then the task set can be feasibly scheduled by the RM algorithm. The minimum of $f(n, \gamma) = (n-1)\left(\gamma^{1/(n-1)} - 1\right) + 2/\gamma - 1$ *is achieved when* $\gamma = 2^{1-1/n}$.

Proof: By Lemma 2.2, we can assume without loss of generality that the task periods satisfy the following relationship:

$$T_1 \le T_2 \le \dots \le T_n < 2T_1.$$

Let $\gamma = T_n / T_1$. Hence $\gamma < 2$.

Then by Lemma 2.3, the minimum utilization of the task set is achieved when

$$C_1 = T_2 - T_1,$$

$$C_2 = T_3 - T_2,$$

$$\dots$$

$$C_{n-1} = T_n - T_{n-1},$$

$$C_n = T_1 - \sum_{i=1}^{n-1} C_i = 2T_1 - T_n.$$

The total utilization of the task set is given by

$$U = \sum_{i=1}^{n} C_i / T_i.$$

Rewrite U as $U = \sum_{i=1}^{n-1} (T_{i+1} - T_i) / T_i + (2T_1 - T_n) / T_n$.

Let $x_i = \frac{T_{i+1}}{T_i}$ for $i = 1, 2, \dots, n-1$. Then $C_n / T_n = (2T_1 - T_n) / T_n = 2 / \prod_{i=1}^{n-1} x_i - 1$ and $T_n / T_1 = \gamma = \prod_{i=1}^{n-1} x_i$.

$$U = \sum_{i=1}^{n-1} x_i + 2 / \prod_{i=1}^{n-1} x_i - n.$$

We need to minimize U subject to the side condition $\gamma = \prod_{i=1}^{n-1} x_i$. This is achieved by forming the Lagrangian

$$L = U + \lambda \left[\gamma - \prod_{i=1}^{n-1} x_i\right]$$

and minimizing the function L over x_i's, and λ.

$$\frac{\partial L}{\partial x_i} = 1 - \frac{2}{x_i \prod_{j=1}^{n-1} x_j} - \lambda \left(\prod_{i=1}^{n-1} x_i\right) / x_i = 0 \text{ for } i = 1, 2, \dots, n-1. \qquad \text{(Eq.2.3)}$$

$$\frac{\partial L}{\partial \lambda} = \gamma - \prod_{i=1}^{n-1} x_i = 0. \qquad \text{(Eq.2.4)}$$

Solving these n equations yields

$$x_i = 2/\gamma + \lambda\gamma = \gamma^{1/(n-1)}.$$

Therefore $U = (n-1)(\gamma^{1/(n-1)} - 1) + 2/\gamma - 1$.

Let $f(n, \gamma) = (n-1)(\gamma^{1/(n-1)} - 1) + 2/\gamma - 1$. Then the function $f(n, \gamma)$ is strictly decreasing in n. To find the minimum of $f(n, \gamma)$ with regard to γ, we take the derivative of the function $f(n, \gamma)$ with regard to γ and solve for γ by setting the resultant equation to zero. We obtain $\gamma = 2^{1-1/n}$. In other words, the minimum of the function is achieved at $f(n, \gamma) = (n-1)(\gamma^{1/(n-1)} - 1) + 2/\gamma - 1 = (n-1)(2^{1/n} - 1) + 2/2^{1-1/n} - 1 = n(2^{1/n} - 1)$, which is exactly the result obtained by Liu and Layland.

It is also apparent that the function $f(n, \gamma)$ is strictly decreasing with regard to γ within the range $[1, 2^{1-1/n})$, and increasing within the range $(2^{1-1/n}, 2)$. ■

We plot the function $f(n, \gamma) = (n-1)(\gamma^{1/(n-1)} - 1) + 2/\gamma - 1$ in Figure 2.2. It is evident that $f(n, \gamma)$ is strictly decreasing in n. When n is large, the increase of $f(n, \gamma)$ in the range of $(2^{1-1/n}, 2)$ becomes insignificant.

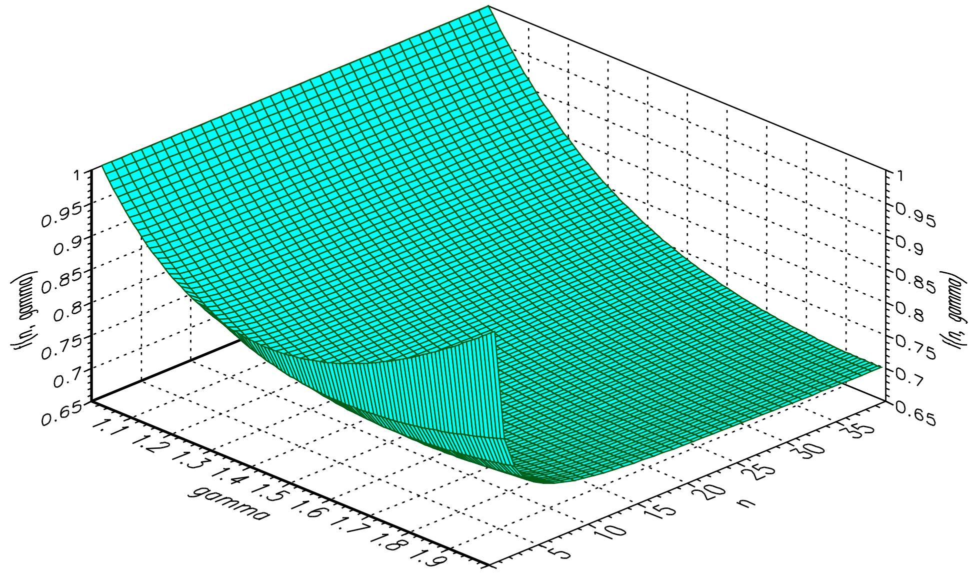


Figure 2.2: The $f(n, \gamma)$ Function

Though the condition given in inequality (2.2) may yield higher utilization, the requirement that the ratio between any two task periods is less than 2 is too strict. In fact, that requirement is unnecessary if we take advantage of the result in Lemma 2.2. This is shown by the two theorems as follows:

Theorem 2.5: *Let* $\{\tau_i = (C_i, T_i) \mid i = 1, 2, \ldots, n\}$ *be a set of n tasks. Define* $V_i = \log_2 T_i - \lfloor \log_2 T_i \rfloor$ *for* $i = 1, 2, \ldots, n$. *Then sort the* V_i*s in the order of increasing value* *and rename them to be* V_i *for* $i = 1, 2, \dots, n$. *If* $\sum_{i=1}^{n} C_i/T_i \le \sum_{i=1}^{n-1} 2^{V_{i+1} - V_i} + 2^{1 + V_1 - V_n} - n$, *then the task set can be feasibly scheduled by the RM algorithm.*

Proof: Since $2^{V_n} < 2$, we can assert, by exactly the same reasoning as in the proof of Theorem 2.4, that the minimum total utilization is achieved when

$$2^{V_1} \le 2^{V_2} \le \dots \le 2^{V_n} < 2 \bullet 2^{V_1},$$

$$C_i = 2^{V_{i+1}} - 2^{V_i},$$

$$C_n = 2 \bullet 2^{V_1} - 2^{V_n},$$

for $i = 1, 2, \dots, n - 1$.

By similar reasoning, we have

$$U = \sum_{i=1}^{n-1} \left(2^{V_{i+1}} - 2^{V_i} \right) / 2^{V_i} + (2 \bullet 2^{V_1} - 2^{V_n}) / 2^{V_n} = \sum_{i=1}^{n-1} 2^{V_{i+1} - V_i} + 2^{1 + V_1 - V_n} - n.$$

The theorem follows through Lemma 2.2. ■

Theorem 2.6: *Let* $\{ \tau_i = (C_i, T_i) \mid i = 1, 2, \dots, n \}$ *be a set of n tasks. Define* $V_i = \log_2 T_i - \lfloor \log_2 T_i \rfloor$ *for* $i = 1, 2, \dots, n$, *and*

$$\beta = max_{1 \le i \le n} V_i - min_{1 \le i \le n} V_i . \qquad \text{(Eq.2.5)}$$

If $\sum_{i=1}^{n} C_i/T_i \le (n - 1) (2^{\beta/(n-1)} - 1) + 2^{1-\beta} - 1$, *then the task set can be feasibly scheduled by the RM algorithm.*

Proof: If we minimize the function $U = \sum_{i=1}^{n-1} 2^{V_{i+1} - V_i} + 2^{1 + V_1 - V_n} - n$ subject to the constraints that $2^{V_n} < 2$ and $0 \le V_i \le V_{i+1}$, then the minimum is achieved when

$$V_{i+1} - V_i = \beta / (n - 1).$$

Therefore, $U = (n - 1) (2^{\beta/(n-1)} - 1) + 2^{1-\beta} - 1$. Together with Theorem 2.5, we have that if $\sum_{i=1}^{n} C_i/T_i \le (n - 1) (2^{\beta/(n-1)} - 1) + 2^{1-\beta} - 1$, then the task set can be feasibly scheduled by the RM algorithm. ■

Corollary 2.1: *Let* $\{ \tau_i = (C_i, T_i) \mid i = 1, 2, \dots, n \}$ *be a set of n tasks and* β *be defined as in (2.5). If*

$$\sum_{i=1}^{n} C_i/T_i \le \max \{ \ln 2, 1 - \beta \ln 2 \} , \qquad \text{(Eq.2.6)}$$

then the task set can be feasibly scheduled by the RM algorithm.

Proof: Since the worst case condition $\sum_{i=1}^{n} C_i / T_i \le n\left(2^{1/n} - 1\right)$ is strictly decreasing with respect to n, we have that

$$n\left(2^{1/n} - 1\right) > \lim_{n \to \infty} n\left(2^{1/n} - 1\right) = \ln 2.$$

Furthermore, $(n-1)\left(2^{\beta/(n-1)} - 1\right) + 2^{1-\beta} - 1 > \lim_{n \to \infty} (n-1)\left(2^{\beta/(n-1)} - 1\right) + 2^{1-\beta} - 1 = \beta\ln 2 + 2/2^{\beta} - 1 > 1 - \beta\ln 2$.

Together with Theorem 2.6, we have proven the corollary. ■

We will call the condition in (2.6) the **PO** (Period-Oriented) condition.

2.4. Utilization-Oriented Schedulability Conditions

Theorem 2.7: *Let $\{\tau_i = (C_i, T_i) \mid i = 1, 2, \ldots, n-1\}$ be a set of $n-1$ tasks and it can be feasibly scheduled by the RM algorithm. Among the $n-1$ tasks, the utilizations of $0 \le m \le n-1$ tasks are known to be $u_1, u_2, \ldots, u_m$, and the total utilization of the rest of the $n-m-1$ tasks is known to be u, i.e., $u = \sum_{i=m+1}^{n-1} C_i / T_i$. A new task $\tau_n = (C_n, T_n)$ can be feasibly scheduled with the $n-1$ tasks on a single processor by the RM algorithm, if*

$$C_n/T_n \le \begin{cases} 2\left[\prod_{i=1}^{m}\left(1+u_i\right)\right]^{-1}\left[1 + u/(n-m-1)\right]^{-(n-m-1)} - 1 & \text{if } m < n-1 \\ 2\left[\prod_{i=1}^{m}\left(1+u_i\right)\right]^{-1} - 1 & \text{if } m = n-1 \end{cases} \qquad \text{(Eq.2.7)}$$

This utilization condition is tight in the sense that there are task sets that actually meet this condition.

When $m = 0$, the expression in (2.7) becomes $C_n / T_n \le 2\left[1 + u/(n-1)\right]^{-(n-1)} - 1$. This condition is the same as the IP condition given by Dhall and Liu, except that tasks are not necessarily assigned in the order of increasing period. The expression in (2.7) provides not only just one condition, but in effect, $(n-1)$ conditions. Most significantly, no restriction is placed on the relative order of the tasks to be scheduled.

When $m = n-1$, the expression in (2.7) becomes

$$C_n / T_n \le 2\left[\prod_{i=1}^{n-1}(1+u_i)\right]^{-1} - 1 \qquad \text{(Eq.2.8)}$$

We will refer this condition as the **UO** (Utilization-Oriented) condition.

If we minimize the expression $U = \sum_{i=1}^{n} C_i / T_i$ over u_i in the UO condition, then the minimum of U is achieved when $u_i = 2^{1/n} - 1$ for $i = 1, 2, \ldots, n$. Thus, $U = n\left(2^{1/n} - 1\right)$. This is exactly the same condition as given by Liu and Layland. In other words, the Liu and Layland's condition is a worst case condition in that it is only achievable when every task has the same utilization of $u_i = 2^{1/n} - 1$. If the utilizations of the tasks are not the same, then the bound for $\sum_{i=1}^{n} C_i / T_i$ can be significantly higher than $n\left(2^{1/n} - 1\right)$ in some cases under this new condition. For example, if $u_1 = 0.6$ and $u_2 = 0.1797$, then the maximum utilization of a task that can be scheduled together with the two tasks according to the UO condition is $C_n / T_n \le 2/[(1 + u_1)(1 + u_2)] - 1 = 0.06$, while it is impossible to schedule a third task on the same processor by the WC condition.

If we view the schedulability conditions for the RM scheduling as points on a spectrum, then on the one end is the worst-case condition by Liu and Layland and on the other end is the sufficient and necessary condition by Lehoczky et al. As we move from one end of the WC condition to the other end of the IFF condition, the schedulability of the conditions increases, as more information of the tasks is taken into account. In the condition $\sum_{i=1}^{n} C_i / T_i \le n\left(2^{1/n} - 1\right)$, only the number of tasks in a set is considered. In expression (2.7), starting from $m = 0$ to $m = n - 1$, not only the number of tasks in a set is taken into account, but also the utilization of each individual task. Furthermore, the conditions can be nicely expressed in well-formed mathematical formula, contrasting the necessary and sufficient condition. The time complexity of all our schedulability conditions remains linear with respect to the number of the tasks in a set.

Proof of Theorem 2.7: We will obtain the expression in (2.7) through two steps: one is under the condition $m < n - 1$ and the other is under the condition $m = n - 1$.

Let us first form a new task set called Σ, from the task τ_n and the set of $(n - 1)$ tasks $\{\tau_i = (C_i, T_i) \mid i = 1, 2, \ldots, n - 1\}$ such that $\Sigma = \{\tau_i \mid i = 1, 2, \ldots, n\}$.

According to Lemma 2.2, we can assume without loss of generality that the periods of the task set satisfy

$$\left(max_{i,j} \frac{T_i}{T_j} \right) < 2.$$

We then sort the tasks in the order of the increasing period and rename them appropriately such that

$$T_1 \le T_2 \le \ldots \le T_\mathrm{n} < 2T_1.$$

Then by Lemma 2.3, the minimum utilization is achieved when

$$C_1 = T_2 - T_1,$$

$$C_2 = T_3 - T_2,$$

$$\ldots$$

$$C_{n-1} = T_n - T_{n-1},$$

$$C_n = T_1 - \sum_{i=1}^{n-1} C_i = 2T_1 - T_n.$$

This set of tasks fully utilizes the processor, even though there is an unknown quantity C_i / T_i in the above conditions that has yet to be determined. The unknown quantity C_i / T_i corresponds to the original C_n / T_n before renaming. Let $U = \sum_{i=1}^{n} C_i / T_i$ denote the total utilization of the new task set. Now notice that the above conditions are symmetric in the sense that if we multiply 2 to the computation time and period of the first task τ_1, then the utilization of each task (and hence the total utilization of the n tasks) remains unchanged and the new task set still fully utilizes the processor. Therefore, by applying the multiplying rule in no more than n steps, we can arrive at a new task set where T_n is the largest period among all n tasks, with C_n/T_n as the unknown quantity. Furthermore, we can let $u_i = C_i / T_i$ for i = 1, 2, …, m and $u = \sum_{i=m+1}^{n-1} C_i / T_i$.

Case 1: $m = n - 1$. Then

$$U = \sum_{i=1}^{n} C_i / T_i = \sum_{i=1}^{m} u_i + C_n / T_n,$$

where $u_i = C_i / T_i = (T_{i+1} - T_i) / T_i = T_{i+1} / T_i - 1$ for i = 1, 2, …, n – 1.

Let $x_i = T_{i+1} / T_i$ for i = 1, 2, …, n – 1. Then $C_n / T_n = (2T_1 - T_n)/T_n = 2 / \prod_{i=1}^{n-1} x_i - 1$. Since $x_i = u_i + 1$, we have

$$C_n / T_n = 2 / \prod_{i=1}^{n-1} x_i - 1 = 2 \left(\prod_{i=1}^{m} (1 + u_i) \right)^{-1} - 1.$$

A task set that actually meets this condition is given as follows:

Let $T_1 = \sigma > 0$. Then $C_1 = \sigma u_1$. Hence we have $T_{i+1} = (1 + u_i)T_i = \sigma \prod_{j=1}^{i} (1 + u_j)$, and $C_{i+1} = u_{i+1} T_{i+1} = \sigma u_{i+1} \prod_{j=1}^{i} (1 + u_j)$ for $i = 1, 2, \ldots, n-2$. $T_n = \sigma \prod_{j=1}^{n-1} (1 + u_j)$, $C_n = \sigma(2 - \prod_{j=1}^{n-1} (1 + u_j))$. In other words, the task set is given by

$$(C_1, T_1) = (\sigma u_1, \sigma)$$

$$(C_2, T_2) = (\sigma u_2(1 + u_1), \sigma(1 + u_1))$$

$$\ldots$$

$$(C_{n-1}, T_{n-1}) = (\sigma u_{n-1} \prod_{j=1}^{n-2} (1 + u_j), \sigma \prod_{j=1}^{n-2} (1 + u_j))$$

$$(C_n, T_n) = (\sigma(2 - \prod_{j=1}^{n-1} (1 + u_j)), \sigma \prod_{j=1}^{n-1} (1 + u_j)).$$

Case 2: $m < n - 1$. The utilization of task τ_n' is determined at the point where the total utilization is minimized.

$$U = \sum_{i=1}^{n} C_i / T_i = \sum_{i=1}^{m} u_i + u + C_n / T_n,$$

where $u_i = C_i / T_i = (T_{i+1} - T_i) / T_i = T_{i+1} / T_i - 1$ for $i = 1, 2, \ldots, m$ and $u = \sum_{i=m+1}^{n-1} C_i / T_i$.

Let $x_i = T_{i+1} / T_i$ for $i = 1, 2, \ldots, n-1$. Then $C_n / T_n = (2T_1 - T_n)/T_n = 2 / \prod_{i=1}^{n-1} x_i - 1$.

$$U = \sum_{i=1}^{m} u_i + u + 2 / \prod_{i=1}^{n-1} x_i - 1. \qquad \text{(Eq.2.9)}$$

$$u_i = x_i - 1 \text{ for } i = 1, 2, \ldots, m. \qquad \text{(Eq.2.10)}$$

$$u = \sum_{i=m+1}^{n-1} C_i / T_i = \sum_{i=m+1}^{n-1} x_i - (n - m - 1). \qquad \text{(Eq.2.11)}$$

To find the minimum, we need to minimize the expression for U as given in equation (2.9) subject to the side conditions (2.10) and (2.11). This is achieved by first forming the Lagrangian

$$L = U + \sum_{i=1}^{m} \lambda_i (x_i - 1 - u_i) + \lambda \left[\sum_{i=m+1}^{n-1} x_i - (n - m - 1) - u\right]$$

and then minimizing the function L over x_i's, λ_i's, and λ. This can be accomplished by first taking the derivatives of L over x_i's, λ_i's, and λ, respectively, and then solving the resultant equations after setting them to be zero.

$$\frac{\partial L}{\partial x_i} = \lambda_i - \frac{2}{x_i \prod_{j=1}^{n-1} x_j} = 0, \text{ for } i = 1, 2, \ldots, m. \qquad \text{(Eq.2.12)}$$

$$\frac{\partial L}{\partial x_i} = \lambda - \frac{2}{x_i \prod_{j=1}^{n-1} x_j} = 0, \text{ for } i = m+1, m+2, \ldots, n-1. \qquad \text{(Eq.2.13)}$$

$$\frac{\partial L}{\partial \lambda_i} = x_i - 1 - u_i = 0, \text{ for } i = 1, 2, \ldots, m. \qquad \text{(Eq.2.14)}$$

$$\frac{\partial L}{\partial \lambda} = [\sum_{i=m+1}^{n-1} x_i - (n-m-1) - u] = 0. \qquad \text{(Eq.2.15)}$$

In the following we show how the above equations can be solved to obtain the final results.

By multiplying the $(n-1)$ equations in (2.12) and in (2.13) together and manipulating the resultant equation, we get

$$\prod_{j=1}^{n-1} x_j = \frac{2^{(n-1)/n}}{\lambda^{(n-m-1)/n}\left(\prod_{i=1}^{m} \lambda_i\right)^{1/n}}. \qquad \text{(Eq.2.16)}$$

By substituting the $\prod_{j=1}^{n-1} x_j$ in (2.12) and in (2.13) by that in (2.14) and solving for x_i's, we have

$$x_i = \frac{2^{1/n}\lambda^{(n-m-1)/n}\left(\prod_{i=1}^{m} \lambda_i\right)^{1/n}}{\lambda_i}, \text{ for } i = 1, 2, \ldots, m. \qquad \text{(Eq.2.17)}$$

$$x_i = \frac{2^{1/n}\left(\prod_{i=1}^{m} \lambda_i\right)^{1/n}}{\lambda^{(m+1)/n}} \text{ for } i = m+1, m+2, \ldots, n-1. \qquad \text{(Eq.2.18)}$$

Since $x_i = 1 + u_i$, we have for $i = 1, 2, \ldots, m$,

$$1 + u_i = \frac{2^{1/n}\lambda^{(n-m-1)/n}\left(\prod_{i=1}^{m} \lambda_i\right)^{1/n}}{\lambda_i} \qquad \text{(Eq.2.19)}$$

By multiplying the m equations in (2.19) together and manipulating the resultant equation, we get

$$\prod_{i=1}^{m} \lambda_i = 2^{m/(n-m)} \lambda^{(m(n-m-1))/(n-m)} \left[\prod_{i=1}^{m} (1+u_i)\right]^{n/(m-n)} \qquad \text{(Eq.2.20)}$$

Since $\sum_{i=m+1}^{n-1} x_i = (n-m-1) + u = (n-m-1)\dfrac{2^{1/n}\left(\prod_{i=1}^{m} \lambda_i\right)^{1/n}}{\lambda^{(m+1)/n}}$, we have

$$\prod_{i=1}^{m} \lambda_i = \frac{\lambda^{m+1}}{2}[1+u/(n-m-1)]^n \qquad \text{(Eq.2.21)}$$

For convenience, we let $\Delta = 1 + \dfrac{u}{n-m-1}$ and $\nabla = \prod_{i=1}^{m} (1+u_i)$. With equations (2.20) and (2.21) together, we can solve for λ:

$$\lambda = \frac{2}{\nabla\Delta^{n-m}}. \qquad \text{(Eq.2.22)}$$

With equations (2.19), (2.21), and (2.22) together we can solve for λ_i's:

$$\lambda_i = \frac{2}{(1+u_i)\nabla\Delta^{n-m-1}}, \text{ for } i = 1, 2, \ldots, m.$$

Then we have

$$\prod_{i=1}^{m} \lambda_i = \frac{2^m}{\nabla^{m+1}\Delta^{(n-m-1)m}}. \qquad \text{(Eq.2.23)}$$

Now we are ready to solve for x_i's. With equations (2.18), (2.22), and (2.23) together we obtain

$$x_i = \frac{2^{1/n}\left(\prod_{i=1}^{m} \lambda_i\right)^{1/n}}{\lambda^{(m+1)/n}} = \Delta, \text{ for } i = m+1, m+2, \ldots, n-1. \qquad \text{(Eq.2.24)}$$

Since $x_i = 1 + u_i$ for $i = 1, 2, \ldots, m$, we have

$$C_n / T_n = 2 / \prod_{i=1}^{n-1} x_i - 1 = \frac{2}{\nabla\Delta^{n-m-1}} - 1.$$

In other words, if

$$C_n / T_n \le \frac{2}{\left[\prod_{i=1}^{m} (1+u_i)\right][1+u/(n-m-1)]^{n-m-1}} - 1,$$

then the new task set can be feasibly scheduled by the RM algorithm.

If $n \to \infty$, then $2/\left(\nabla\Delta^{n-m-1}\right) \to 2e^{-u}\left[\prod_{i=1}^{m} (1+u_i)\right]^{-1} - 1$.

A task set that actually meets this condition is given as follows:

Let $T_1 = \sigma > 0$. Then $C_1 = \sigma u_1$. Hence we have $T_{i+1} = (1 + u_i)T_i = \sigma\prod_{j=1}^{i}(1+u_j)$, and $C_{i+1} = u_{i+1}T_{i+1} = \sigma u_{i+1}\prod_{j=1}^{i}(1+u_j)$, for $i = 1, 2, \ldots, m-1$. $T_{m+1} = x_m T_m = \sigma\prod_{j=1}^{m}(1+u_j)$. $T_{i+1} = x_i T_i = \Delta T_i = \Delta^{i-m}\sigma\prod_{j=1}^{m}(1+u_j)$, $C_i = T_{i+1} - T_i = (\Delta - 1)\Delta^{i-m}\sigma\prod_{j=1}^{m}(1+u_j)$, for $i = m + 1, m + 2, \ldots, n - 2$. $C_n = \sigma(2 - \prod_{j=1}^{n-1}(1+u_j))$. In other words, with σ as a variable, the task set is given as

$$(C_1, T_1) = (\sigma u_1, \sigma)$$

$$(C_2, T_2) = (\sigma u_2(1+u_1), \sigma(1+u_1))$$

$$\ldots$$

$$(C_{m-1}, T_{m-1}) = (\sigma u_{m-1}\prod_{j=1}^{m-2}(1+u_j), \sigma\prod_{j=1}^{m-2}(1+u_j))$$

$$(C_m, T_m) = (\sigma u_m\prod_{j=1}^{m-1}(1+u_j), \sigma\prod_{j=1}^{m-1}(1+u_j))$$

$$(C_{m+1}, T_{m+1}) = ((\Delta-1)\sigma\prod_{j=1}^{m}(1+u_j), \sigma\prod_{j=1}^{m}(1+u_j))$$

$$\ldots$$

$$(C_{n-2}, T_{n-2}) = ((\Delta-1)\Delta^{n-m-3}\sigma\prod_{j=1}^{m}(1+u_j), \Delta^{n-m-3}\sigma\prod_{j=1}^{m}(1+u_j))$$

$$(C_{n-1}, T_{n-1}) = (\sigma(\nabla\Delta^{n-m-1} - \Delta^{n-m-2}), \Delta^{n-m-2}\sigma\prod_{j=1}^{m}(1+u_j))$$

$$(C_n, T_n) = (\sigma(2 - \nabla\Delta^{n-m-1}), \sigma\Delta^{n-m-1}\nabla).$$ ■

The following theorem is a generalized version of Theorem 2.2 by Dhall.

Theorem 2.8: *Let* $\{\tau_i = (C_i, T_i) \mid i = 1, 2, \ldots, n\}$ *be a set of n tasks and* $u_1 = C_1/T_1$. *If the utilization* $u = \sum_{i=2}^{n} C_i/T_i$ *of the (n – 1) tasks* $\{\tau_i \mid i = 2, 3, \ldots\ldots, n\}$ *is less than or equal to* $(n-1)\{[2/(1+u_1)]^{1/(n-1)} - 1\}$, *then the given set of n tasks can be feasibly scheduled by the RM scheduling algorithm. When* $n \to \infty$, $(n-1)\{[2/(1+u_1)]^{1/(n-1)} - 1\} \to \ln(2/(1+u_1))$.

Proof: This theorem can be proven as the above theorem.

According to Lemma 2.2, we can assume without loss of generality that the task periods satisfy

$$T_i / T_j < 2, \text{ for } i, j = 1, 2, \ldots, n \text{ and } i \neq j.$$

We then sort the tasks in the order of the increasing period and rename them appropriately such that

$$T_1 \leq T_2 \leq \ldots \leq T_{\mathrm{n}} < 2T_1.$$

Then by Lemma 2.3, the minimum utilization is achieved when

$$C_1 = T_2 - T_1,$$

$$C_2 = T_3 - T_2,$$

$$\ldots$$

$$C_{n-1} = T_n - T_{n-1},$$

$$C_n = T_1 - \sum_{i=1}^{n-1} C_i = 2T_1 - T_n.$$

This set of tasks fully utilizes the processor, even though there is an unknown quantity, $u = \sum_{i=2}^{n} C_i/T_i$ in the above conditions that has yet to be determined. Note that the original unknown quantity, C_1 / T_1, corresponds to the quantity of C_i / T_i after renaming, with $i \in [1 \ldots n]$. Let $U = \sum_{i=1}^{n} C_i/T_i$ denote the total utilization of the new task set. Now notice that the above conditions are symmetric in the sense that if we multiply 2 to the computation time and period of the first task τ_1, then the utilization of each task (and hence the total utilization of the *n* tasks) remains unchanged and the new task set still fully utilizes the processor. Therefore, by applying the multiplying rule in no more than *n* steps, we can arrive at a new task set where T_1 is the shortest period among all *n* tasks, with C_1 / T_1 as the known quantity. Furthermore, we can let $u_i = C_i/T_i$ for $i = 2, \ldots, n$.

Then $U = \sum_{i=1}^{n} C_i/T_i = \sum_{i=2}^{n} u_i + C_1 / T_1 = u + u_1$, where $u_i = C_i/T_i = (T_{i+1} - T_i) / T_i = T_{i+1} / T_i - 1$ for $i = 1, 2, \ldots, n - 1$.

Let $x_i = T_{i+1}/T_i$ for $i = 1, 2, \ldots, n - 1$. Then $C_n / T_n = (2T_1 - T_n)/T_n = 2 / \prod_{i=1}^{n-1} x_i - 1$.

Since $x_i = u_i + 1$, we have

$$U = \sum_{i=2}^{n} u_i + u_1 = u_1 + \sum_{i=2}^{n-1} (x_i - 1) + 2 / [(1 + u_1)\prod_{i=2}^{n-1} x_i] - 1$$

We want to find the minimum of $u = \sum_{i=2}^{n} u_i$ with u_1 as a known quantity. This is achieved by finding the minimum of U since $U = \sum_{i=2}^{n} u_i + u_1$.

To find the minimum of U, we use the familiar method of taking the derivative of U over x_i and solving for x_i the resultant equations when they are set to zero.

$$\frac{\partial U}{\partial x_i} = 1 - 2 / [(1+u_1) x_i \prod_{i=2}^{n-1} x_i] = 0, \text{ for } i = 2, \dots, n-1. \qquad \text{(Eq.2.25)}$$

Through some manipulation of equations (2.25) we obtain that

$$x_i = [2 / (1+u_1)]^{1/(n-1)}, \text{ for } i = 2, \dots, n-1. \qquad \text{(Eq.2.26)}$$

Therefore, the minimum of $u = \sum_{i=2}^{n} u_i$ is given by

$$u = \sum_{i=2}^{n-1} (x_i - 1) + 2 / [(1+u_1) \prod_{i=2}^{n-1} x_i] - 1 = (n-1)\{ [2/(1+u_1)]^{1/(n-1)} - 1\}$$

A task set that actually meets the condition is given as follows:

Let $T_1 = \sigma > 0$. Then $C_1 = \sigma u_1$. We also let $\Delta = [2/(1+u_1)]^{1/(n-1)}$. Hence we have $T_{i+1} = x_i T_i = \sigma \prod_{j=1}^{i} x_j = \sigma(1+u_1)\Delta^{i-2}$ and $C_{i+1} = u_{i+1} T_{i+1} = (x_{i+1} - 1) T_{i+1} = \sigma(\Delta - 1)(1+u_1)\Delta^{i-2}$, for $i = 1, \dots, n-1$.

In other words, the task set is given by

$$(C_1, T_1) = (\sigma u_1, \sigma)$$

$$(C_2, T_2) = (\sigma(\Delta - 1)(1+u_1), \sigma(1+u_1))$$

$$\dots$$

$$(C_{n-1}, T_{n-1}) = (\sigma(\Delta-1)(1+u_1)\Delta^{n-3}, \sigma(1+u_1)\Delta^{n-3})$$

$$(C_n, T_n) = (\sigma(\Delta-1)(1+u_1)\Delta^{n-2}, \sigma(1+u_1)\Delta^{n-2}).$$ ■

Note that the only difference between Theorem 2.8 and Theorem 2.3 is that in Theorem 2.8, no restriction is placed on the period of the task whose utilization is known.

2.5. Miscellaneous Schedulability Conditions

In this section, we present two scheduling conditions that explicitly take into account the relation among task periods. Though these results are not used in this thesis, they may be useful elsewhere.

Theorem 2.9: *For a set of two tasks with fixed priority assignment, the least* *upper bound to the processor utilization is* $U = 2(\sqrt{q\,(q+1)} - q)$*, where* $q = \lfloor T_2 / T_1 \rfloor \le 1$*, and* T_1 *and* T_2 *are the periods of the two tasks.*

Proof: Let τ_1 and τ_2 be two tasks with their periods being T_1 and T_2 and their run-time being C_1 and C_2, respectively. Assume that $T_1 \le T_2$ and $T_2 = qT_1 + r$, where $q \ge 1$ and $r \ge 0$. According to the RM algorithm, τ_1 has higher priority than τ_2.

We first claim that the least upper bound of processor utilization is achieved when $C_1 = r$ or $C_1 = T_2 - qT_1$.

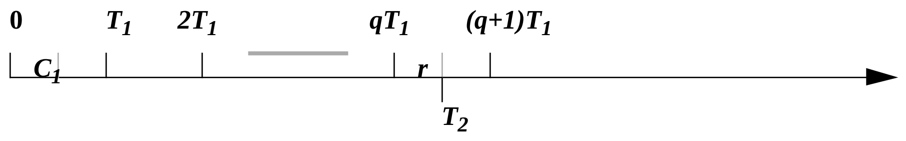


Figure 2.3: Relationship between T_1 and T_2.

Suppose that the two tasks fully utilize the processor with utilization equal to $U = C_1 / T_1 + C_2 / T_2$.

If $C_1 = r + \Delta$, where $0 < \Delta < T_1$, let us replace τ_1 by τ_1' such that $T_1' = T_1$ and $C_1' = r$, and increase C_2 by the amount of $q\Delta$ needed to again fully utilize the processor. This increase is the time within the critical zone $[0, T_2]$ of τ_2 occupied by τ_1 but not by τ_2. Let U' be the total utilization of such a set of tasks. We have

$$U - U' = \Delta / T_1 - q\Delta / T_2 = \Delta / T_1T_2\,(T_2 - qT_1) \ge 0.$$

Therefore, $U \ge U'$.

If $C_1 = r - \Delta$, where $0 < \Delta < T_1$, let us replace τ_1 by τ_1' such that $T_1' = T_1$ and $C_1' = r$, and decrease C_2 by the amount of $(q+1)\Delta$ needed to again fully utilize the processor.This decrease is the time within the critical zone $[0, T_2]$ of τ_2 not occupied by τ_1 but by τ_2. Let U' be the total utilization of such a set of tasks. We have

$$U - U' = -\Delta / T_1 + (q+1)\Delta / T_2 = \Delta / T_1T_2\,((q+1)T_1 - T_2) \ge 0.$$

Therefore, $U \ge U'$.

Hence the least upper bound of processor utilization is achieved when

$$U = C_1 / T_1 + C_2 / T_2 = C_1 / T_1 + q(T_1 - C_1) / (qT_1 + C_1)$$

Let $x = C_1 / T_1$, then $U = x + q(1 - x) / (q + x)$. To minimize U, we set the first derivative of U with respect to x equal to zero and solve the resultant difference equation for x. We get $x = \sqrt{q(q+1)} - q$. $U = 2(\sqrt{q(q+1)} - q)$. ■

Theorem 2.10: *For a set of three tasks with fixed priority assignment, the least upper bound to the processor utilization factor is* $U = (aq^2 + b^2) / qr$, *where* $T_2 = qT_1$, $T_3 = rT_1$, $r = aq + b$, *and* T_1, T_2, *and* T_3 *are the periods of the three tasks, with* $T_1 \le T_2 \le T_3$.

Proof: Since $T_1 \le T_2 \le T_3$, $T_2 = qT_1$, $T_3 = rT_1$, $r = aq + b$, we have $a \ge 1, b \ge 0.$

Normalize the periods of the tasks by letting $T_1 = 1$. The total utilization of the three tasks are $U = C_1 / T_1 + C_2 / T_2 + C_3 / T_3 = C_1 + C_2 / q + C_3 / r$. There are two cases to consider:

Case 1: $bC_1 + C_2 > b$. Since $bC_1 + C_2 > b$, $C_3 = aq - aC_2 - aqC_1$.

$U = C_1 + C_2 / q + (aq - aC_2 - aqC_1) / r$

$= aq / r + (r - aq)C_2 / qr + (r - aq)C_1 / r$

$> aq / r + (r - aq)(b - bC_1)/ qr + (r - aq)C_1 / r$

$= (aq^2 + b^2) / qr + (r - aq)(q - b)\, C_1 / qr.$

Since $r - aq = b \ge 0$ and $q - b > 0$, U is minimized when $C_1 \to 0$. Then $U = (aq^2 + b^2) / qr.$

Case 2: $bC_1 + C_2 \le b$. Since $bC_1 + C_2 > b$, we have $C_3 = r - rC_1 - (a + 1)C_2$.

$U = C_1 + C_2 / q + (r - rC_1 - (a + 1)C_2) / r$

$= 1 + (b - q)C_2 / qr$

Since $bC_1 + C_2 \le b$, then $C_2 \le b$. U is minimized when $C_2 = b$, since $b - q < 0$. $U = (aq^2 + b^2) / qr.$ ■

Chapter 3 Rate-Monotonic Scheduling on a Multiprocessor System

"Try your best."
-- Common Sense

In this chapter, we first present some results that are fundamental to rate-monotonic scheduling on a multiprocessor system. Then we embark on our search for the best heuristic algorithms for rate-monotonic scheduling on a multiprocessor system, in terms of worst case performance. At the end of this chapter, we present simulation results on the average case behavior of various algorithms.

3.1. Fundamental Results of RM Scheduling on Multiprocessor

Theorem 3.1: *If a set of tasks* $\Sigma = \{\tau_i = (C_i, T_i) \mid i = 1, 2, \ldots, n\}$ *cannot be scheduled on N processors, then the set of tasks* $\Sigma = \{\tau_i' = (C_i', T_i') \mid i = 1, 2, \ldots, n\}$ *given by*

$$C_i' = T_i' \bullet C_i / T_i,\ T_i' = 2^{V_i},\ \textit{and}\ V_i = \log_2 T_i - \lfloor \log_2 T_i \rfloor$$

cannot be scheduled on the N processors either.

Proof: Let us define $T_{max} = \max_i (T_i)$. For the task set Σ, we select $T_{min} = \min_i (T_i)$ and replace the task $\tau_{min} = (C_{min}, T_{min})$ by a new task $(2C_{min}, 2T_{min})$ if $2T_{min} < T_{max}$. Clearly the resulting task set due to this replacement cannot be scheduled on N processors by Lemma 2.2. Furthermore, the utilization of the task τ_{min} is not changed. We repeat this process until we arrive at a task set such that $2T_{min} > T_{max}$.

Since scaling a task set by any positive number does not change its schedulability according to the necessary and sufficient condition, we replace every task $\tau_i = (C_i, T_i)$ by a task $(2^{-\lfloor \log_2 T_i \rfloor} C_i, 2^{-\lfloor \log_2 T_i \rfloor} T_i)$ in the task set above. Then we have arrived at a task set that was to be obtained. ■

Next we answer the question of what the minimum utilization of a set of *n* tasks is such that any set of *n* tasks with a smaller utilization is guaranteed to be feasible on $n - 1$ processors by the RM algorithm. This question is answered in the following two theorems. The first theorem, first outlined and partially proven by Dhall in his thesis [19], is the key to the proof of the second theorem. The proof of these two theorems relies heavily on Theorem 3.1.

Theorem 3.2: *If a set of* $n > 1$ *tasks, each with a utilization less than* 1/2, *cannot be feasibly scheduled on* $n - 1$ *processors by the RM algorithm, then the total utilization of the set must be greater than* $n/\left(1 + 2^{1/n}\right)$.

Proof: Let the set of *n* tasks be $\Sigma = \{\tau_i = (C_i, T_i) \mid i = 1, 2, \ldots, n\}$ and $u_i = C_i/T_i < 1/2$.

According to Theorem 3.1, we can assume without loss of generality that

$$T_1 \le T_2 \le \ldots \le T_n < 2T_1 \qquad \text{(Eq.3.1)}$$

Since no two of the *n* tasks can be feasibly scheduled together, the following conditions must hold according to the IFF condition.

$$\begin{cases} C_i + C_j > T_i \\ 2C_i + C_j > T_j \end{cases} \qquad 1 \le i < j \le n \qquad \text{(Eq.3.2)}$$

Furthermore, since $u_i < 1/2$, by (3.1), (3.2) we have

$$T_1 < T_2 < \ldots < T_n < 2T_1 \qquad \text{(Eq.3.3)}$$

We want to find the minimum of $U = \sum_{i=1}^{n} C_i/T_i$ subject to the constraints of (3.2), (3.3), and (3.4).

$$C_i/T_i < 1/2 \qquad \text{(Eq.3.4)}$$

In order to ensure that the minimum is obtained at some point, we replace “>” by “≥” in (3.2). This replacement will not affect the minimum.

We proceed in three steps to obtain the minimum of U:

(1) Fix the values $\bar{C} = (C_1, C_2, \ldots, C_n)$ and express $\bar{T} = (T_1, T_2, \ldots, T_n)$ in terms of $\bar{C}$ in the minimization problem.

(2) Prove that $C_1 < C_2 < \ldots < C_n < 2C_1$ if the minimum is achieved and reduce the minimization problem to a convex minimization problem.

(3) Solve the minimization problem using standard methods.

First, let us assume that $\bar{C} = (C_1, C_2, \ldots, C_n)$ is fixed.

Since

$$\frac{\partial U}{\partial T_i} = -\frac{C_i}{T_i^2}, \qquad \text{(Eq.3.5)}$$

U decreases as we increase T_i. But the increase of T_i cannot exceed the limit that is imposed by the constraints in (3.2). In other words, U is minimized when

$$T_i = min\{2C_1, \ldots, 2C_{i-1}, C_{i+1}, \ldots, C_n\} + C_i . \qquad \text{(Eq.3.6)}$$

for $i = 1, 2, \ldots, n$. Let $m_i = min\{2C_1, \ldots, 2C_{i-1}, C_{i+1}, \ldots, C_n\}$. Then the minimization problem becomes

$$U = \sum_{i=1}^{n} C_i / T_i = \sum_{i=1}^{n} C_i / (C_i + m_i) .$$

Next we claim that $C_1 < C_2 < \ldots < C_n < 2C_1$ if the minimum is achieved.

Suppose that the above claim is false. Then we have either $C_i \geq C_{i+1}$ or $2C_1 \geq C_n$. We will only present the proof to the case of $C_i \geq C_{i+1}$, since the proof to the case of $2C_1 \geq C_n$ is completely analogous.

If $C_i \geq C_{i+1}$, then we have by the constraints in (3.2) that

$$2C_i \geq C_i + C_{i+1} \geq T_i.$$

Hence $u_i = C_i / T_i \geq 1/2$, which is a contradiction to (3.4).

Therefore, the minimum is achieved when

$$m_i = C_{i+1},$$
$$m_n = 2C_1,$$

for $i = 1, 2, \ldots, n-1$.

Accordingly, the minimum of $U = \sum_{i=1}^{n} C_i / T_i$ is achieved when

$$T_i = C_i + m_i = C_i + C_{i+1},$$
$$T_n = C_n + m_n = C_n + 2C_1,$$

for $i = 1, 2, \ldots, n-1$

This becomes a convex minimization problem.

Finally, we solve the problem by using one of the standard methods.

$$\begin{aligned} \mathrm{U} &= \sum_{i=1}^{n-1} C_i / (C_i + C_{i+1}) + C_n / (C_n + 2C_1) \\ &= \sum_{i=1}^{n-1} 1 / (1 + b_i) + \prod_{i=1}^{n-1} b_i / (2 + \prod_{i=1}^{n-1} b_i) \\ &= \sum_{i=1}^{n-1} 1 / (1 + b_i) + 1 - 2 / (2 + \prod_{i=1}^{n-1} b_i), \end{aligned}$$

where $b_i = C_{i+1} / C_i$ for $i = 1, 2, \ldots, m-1$ and $2C_1 / C_n = 2 / \prod_{i=1}^{n-1} b_i$.

$$\partial \mathrm{U} / \partial b_i = -1 / (1 + b_i)^2 + 2 \prod_{j=1, j \neq i}^{n-1} b_i / (2 + \prod_{i=1}^{n-1} b_i)^2 = 0.$$

$$2 \prod_{j=1, j \neq i}^{n-1} b_i \, (1 + b_i)^2 = (2 + \prod_{i=1}^{n-1} b_i)^2 \qquad \text{(Eq.3.7)}$$

for $i = 1, 2, \ldots, n-1$. Then we have

$$b_j (1 + b_i)^2 = b_i (1 + b_j)^2.$$

Solving these equations yields $b_j = b_i = 2^{1/n}$.

Therefore, $U = n / \left(1 + 2^{1/n} \right)$.

This bound is tight in the sense that there are in fact some task sets that actually meet this condition. One of such task sets is given as follows:

Let $\varphi > 0$ and $C_i = \varphi 2^{i/n}$. Then $T_i = \varphi 2^{i/n} (1 + 2^{1/n})$ for $i = 1, 2, \ldots, n$. ■

Next we prove that the result given in Theorem 3.2 holds for any task set.

Theorem 3.3: *If a set of $n > 1$ tasks cannot be feasibly scheduled on $n - 1$ processors by the RM algorithm, then the total utilization of the set must be greater than* $n / \left(1 + 2^{1/n} \right)$.

Proof: In Theorem 3.2, we have proven that if the utilization of each of the n tasks is less than 1/2, then the total utilization must be greater than $n/\left(1+2^{1/n}\right)$.

We will prove that the bound of $n/\left(1+2^{1/n}\right)$ is indeed the minimum for any task set by method of contradiction.

Suppose that a lower bound than $n/\left(1+2^{1/n}\right)$ is achieved with a set of n tasks, in which $0 < k < n$ of them have utilizations equal to or greater than 1/2. Then there are $n - k$ tasks each of whose utilizations is less than 1/2.

Since the $n - k$ tasks cannot be scheduled on $n - k - 1$ processors and each of them has a utilization less than 1/2, we have $\sum_{i=1}^{n-k} u_i > (n-k)/\left(1+2^{1/(n-k)}\right)$ by Theorem 3.2.

Since $\sum_{i=1}^{n} u_i \geq (n-k)/\left(1+2^{1/(n-k)}\right) + k/2 > n/\left(1+2^{1/n}\right)$, it contradicts the assumption that a lower bound is achieved when some of the task utilizations is equal to or greater than 1/2. Therefore, the theorem must be true. ■

3.2. Scheduling Heuristics and Their Worst Case Performance Analysis

With the task model as described in Chapter 2, the problem of scheduling a set of periodic tasks on a multiprocessor system such that task deadlines are met on each processor by the RM algorithm can be described as follows:

Given a set of n tasks $\Sigma = \{\tau_i = (C_i, T_i) \mid i = 1, 2, \ldots\ldots, n\}$, what is the minimum number of processors required to execute the n tasks such that their deadlines are met by the RM algorithm on each individual processor?

Various scheduling heuristics can be developed to solve this problem. One set of heuristic algorithms to solve this problem can be formed by combining any of the bin-packing heuristics with any of the schedulability conditions for the RM algorithm. This set of algorithms can be described as follows:

$$\mathrm{A} = \{NF, FF, BF, FFD, BFD, \ldots\} \times \{WC, IFF, IP, UO, \ldots\},$$

where *NF, FF, BF, FFD, BFD* are bin-packing heuristics and *WC, IFF, IP, UO* are the schedulability conditions for the RM algorithm.

For example, the Rate-Monotonic-Next-Fit (RMNF) algorithm proposed by Dhall and Liu can be categorized as RMNF-IP and their Rate-Monotonic-First-Fit (RMFF) as RMFF-IP, since the IP condition is used.

In order to show how the worst case performance bound is generally derived for the scheduling heuristic algorithms, we present an algorithm called the Rate-Monotonic-Next-Fit-WC (RM-NF-WC) using the WC condition for the RMMS problem.

The RM-NF-WC algorithm is given in Figure 3.1.

Rate-Monotonic-Next-Fit-WC (RM-NF-WC) (Input: task set Σ; Output: *m*)

```
(1). i := 1; m := 1; /* i denotes the ith task, m the number of
     processors allocated */
(2). Assign task $\tau_i$ to processor $P_m$ if this task together with the
     tasks that have been assigned to $P_m$ can be feasibly scheduled
     on $P_m$ according to the WC condition. If not, assign task $\tau_i$ to
     $P_{m+1}$ and set m = m + 1.
(3). If i < n, then i := i + 1 and go to (2) else stop.
```

Figure 3.1: Algorithm RM-FF-WC

When the algorithm finishes, *m* is the number of processors required to execute a given set of tasks.

If we let *N* and N_0 be the number of processors required by RM-NF-WC and the minimum number of processors required to feasibly schedule a given set of tasks, respectively, then we want to find the worst case performance bound of RM-NF-WC, i.e., $\Re^{\infty}_{RM-NF-WC}$. If we can prove that $\Re^{\infty}_{RM-NF-WC} \leq \alpha$, then we say that the performance of RM-NF-WC is upper bounded by α. If we show that $\Re^{\infty}_{RM-NF-WC} \geq \alpha$, usually by constructing some task sets, then we say that the performance of RM-NF-WC is lower bounded by α. If we prove both conditions, then we can conclude that $\Re^{\infty}_{RM-NF-WC} = \alpha$, i.e, the algorithm RM-NF-WC has a tight bound of α.

The general approach to prove the tight bound for a heuristic algorithm involves two procedures that must be performed simultaneously: on the one hand, we should try to find patterns of input that will result in the worst case performance for the algorithm. On the other hand, we should try to lower the upper bound by analytic reasoning. There may be many alternations between the two procedures before the final tight bound is obtained. The lack of either effort will make a solution incomplete. Although this process of obtaining the tight bound for an algorithm may be a long one, as it is the case for many of our algorithms, we will not describe the process in this thesis, since finding a bound is one matter; presenting it may be an entirely different matter.

Next we show that the worst case performance of RM-NF-WC is upper bounded by 2 / ln2. Then for any number of processors in an optimal schedule, a task set is constructed that results in the nearly upper bounded number of processors required by RM-NF-WC.

Theorem 3.4: *Let N and N_0 be the number of processors required by RM-NF-WC and the minimum number of processors required to feasibly schedule a given set of tasks, respectively. Then* $N \le (2/(ln2))N_0 + 1 \approx 2.88N_0 + 1$.

Proof: For a processor P_j, let $\tau_1, \tau_2, \ldots, \tau_s$ be the tasks that have been assigned to it and τ_{s+1} be the first task assigned to processor P_{j+1}. According to the WC condition, we have

$$\sum_{k=1}^{s} u_k + u_{s+1} > (s+1)\left[2^{1/(s+1)} - 1\right] > \ln 2. \qquad \text{(Eq.3.8)}$$

Let $U_j = \sum_{k=1}^{s} u_k$ for $1 \le j \le N$.

Since $U_{j+1} \ge u_{s+1}$ for $1 \le j \le N-1$, we have

$$U_j + U_{j+1} > \ln 2 \qquad \text{(Eq.3.9)}$$

from inequality (3.8).

Summing up the $N-1$ inequalities in (3.9) yields $2\sum_{j=1}^{N} U_j - U_1 - U_N > (N-1)$ ln2. In other words, $2\sum_{j=1}^{N} U_j > (N-1)\ln 2 + U_1 + U_N > (N-1)\ln 2$.

Since $N_0 \ge \sum_{j=1}^{N} U_j$, we have $N \le (2/(\ln 2))N_0 + 1$.

Therefore, $\Re^{\infty}_{RM-NF-WC} \le 2 / \ln 2$. ■

Theorem 3.5: *Let N and N_0 be the number of processors required by RM-NF-*

WC and the minimum number of processors required to feasibly schedule a given set of tasks, respectively. Then $\Re^{\infty}_{RM-NF-WC} \geq 2.87$.

Proof: Let K be a positive integer divisible by 7, i.e., $K = 7m$, where m is a natural number and let δ be a very small positive number such that $\delta = n\varepsilon$, where n is a very large positive integer and ε is a very small positive number. The relationship between n and ε is given as follows: given any small number δ, n is chosen large enough and ε small enough such that $\ln 2 + n\varepsilon \geq n\left(2^{1/n} - 1\right)$ and $\delta = n\varepsilon$.

The set of tasks consists of two sets of task groups, with the numbers of groups equal to $20K/7$ in the first set, and $\lfloor((14K)/7)/20\rfloor$ in the second set, where $\alpha = 1 - 5(\ln 2 - 1/2) = 0.034264$. In terms of m, the number of task groups in the first set is given by $20m$ and the number of task groups in the second set is given by $\lfloor(2m)/20\rfloor$. In the first set of task groups, it consists of $10m$ pairs of task groups, each of which has $(n + 1)$ tasks. Note that in the (x, y) notation, x and y denote the computation time and the period of a task, respectively. A pair of task groups is given by

$$10m\begin{cases}(\ln 2 - 1/2, 1),\ \underbrace{(\varepsilon, 1), \ldots, (\varepsilon, 1)}_{n} \\ (1/2, 1),\ \underbrace{(\varepsilon, 1), \ldots, (\varepsilon, 1)}_{n}\end{cases}$$

In the second set of groups, it has $\lfloor(2m)/20\rfloor$ groups, each of which has 20 tasks, as given by

$$\lfloor(2m)/20\rfloor\left\{\ \underbrace{(\alpha - 10\delta, 1), \ldots, (\alpha - 10\delta, 1)}_{20}\right.$$

In the RM-NF-WC schedule, the first set of task groups uses $20m$ processors, since $\ln 2 - 1/2 + n\varepsilon + 1/2 > n\left(2^{1/n} - 1\right)$, as illustrated by Figure 3.2. The second set of task groups uses $\lfloor(2m)/20\rfloor$ processors in total, since $20(\alpha - 10\delta) + (\alpha - 10\delta) \approx 0.719 - 210\delta > 20\left(2^{1/21} - 1\right) \approx 0.705$, for small δ.

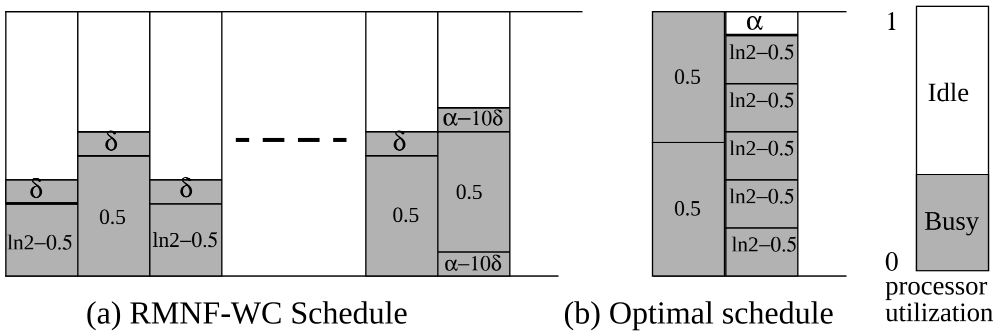


Figure 3.2: RM-NF-WC vs. Optimal Schedules

In the optimal schedule, the $10m$ tasks each with utilization factor of 1/2 can be scheduled using $5m$ processors. The $10m$ tasks each with utilization factor of ln2 and the $20mn$ tasks each with utilization factor of ε can be scheduled on $2m$ processors, with a total utilization of $2m(\alpha - 10\delta)$ left unused for the $2m$ processors. This amount of utilization, i.e., $2m(\alpha - 10\delta)$, is used to execute the task groups in the second set, since $\lfloor (2m)/20 \rfloor$ $(\alpha - 10\delta)\,20 < 2m(\alpha - 10\delta)$.

Therefore, the total number of processors required by an optimal algorithm is N_0 = $5m + 2m = 7m$, while the total number of processors required by RM-NF-WC is $N = 20m$ $+ \lfloor (2m)/20 \rfloor$. The performance bound is thus given by

$$\frac{N}{N_0} = \frac{20m + \lfloor 2m/20 \rfloor}{7m} \geq 2.87, \text{ for } m \to \infty .$$

Hence $\Re^{\infty}_{RM-NF-WC} \geq 2.87$.

Since $\Re^{\infty}_{RM-NF-WC} \leq 2.88$, from Theorem 3.4, it is concluded that the bound is nearly tight. ■

3.3. Rate-Monotonic-First-Fit

In assigning tasks to processors, RM-NF-WC only checks the current processor to see whether a task together with those tasks that have already been assigned on it can be feasibly scheduled or not. If not, the task must be scheduled on an idle processor, even though the task may be scheduled on those processors previously used. To overcome this waste of processor utilization, we develop an allocation algorithm, RMFF, which always checks the feasibility of a task from the first processor to the one on which the task can be scheduled. Note that we will instead use the new condition, the UO condition in RM-FF.

The Rate-Monotonic-First-Fit (RM-FF) is designed as follows: let the processors be indexed as $P_1, P_2, \ldots$, with each one initially in the idle state, i.e., with zero utilization. The tasks $\tau_1, \tau_2, \ldots, \tau_n$ will be scheduled in that order. To schedule τ_i, find the least j such that task τ_i, together with all the tasks that have been assigned to processor P_j, can be feasibly scheduled according to the UO condition for a single processor, and assign task τ_i to P_j.

To describe RM-FF in a more algorithmic format, let k_j and U_j denote the number of tasks that have already been assigned to processor P_j so far and the total utilization of the k_j tasks, respectively. Note that u_i denotes the utilization of task τ_i and $u_{i,j}$ denotes the utilization of the jth task assigned to processor P_i. The RM-FF algorithm is given in Figure 3.3.

Rate-Monotonic-First-Fit (**RM-FF**) (Input: task set Σ; Output: m)

```
(1) i := 1; m := 1;
(2) j := 1; While $(u_i > 2/\left(\prod_{l=1}^{k_j}(u_{j,l}+1)\right) - 1)$ Do {j := j + 1;};
(3) $k_j := k_j + 1$; $U_j := U_j + u_i$; /* Assign task $\tau_i$ to $P_j$ */
(4) If (j > m) Then {m := j;}
(5) i := i + 1;
(6) If (i > n) Then {Stop;} Else {Goto 2;}
```

Figure 3.3: Algorithm RM-FF

When the algorithm terminates, m is the number of processors required by RM-FF to schedule the given set of tasks. The RM-FF algorithm has the following distinguished property: No incoming task is assigned to an idle processor unless it cannot be assigned to any processor that has already been assigned some tasks. We will implicitly use this property throughout the analysis. In order to obtain the worst case bound, we need some lemmas.

Lemma 3.1: *In the completed RM-FF schedule, if n tasks cannot be feasibly scheduled on $n-1$ processors, then the total utilization of the n tasks is greater than*

$n/\left(1+2^{1/2}\right) = n\left(2^{1/2}-1\right)$.

Proof: The proof is by induction.

(1) $n = 2$. Suppose u_1 and u_2 are the utilizations of two tasks which cannot be scheduled on a processor according to the UO condition, i.e., $u_2 > 2(1+u_1)^{-1} - 1$. $u_1 + u_2 = u_1 + 2(1+u_1)^{-1} - 1$. To find the minimum of $f(u_1) = u_1 + 2(1+u_1)^{-1} - 1$, we take the derivative of the function $f(u_1)$, and solve for u_1 after setting the resultant equation to zero. The minimum of $f(u_1)$ is achieved when $u_1 = 2^{1/2} - 1$. Therefore $u_1 + u_2 > 2(2^{1/2} - 1)$.

(2) Suppose the lemma is true for $n = k$, i.e.,

$$\sum_{i=1}^{k} u_i > k\left(2^{1/2}-1\right). \tag{Eq.3.10}$$

Then when $n = k+1$, the $(k+1)$th task cannot be scheduled to any of the k processors. According to RM-FF,

$$u_i + u_{k+1} > 2\left(2^{1/2}-1\right) \text{ for } 1 \le i \le k. \tag{Eq.3.11}$$

Summing up the above k inequalities yields

$$\sum_{i=1}^{k} u_i + k u_{k+1} > 2k\left(2^{1/2}-1\right) \tag{Eq.3.12}$$

Multiplying $k - 1$ on both sides of inequality (3.10) yields

$$(k-1)\sum_{i=1}^{k} u_i > (k-1)\,k\left(2^{1/2}-1\right) \tag{Eq.3.13}$$

Adding up inequalities (3.12) and (3.13) and dividing the new inequality on both sides by k yields $\sum_{i=1}^{k+1} u_i > (k+1)\left(2^{1/2}-1\right)$. Therefore the lemma follows. ■

Lemma 3.2: *In the completed RM-FF schedule, among all the processors on which $n \ge c \ge 1$ tasks are assigned, there is at most one processor with a utilization no greater than $c\left(2^{1/(c+1)}-1\right)$.*

Proof: The lemma is proven by contradiction. Suppose there are two processors each of which has a utilization no greater than $c\left(2^{1/(c+1)}-1\right)$, and let P_a and P_b be the two processors and n_i be the number of tasks assigned to processor P_i with $n_i \ge c$, $a < b$, and $i = a, b$. Let $u_{i,j}$ be the utilization of the jth task that is assigned to processor P_i for $i =$

a, b and $1 \le j \le n_i$. Then $\sum_{j=1}^{n_i} u_{i,j} \le c\left(2^{1/(c+1)} - 1\right)$ for $i = a, b$.

If there exists a number x such that $1 \le x \le n_b$ and $u_{b,x} \ge \left(2^{1/(c+1)} - 1\right)$, then there must exist a task with a utilization $u_{b,y} \le \left(2^{1/(c+1)} - 1\right)$, since there are totally $n_b \ge c$ tasks on processor P_b and $\sum_{j=1}^{n_b} u_{b,j} \le c\left(2^{1/(c+1)} - 1\right)$, where $x \ne y$ and $1 \le y \le n_b$. In other words, there exists a task $\tau_{b,z}$ on processor P_b such that $u_{b,z} \le \left(2^{1/(c+1)} - 1\right)$ and $z \in \{1, 2, \ldots, n_b\}$.

Since $\sum_{j=1}^{n_a} u_{i,j} + u_{b,z} \le c\left(2^{1/(c+1)} - 1\right) + \left(2^{1/(c+1)} - 1\right) = (c+1)\left(2^{1/(c+1)} - 1\right)$, $u_{b,z}$ can be assigned on processor P_a. This is a contradiction to the way RM-FF assigns tasks to processors. Therefore the lemma must be true. ■

Theorem 3.6: *Let N and N_0 be the number of processors required by RM-FF and the minimum number of processors required to schedule a given set of tasks, respectively. Then* $N \le [2 + \left(3 - 2^{3/2}\right)/\left(2^{4/3} - 2\right)]N_0 + 1 \approx 2.33N_0 + 1$. $\Re_{RM-FF}^{\infty} \le 2.33$.

In order to prove the above bound, we define a weighting function that maps the utilization of a task to a value in the interval of (0, 1] as follows:

$$W(u) = \begin{cases} u/a & 0 < u < a \\ 1 & a \le u \le 1 \end{cases}, \text{ where } a = 2\left(2^{1/3} - 1\right).$$

We call that value the weight of the task and the sum of the weights of the tasks assigned to a processor the weight of the processor. The weighting function is designed in such a way that except for some bounded number of processors, the weight of every processor in the RM-FF schedule is equal to or greater than 1. At the meantime, the weight of a processor in the optimal schedule is no greater than $1/a$. We first prove that weight of a processor in the optimal schedule is no greater than $1/a$. Then we prove that with few exceptions, the weight of every processor P in the completed RM-FF schedule is equal to or greater than 1, i.e., $W(P) = \sum_{i=1}^{k} W(u_i) \ge 1$, where k is the number of tasks assigned to the processor.

Lemma 3.3: *If a processor is assigned a number of tasks* $\tau_1, \tau_2, \ldots, \tau_m$, *with utilizations* $u_1 \ge u_2 \ge \ldots \ge u_m$, *then* $\sum_{i=1}^{m} W(u_i) \le 1/a$, *where* $a = 2\left(2^{1/3} - 1\right)$.

Proof: If $u_1 \ge a$, then $u_2 < a$, since $a \approx 0.52$. $\sum_{i=1}^{m} W(u_i) = W(u_1) + \sum_{i=2}^{m} W(u_i)$

$= 1 + (\sum_{i=2}^{m} u_i) / a \le 1 + (1 - a) / a = 1 / a$. Otherwise ($u_1 < a$), then $\sum_{i=1}^{m} W(u_i) = \sum_{i=1}^{m} u_i / a \le 1 / a$. ■

Lemma 3.4: *Suppose tasks are assigned to processors according to RM-FF. If a processor is assigned $m \ge 2$ tasks and $\sum_{i=1}^{m} u_i \ge 2\left(2^{1/3} - 1\right)$, then $\sum_{i=1}^{m} W(u_i) \ge 1$, where $u_1 \ge u_2 \ge \ldots \ge u_m$ are utilizations of the m tasks $\tau_1, \tau_2, \ldots, \tau_m$ that are assigned to it.*

Proof: Since $\sum_{i=1}^{m} u_i \ge 2\left(2^{1/3} - 1\right)$, we either have $u_1 \ge a$ or $u_1 < a$. If $u_1 \ge a$, then $\sum_{i=1}^{m} W(u_i) \ge 1$ according to the definition of weighting function. Otherwise, $\sum_{i=1}^{m} W(u_i) = \sum_{i=1}^{m} u_i / a > 1$, since $a = 2\left(2^{1/3} - 1\right)$. ■

Proof of Theorem 3.6: Let $\Sigma = \{\tau_1, \tau_2, \ldots, \tau_m\}$ be a set of m tasks, with their utilizations $u_1, u_2, \ldots, u_m$ and $\varpi = \sum_{i=1}^{m} W(u_i)$. By Lemma 3.3, $\varpi \le N_0 / a$, where $a = 2\left(2^{1/3} - 1\right)$.

Suppose that among the N processors that are used by RM-FF to schedule a given set Σ of tasks, L of them have $\sum_j W(u_j) = 1 - \beta_i$ with $\beta_i > 0$, where j ranges over all tasks in processor i among the L processors. Let us divide these processors into two different classes:

(1) Processors to each of which only one task is assigned. Suppose there are n_1 of them.

(2) Processors to each of which two or more tasks are assigned. Let n_2 denote the number of processors in this class. According to Lemma 3.2, there is at most one processor whose utilization in the RM-FF schedule is no greater than $a = 2\left(2^{1/3} - 1\right)$. Therefore $n_2 \le 1$.

Obviously, $L = n_1 + n_2$. For each of the rest $N - L$ processors, $\sum_j W(u_j) \ge 1$, where j ranges over all tasks in a processor.

For the processors in class (1), $\sum_{i=1}^{n_1} u_i > n_1(2^{1/2} - 1)$ according to Lemma 3.1. Since $\sum_{i=1}^{n_1} W(u_i) < 1$, we must have $u_i < a$. Hence $\sum_{i=1}^{n_1} W(u_i) > n_1(2^{1/2} - 1) / a$. Moreover, according to Lemma 3.2, there is at most one task whose utilization is no greater than $(2^{1/2} - 1)$. In the optimal assignment of these tasks, the optimal number N_0 of processors used cannot be less than n_1 /2, i.e., $N_0 \geq n_1/2$, since possibly with one exception, any three tasks among these tasks cannot be scheduled on one processor.

Now we are ready to find out the relationship between N and N_0.

$\varpi = \sum_{i=1}^{m} W(u_i) \geq (N - L) + n_1 (2^{1/2} - 1) / a = N - n_1 - n_2 + n_1 (2^{1/2} - 1) / a$

$= N - n_1(1 - (2^{1/2} - 1) / a) - n_2$

$\geq N - 2N_0(1 - (2^{1/2} - 1) / a) - n_2$, where $a = 2\left(2^{1/3} - 1\right)$.

Since $\varpi \leq N_0 / a$ by Lemma 3.3,

$N_0 / a \geq N - 2N_0(1 - (2^{1/2} - 1) / a) - n_2 \geq N - 2N_0(1 - (2^{1/2} - 1) / a) - 1$.

Therefore, $N \leq [2 + (3 - 2^{3/2}) / a]\, N_0 + 1$. ■

Having proven the upper bound for RM-FF, we construct a task set which, when scheduled by RM-FF, requires nearly the same upper bounded number of processors. This theorem also serves as a counter example for the claim that Dhall and Liu's RMFF is upper bounded by 2.23 in [20]. Since the tasks in each group has the same utilization and to show the incorrectness of the upper bound for RMFF in [20], we use the condition, $C_n / T_n \leq 2\,[1 + u/(n-1)]^{-(n-1)} - 1$, instead of the condition $C_n / T_n \leq 2/\left(\prod_{l=1}^{n-1} (u_l + 1)\right) - 1$, without affecting the final result.

Theorem 3.7: *Let N and N_0 be the number of processors required by RM-FF and the minimum number of processors required to feasibly schedule a given set of tasks, respectively. Then*

$$\Re_{RM-FF}^{\infty} \geq 2.2833 \ldots$$

Proof: The proof consists of constructing a task set such that RM-FF exhibits the worst case performance when it is used to schedule the task set.

Let $n = 120k$ with $k \geq 1$. The task set is given as follows: There are n tasks, each with a utilization of $u_1 = 2^{1/31} - 1 + \varepsilon$, where $\varepsilon > 0$ is a small number. For our construction, it suffices to let $\varepsilon < 0.00006$. Next come n tasks, each with a utilization of $u_2 = 2^{1/5} - 1 + \varepsilon$. Finally, there are $2n$ tasks, each with a utilization of $u_3 = 2^{1/2} - 1 + \varepsilon$.

When this task set is scheduled by RM-FF, the first n tasks will use $\lfloor n/30 \rfloor$ processors, since $2^{1/31} - 1 + \varepsilon > 2\left(1 + \left(30\left(2^{1/31} - 1 + \varepsilon\right)\right)/30\right)^{-30} - 1$. The next n tasks will take up $\lfloor n/4 \rfloor$ processors, since $2^{1/5} - 1 + \varepsilon > 2\left(1 + \left(4\left(2^{1/5} - 1 + \varepsilon\right)\right)/4\right)^{-4} - 1$. The last $2n$ tasks will take up $2n$ processors for similar reason. Thus, the total number of processors allocated for this task set by RM-FF is $N = 2n + \lfloor n/4 \rfloor + \lfloor n/30 \rfloor = 274k$.

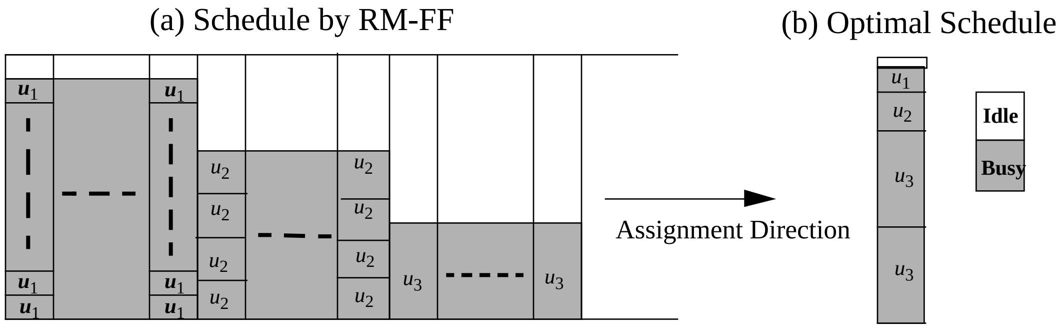


Figure 3.4: RM-FF vs. Optimal Schedules

For the optimal schedule, each processor is assigned four tasks and totally n processors are required. For each processor, it is assigned two tasks each with a utilization of $(2^{1/2} - 1 + \varepsilon)$, one task of $(2^{1/5} - 1 + \varepsilon)$, and one task of $(2^{1/31} - 1 + \varepsilon)$, since $2(2^{1/2} - 1 + \varepsilon) + (2^{1/5} - 1 + \varepsilon) + (2^{1/31} - 1 + \varepsilon) < 1$ for $\varepsilon < 0.00006$. Therefore, $N_0 = n = 120k$.

Then $N / N_0 = 274k / (120k)$. The bound of RM-FF satisfies

$$\Re^{\infty}_{RM-FF} = 274k / (120k) \geq 2.2833 \qquad \blacksquare$$

The bounds for RM-FF were derived under the general assumption that the utilization of a task can take any value between zero and one. If the utilization of a task is small compared to the processing power of a processor, we show that the worst case performance of RM-FF can be significantly improved.

Let α be the maximum allowable utilization of a task, i.e., $\alpha = max_i (C_i/T_i)$. Then we have the following theorem.

Theorem 3.8: *Let N and N_0 be the number of processors required by RM-FF and the minimum number of processors required to feasibly schedule a given set of tasks, respectively. If* $\alpha = max_{1 \leq i \leq n} \{u_i\}$ *and* $\alpha \leq 2^{1/(1+c)} - 1$, *then*

$\Re^{\infty}_{RM-FF}(\alpha) \le 1 / [(c+1)(2^{1/(2+c)} - 1)]$ *for* $c = 0, 1, 2, \ldots$

Table 3.1: Worst Case Performance Bounds of RM-FF under α

α	≥ 0.4142	< 0.4142	< 0.2599	< 0.1892	< 0.1487	< 0.02
$\Re^{\infty}_{RM-FF}(\alpha)$	2.33	1.92	1.76	1.68	1.63	1.47

When $c \to \infty$, $[(c+1)(2^{1/(2+c)} - 1)] \to \ln 2$. *Then* $\Re^{\infty}_{RM-FF}(\alpha) \le 1/\ln 2$. *The upper bounds of RM-FF with regard to* α *are given in Table 3.1 for a few values of* α.

Proof: For any set of n tasks, let $\sum_{i=1}^{n} u_i$ be the total utilization of the task set. According to RM-FF, if $\alpha \le 2^{1/(1+c)} - 1$, then each processor must be assigned at least $(c+1)$ tasks since $(c+1)\alpha \le (c+1)\,[2^{1/(1+c)} - 1]$ except possibly for the last processor. According to Lemma 3.2, among all the processors to each of which at least $(c+1)$ tasks are assigned, there is at most one processor whose utilization is no greater than $(c+1)\,[2^{1/(2+c)} - 1]$, for $c = 0, 1, 2, \ldots$ Since $(N-2)\{(c+1)\,[2^{1/(2+c)} - 1]\} \le \sum_{i=1}^{n} u_i \le N_0$, we have

$$N \le N_0 + 2(c+1)\,[2^{1/(2+c)} - 1].$$

Hence, $\Re^{\infty}_{RM-FF}(\alpha) \le 1 / [(c+1)(2^{1/(2+c)} - 1)]$, for $c = 0, 1, 2, \ldots$

When $c \to \infty$, $[(c+1)(2^{1/(2+c)} - 1)] \to \ln 2$. Then $\Re^{\infty}_{RM-FF}(\alpha) \le 1/\ln 2$. ■

For future reference, we also prove the following lemma here.

Lemma 3.5: *Suppose that in the RM-FF schedule, there are n processors on each of which exactly m* ≥ 1 *tasks are assigned. Then* $\sum_{i=1}^{n} U_i > nm\left(2^{1/(m+1)} - 1\right)$ *for* $n > m$, *where* U_i *is the total utilization of the m tasks assigned to processor* P_i. *If* $n \le m$, *then* $\sum_{i=1}^{n} U_i > nm\left(2^{1/(m+1)} - 1\right) - (m+1-n)\, m\left(2^{1/m} - 2^{1/(m+1)}\right)$.

Proof: Let us index the n processors from 1 to n according to the order in which they are assigned tasks in the completed RM-FF schedule. According to Lemma 3.2, there is at most one processor P_i with $\sum_{k=1}^{m} u_{i,k} \le m\left(2^{1/(m+1)} - 1\right)$ among n processors.

If $i < n$, then we can assume that $\sum_{k=1}^{m} u_{i,k} = m\left(2^{1/(m+1)} - 1\right) - \Delta$, where $\Delta > 0$. For each task on the last processor P_n, its utilization satisfied:

$$u > (m+1)\left(2^{1/(m+1)}-1\right) - \sum_{k=1}^{m} u_{i,k}$$
$$= (m+1)\left(2^{1/(m+1)}-1\right) - m\left(2^{1/(m+1)}-1\right) + \Delta = 2^{1/(m+1)} - 1 + \Delta.$$

Then the total utilization of processor P_n is given by

$$\sum_{k=1}^{m} u_{n,k} \geq mu > m\left(2^{1/(m+1)}-1\right) + m\Delta.$$

Since each processor has a utilization greater than $m\left(2^{1/(m+1)}-1\right)$ for the rest of $n-2$ processors, $\sum_{i=1}^{n} U_i > (n-2)m\left(2^{1/(m+1)}-1\right) + m\left(2^{1/(m+1)}-1\right) + m\Delta + m\left(2^{1/(m+1)}-1\right) - \Delta = nm\left(2^{1/(m+1)}-1\right) + (m-1)\Delta \geq nm\left(2^{1/(m+1)}-1\right)$.

If $i = n$, then suppose the least total utilization among the rest of $(n-1)$ processors is $\sum_{k=1}^{m} u_{i,k} = m\left(2^{1/(m+1)}-1\right) + \Delta$, where $\Delta > 0$. Then for each task on processor P_n,

$$\text{u} > (m+1)\left(2^{1/(m+1)}-1\right) - \sum_{k=1}^{m} u_{i,k}$$
$$= (m+1)\left(2^{1/(m+1)}-1\right) - m\left(2^{1/(m+1)}-1\right) - \Delta = 2^{1/(m+1)} - 1 - \Delta.$$

Then the total utilization of processor P_n is given by

$$\sum_{k=1}^{m} u_{n,k} \geq mu > m\left(2^{1/(m+1)}-1\right) - m\Delta.$$

Since each processor has a utilization greater than or equal to $m\left(2^{1/(m+1)}-1\right)$ $+ \Delta$ for the first $n-1$ processors, we have

$$\sum_{i=1}^{n} U_i > (n-1)m\left(2^{1/(m+1)}-1\right) + (n-1)\Delta + m\left(2^{1/(m+1)}-1\right) - \text{m}\Delta$$
$$= nm\left(2^{1/(m+1)}-1\right) + (n-m-1)\Delta.$$

If $n > m$, then $\sum_{i=1}^{n} U_i \geq nm\left(2^{1/(m+1)}-1\right)$.

If $n \leq m$, then $\sum_{i=1}^{n} U_i \geq nm\left(2^{1/(m+1)}-1\right) - (m+1-n)\Delta$. We need to find the upper bound for Δ. For each processor P_i with $i < n$, $\sum_{k=1}^{m} u_{i,k} \leq m\left(2^{1/m}-1\right)$ and $\sum_{k=1}^{m} u_{i,k} \geq m\left(2^{1/(m+1)}-1\right) + \Delta$. Therefore, $\Delta \leq m\left(2^{1/m} - 2^{1/(m+1)}\right)$. We have $\sum_{i=1}^{n} U_i \geq nm\left(2^{1/(m+1)}-1\right) - (m+1-n)\, m\left(2^{1/m} - 2^{1/(m+1)}\right)$. $\Delta \to 0$ and $\text{m}\Delta \to 0$, when $m \to \infty$.

Examples can be constructed to show that both bounds are tight. ■

3.4. Rate-Monotonic-Best-Fit

When RM-FF schedules a task, it always assigns it to the lowest indexed processor on which the task can be scheduled. This strategy may not be optimal in some cases. For example, the lowest indexed processor on which a task is scheduled may be the one with the largest available utilization among all those busy (non-idle) processors. This processor could have been used to execute a future task with large enough utilization so that it could not be scheduled on any busy processors, had it not been assigned a task with a small utilization earlier on. In order to overcome these likely disadvantages, a new algorithm is designed as follows, which is based on the Best-Fit bin-packing algorithm.

It is a well-known fact that the Best-Fit heuristic has the same worst case performance bound as the First-Fit in bin-packing [15]. Yet we cannot automatically conclude from the bin-packing results that RM-FF and RM-BF will have the same worst case performance bound, since the RMMS problem differs from the bin-packing problem (i.e., the classical one-dimensional bin-packing). The major difference is that the size of each bin in bin-packing is unitary and the utilization of a processor can assume a value ranging from ln2 to 1 as given by the schedulability condition.

In bin-packing, when an item is allocated by the Best-Fit, the lowest indexed bin in which the item can be fit and whose content is the largest among all the non-empty bins in which the item can be fit, is chosen to contain the item. Since the “sizes” of the bins are unitary, finding the fitting bin whose content is the largest among all the non-empty fitting bins is equivalent to finding the fitting bin whose available space is the smallest among all the non-empty fitting bins. This “equivalence” property of Best-Fit does not hold when Best-Fit is used to schedule tasks on processors. The “unfilled” utilization of a processor is not only determined by the total utilization of the tasks assigned to it, but also by the number of tasks. Therefore, it is possible that the available utilization of a processor with a currently large utilization is larger than that of a processor with a currently small utilization. For example, processor P_1 is currently assigned two tasks, each with a utilization of $u = (2^{1/3} - 1) \approx 0.259$. Then the total utilization of processor P_1 is $U_1 = 2(2^{1/3} - 1) < 0.52$. The available (or unfilled) utilization of P_1 is given by $3/(1+u)^2 - 1 = (2^{1/3} - 1)$. For another processor P_2 to which one task with a utilization of $U_2 = 0.52$ is assigned, its available utilization is given by $(1 - 0.52)/(1 + 0.52) > 0.31$. Therefore, the available utilization of processor P_2 is larger than that of processor P_1 even though $U_1 < U_2$. The schedulability condition used in both the calculations is the UO condition.

In other words, there are at least two notably different ways in which the Best-Fit heuristic can be applied to allocating tasks to processors: one is to find the “fitting” processor with the largest utilization, and the other is to find the “fitting” processor with the smallest available utilization. Presumably these two variations might have different worst case performance bounds. In the following, we only investigate one variation of the Best-Fit strategy, where the “best fit” is the “fitting” processor with the smallest available utilization.

Algorithm RM-BF: Let the processors be indexed as $P_1, P_2, \ldots$, with each initially in the idle state, i.e., with zero utilization. The tasks $\tau_1, \tau_2, \ldots, \tau_m$ will be scheduled in that order. To schedule τ_i, find the least j such that task τ_i together with all the tasks that have been assigned to processor P_j, can be feasibly scheduled according to the condition $2\,(1 + U_j/k_j)^{-\kappa_j} - 1$ for a single processor, and $2\,(1 + U_j/k_j)^{-\kappa_j} - 1$ be as small as possible, and assign task τ_i to P_j, where k_j and U_j are the number of tasks already assigned to processor P_j and the total utilization of the k_j tasks, respectively.

With its “minimal unfilled utilization” strategy in assigning tasks to processors, the RM-BF algorithm does not outperform RM-FF in the worst-case, as shown by Theorem 3.9. Before we prove the bounds for RM-BF, we need to establish a few lemmas.

Definition 3.1: For all the processors required to schedule a given set of tasks by RM-BF, they are divided into two types of processors:

Type (I): For all the tasks $\tau_1, \tau_2, \ldots, \tau_m$ with utilizations $u_1, u_2, \ldots, u_m$ that were assigned to a processor P_x in the completed RM-BF schedule, there exists at least one task τ_i with $i \geq 2$ that was assigned to P_x, not because it could not be assigned on any processor P_y with lower index, i.e., $y < x$, but because $2\left(1 + \left(\sum_{l=1}^{i-1} u_l\right) / (i-1)\right)^{-(i-1)} - 1 < 2\left(1 + \left(\sum_{l=1}^{n_y} u_l\right) / n_y\right)^{-n_y} - 1$, where n_y is the number of tasks assigned to processor P_y.

Processor P_x is called a Type (I) processor. Such a task τ_i is, for convenience, referred to as a task with Type (I) property.

Type (II): They consist of all the processors that do not belong to Type (I).

Lemma 3.6: *If m tasks cannot be feasibly scheduled on $m - 1$ processors according to RM-BF, then the total utilization of the tasks is greater than* $m\left(2^{1/2} - 1\right)$.

The proof of this lemma is the same as the proof to Lemma 3.1.

Lemma 3.7: *In the completed RM-BF schedule, if the mth task on any of the Type (I) processors has Type (I) property, where $m \ge 2$, then the total utilization of the first $(m - 1)$ tasks on that processor is greater than* $(m-1)\left(2^{1/m} - 1\right)$.

Proof: Let $\tau_{k,1}, \tau_{k,2}, \ldots, \tau_{k,m-1}$ be the tasks that were assigned a processor P_k of Type (I), and P_y, with $y < k$, is one of the processors on which τ_m could have been scheduled, but $2\left(1 + \left(\sum_{l=1}^{m-1} u_{k,l}\right) / (m-1)\right)^{-(m-1)} - 1 < 2\left(1 + \left(\sum_{l=1}^{n_y} u_{y,l}\right) / n_y\right)^{-n_y} - 1$, where n_y is the number of tasks assigned to processor P_y, and where $u_{x,l}$ is the utilization of task $\tau_{x,l}$ on processor P_x.

Since $u_{k,i} > 2\left(1 + \left(\sum_{l=1}^{n_y} u_{y,l}\right) / n_y\right)^{-n_y} - 1$ (note that this is true even though $\tau_{k,i}$ is assigned to processor P_k before some of tasks among the n_y tasks are assigned to processor P_y), for $1 \le i \le m - 1$, we have

$$u_{k,i} > 2\left(1 + \left(\sum_{l=1}^{n_y} u_{y,l}\right) / n_y\right)^{-n_y} - 1 > 2\left(1 + \left(\sum_{l=1}^{m-1} u_{k,l}\right) / (m-1)\right)^{-(m-1)} - 1.$$

Summing up these $(m - 1)$ inequalities yields

$$\sum_{j=1}^{m-1} u_{k,j} > 2(m-1)\left(1 + \left(\sum_{l=1}^{m-1} u_{k,l}\right) / (m-1)\right)^{-(m-1)} - (m-1).$$

Solving the above equation yields

$$\sum_{j=1}^{m-1} u_{k,j} > (m-1)\left(2^{1/m} - 1\right). \qquad \blacksquare$$

The following lemma is key to the proof of Theorem 3.9.

Lemma 3.8: *In the completed RM-BF schedule, among the processors of Type (I) on which the second task has Type (I) property, there are at most three of them, each of which has a total utilization less than* $2\left(2^{1/3} - 1\right)$.

Proof: This lemma is proven by contradiction. Let P_i, P_j, P_k, and P_l be the four processors, each of which has a total utilization less than $2\left(2^{1/3}-1\right)$ with $i < j < k < l$, i.e.,

$$\sum_{x=1}^{n_i} u_{i,x} < 2\left(2^{1/3}-1\right)$$
$$\sum_{x=1}^{n_j} u_{j,x} < 2\left(2^{1/3}-1\right)$$
$$\sum_{x=1}^{n_k} u_{k,x} < 2\left(2^{1/3}-1\right)$$
$$\sum_{x=1}^{n_l} u_{l,x} < 2\left(2^{1/3}-1\right)$$

where $n_i \geq 2$, $n_j \geq 2$, $n_k \geq 2$, and $n_l \geq 2$ are the number of tasks assigned to processors P_i, P_j, P_k, and P_l, respectively.

Let's define $u_{i,1}$ and $u_{i,2}$ to be the utilizations of the first task $\tau_{i,1}$ and second task $\tau_{i,2}$ assigned to processor P_i, $u_{j,1}$ and $u_{j,2}$ to be the utilizations of the first task $\tau_{j,1}$ and second task $\tau_{j,2}$ assigned to processor P_j. $u_{k,1}$ and $u_{k,2}$, $u_{l,1}$ and $u_{l,2}$ are similarly defined. We further assume that n_y is the number of tasks which have been assigned to processor P_i, when the second task on processor P_j is assigned. Note that $i < j$ and $1 \leq n_y \leq n_j$.

There are three cases to consider.

Case 1: Tasks $\tau_{j,1}$ and $\tau_{j,2}$ are assigned to processor P_j after task $\tau_{i,2}$ is assigned to processor P_i. Since task $\tau_{j,2}$ is a Type (I) task, the following inequality must hold

$$2\,(1+u_{j,1})^{-1}-1 < 2\left(1+\left(\sum_{x=1}^{n_y} u_{i,x}\right)/n_y\right)^{-n_y}-1$$

Note that $n_y \geq 2$, i.e., other tasks may have been assigned to processor P_i after task $\tau_{i,2}$ but before $\tau_{j,1}$ is assigned to processor P_j.

Since $2\left(1+\left(\sum_{x=1}^{n_y} u_{i,x}\right)/n_y\right)^{-n_y} - 1 \leq 2\,(1+(u_{i,1}+u_{i,2})/2)^{-2} - 1 < 2\,(1+u_{i,1}/2)^{-2}-1$, we have

$$2\,(1+u_{j,1})^{-1}-1 < 2\,(1+u_{i,1}/2)^{-2}-1\text{, i.e., } 1+u_{j,1} > (1+u_{i,1}/2)^{2}.$$

Case 2: Tasks $\tau_{j,1}$ and $\tau_{j,2}$ are assigned to processor P_j after task $\tau_{i,1}$ is assigned to processor P_i but before task $\tau_{i,2}$ is assigned to processor P_i.

This case is impossible with RM-BF scheduling. Since $\sum_{x=1}^{n_i} u_{i,x} < 2(2^{1/3}-1)$ and $u_{i,1} > (2^{1/2}-1)$ according to Lemma 3.7, $u_{i,2} < 2(2^{1/3}-1)-(2^{1/2}-1) \approx 0.1056$.

Since task $\tau_{j,2}$ is assigned to processor P_j before task $\tau_{i,2}$ is assigned to processor P_i, and task $\tau_{j,2}$ is a Type (I) task, $2(1+u_{i,1})^{-1}-1 > 2((1+u_{j,1})^{-1}-1$, i.e.,

$$u_{i,1} < u_{j,1}. \qquad \text{(Eq.3.14)}$$

Since task $\tau_{i,2}$ is also a Type (I) task, it must be true according to the definition that

$$2(1+u_{i,1})^{-1}-1 < 2\left(1+\left(\sum_{x=1}^{n_z} u_{j,x}\right)/n_z\right)^{-n_z}-1,$$

where n_z is the number of tasks that have been assigned to processor P_j after task $\tau_{j,2}$, but before task $\tau_{i,2}$ is assigned to processor P_i. Note that it is conceivable that other tasks may have been assigned to processor P_j after task $\tau_{j,2}$ but before task $\tau_{i,2}$ is assigned to processor P_i.

Since $2(1+u_{i,1})^{-1}-1 < 2\left(1+\left(\sum_{x=1}^{n_z} u_{j,x}\right)/n_z\right)^{-n_z}-1 < 2(1+u_{j,1})^{-1}-1$, we have $u_{i,1} > u_{j,1}$. This is a contradiction to inequality (3.14).

Case 3: Task $\tau_{j,1}$ is assigned to processor P_j after task $\tau_{i,1}$ is assigned to processor P_i, and task $\tau_{j,2}$ is assigned to processor P_j after task $\tau_{i,2}$ is assigned to processor P_i. Since task $\tau_{j,2}$ is a Type (I) task, the following inequality must hold

$$2(1+u_{j,1})^{-1}-1 < 2\left(1+\left(\sum_{x=1}^{n_y} u_{i,x}\right)/n_y\right)^{-n_y}-1$$

Note that $n_y \geq 2$, i.e., other tasks may have been assigned to processor P_i after task $\tau_{i,2}$ but before $\tau_{j,2}$ is assigned to processor P_j.

Since $2\left(1+\left(\sum_{x=1}^{n_y} u_{i,x}\right)/n_y\right)^{-n_y} - 1 \leq 2(1+(u_{i,1}+u_{i,2})/2)^{-2} - 1 < 2(1+u_{i,1}/2)^{-2}-1$, we have

$$2(1+u_{j,1})^{-1}-1 < 2(1+u_{i,1}/2)^{-2}-1, \text{ i.e., } 1+u_{j,1} > (1+u_{i,1}/2)^2.$$

Therefore for processors P_i and P_j, we have

$$1+u_{j,1} > (1+u_{i,1}/2)^2. \qquad \text{(Eq.3.15)}$$

For the tasks assigned on processors P_j and P_k, and P_k and P_l, it can be similarly proven that

$$1+u_{k,1} > (1+u_{j,1}/2)^2 \qquad \text{(Eq.3.16)}$$

$$1+u_{l,1} > (1+u_{k,1}/2)^2 \qquad \text{(Eq.3.17)}$$

Summing up inequalities (3.15), (3.16), and (3.17) yields $u_{l,1} > (u_{i,1}^2 + u_{j,1}^2 + u_{k,1}^2) / 4 + u_{i,1}$. Since $u_{i,1} > (2^{1/2} - 1)$, $u_{j,1} > (2^{1/2} - 1)$, and $u_{k,1} > (2^{1/2} - 1)$ according to Lemma 3.7, $u_{l,1} > 3(2^{1/2} - 1)^2 / 4 + (2^{1/2} - 1) = 0.5429 > 2(2^{1/3} - 1)$. This results in a contradiction to the assumption that $\sum_{x=1}^{n_l} u_{l,x} < 2(2^{1/3} - 1)$. ■

Theorem 3.9: *Let N and N_0 be the number of processors required by RM-BF and the minimum number of processors required to feasibly schedule a given set of tasks, respectively. Then $2.2833 \le \Re_{RM-BF}^{\infty} \le 2.33$.*

Proof: Similar to what we have done in Section 3.3 for the RM-FF algorithm, we use the same weighting function to map the utilization of a task into the real interval [0, 1]. Note that all the relevant lemmas in Section 3.3 hold for those processors of Type (II) in the RM-BF schedule.

Let $\Sigma = \{\tau_1, \tau_2, \ldots, \tau_m\}$ be a set of m tasks, with their utilizations $u_1, u_2, \ldots, u_m$ respectively, and $\varpi = \sum_{i=1}^{m} W(u_i)$. By Lemma 3.3, $\varpi \le N_0 / a$, where $a = 2\left(2^{1/3} - 1\right)$.

Suppose that among the N number of processors required by RM-BF to schedule a given set Σ of tasks, M of them are processors of Type (I). Since all processors of Type (I) must be assigned at least two tasks, there exists for each processor a number $m \ge 2$ such that the mth task is a Type (I) task. For all the processors of Type (I) on each of which the mth task is a Type (I) task with $m \ge 3$, we have $\sum_j W(u_j) > 1$ since $\sum_j u_j > 2(2^{1/3} - 1)$ according to Lemma 3.7.

When $m = 2$, there are at most three of the processors, each of which has a total utilization less than $2(2^{1/3} - 1)$. Therefore, for all the processors of Type (I), there are at most three processors whose $\sum_j W(u_j)$ is less than 1 in the RM-BF schedule.

Now let $L = n_1 + n_2$ be defined similarly as in Section 3.3, except that they are for processors of Type (II). All the results derived in Section 3.3 are applicable to the set of Type (II) processors in the RM-BF schedule.

The upper bound of RM-BF can now be determined.

$\varpi = \sum_{i=1}^{m} W(u_i) \ge (N - L - M) + n_1 (2^{1/2} - 1) / a$

$= N - n_1 - n_2 + n_1(2^{1/2} - 1)/a - 3$

$\geq N - 2N_0[1 - (2^{1/2} - 1)/a] - n_2 - 3$

Since $\varpi \leq N_0 / a$ and $n_2 \leq 1$, we have

$N \leq N_0 [2a + 1 - 2(2^{1/2} - 1)] / a + 4$. Hence,

$\Re^{\infty}_{RM-BF} \leq [2a + 1 - 2(2^{1/2} - 1)] / a \approx 2.33$, where $a = 2\left(2^{1/3} - 1\right)$.

The lower bound is proven by repeating the same argument as in Theorem 3.7 for RM-BF. Therefore, $2.2833 \leq \Re^{\infty}_{RM-BF}$. ■

Note that even though RM-BF has the same worst case performance bound as RM-FF, special cases exist where RM-BF performs better than RM-FF, and vice versa. For example, for a set of four tasks with their utilizations given as follows, two processors are needed by RM-BF while three processors are required by RM-FF to schedule it.

$u_1 = 2/5$, $u_2 = 3/7 + \varepsilon$, $u_3 = (4 - 7\varepsilon) / (10 + 7\varepsilon)$, and $u_4 = 3/7$ for arbitrarily small $\varepsilon > 0$.

We can also derive similar bounds for RM-BF with respect to the maximum allowable utilization of a task.

Theorem 3.10: *Let N and N_0 be the number of processors required by RM-BF and the minimum number of processors required to feasibly schedule a given set of tasks, respectively. If $\alpha = max_{1 \leq i \leq n}\{u_i\}$ and $\alpha \leq 2^{1/(1+c)} - 1$, then*

$\Re^{\infty}_{RM-BF}(\alpha) \leq 1 / [(c+1)(2^{1/(2+c)} - 1)]$, *for c = 0, 1, 2, …*

When $c \to \infty$, $[(c+1)(2^{1/(2+c)} - 1)] \to \ln 2$. Then $\Re^{\infty}_{RM-BF}(\alpha) \leq 1/\ln 2$. The upper bounds of RM-BF with regard to α are given in Table 3.1 for a few values of α.

3.5. The Refinements of RM-FF and RM-BF

It is clear that if α is small, RM-FF performs well. However, its performance degrades rapidly when $\alpha > 0.4142$. If we can find a better way to schedule the tasks with large utilization, and use the RM-FF to schedule the tasks with small utilization, then the overall performance of the combined algorithm will be improved. We are thus motivated to develop a new allocation algorithm that is based on the well-known divide-and-conquer strategy. It is called the Refined-Rate-Monotonic-First-Fit (RRM-FF).

RRM-FF divides the processors into two groups such that within each group there is an infinite number of processors. It also divides the task set into two groups according to their utilizations such that tasks within a group are assigned to the same group of processors. Let the processors in the first group (or the P group) be indexed as $P_1, P_2, \ldots$, and processors in the second group (or the Q group) be indexed as $Q_1, Q_2, \ldots$, with each one initially in the idle state. A task τ_i belongs to the first group if its utilization is no greater than $2^{1/3} - 1$, i.e., $u_i \leq 2^{1/3} - 1$, otherwise it belongs to the second group. The tasks $\tau_1, \tau_2, \ldots, \tau_m$ will be scheduled in that order. To schedule τ_i, RRM-FF first identifies the task group it belongs to and then finds the least j such that task τ_i, together with all the tasks that have been assigned to processor P_j (or Q_j), can be feasibly scheduled, and assign task τ_i to P_j. The First-Fit heuristic is used to assign tasks in both groups.

RRM-FF can be described in a more algorithmic format in Figure 3.5. Note that the grouping of tasks is “imaginary” and the algorithm is clearly on-line. The schedulability condition used for scheduling tasks in the first group is the IFF condition. Since the algorithm assumes that at most two tasks can be assigned to any processor in the second processor group Q, the IFF schedulability test is reduced to just two comparison operations. Therefore, the overall time complexity of the RRM-FF algorithm is still $O(n \log n)$. The worst case performance bound for RRM-FF is given in Theorem 3.11. Theorem 3.3 is key to the proof of the upper bound.

Theorem 3.11: *Let N and N_0 be the number of processors required by RRM-FF and the minimum number of processors required to feasibly schedule a given set of tasks, respectively. Then* $\Re^{\infty}_{RRM-FF} \leq 1.96$. *The upper bounds of RRM-FF with regard to* α *are given in Table 3.2 for a few values of* α.

Table 3.2: Worst Case Performance Bounds of RRM-FF under α

α	≥ 0.4142	< 0.4142	< 0.2598	< 0.1892	< 0.1487	< 0.02
$\Re^{\infty}_{RRM-FF}(\alpha)$	1.96	1.92	1.76	1.68	1.63	1.47

Refined-Rate-Monotonic-First-Fit (RRM-FF) (Input: task set Σ; Output: *m*)

```
(1) Determine the group member of an incoming task $\tau_i$ as follows:
        $P = \{\tau_i | u_i \le 2^{1/3} - 1\}$ and $Q = \{\tau_i | u_i > 2^{1/3} - 1\}$
(2) If the task belongs to the first group (P), then assign it to a
    processor in the first processor group using the RM-FF algorithm.
(3) If the task belongs to the second group (Q), then assign it to a
    processor in the second group as follows:
    Use the First-Fit heuristic to find a processor $Q_i$ that contains
    exactly one task and assign task $\tau_i$ to processor $Q_i$ if the two
    tasks can be feasibly scheduled according to the following con-
    dition:
        min := i; max := j;
        If( $T_j > T_i$ ) Then {min := j; max := i;};
        If ( $\lfloor T_{max}/T_{min} \rfloor C_{min} + C_{max} \le \lfloor T_{max}/T_{min} \rfloor T_{min}$ ) Or ( $\lceil T_{max}/T_{min} \rceil$
          $C_{min} + C_{max} \le T_{max}$ ) Then feasible := True
        Else feasible := False
    Otherwise, assign a task to an empty processor.Terminate when all
    tasks in the group have been assigned.
(4) The total number of processors required is the sum of the total
    number of processors used in both processor groups.
```

Figure 3.5: Algorithm RRM-FF

Proof: Let $\Sigma = \{\tau_1, \tau_2, \ldots, \tau_m\}$ be a set of *m* tasks, with their utilizations being $u_1, u_2, \ldots, u_m$. Then the total utilization of the task set is given by $\sum_{i=1}^{m} u_i$. Suppose that $N = N_P + N_Q$ processors are used by RRM-FF to schedule the task set Σ, where N_P and N_Q are the number of processors allocated in processor group *P* and that in processor group *Q*, respectively. Among the N_Q processors, let n_1 be the number of processors assigned one task and n_2 be the number of processors assigned two tasks. Then $N_Q = n_1 + n_2$. For convenience, we let $a = \left(2^{1/3} - 1\right)$.

Among the N_Q processors, for the n_1 processors to each of which one task is assigned, we have by Theorem 3.3 that

$$\sum_{i=1}^{n_1} u_i > n_1 \big/ \left(1 + 2^{1/n_1}\right) > n_1 / 2 - \ln 2 / 4. \qquad \text{(Eq.3.18)}$$

For the n_2 processors to each of which two tasks are assigned, it is apparent that

$$\sum_{i=1}^{n_2} (u_{i,1} + u_{i,2}) > 2an_2 \qquad \text{(Eq.3.19)}$$

since $u_{i,1} > a$ and $u_{i,2} > a$.

Since $u_i \le a$, each of the N_P processors must be assigned at least three tasks, possibly except the last processor. According to Lemma 3.2, among all processors on each of which at least three tasks are assigned, there are at most one processor whose utilization is no greater than $3(2^{1/4} - 1)$. Then we have

$$\sum_{i=1}^{N_P} u_i \ge 3\left(2^{1/4} - 1\right)(N_P - 1) \qquad \text{(Eq.3.20)}$$

According to inequalities (3.18), (3.19), and (3.20), we have

$$\sum_{i=1}^{m} u_i = \sum_{i=1}^{n_1} u_i + \sum_{i=1}^{n_2} (u_{i,1} + u_{i,2}) + \sum_{i=1}^{N_P} U_i$$
$$\ge n_1 / 2 - \ln 2 / 4 + 2an_2 + 3\left(2^{1/4} - 1\right)(N_P - 1) .$$

Since $N_0 \ge \sum_{i=1}^{m} u_i$ and $N = n_1 + n_2 + N_p$, it is immediate that

$$N_0 \ge 2aN - \ln 2 / 4 - 3\left(2^{1/4} - 1\right) - (2a - 0.5)n_1.$$

Since any two of the tasks that are assigned to the n_1 processors cannot be scheduled on a single processor, we have $N_0 \ge n_1$.

Then $N_0 \ge 2aN - \ln 2 / 4 - 3\left(2^{1/4} - 1\right) - (2a - 0.5)n_1$

$$\ge 2aN - \ln 2 / 4 - 3\left(2^{1/4} - 1\right) - (2a - 0.5)N_0$$

$$\frac{N}{N_0} \le \frac{2a + 0.5}{2a} - \left(\frac{\ln 2}{4} + 3\left(2^{1/4} - 1\right)\right)\frac{1}{2aN_0} \qquad \text{(Eq.3.21)}$$

Hence, $\Re^{\infty}_{RRM-FF} \le 1.96$.

For $\alpha \le a$, by arguments similar to the above one and the one in the proof of Theorem 3.8, we obtain the rest of the results that are listed in Table 3.2. ■

At this point, it is interesting to note that although we choose $(2^{1/3} - 1)$ as value used to divide a task set into two groups, it is in part for convenience of proof and presentation. In fact, any value in the range of $(2^{1/3} - 1, 2^{1/2} - 1)$ can do. In other words, if we divide a task set into groups by choosing any value between $(2^{1/3} - 1)$ and $(2^{1/2} - 1)$, the algorithm RRM-FF still has the same worst case upper bound of 1.96. This claim can be readily proven.

Let α be the maximum allowable utilization of a task, i.e., $\alpha = max_i (C_i / T_i)$. Then we can prove, similar to what we have done in Section 3.3 that when α is small, the worst case performance of RM-BF can be significantly improved, as stated in Theorem 3.8. Based on similar observation, we can modify RM-BF to develop a new algorithm called Refined-Rate-Monotonic-Best-Fit (RRM-BF) to cope with situations where α is large.

RRM-BF works as follows: It divides the processors into two groups such that within each group there is an infinite number of processors. It also divides the task set into two groups in just the same manner as RRM-FF does. Also, RRM-BF works the same way as RRM-FF does, except that the Best-Fit heuristic is used to assign tasks in both groups for RRM-BF. The following result can be proven similar to that of Theorem 3.11.

Theorem 3.12:*Let N and N_0 be the number of processors required by RRM-BF and the minimum number of processors required to feasibly schedule a given set of tasks, respectively. Then* $\Re^{\infty}_{RRM-BF} \leq 1.96$. *The upper bounds of RRM-BF with regard to* α *are given in Table 3.2 for a few values of* α.

3.6. Period-Oriented Heuristic Algorithms

As we have seen, the performance of a multiprocessor scheduling algorithm depends not only upon the allocation scheme, but also upon the schedulability condition used for each processor. The schedulability conditions that we have used in various scheduling algorithms so far are oriented towards utilization, i.e., the relative values of task utilizations are taken into account. The performance of the algorithms is therefore limited because they fail to consider the relative values of task periods.

Though task periods have been assumed to be arbitrary in those utilization-oriented schedulability conditions, they are in fact derived under the condition that the ratio between any two task periods is no more than 2. The task sets that are given in showing the lower bounds for the algorithms might require few processors to execute them if their periods are taken into consideration as well. One of the schedulability conditions that explicitly takes into account the periods of tasks, besides the necessary and sufficient condition, is the PO condition present in Section 2.3.

Next we will develop three scheduling algorithms that are based on the PO condition. The first two algorithms, RMST and RMGT, first order the tasks according to their periods and then schedule them. Accordingly they are off-line algorithms. The third one, RMGT-M, schedules tasks without assuming any knowledge about the incoming tasks, and hence it is a on-line algorithm.

One of the salient features of these three algorithms is that their performance increases as α, the maximum allowable utilization of a task, decreases. Though it may also be true that the worst case performance of other algorithms increases as α decreases, the increase in the performance of these three algorithms is very rapid.

It is apparent that in the PO condition, the utilization bound increases as β decreases. β is defined as the largest difference of the V values between any two tasks and the V value of a task is defined as $V_i = \log_2 T_i - \lfloor \log_2 T_i \rfloor$. In the PO condition, $1 - \beta \ln 2 \rightarrow 1$ as $\beta \rightarrow 0$. This suggests that if we assign the tasks having almost the same V values together on a processor, then the total utilization of a processor can be increased. Therefore, a natural way to schedule a set of tasks is first to sort tasks according to the order of increasing or decreasing V value and then schedule them in the new order.

The first algorithm is thus developed and it is called the Rate-Monotonic-Small-Task (RMST) because it favors task sets that have small α. RMST assigns tasks to processors in almost the same manner as the Next-Fit bin-packing heuristic. The algorithm is described in Figure 3.6.

Note that U_m denotes the total utilization of the tasks that have been assigned to

Rate-Monotonic-Small-Task (RMST) (Input: task set Σ; Output: m)

```
(1) Sort the task set such that $0 \le V_1 \le \ldots \le V_n < 1$.
(2) i := 1; m := 1; $S_m := V_i$;
(3) Assign task $\tau_i$ to processor $P_m$ if this task together with the
    tasks that have already been assigned $P_m$ to can be feasibly
    scheduled on $P_m$ according to the following condition:
    $U_m + u_i \le \max\{\ln 2, 1 - \beta \ln 2\}$, where $\beta = V_i - S_m$.
    If not, assign task $\tau_i$ to $P_{m+1}$ and m := m + 1, $S_m := V_i$.
(4) If i < n, then i := i + 1 and go to (3) else stop.
```

Figure 3.6: Algorithm RMST

processor P_m and u_i denotes the utilization of task τ_i.

Theorem 3.13: *Let N and N_0 be the number of processors required by RMST and the minimum number of processors required to feasibly schedule a given set of tasks, respectively. Define* $\alpha = \max_{i=1,\ldots,n}(C_i/T_i)$. *If* $\alpha \ge 1/2$, *then*

$$N \le 2N_0 + 1 + \ln 2. \tag{Eq.3.22}$$

If $\alpha < 1/2$, *then*

$$\frac{N}{N_0} < \frac{1}{1-\alpha} + \left(1 + \frac{\ln 2}{1-\alpha}\right)\frac{1}{N_0}. \tag{Eq.3.23}$$

Proof: In the completed RMST schedule, let $\tau_{j,1}, \tau_{j,2}, \ldots, \tau_{j,s_j}$ be the s_j tasks that are assigned to a processor P_j and $U_j = \sum_{k=1}^{s_j} u_{j,k}$ for $j = 1, \ldots, N$. Furthermore, let $V_{j,i}$ be the V value of task $\tau_{j,i}$ and $\beta_j = V_{j+1,1} - V_{j,1}$. Then $\sum_{j=1}^{N} \beta_j \le 1$. According to RMST, we have

$$\sum_{k=1}^{s_j} u_{j,k} + u_{j+1,1} > \max\{\ln 2, 1 - \beta_j \ln 2\} \ge 1 - \beta_j \ln 2 \tag{Eq.3.24}$$

for $j = 1, \ldots, N - 1$.

Since $U_{j+1} \ge u_{j+1,1}$, we have

$$U_j + U_{j+1} \ge 1 - \beta_j \ln 2 \tag{Eq.3.25}$$

from (3.24), where $j = 1, \ldots, N - 1$.

Summing up the $N - 1$ inequalities in (3.25) yields

$$2\sum_{j=1}^{N} U_j - U_1 - U_N \geq (N-1) - \ln 2 \sum_{j=1}^{N-1} \beta_j \geq (N-1) - \ln 2$$

since $\sum_{j=1}^{N-1} \beta_j \leq \sum_{j=1}^{N} \beta_j \leq 1$.

In other words,

$$2\sum_{j=1}^{N} U_j > (N-1) + U_1 + U_N - \ln 2 \geq N - 1 - \ln 2.$$

Since $N_0 \geq \sum_{j=1}^{N} U_j = \sum_{j=1}^{n} u_j$, we have $N \leq 2N_0 + 1 + \ln 2$.

If $\alpha = \max_{i=1,\ldots,n} (C_i/T_i)$, then $\alpha \geq u_{j+1,1}$ and

$$U_j + \alpha > 1 - \beta_j \ln 2 \qquad \text{(Eq.3.26)}$$

from (3.24), where $j = 1, \ldots, N-1$.

Summing up the $N-1$ inequalities in (3.26) yields

$$\sum_{j=1}^{N} U_j + (N-1)\alpha > (N-1) - \ln 2 \sum_{j=1}^{N-1} \beta_j > (N-1) - \ln 2$$

since $\sum_{j=1}^{N-1} \beta_j \leq \sum_{j=1}^{N} \beta_j \leq 1$.

In other words,

$$\sum_{j=1}^{N} U_j - \alpha > N(1-\alpha) - 1 - \ln 2. \qquad \text{(Eq.3.27)}$$

$$N < \frac{1}{1-\alpha} \sum_{j=1}^{N} U_j + \left(1 + \frac{\ln 2}{1-\alpha}\right) \qquad \text{(Eq.3.28)}$$

Since $N_0 \geq \sum_{j=1}^{n} u_j$ and $\sum_{j=1}^{N} U_j = \sum_{j=1}^{n} u_j$, we have

$$\frac{N}{N_0} < \frac{1}{1-\alpha} + \left(1 + \frac{\ln 2}{1-\alpha}\right)\frac{1}{N_0}. \qquad ■$$

Next we prove that the bounds given about are in fact tight.

Theorem 3.14: $\Re_{RMST}^{\infty} = 2$. $\Re_{RMST}^{\infty}(\alpha) = \frac{1}{1-\alpha}$ *for* $\alpha = \max_{i=1,\ldots,n} (C_i/T_i) < 1/2$.

Proof: Since $N \leq 2N_0 + 1 + \ln 2$, it is immediate that $\Re_{RMST}^{\infty} \leq 2$.

Since $\frac{N}{N_0} < \frac{1}{1-\alpha} + \left(1 + \frac{\ln 2}{1-\alpha}\right)\frac{1}{N_0}$ for $\alpha = \max_{i=1,\ldots,n} (C_i/T_i) < 1/2$, we have $\Re_{RMST}^{\infty}(\alpha) \leq \frac{1}{1-\alpha}$.

To prove that the above bounds are tight, we need only to construct task sets that require the upper-bounded numbers of processors when they are scheduled by RMST.

Let $n = 4k$ where k is a positive integer and ε be an arbitrarily small number such that $0 < \varepsilon << 1/n$. Furthermore, define $\delta > 0$ such that $2^{n\delta} < 1 + \varepsilon$.

Then for the first bound, the set of n tasks $\Sigma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ is constructed as follows:

$$\tau_i = (C_i, T_i) = (1/2,\ 2^{i\delta}) \text{ for } i = 2j \text{ and } j = 0, 1, \ldots, 2k-1;$$

$$\tau_i = (C_i, T_i) = (\varepsilon,\ 2^{i\delta}) \text{ for } i = 2j+1, j = 0, 1, \ldots, 2k-1.$$

Since $V_{i+1} - V_i = \delta$, the tasks are in the order of increasing V value.

We first claim that $2k$ processors are required to schedule the task set by RMST.

According to the schedulability condition used by RMST,

$$\frac{1/2}{2^{2j\delta}} + \frac{\varepsilon}{2^{(2j+1)\delta}} + \frac{1/2}{2^{(2j+2)\delta}} > \frac{1/2}{2^{n\delta}} + \frac{\varepsilon}{2^{n\delta}} + \frac{1/2}{2^{n\delta}} > 1 > 1 - 2\delta\ln 2 = 1 - \beta\ln 2,$$

where $\beta = 2\delta$ and $j = 0, 1, \ldots\ldots, 2k-1$.

Hence, tasks τ_{2j+1} and τ_{2j} are assigned to a processor, for $j = 0, 1, \ldots, 2k-1$, in the completed RMST schedule. Then a total number of $2k$ processors is required by RMST.

We next claim that $k+1$ processors are needed to schedule the same task set in the optimal schedule.

Since $1/2 + 1/2 = 1 \le 2^{i\delta}$ for $i = 2j$ and $j = 0, 1, \ldots, 2k-1$, any two of these $2k$ tasks can be scheduled on a processor. Yet any three of these tasks cannot be scheduled on a processor since $1/2 + 1/2 + 1/2 > 1 + 1/n > 1 + \varepsilon > 2^{n\delta}$. Therefore, exactly k processors are needed to schedule these $2k$ tasks. For the other $2k$ tasks with $\tau_i = (C_i, T_i) = (\varepsilon,\ 2^{i\delta})$ for $i = 2j+1, j = 0, 1, \ldots, 2k-1$, and $\varepsilon > 0$, one processor is needed to schedule them since $n\varepsilon << 1 \le 2^{i\delta}$.

Let N and N_0 be the number of processors required by RMST and the minimum number of processors required to schedule this task set, respectively. Then $N = 2k$ and $N_0 = k+1$. Hence $\Re^{\infty}_{RMST} = 2$.

For the second bound, task sets can be similarly constructed to prove that the upper-bounded number of processors is required by RMST in each case. Hence we can conclude that

$$\Re^{\infty}_{RMST}(\alpha) = \frac{1}{1-\alpha}. \qquad \blacksquare$$

Next we present an improved version of RMST. Since RMST favors task sets with small task utilization, its performance degrades as the maximum allowable utilization of a task increases. In order to obtain better performance, we modify RMST in such a way that tasks with large utilizations are scheduled together. The new algorithm is called Rate-Monotonic-General-Task (RMGT). It is given in Figure 3.7.

```
Rate-Monotonic-General-Task (RMGT) (Input: task set $\Sigma$; Output: $m$)
(1) Partition the task set $\Sigma$ into two groups:
        $\Re_1 = \{\tau_i | (u_i \le 1/3)\}$ and $\Re_2 = \{\tau_i | u_i > 1/3\}$
        Processors are also partitioned into two groups such that tasks
        in a group must be assigned to processors in a group.
(2) Assign tasks in the first group, $\Re_1$, to processors using the RMST
        algorithm.
(3) Assign tasks in the second group, $\Re_2$, to processors as follows:
     Use the First-Fit heuristic to find a processor $P_i$ that contains
     exactly one task and assign task $\tau_i$ to processor $P_i$ if the two
     tasks can be feasibly scheduled according to the following con-
     dition:
          min := i; max := j;
          If ($T_j > T_i$) Then {min := j; max := i;};
          If ($\lfloor T_{max}/T_{min} \rfloor C_{min} + C_{max} \le \lfloor T_{max}/T_{min} \rfloor T_{min}$) Or ($\lceil T_{max}/T_{min} \rceil$
            $C_{min} + C_{max} \le T_{max}$) Then feasible := True
          Else feasible := False
     Otherwise, assign a task to an empty processor.Terminate when all
     tasks in the group have been assigned
```

Figure 3.7: Algorithm RMGT

The reason that 1/3 is chosen in dividing the task set will become clear after the proof of the following theorem is presented.

Theorem 3.15: $\Re^{\infty}_{RMGT} = 7/4$.

Proof: Let $\Sigma = \{\tau_1, \tau_2, \ldots, \tau_m\}$ be a set of m tasks with their utilizations $u_1, u_2, \ldots, u_m$. Then the total utilization of the task set is given by $\sum_{i=1}^{m} u_i$. Let N and N_0 be the number of processors required by RMGT and the minimum number of processors required to schedule Σ, respectively. Suppose that $N = N_1 + N_2$ processors are used by RMGT to schedule the task set Σ, where N_1 and N_2 are the numbers of processors allocated in the first processor group and the second processor group, respectively. Among the N_2 processors, let n_1 be the number of processors assigned one task and n_2 be the number of processors assigned two tasks. Then $N_2 = n_1 + n_2$.

In the first processor group, the following holds from (3.28) in Theorem 3.14

$$N_1 \le \frac{1}{1-\alpha}\sum_{j=1}^{N_1} U_j + \left(1 + \frac{\ln 2}{1-\alpha}\right) \qquad \text{(Eq.3.29)}$$

where $\alpha = \max\limits_{i=1,\ldots,n} (C_i / T_i)$.

Since $\alpha = 1/3$ in the first task group, it follows from (3.29) that

$$\frac{2}{3} + \ln 2 + \frac{2}{3}N_1 < \sum_{j=1}^{N_1} U_j \qquad \text{(Eq.3.30)}$$

In the second processor group, for the n_1 processors to each of which one task is assigned,

$$\sum_{i=1}^{n_1} u_i > n_1 / \left(1 + 2^{1/n_1}\right) > n_1 / 2 - \ln 2 / 4. \qquad \text{(Eq.3.31)}$$

For the n_2 processors to each of which two tasks are assigned, we have

$$\sum_{i=1}^{n_2} (u_{i,1} + u_{i,2}) > (2n_2)/3 \qquad \text{(Eq.3.32)}$$

since $u_{i,1} > 1/3$ and $u_{i,2} > 1/3$.

Since $\sum_{i=1}^{n_1} u_i > n_1 / 2 - \ln 2 / 4$, $\sum_{i=1}^{n_2} (u_{i,1} + u_{i,2}) > (2n_2)/3$, and $\sum_{i=1}^{N_1} u_i \ge \frac{2}{3} + \ln 2 + \frac{2}{3}N_1$, we have

$$\sum_{i=1}^{m} u_i = \sum_{i=1}^{n_1} u_i + \sum_{i=1}^{n_2} (u_{i,1} + u_{i,2}) + \sum_{i=1}^{N_1} U_i$$
$$\ge n_1 / 2 - \ln 2 / 4 + (2n_2)/3 + \frac{2}{3} + \ln 2 + \frac{2}{3}N_1 .$$

Since $N_0 \ge \sum_{i=1}^{m} u_i$ and $N = n_1 + n_2 + N_1$, it is immediate that

$$N_0 \ge \frac{2N}{3} + \frac{2}{3} + \frac{3\ln 2}{4} - \frac{n_1}{6}$$

Since any two of the tasks that are assigned to the n_1 processors cannot be scheduled on a single processor, we have $N_0 \ge n_1$.

Then $N_0 \ge \frac{2N}{3} + \frac{2}{3} + \frac{3\ln 2}{4} - \frac{n_1}{6} \ge \frac{2N}{3} + \frac{2}{3} + \frac{3\ln 2}{4} - \frac{N_0}{6}$.

$$\frac{N}{N_0} \le \frac{7}{4} - \left(1 + \frac{3\ln 2}{4}\right)\frac{1}{N_0} \qquad \text{(Eq.3.33)}$$

Therefore, $\Re^{\infty}_{RMGT} \le 7/4$.

In order to show that the bound as given above is tight, we construct the following task set such that $N / N_0 = 7/4$.

For any positive integer *m*, the task set consists of 13*m* tasks. We select two sufficiently small positive number ε and δ such that $\delta << 1/(6m)$ and

$$\frac{1}{2}2^{-\varepsilon} + \frac{1}{6} - \frac{2}{3}2^{-2\varepsilon} \le \delta \le \frac{3}{2}\varepsilon\ln 2 \qquad \text{(Eq.3.34)}$$

We label the tasks as $\tau_{i,j}$ for $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, 13$. The order of the tasks that are given is not important, since the tasks will be sorted according to their V values.

The tasks are given by

$$C_{i,j} = u_{i,j}T_{i,j},\ T_{i,j} = \varphi 2^{V_{i,j}},\ \varphi > 0,$$

$$V_{i,j} = \begin{cases} (12i + j)\,\varepsilon & 1 \le j \le 12 \\ (12i + 11)\,\varepsilon & i = 13 \end{cases}, \text{ and}$$

$$u_{i,j} = \begin{cases} 1/2 & j = 2, 5, 8, 11 \\ 1/3 & j = 1, 6, 10 \\ 1/6 - \delta & j = 3, 4, 7, 9, 12, 13 \end{cases}$$

For RMGT to schedule the task set, it divides it into two task groups:

$$\Re_1 = \{\tau_{i,j} | i = 1, \ldots, m, j = 1, 3, 4, 6, 7, 9, 10, 12, 13\}$$

$$\Re_2 = \{\tau_{i,j} | i = 1, \ldots, m, j = 2, 5, 8, 11\}$$

In the completed RMGT schedule, 4*m* processors are required to schedule the tasks in the $\Re_2$ since no two tasks in the group can be scheduled on a processor. This can be verified by the necessary and sufficient condition.

For the $\Re_1$ task group, 3*m* processors are required to schedule by RMGT. The processor assignment is given by

$$\{\tau_{i,1}, \tau_{i,3}, \tau_{i,4}\},\ \{\tau_{i,6}, \tau_{i,7}, \tau_{i,9}\},\ \{\tau_{i,10}, \tau_{i,13}, \tau_{i,12}\}, \text{ for } i = 1, 2, \ldots, m.$$

This can be verified by the schedulability condition given in the RMGT algorithm.

In the optimal schedule, a total of $4m$ processors is required. The processor assignment is given by

$$\{\tau_{i,1}, \tau_{i,2}, \tau_{i,3}\}, \{\tau_{i,4}, \tau_{i,5}, \tau_{i,6}\}, \{\tau_{i,7}, \tau_{i,8}, \tau_{i,9}, \tau_{i,13}\}, \{\tau_{i,10}, \tau_{i,11}, \tau_{i,12}\}$$

for $i = 1, 2, \ldots, m$. This can be verified to be so by the necessary and sufficient condition. This assignment is optimal since the total utilization (load) of the task set is given by $(4 - 6\delta)m > 4m - 1$ since $\delta << 1/(6m)$.

Hence, $N / N_0 = 7/4$.

Therefore, $\Re^{\infty}_{RMGT} = 7/4$. ■

It is clear that 1/3 is chosen in dividing the task set because $\alpha = 1/3$ satisfying the relationship $1 - \alpha = 2\alpha$.

In the following, we present an on-line version of the RMST algorithm described earlier. The idea is to divide the incoming tasks into classes such that the utilization of a processor can be increased by lowering the value of β in the schedulability condition:

$$U_m + u_i \le \max\{\ln 2, 1 - \beta \ln 2\} \qquad \text{(Eq.3.35)}$$

We refer the new algorithm as Rate-Monotonic-General-Task-M (RMGT-M). The parameter in the algorithm, M, denotes the number of classes a task set is divided into. The processors are also divided into M classes such that tasks in the kth class are assigned to processors in the kth class. The class membership of a task is determined by the following expression:

$$m = \lfloor M(\log_2(T) - \lfloor \log_2(T) \rfloor) \rfloor + 1 .$$

Then for each processor the value of β as defined in (3.35) is bounded above by 1/ M. For each class, the algorithm keeps one so-called current processor. If a new task from class k is added to the task set, then the algorithm first attempts to assign the task to the current processor in the kth class. If the task can be scheduled on the current processor according to the above condition (3.35), then add the task to it. Otherwise, the task is assigned to an empty processor, which in turn becomes the current processor. Note that an improvement can be made here: instead of choosing the newly used processor as the current processor, we can choose the one with a smaller utilization (load) between the newly used one and the "current" processor. Since this modification does not improve the worst case performance (but improves average case performance), we will not consider it here. A complete description of the algorithm is given in Figure 3.8.

```
**Rate-Monotonic-General-Task-M(RMGT-M)** (Input: task set Σ; Output: m)
(1) $m := \lfloor M(\log_2(T) - \lfloor \log_2(T) \rfloor) \rfloor + 1$.
(2) Assign task $\tau_i$ to the current processor $P_m$ in the mth class if
    this task together with the tasks that have been assigned to $P_m$
    can be feasibly scheduled according to the condition:
    $U_m + u_i \le 1 - (\ln 2)/M$
    If not, assign task $\tau_i$ to $P_{m+1}$ and let $P_{m+1}$ becomes the current
      processor.
```

Figure 3.8: Algorithm RMGT-M

If we define $\alpha = \max_{i=1,\ldots,n}(C_i/T_i)$, we have the following theorem.

Theorem 3.16: *Let N and N_0 be the number of processors required by RMGT-M and the minimum number of processors required to feasibly schedule a given set of tasks, respectively. Then*

$$N < \frac{N_0}{1 - (\ln 2)/M - \alpha} + M \tag{Eq.3.36}$$

if $\alpha \le (1 - (\ln 2)/M)/2$;

$$N < \frac{2N_0}{1 - (\ln 2)/M} + M \tag{Eq.3.37}$$

if $\alpha > (1 - (\ln 2)/M)/2$.

Proof: According to the schedulability condition used in RMGT-M, the total utilization of any busy processor except the current processors exceeds $1 - (\ln 2)/M - \alpha$. In other words,

$$\sum_{i=1}^{n} u_i \ge (N - M)[1 - (\ln 2)/M - \alpha].$$

Since $N_0 \ge \sum_{i=1}^{n} u_i$, we have $N_0 \ge (N - M)[1 - (\ln 2)/M - \alpha]$.

If $\alpha \leq (1-(\ln 2)/M)/2$, we have (3.36). If $\alpha > (1-(\ln 2)/M)/2$, we have (3.37). ■

Corollary 3.1**:** $\Re^{\infty}_{\text{RMGT-M}} \leq \dfrac{1}{1-(\ln 2)/M}$.

Proof: For RMGT-M to be used as an on-line algorithm, the utilization of a task can assume an arbitrary value in the range of zero and one since the characteristics of the incoming tasks are unknown. Therefore, in the worst case,

$$N < \frac{2N_0}{1-(\ln 2)/M} + M$$

When N_0 becomes large, the term M disappears. Hence, $\Re^{\infty}_{\text{RMGT-M}} \leq \dfrac{1}{1-(\ln 2)/M}$. ■

From the derivation of the bounds we can see that the performance of RMGT-M is sensitive to the choice of M, the number of classes in a task set. In practice, it is sufficient for M to assume a value in the range of [5, 100]. The worst case bounds improve for large values of M. However, M also determines the number of current processors, which may not be fully utilized. Next we consider the problem of optimally selecting the value of M such that the worst case bounds are lowest.

Let us assume that the total load of the task set is known. To find the value of M that gives the lowest worst case bound for the number of processors in (3.37), we fix the value of $U = \sum_{i=1}^{n} u_i$. Since we derive both (3.36) and (3.37) through replacing $\sum_{i=1}^{n} u_i$ by N_0, the inequalities hold when we use U in the place of N_0. Since both of the right hand sides of (3.36) and (3.37) are convex functions in terms of M, we solve them for the minimum and we obtain

$$M^* = \sqrt{2U\ln 2} + \ln 2 \qquad \text{(Eq.3.38)}$$

for the right hand side of (3.37), and

$$M^* = \frac{\sqrt{U\ln 2} + \ln 2}{1-\alpha} \qquad \text{(Eq.3.39)}$$

for the right hand side of (3.36).

This suggests to us that if we choose $M \sim \sqrt{U}$, we obtain

$$\frac{N}{U} \le \frac{2M}{M - \ln 2} + \frac{M}{U} \to 2 + O(1/(\sqrt{U})) \qquad \text{(Eq.3.40)}$$

and

$$\frac{N}{U} \le \frac{MU}{(1-\alpha)M - \ln 2} + M \to \frac{1}{1-\alpha} + O(1/(\sqrt{U})) \qquad \text{(Eq.3.41)}$$

when $U \to \infty$.

Therefore, the lowest worst case bounds we can get for RMGT-M is 2 for $\alpha > (1 - \alpha \ln 2 / M)/2$ and $1 / (1 - \alpha)$ for $\alpha \le (1 - \alpha \ln 2/M)/2$.

Since $U \to \infty$ and $M \sim \sqrt{U}$, M must also approach infinity. Therefore, these bounds may only be of theoretical interest.

3.7. Rate-Monotonic-First-Fit-Decreasing-Utilization

The algorithm RM-FFDU is based on the bin-packing heuristic of First-Fit-Decreasing (FFD). It has been known that if a list of items is sorted in the order of non-increasing size, then the performance of such bin-packing heuristics as First-Fit and Best-Fit can be improved significantly. Davari and Dhall's FFDUF is an example of applying the First-Fit heuristic to schedule a set of tasks sorted in the order of non-increasing utilization. However, their FFDUF only has a worst case bound of 2, in part because of the WC schedulability condition used. Next, we will describe a new algorithm which is also based on the FFD heuristic, but uses the UO condition for schedulability test on each processor. The algorithm is called RM-FFDU and has a worst case bound of 5/3, the best in the literature to date. RM-FFDU is given in Figure 3.9.

When the algorithm returns, m is the number of processors required by RM-FFDU to schedule the task set Σ. k_j is the number of tasks assigned on processor P_j. We state the upper bound in the following theorem.

Theorem 3.17: $\Re^{\infty}_{RM-FFDU} \le \frac{5}{3}$.

To prove Theorem 3.17, we need to introduce some notations.

Let x be the first item assigned to the last processor in the completed RM-FFDU

RM-First-Fit-Decreasing-Utilization (RM-FFDU) (Input: task set Σ; Output: m)

```
(1) Sort the task set in the order of non-increasing utilization.
(2) i:= 1; m:= 1;
(3) j:= 1; While ($u_i > 2/\left(\prod_{l=1}^{k_j}(u_{j,l}+1)\right) - 1$) Do {j := j + 1;};
(4) $k_j := k_j + 1$; $U_j := U_j + u_i$; /* Assign task $\tau_i$ to $P_j$ */
(5) If (j > m) Then {m := j;}
(6) i := i + 1;
(7) If (i > n) Then {Exit;} Else {Goto 3;}
```

Figure 3.9: Algorithm RM-FFDU

schedule. If $x \le \ln 2 - 3/5$, then every busy processor is allocated a utilization at a level that is at least $\ln 2 - x \ge 3/5$. Then $\Re^{\infty}_{RM-FFDU} \le \frac{5}{3}$. If $x > 1/2$, then $N = N_0$, where N_0 is the minimum number of processors required to schedule the task set Σ and N is the number of processors required by RM-FFDU to schedule Σ.

Lemma 3.9: *If $2^{1/(c+1)} - 1 < y \le 2^{1/c} - 1$ for $c \ge 1$ in the completed RM–FFDU schedule, then among all processors on each of which at least c tasks are assigned, there are at most one processor to which not all the first c tasks are assigned tasks each with a utilization greater than $2^{1/(c+1)} - 1$.*

Proof: This lemma is proven by contradiction.

Suppose that there are two such processors P_i and P_j with $i < j$ such that each of them is assigned at least c tasks. Furthermore, let $\tau_{i,k}$ and $u_{i,k}$ be the task assigned to processor P_i and its utilization, respectively. Then for processor P_i, there exists at least one task $\tau_{i,m}$ with $m \le c$ having a utilization $u_{i,m} \le 2^{1/(c+1)} - 1$.

For processor P_i, since $2^{1/(c+1)} - 1 < y \le 2^{1/c} - 1$ for $c \ge 1$ ($y = u_{i,1}$ by definition), all the tasks yet to be assigned after task $\tau_{i,1}$ have utilizations no greater than $u_{i,1}$. Furthermore, at least c tasks can be assigned on processor P_i because $u_{i,1} \le 2^{1/c} - 1$.

Since $2^{1/(c+1)} - 1 < y \le 2^{1/c} - 1$ for processor P_j with $i < j$, $\tau_{i,m}$ with $m \le c$ must be assigned to processor P_i after the task $\tau_{j,1}$ is assigned to processor P_j. This could only happen when the first task assigned to processor P_j cannot be assigned to processor P_i, since $u_{j,1} > 2^{1/(c+1)} - 1 \ge u_{i,m}$.

Since $U_i + u_{j,1} \geq c(2^{1/c} - 1)$, where U_i is the total utilization assigned to processor P_i when processor P_j was first assigned the task $\tau_{j,1}$, processor P_i must have been assigned c or more tasks each with a utilization equal to or greater than $u_{j,1} > 2^{1/(c+1)} - 1$. This is a contradiction to the assumption that there exists at least one task $\tau_{i,m}$ having a utilization $u_{i,m} \leq 2^{1/(c+1)} - 1$ with $m \leq c$ on processor P_i.

Therefore, the lemma must be true. ■

Before we move on, let us obtain the upper bound (not tight) for some of the values of x.

Lemma 3.10: *For some values of* x, $\Re^{\infty}_{RM-FFDU}$ *is given as in Table 3.3.*

Table 3.3: Performance of RM–FFDU for some values of x

c	$x \geq$	$\Re^{\infty}_{RM-FFDU} \leq$	c	$x \geq$	$\Re^{\infty}_{RM-FFDU} \leq$
2	0.4142	2.4	5	0.1487	1.68
3	0.2599	1.92	6	0.1245	1.63
4	0.1892	1.76	∞	$\to 0$	1/ln2

Proof: Since x is the utilization of the first task on the last processor, we can assume, without loss of generality, that the first task on the last processor is the last task in the task set after sorting. Note that the tasks following the first task on the last processor do not affect the number of processors used by RM-FFDU if they are included in the task set. Therefore, we can further assume that the utilization of any other task in the task set is equal to or larger than x.

For any value of c such that $(2^{1/(c+1)} - 1) \leq x \leq (2^{1/c} - 1)$, suppose $n \geq 1$ tasks are assigned to a processor P_i with a total utilization of U_i, for $c = 0, 1, 2, \ldots$

If $n \geq c$, then $U_i \geq c(2^{1/(c+1)} - 1)$.

If $n < c$, then $c \geq 2$ and $x > n(2^{1/n} - 1) - U_i > c(2^{1/c} - 1) - U_i$. $U_i > (c - 1)(2^{1/c} - 1)$.

In summary, every processor has a utilization greater than $(c - 1)(2^{1/c} - 1)$ for $c \geq$

2. Since the utilization of a processor cannot exceed one in the optimal schedule. $\Re^{\infty}_{RM-FFDU} \leq 1/[(c-1)*(2^{1/c}-1]$ for $c > 1$. A few values of $\Re^{\infty}_{RM-FFDU}$ are given in Table 3.3 for some values of c. ■

In the RM-FFDU schedule, let N_i be the number of processors to each of which i tasks are assigned. Then $N = \sum_{i=1}^{\kappa} N_i$, where κ is the maximum number of tasks assigned to a processor. Then the total number of tasks in the task set is given by $n = \sum_{i=1}^{\kappa} iN_i$. In the optimal schedule, let M_i be the number of processors to each of which i tasks are assigned. The minimum number of processors required is $N_0 = \sum_{i=1}^{\kappa} M_i$ and $n = \sum_{i=1}^{\kappa} iM_i$. We are trying to find the maximum of $\Re^{\infty}_{RM-FFDU}$ for any value of x. Let us define y_i to be the utilization of the first task assigned to a processor P_i. Then it is immediate for RM-FFDU that $y_i \geq y_j$ if $i < j$. Where there is not confusion, we simply use y to denote the utilization of the first task assigned to a processor.

For those processors to each of which n tasks are assigned, their minimum total utilization can be determined by the following method: Since $x > 2/\prod_{i=1}^{n}(1+u_i) - 1$, the minimum of $U = \sum_{i=1}^{n} u_i$ is achieved at $U = n\{[2/(1+x)]^{1/n} - 1\}$ when $u_1 = u_2 = \ldots = u_n = [2/(1+x)]^{1/n} - 1$.

In the subsequent lemmas, we will prove that $\Re^{\infty}_{RM-FFDU} \leq 5/3$ with regard to x. We divide the range of values x can assume into several intervals and prove that $\Re^{\infty}_{RM-FFDU} \leq 5/3$ for each interval:

$x \in (1/3, 1/2]$, $x \in (1/4, 1/3]$, $x \in (1/5, 1/4]$, $x \in (2^{1/4} - 1, 1/5]$, $x \in (1/6, 2^{1/4} - 1]$, $x \in (5(2^{1/5} - 1) - 3/5, 1/6]$, and $x \in (6(2^{1/6} - 1) - 3/5, 5(2^{1/5} - 1) - 3/5]$.

The final proof of Theorem 3.17 appears after the lemmas.

Lemma 3.11: *If* $x \in (1/3, 1/2]$, *then* $\Re^{\infty}_{RM-FFDU} \leq 3/2$.

Proof: Since $x > 1/3$, a processor cannot be assigned more than 2 tasks, i.e., $\kappa < 3$. Each processor is assigned one or two tasks in either the RM-FFDU schedule or the optimal schedule.

Let n be the total number of tasks in a task set. The optimal number of processors required is $N_0 = \sum_{i=1}^{2} M_i$. Furthermore, $n = \sum_{i=1}^{2} iM_i$. In the RM-FFDU schedule, $N = \sum_{i=1}^{2} N_i$ and $n = \sum_{i=1}^{2} iN_i$.

Then the ratio $\Re_{RM-FFDU}^{\infty} = N/N_0$ is maximized when and $M_1 = 0$ and $M_2 = n/2$. The maximum value is achieved at $\Re_{RM-FFDU}^{\infty} = 3/2$. ■

Lemma 3.12: *If* $x \in (1/4, 1/3]$, *then* $\Re_{RM-FFDU}^{\infty} \le 3/2$.

Proof: Since $x > 1/4$, a processor can be assigned no more than 3 tasks, i.e., $\kappa = 3$. In the RM-FFDU schedule, let us consider all the processors to each of which one task is assigned. Let u be the utilization of the only task assigned to a processor. Then $u > (1 - 1/3) / (1 + 1/3) = 1/2$. In other words, among all the processors to each of which one task is assigned, the utilization of the task is greater than 1/2. Therefore, if there are N_1 such processors in the RM-FFDU schedule, then at least N_1 processors are needed in the optimal schedule.

For $\Re_{RM-FFDU}^{\infty} = N / N_0$, suppose $N_1 = N_2 = 0$, then $\Re_{RM-FFDU}^{\infty} = 1$ since a processor is assigned at most three tasks. If $N_1 = N_3 = 0$, then the maximum value $\Re_{RM-FFDU}^{\infty}$ can achieve is 3/2 since at most three tasks can be assigned on one processor. If $N_2 = N_3 = 0$, then $\Re_{RM-FFDU}^{\infty} = 1$.

If $N_1 = 0$, then the maximum of $\Re_{RM-FFDU}^{\infty}$ is 3/2 since at most three tasks can be assigned on one processor. If $N_2 = 0$, then for the N_1 processors in the optimal schedule, each can only be assigned at most two tasks. Therefore the maximum of $\Re_{RM-FFDU}^{\infty}$ is achieved when $N_1 = 3N_3$ such that $\Re_{RM-FFDU}^{\infty} \le 4/3$. If $N_3 = 0$, $\Re_{RM-FFDU}^{\infty} \le 3/2$ for similar reason.

If $N_i \ne 0$ in the RM-FFDU schedule, then $\Re_{RM-FFDU}^{\infty} \le 3/2$. Suppose that in the best case where each of the N_1 processors on each of which a task with a utilization greater than 1/2 is assigned is assigned two tasks totally. Then the minimum number of processors required (in the optimal schedule) is at least $N_1 + (2N_2 + 3N_3 - N_1)/3$. Therefore $\Re_{RM-FFDU}^{\infty} \le (N_1 + N_2 + N_3) / (N_1 + (2N_2 + 3N_3 - N_1)/3) \le 3/2$. ■

Lemma 3.13: *If* $x \in (1/5, 1/4]$, *then* $\Re^{\infty}_{RM-FFDU} \le 3/2$.

Proof: Since $x > 1/5$, a processor is assigned at most four tasks, i.e., $\kappa = 4$.

For those processors to each of which one task is assigned, the utilization of the task is greater than $(1 - 1/4) / (1 + 1/4) = 3/5$.

For those processors to each of which two tasks are assigned, the minimum of $u_1 + u_2$ is achieved at $2[\sqrt{2/(1+x)} - 1]$ when $u_1 = u_2 = \sqrt{2/(1+x)} - 1$. Then $u_1 = u_2 = \sqrt{8/5} - 1 = 0.2649$, and $U = 0.529$ for $x = 1/4$. Note that for $x > 1/4$, $U > 0.529$. Two more tasks can be assigned on these processors in the optimal schedule.

For those processors to each of which three or four tasks are assigned, their minimum total utilization is determined at $U \ge 3x > 0.6$, when $u_i = 1/5$.

In the following, we define a function that maps the utilization of a task to a value that is in the range of 0 and 1, as given in Table 3.4. We call that value the weight of the task, and the sum of the weights of the tasks assigned to a processor the weight of the processor. The weighting function is designed in such a way that for every processor in the RM-FFDU schedule, its weight is equal to or greater than 1. At the meantime, the weight of a processor in the optimal schedule is no greater than 5/3. We first claim that for any processor P in the completed RM-FFDU schedule, the total weight of processor P is equal to or greater than 1, i.e., $W(P) = \sum_{i=1}^{k} W(u_i) \ge 1$, where k is the number of tasks assigned to processor *P*.

Table 3.4: Weighting Function for x $\in (1/5, 1/4]$

$W(u) =$	$u \in$
0	$(0, 1/5]$
1/3	$(1/5, \sqrt{8/5} - 1]$
1/2	$(\sqrt{8/5} - 1, 2^{1/2} - 1]$
2/3	$(2^{1/2} - 1, 3/5]$
1	$(3/5, 1]$

In the completed RM-FFDU schedule, the utilization of the first task assigned on any processor must be equal to or greater than x, i.e., $y \geq x$. Let us consider a processor to which is first assigned a task with a utilization of y.

Case 1: $1/5 < y \leq \sqrt{8/5} - 1$. Then the processor must be assigned at least three tasks each with a utilization greater than 1/5. Therefore, $W(P) \geq 3 \bullet 1/3 = 1$.

Case 2: $\sqrt{8/5} - 1 < y \leq 2^{1/2} - 1$. Then the processor must be assigned at least two tasks. Furthermore, except for possibly one processor by Lemma 3.9, each of the first two tasks must have a utilization greater than $\sqrt{8/5} - 1$. Therefore, $W(P) \geq 1$.

Case 3: $2^{1/2} - 1 < y \leq 3/5$. Then the processor must be assigned at least two tasks. Since the second task must be a task with a utilization greater than 1/5, we have W(P) ≥ 1.

Case 4: $3/5 < y \leq 1$. Then W(P) ≥ 1 by definition.

We then claim that for any processor *P* in the optimal schedule, W(P) ≤ 3/2.

Let us assume that a processor in the optimal schedule is assigned *m* tasks with their utilizations as $u_1 \geq u_2 \geq \ldots\ldots \geq u_m$.

Case I: $u_1 < \sqrt{8/5} - 1$. Then at most four tasks each with a utilization greater than 1/5 can be assigned on it. Therefore, W(P) ≤ 4/3.

Case II: $\sqrt{8/5} - 1 < u_1 < 2^{1/2} - 1$ and $\sqrt{8/5} - 1 < u_2$. Then at most one more task can be assigned to the processor. If $u_3 < \sqrt{8/5} - 1$, then W(P) = 1/2 + 1/2 + 1/3 = 4/3. If $u_3 > \sqrt{8/5} - 1$, then W(P) = 1/2 + 1/2 + 1/2 = 3/2. If $\sqrt{8/5} - 1 < u_1 < 2^{1/2} - 1$ and $u_2 < \sqrt{8/5} - 1$, then at most two more tasks are assigned to the processor. Therefore, W(P) = 1/2 + 1/3 + 1/3 + 1/3 = 3/2.

Case III: $2^{1/2} - 1 < u_1 < 3/5$ and $u_2 > 2^{1/2} - 1$. Then no more task with a utilization greater than 1/5 can be assigned to the processor. Therefore, W(P) = 2/3 + 2/3 = 4/3.

Case IV: $2^{1/2} - 1 < u_1 < 3/5$ and $\sqrt{8/5} - 1 < u_2 < 2^{1/2} - 1$. Then no more task with a utilization greater than 1/5 can be assigned to the processor. Therefore, W(P) = 2/3 + 1/2 = 7/6. If $2^{1/2} - 1 < u_1 < 3/5$ and $u_2 < \sqrt{8/5} - 1$, then at most one more task with a utilization greater than 1/5 can be assigned to the processor. Therefore, W(P) = 2/3 + 1/3 + 1/3 = 4/3.

Case V: $3/5 < u_1 < 1$. Then at most one more task with a utilization greater than 1/5 can be assigned to the same processor. Furthermore, $u_2 < 2^{1/2} - 1$. Then W(P) ≤ 1 + 1/2 = 3/2.

Let N and N_0 be number of processors required by RM-FFDU and the minimum number of processors required to schedule a given set Σ of n tasks, respectively. Then the total weight of the task set is given by $\sum_{i=1}^{n} W(u_i)$. Since, except for one processor, W(P) ≥ 1 for every processor in the RM-FFDU schedule, then $\sum_{i=1}^{n} W(u_i) \geq N - 1$. Since W(P) ≤ 3/2 for every processor in the optimal schedule, $3N_0/2 \geq \sum_{i=1}^{n} W(u_i)$. Therefore, $\Re_{RM-FFDU}^{\infty} \leq 3/2$. ■

Lemma 3.14: *If* $x \in (2^{1/4} - 1, 1/5]$, *then* $\Re_{RM-FFDU}^{\infty} \leq 5/3$.

Proof: Since $x > 2^{1/4} - 1 \approx 0.1892$, a processor is assigned at most five tasks, i.e., $\kappa = 5$.

For those processors to each of which one task is assigned, the utilization of each task is greater than (1 – 1/5) / (1+ 1/5) = 2/3.

For the processor to which two tasks are assigned, the minimum of $u_1 + u_2$ is achieved at $U = 2[\sqrt{2/(1+x)} - 1]$ when $u_1 = u_2 = \sqrt{2/(1+x)} - 1$. Then $u_1 = u_2 = \sqrt{5/3} - 1 \approx 0.29$, and $U = 2(\sqrt{5/3} - 1) \approx 0.58$ for $x = 1/5$. Note that for $x < 1/5$, $U > 0.58$.

For a processor to which three tasks are assigned, their minimum utilization is achieved at $U = 3\{[2/(1+x)]^{1/3} - 1\}$ when $u_i = [2/(1+x)]^{1/3} - 1$. We want to find the x such that $x \leq [2/(1+x)]^{1/3} - 1$. Solving the inequality $x \leq [2/(1+x)]^{1/3} - 1$ yields $x \leq 2^{1/4} - 1$. In other words, for every processor to which three tasks are assigned in the completed RM–FFDU schedule, their total utilization is greater than $3(2^{1/4} - 1) \approx 0.5676$.

In the following, we define a function that maps the utilization of a task to a value that is in the range of 0 and 1, as given by Table 3.5. The weighting function is designed in such a way that for every processor in the RM-FFDU schedule, its weight is equal to or greater than 1. At the meantime, the weight of a processor in the optimal schedule is no greater than 5/3.

Table 3.5: Weighting Function for x $\in (2^{1/4} - 1, 1/5]$

W(u) =	$u \in$
0	$(0, 2^{1/4} - 1]$
1/3	$(2^{1/4} - 1, \sqrt{5/3} - 1]$
1/2	$(\sqrt{5/3} - 1, 2^{1/2} - 1]$
2/3	$(2^{1/2} - 1, 2/3]$
1	(2/3, 1]

We first claim that for every processor *P* in the completed RM-FFDU schedule, W(P) ≥ 1.

Since $x \in (2^{1/4} - 1, 1/5]$, the utilization of the first task assigned on any processor in the completed RM-FFDU schedule must be equal to or greater than *x*, i.e., $y \geq x$. Let us consider a processor to which is first assigned a task with a utilization of *y*.

Case 1: $2^{1/4} - 1 < y \leq \sqrt{5/3} - 1$. Then the processor must be assigned at least three tasks. Therefore, W(P) ≥ 1/3 + 1/3 + 1/3 = 1.

Case 2: $\sqrt{5/3} - 1 < y \leq 2^{1/2} - 1$. Then the processor must be assigned at least two tasks. Furthermore, except for one processor by Lemma 3.9, each of the first two tasks must have a utilization greater than $\sqrt{5/3} - 1$. Therefore, W(P) ≥ 1/2 + 1/2 = 1.

Case 3: $2^{1/2} - 1 < y \leq 2/3$. Then the processor must be assigned at least two tasks. Since the second task must be a task with a utilization greater than $2^{1/2} - 1$, we have W(P) ≥ 2/3 + 1/3 = 1.

Case 4: 2/3 < y ≤ 1, W(P) ≥ 1 by definition.

We then claim that for any processor in the optimal schedule, *W(P)* ≤ 5/3.

Let us assume that a processor in the optimal schedule is assigned *m* tasks with their utilizations as $u_1 \geq u_2 \geq \ldots\ldots \geq u_m$.

Case I: $u_1 < \sqrt{5/3} - 1$. Then at most five tasks each with a utilization greater than $2^{1/4} - 1$ can be assigned on a processor. Therefore, W(P) ≤ 5/3.

Case II: $\sqrt{5/3} - 1 < u_1 < 2^{1/2} - 1$ and $\sqrt{5/3} - 1 < u_2$. Then at most two more tasks can be assigned to the processor. If $u_3 < \sqrt{5/3} - 1$, then W(P) ≤ 1/2 + 1/2 + 1/3 + 1/3 = 5/3. If $u_3 > \sqrt{5/3} - 1$, then W(P) = 1/2 + 1/2 + 1/2 = 3/2. If $\sqrt{5/3} - 1 < u_1 < 2^{1/2} - 1$ and $u_2 < \sqrt{5/3} - 1$, then at most two more tasks each with a utilization less than $\sqrt{5/3} - 1$ can be assigned to the processor. Therefore, W(P) ≤ 1/2 + 1/3 + 1/3 + 1/3 = 3/2.

Case III: $2^{1/2} - 1 < u_1 < 2/3$ and $u_2 > 2^{1/2} - 1$. Then no more task with a utilization greater than $2^{1/4} - 1$ can be assigned to the processor. Therefore, W(P) ≤ 2/3 + 2/3 = 4/3.

If $2^{1/2} - 1 < u_1 < 2/3$ and $\sqrt{5/3} - 1 < u_2 < 2^{1/2} - 1$, then at most one task with a utilization greater than $2^{1/4} - 1$ and less than $2^{1/2} - 1$ can be assigned to the processor. Therefore, W(P) = 2/3 + 1/2 + 1/2= 5/3.

If $2^{1/2} - 1 < u_1 < 2/3$ and $u_2 < \sqrt{5/3} - 1$, then at most two more task with a utilization greater than $2^{1/4} - 1$ can be assigned to the processor. Therefore, W(P) ≤ 2/3 + 1/3 + 1/3 + 1/3 = 5/3.

Case IV: 2/3 < u_1 < 1. Then at most one more task with a utilization greater than $2^{1/4} - 1$ and less than $2^{1/4} - 1$ can be assigned to the same processor. Furthermore, $u_2 < 2^{1/2} - 1$. Then W(P) ≤ 1 + 1/2 = 3/2.

Let N and N_0 be number of processors required by RM-FFDU and the minimum number of processors required to schedule a given set Σ of *n* tasks, respectively. Then the total weight of the task set is given by $\sum_{i=1}^{n} W(u_i)$. Since W(P) ≥ 1 for every processor in the RM-FFDU schedule, then $\sum_{i=1}^{n} W(u_i) \geq N - 1$. Since W(P) ≤ 5/3 for every processor in the optimal schedule, $5N_0/3 \geq \sum_{i=1}^{n} W(u_i)$. Therefore, $\Re^{\infty}_{RM-FFDU} \leq 5/3$. ■

Lemma 3.15: *If* $x \in (1/6,\ 2^{1/4} - 1]$, *then* $\Re^{\infty}_{RM-FFDU} \leq 5/3$.

Proof: Since *x* > 1/6, a processor is assigned at most five tasks, i.e., κ = 5.

For those processors to each of which one task is assigned, the utilization of each task is greater than $[1 - (2^{1/4} - 1)] / (1 + 2^{1/4} - 1) = 2^{3/4} - 1 \approx 0.68$.

For the processor to which two tasks are assigned, the minimum of $u_1 + u_2$ is achieved at $U = 2[\sqrt{2/(1+x)} - 1]$ when $u_1 = u_2 = \sqrt{2/(1+x)} - 1$. Then $u_1 = u_2 = 2^{3/8} - 1 \approx 0.297$, and $U = 2(2^{3/8} - 1) \approx 0.594$ for $x = 2^{1/4} - 1 \approx 0.1892$. Note that for $x < 2^{1/4} - 1$, $U > 0.594$.

For a processor to which three tasks are assigned, their minimum utilization is achieved at $U = 3\{[2/(1+x)]^{1/3} - 1\}$ when $u_i = [2/(1+x)]^{1/3} - 1$. We want to find the x such that $x \le [2/(1+x)]^{1/3} - 1$. Solving the inequality $x \le [2/(1+x)]^{1/3} - 1$ yields $x \le 2^{1/4} - 1$. In other words, for every processor to which three tasks are assigned in the completed RM-FFDU schedule, their total utilization is greater than $3(2^{1/4} - 1) \approx 0.5676$.

For $y < 2^{1/4} - 1$, each processor must be assigned at least four tasks each with a utilization less than $2^{1/4} - 1$.

In the following, we define a function that maps the utilization of a task to a value that is in the range of 0 and 1, as given by Table 3.5. The weighting function is designed in such a way that for every processor in the RM-FFDU schedule, its weight is equal to or greater than 1. At the meantime, the weight of a processor in the optimal schedule is no greater than 5/3.

We first claim that for every processor P in the completed RM-FFDU schedule,

Table 3.6: Weighting Function for x $\in (1/6,\ 2^{1/4} - 1]$

W(u) =	u $\in$
0	(0, 1/6]
1/3	$(1/6,\ 2^{3/8} - 1]$
1/2	$(2^{3/8} - 1, 2^{1/2} - 1]$
2/3	$(2^{1/2} - 1,\ 2^{3/4} - 1]$
1	$(2^{3/4} - 1,\ 1]$

W(P) $\ge$ 1.

Since $x \in (1/6,\ 2^{1/4} - 1]$, the utilization of the first task assigned on any processor in the completed RM-FFDU schedule must be equal to or greater than x, i.e., $y \ge x$. Let us consider a processor to which is first assigned a task with a utilization of y.

Case 1: $1/6 < y \le 2^{1/4} - 1$. Then the processor must be assigned at least four tasks. Therefore, W(P) ≥ 4 • 1/3 > 1.

Case 2: $2^{1/4} - 1 < y \le 2^{3/8} - 1$. Then the processor must be assigned at least three tasks. Therefore, W(P) ≥ 1/3 + 1/3 + 1/3 = 1.

Case 3: $2^{3/8} - 1 < y \le 2^{1/2} - 1$. Then the processor must be assigned at least two tasks. Furthermore, except for one processor by Lemma 3.9, each of the first two tasks must have a utilization greater than $2^{3/8} - 1$. Therefore, W(P) ≥ 1/2 + 1/2 = 1.

Case 4: $2^{1/2} - 1 < y \le 2^{3/4} - 1$. Then the processor must be assigned at least two tasks. Since the second task must be a task with a utilization greater than $2^{1/2} - 1$, we have W(P) ≥ 2/3 + 1/3 = 1.

Case 5: $2^{3/4} - 1 < y \le 1$. Then W(P) ≥ 1 by definition.

We then claim that for any processor in the optimal schedule, W(P) ≤ 5/3.

Let us assume that a processor in the optimal schedule is assigned *m* tasks with their utilizations as $u_1 \ge u_2 \ge \ldots\ldots \ge u_m$.

Case I: $1/6 < u_1 < 2^{3/8} - 1$. Then at most four tasks each with a utilization greater than 1/6 and less than $2^{3/8} - 1$ can be assigned on a processor. Therefore, W(P) ≤ 5 • 1/3 = 5/3.

Case II: $2^{3/8} - 1 < u_1 < 2^{1/2} - 1$ and $2^{3/8} - 1 < u_2$. Then at most two more tasks can be assigned to the processor. If $u_3 < 2^{3/8} - 1$, then W(P) ≤ 1/2 + 1/2 + 1/3 + 1/3 = 5/3. If $u_3 > 2^{3/8} - 1$, then no more task with a utilization greater than 1/6 can be assigned to the processor, and thus W(P) ≤ 1/2 + 1/2 + 1/2 = 3/2.

Case III: $2^{3/8} - 1 < u_1 < 2^{1/2} - 1$ and $u_2 < 2^{3/8} - 1$. Then at most two more tasks each with a utilization less than $2^{3/8} - 1$ can be assigned to the processor. Therefore, W(P) ≤ 1/2 + 1/3 + 1/3 + 1/3 = 3/2.

Case IV: $2^{1/2} - 1 < u_1 < 2^{3/4} - 1$ and $u_2 > 2^{1/2} - 1$. Then at most one task with a utilization greater than 1/6 and less than $2^{1/4} - 1$ can be assigned to the processor. Therefore, W(P) ≤ 2/3 + 2/3 + 1/4 < 5/3.

If $2^{1/2} - 1 < u_1 < 2^{3/4} - 1$ and $2^{3/8} - 1 < u_2 < 2^{1/2} - 1$, then at most one task with a utilization greater than 1/6 and less than $2^{3/4} - 1$ can be assigned to the processor. Therefore, W(P) ≤ 2/3 + 1/2 + 1/3= 3/2.

If $2^{1/2} - 1 < u_1 < 2^{3/4} - 1$ and $u_2 < 2^{3/8} - 1$, then at most two more task with a utilization greater than $2^{1/4} - 1$ can be assigned to the processor. Therefore, W(P) ≤ 2/3 + 1/3 + 1/3 + 1/3 = 5/3.

Case V: $2^{3/4} - 1 < u_1 < 1$. Then at most one more task with a utilization greater than 1/6 and less than $2^{3/8} - 1$ can be assigned to the same processor. Then W(P) ≤ 1 + 1/3 = 4/3.

Let N and N_0 be number of processors required by RM-FFDU and the minimum number of processors required to schedule a given set Σ of n tasks, respectively. Then the total weight of the task set is given by $\sum_{i=1}^{n} W(u_i)$. Since W(P) ≥ 1 for every processor in the RM-FFDU schedule, then $\sum_{i=1}^{n} W(u_i) \geq N - 1$. Since W(P) ≤ 5/3 for every processor in the optimal schedule, $5N_0/3 \geq \sum_{i=1}^{n} W(u_i)$. Therefore, $\Re_{RM-FFDU}^{\infty} \leq 5/3$. ■

Lemma 3.16: *If* $x \in (5(2^{1/5} - 1) - 3/5, 1/6]$, *then* $\Re_{RM-FFDU}^{\infty} \leq 5/3$.

Proof: Since $5(2^{1/5} - 1) - 3/5 \approx 0.14349 > 1/7$, a processor is assigned at most six tasks, i.e., κ = 6. For convenience, let us denote $\delta = 5(2^{1/5} - 1) - 3/5$.

For those processors to each of which one task is assigned, the utilization of each task is greater than (1 – 1/6) / (1+ 1/6) = 5/7 ≈ 0.71.

For the processor to which two tasks are assigned, the minimum of $u_1 + u_2$ is achieved at $U = 2[\sqrt{2/(1+x)} - 1]$ when $u_1 = u_2 = \sqrt{2/(1+x)} - 1$. Then $u_1 = u_2 = \sqrt{12/7} - 1 \approx 0.31$, and $U = 2(\sqrt{12/7} - 1) \approx 0.62$ for $x = 1/6$. Note that for $x < 1/6$, $U > 0.62$.

For a processor to which three tasks are assigned, their minimum utilization is achieved at $U = 3\{[2/(1+x)]^{1/3} - 1\}$ when $u_i = [2/(1+x)]^{1/3} - 1$. Then $u_1 = u_2 = u_3 = (12/7)^{1/3} - 1 \approx 0.197$, and $U = 3[(12/7)^{1/3} - 1] \approx 0.59$ for $x = 1/6$. Note that for $x < 1/6$, $U > 0.59$.

For a processor to which four tasks are assigned, their minimum utilization is achieved at $U = 4\{[2/(1+x)]^{1/4} - 1\}$ when $u_i = [2/(1+x)]^{1/4} - 1$. We want to find x such that $x \le [2/(1+x)]^{1/4} - 1$. Solving the inequality $x \le [2/(1+x)]^{1/4} - 1$ yields $x \le 2^{1/5} - 1 \approx 0.1487$. In other words, for every processor to which four tasks are assigned in the completed RM–FFDU schedule, their total utilization is greater than $4(2^{1/5} - 1) \approx 0.595$.

As having done so above, we define a function that maps the utilization of a task to a value that is in the range of 0 and 1, as given by Table 3.5.

We first claim that for every processor P in the completed RM-FFDU schedule, W(P) ≥ 1.

Since $x \in (5(2^{1/5} - 1) - 3/5, 1/6]$, the utilization of the first task assigned on any processor in the completed RM-FFDU schedule must be equal to or greater than x, i.e., $y \ge x$. Let us consider a processor to which is first assigned a task with a utilization of y.

Table 3.7: Weighting Function for x $\in (5(2^{1/5} - 1) - 3/5, 1/6]$

W(u) =	*u* ∈
0	$(0, \delta]$
1/4	$(\delta, (12/7)^{1/3} - 1]$
1/3	$((12/7)^{1/3} - 1, 2^{1/3} - 1]$
3/8	$(2^{1/3} - 1, \sqrt{12/7} - 1]$
1/2	$(\sqrt{12/7} - 1, 2^{1/2} - 1]$
2/3	$(2^{1/2} - 1, (12/7)^{2/3} - 1]$
3/4	$((12/7)^{2/3} - 1, 5/7]$
1	$(5/7, 1]$

Case 1: $y < (12/7)^{1/3} - 1$. Except for the last processor, the processor must be assigned at least four tasks each with a utilization less than $(12/7)^{1/3} - 1$ but greater than δ. Therefore, $W(P) \ge 4 \cdot 1/4 = 1$.

Case 2: $(12/7)^{1/3} - 1 < y \le 2^{1/3} - 1$. Except for one processor by Lemma 3.9, the processor must be assigned at least three tasks each with a utilization *u* such that $(12/7)^{1/3} - 1 < u \le 2^{1/3} - 1$. Therefore, W(P) ≥ 3 • 1/3 = 1.

Case 3: $2^{1/3} - 1 < y \le \sqrt{12/7} - 1$. Except for one processor by Lemma 3.9, the processor must be assigned at least three tasks. Furthermore, each of the first two tasks must have a utilization greater than $2^{1/3} - 1$. Therefore, W(P) ≥ 3/8 + 3/8 + 1/4 = 1.

Case 4: $\sqrt{12/7} - 1 < y \le 2^{1/2} - 1$. Except for one processor by Lemma 3.9, the processor must be assigned at least two tasks, each of which must have a utilization greater than $\sqrt{12/7} - 1$. Therefore we have W(P) ≥ 1/2 + 1/2 = 1.

Case 5: $2^{1/2} - 1 < y \le (12/7)^{2/3} - 1 \approx 0.432$. Since x ∈ (δ, 1/6], the processor must be assigned at least two tasks each with a utilization $\delta < u \le (12/7)^{2/3} - 1$. If the utilization of the second task is greater than $(12/7)^{1/3} - 1$, then W(P) ≥ 2/3 + 1/3 = 1. If the utilization u_2 of the second task is equal to or less than $(12/7)^{1/3} - 1$, then one more task with a utilization $u_3 \in (\delta, 1/6]$ must be assigned on the processor. This is because

$$2/\left(\{1 + [(12/7)^{1/3} - 1]\}\{1 + [(12/7)^{2/3} - 1]\}\right) - 1 = 7/6 - 1 = 1/6 \ge u_3.$$

Then W(P) ≥ 2/3 + 1/4 + 1/4 > 1.

Case 6: $(12/7)^{2/3} - 1 < y \le 5/7$. Except for one processor by Lemma 3.9, the processor must be assigned at least two tasks. The second task e must have a utilization greater than δ. Therefore we have W(P) ≥ 3/4 + 1/4 = 1.

Case 7: 5/7 < y ≤ 1. W(P) ≥ 1 by definition.

We then claim that for any processor in the optimal schedule, W(P) ≤ 5/3.

Let us assume that a processor *P* in the optimal schedule is assigned *m* tasks with their utilizations as $u_1 \ge u_2 \ge \ldots\ldots \ge u_m \ge x$

Case I: $\delta < u_1 \le (12/7)^{1/3} - 1$. Then at most six tasks each with a utilization greater than δ (and $\le u_1$) can be assigned on a processor. Therefore, W(P) ≤ 6/4.

Case II: $(12/7)^{1/3} - 1 < u_1 \le 2^{1/3} - 1$. There are several sub-cases to consider. If $u_2 \le (12/7)^{1/3} - 1$, then at most four more tasks each with a utilization greater than δ can be assigned to the processor, i.e., $m \le 6$. Then W(P) ≤ 1/3 + 5 • 1/4 < 5/3.

If $u_2 > (12/7)^{1/3} - 1$ and $u_3 \le (12/7)^{1/3} - 1$, then at most three more tasks each with a utilization greater than δ can be assigned to the processor, i.e., $m \le 6$. Then W(P) ≤ 1/3 + 1/3 + 4 • 1/4 = 5/3.

If $u_3 > (12/7)^{1/3} - 1$ and $u_4 \le (12/7)^{1/3} - 1$, then at most one more task with a utilization greater than δ can be assigned to the processor, i.e., $m \le 5$. This is because $3[(12/7)^{1/3} - 1] + 3\delta > 1$. Then W(P) ≤ 3 • 1/3 + 2 • 1/4 < 5/3.

If $u_4 > (12/7)^{1/3} - 1$ and $u_5 \le (12/7)^{1/3} - 1$, then no more task with a utilization greater than δ can be assigned to the processor, i.e., $m \le 5$. Then W(P) ≤ 4 • 1/3 + 1/4 < 5/3. If $u_5 > (12/7)^{1/3} - 1$, then W(P) ≤ 5 • 1/3 = 5/3.

Case III: $2^{1/3} - 1 < u_1 \le \sqrt{12/7} - 1$. There are several sub-cases to consider. If $u_2 \le (12/7)^{1/3} - 1$, then at most four more tasks each with a utilization greater than δ (and less than can $(12/7)^{1/3} - 1$) be assigned to the processor, i.e., $m \le 6$. Then W(P) ≤ 3/8 + 5 • 1/4 < 5/3.

If $(12/7)^{1/3} - 1 < u_2 \le 2^{1/3} - 1$ and $u_3 \le (12/7)^{1/3} - 1$, then at most two more tasks each with a utilization greater than δ can be assigned to the processor, i.e., $m \le 5$. Then W(P) ≤ 3/8 + 1/3 + 3 • 1/4 < 5/3. If $u_2 \le 2^{1/3} - 1$, $(12/7)^{1/3} - 1 < u_3 \le 2^{1/3} - 1$, and $u_4 \le (12/7)^{1/3} - 1$, then at most one more task with a utilization greater than δ can be assigned to the processor, i.e., $m \le 5$. Then W(P) ≤ 3/8 + 2 • 1/3 + 2 • 1/4 < 5/3. If $u_2 \le 2^{1/3} - 1$, $(12/7)^{1/3} - 1 < u_4 \le 2^{1/3} - 1$, and $u_5 \le (12/7)^{1/3} - 1$, then no more task with a utilization greater than δ can be assigned to the processor, i.e., $m \le 5$. Then W(P) ≤ 3/8 + 3 • 1/3 + 1/4 < 5/3.

If $2^{1/3} - 1 < u_2$ and $(12/7)^{1/3} - 1 < u_3 \le 2^{1/3} - 1$, then at most one more task with a utilization greater than δ can be assigned to the processor, i.e., $m \le 4$. Therefore W(P) ≤ 2 • 3/8 + 1/3 + 1/4 < 5/3. If $2^{1/3} - 1 < u_3$, then at most more task with a utilization greater than δ and less than $2^{1/3} - 1$ can be assigned to the processor, i.e., $m \le 4$. Therefore W(P) ≤ 3 • 3/8 + 1/3 < 5/3.

Case IV: $\sqrt{12/7} - 1 < u_1 \le 2^{1/2} - 1$ There are several sub-cases to consider. If $u_2 \le (12/7)^{1/3} - 1$, then at most three more tasks each with a utilization greater than δ (and less than can $(12/7)^{1/3} - 1$) be assigned to the processor, i.e., $m \le 5$, since $\sqrt{12/7} - 1 + 5\delta > 1$. Then W(P) ≤ 1/2 + 4 • 1/4 < 5/3.

If $(12/7)^{1/3} - 1 < u_2 \le 2^{1/3} - 1$ and $u_3 \le (12/7)^{1/3} - 1$, then at most two more tasks each with a utilization greater than δ can be assigned to the processor, i.e., $m \le 5$, since $\sqrt{12/7} - 1 + (12/7)^{1/3} - 1 + 4\delta > 1$. Then W(P) ≤ 1/2 + 1/3 + 3 • 1/4 < 5/3. If $u_2 \le 2^{1/3} - 1$ and $(12/7)^{1/3} - 1 < u_3 \le 2^{1/3} - 1$, then at most two more tasks each with a utilization greater than δ and less than $(12/7)^{1/3} - 1$ can be assigned to the processor, i.e., $m \le 5$. Then W(P) ≤ 1/2 + 2 • 1/3 + 2 • 1/4 = 5/3. If $u_2 \le 2^{1/3} - 1$ and $(12/7)^{1/3} - 1 < u_4 \le 2^{1/3} - 1$, then no more task with a utilization greater than δ can be assigned to the processor, i.e., $m \le 4$. Then W(P) ≤ 1/2 + 3 • 1/3 < 5/3.

If $2^{1/3} - 1 < u_2 \le \sqrt{12/7} - 1$ and $u_3 \le (12/7)^{1/3} - 1$, then at most two more tasks each with a utilization greater than δ can be assigned to the processor, i.e., $m \le 5$. Therefore W(P) ≤ 1/2 + 3/8 + 3 • 1/4 < 5/3. If $2^{1/3} - 1 < u_2 \le \sqrt{12/7} - 1$ and $(12/7)^{1/3} - 1 < u_3 \le 2^{1/3} - 1$, then at most one more task with a utilization greater than δ and less than $(12/7)^{1/3} - 1$ can be assigned to the processor, i.e., $m \le 4$. Therefore W(P) ≤ 1/2 + 3/8 + 1/3 + 1/4 < 5/3. If $2^{1/3} - 1 < u_2 \le \sqrt{12/7} - 1$ and $(12/7)^{1/3} - 1 < u_4 \le u_3 \le 2^{1/3} - 1$, then no more task with a utilization greater than δ can be assigned to the processor, i.e., $m \le 4$. Therefore W(P) ≤ 1/2 + 3/8 + 1/3 + 1/3 < 5/3. If $u_2 \le \sqrt{12/7} - 1$ and $2^{1/3} - 1 < u_3 \le \sqrt{12/7} - 1$, then at most one more task with a utilization greater than δ and less than $(12/7)^{1/3} - 1$ can be assigned to the processor, i.e., $m \le 4$. Therefore W(P) ≤ 1/2 + 2 • 3/8 + 1/4 < 5/3.

If $\sqrt{12/7} - 1 < u_2 \le 2^{1/2} - 1$ and $u_3 \le (12/7)^{1/3} - 1$, then at most one more task with a utilization greater than δ can be assigned to the processor, i.e., $m \le 4$. Therefore W(P) ≤ 2 • 1/2 + 2 • 1/4 < 5/3. If $\sqrt{12/7} - 1 < u_2 \le 2^{1/2} - 1$ and $(12/7)^{1/3} - 1 < u_3 \le 2^{1/3} - 1$, then at most one more task with a utilization greater than δ and less than

$(12/7)^{1/3} - 1$ can be assigned to the processor, i.e., $m \le 4$, since $2(\sqrt{12/7} - 1) + 2[(12/7)^{1/3} - 1] > 1$. Therefore W(P) $\le 2 \bullet 1/2 + 1/3 + 1/4 < 5/3$. If $\sqrt{12/7} - 1 < u_2 \le 2^{1/2} - 1$ and $2^{1/3} - 1 < u_3 \le \sqrt{12/7} - 1$, then no more task with a utilization greater than δ can be assigned to the processor, i.e., $m \le 3$, since $2(\sqrt{12/7} - 1) + 2^{1/3} - 1 + \delta > 1$. Therefore W(P) $\le 2 \bullet 1/2 + 3/8 < 5/3$. If $\sqrt{12/7} - 1 < u_3 \le u_2 \le 2^{1/2} - 1$, then no more task with a utilization greater than δ can be assigned to the processor, i.e., $m \le 3$, since $3(\sqrt{12/7} - 1) + \delta > 1$. Therefore W(P) $\le 3 \bullet 1/2 < 5/3$.

Case V: $2^{1/2} - 1 < u_1 \le (12/7)^{2/3} - 1$. There are several sub-cases to consider.

If $u_2 \le (12/7)^{1/3} - 1$, then at most three more tasks each with a utilization greater than δ can be assigned to the processor, i.e., $m \le 5$. Therefore W(P) $\le 2/3 + 4 \bullet 1/4 = 5/3$.

If $(12/7)^{1/3} - 1 < u_2 \le 2^{1/3} - 1$ and $u_3 \le (12/7)^{1/3} - 1$, then at most one more task with a utilization greater than δ can be assigned to the processor, i.e., $m \le 4$, since $2^{1/2} - 1 + (12/7)^{1/3} - 1 + 3\delta > 1$. Then W(P) $\le 2/3 + 1/3 + 2 \bullet 1/4 < 5/3$. If $u_2 \le 2^{1/3} - 1$ and $(12/7)^{1/3} - 1 < u_3 \le 2^{1/3} - 1$, then at most one more task with a utilization greater than δ and less than $(12/7)^{1/3} - 1$ can be assigned to the processor, i.e., $m \le 4$, since $2^{1/2} - 1 + 3[(12/7)^{1/3} - 1] > 1$. Then W(P) $\le 2/3 + 2 \bullet 1/3 + 1/4 < 5/3$.

If $2^{1/3} - 1 < u_2 \le \sqrt{12/7} - 1$ and $u_3 \le (12/7)^{1/3} - 1$, then at most one more task with a utilization greater than δ can be assigned to the processor, i.e., $m \le 4$. Therefore W(P) $\le 2/3 + 3/8 + 2 \bullet 1/4 < 5/3$. If $2^{1/3} - 1 < u_2 \le \sqrt{12/7} - 1$ and $(12/7)^{1/3} - 1 < u_3 \le 2^{1/3} - 1$, then no more task with a utilization greater than δ can be assigned to the processor, i.e., $m \le 3$. Therefore W(P) $\le 2/3 + 3/8 + 1/3 < 5/3$. If $2^{1/3} - 1 < u_3 \le u_2 \le \sqrt{12/7} - 1$, then no more task with a utilization greater than δ and can be assigned to the processor, i.e., $m \le 3$. Therefore W(P) $\le 2/3 + 2 \bullet 3/8 < 5/3$.

If $\sqrt{12/7} - 1 < u_2 \le 2^{1/2} - 1$ and $u_3 \le 2^{1/3} - 1$, then no more task with a utilization greater than δ can be assigned to the processor, i.e., $m \le 3$, since $2^{1/2} - 1 + \sqrt{12/7} - 1 + 2\delta > 1$. Therefore W(P) $\le 2/3 + 1/2 + 1/3 < 5/3$. If $\sqrt{12/7} - 1 < u_2 \le 2^{1/2} - 1$ and $2^{1/3} - 1 < u_3 \le \sqrt{12/7} - 1$, then no more task with a utilization greater than

δ can be assigned to the processor, i.e., $m \leq 3$, since $2^{1/2} - 1 + \sqrt{12/7} - 1 + 2^{1/3} - 1 + \delta > 1$. Therefore W(P) ≤ 2/3 + 1/2 + 3/8 < 5/3.

If $2^{1/2} - 1 < u_2 \leq (12/7)^{2/3} - 1$, then at most one more task with a utilization greater than δ and less than $(12/7)^{1/3} - 1$ can be assigned to the processor, i.e., $m \leq 3$. Therefore W(P) ≤ 2/3 + 2/3 + 1/4 < 5/3.

Case VI: $(12/7)^{2/3} - 1 < u_1 \leq 5/7$. There are several sub-cases to consider.

If $\delta < u_2 \leq (12/7)^{1/3} - 1$, then at most two more tasks each with a utilization greater than δ can be assigned to the processor, i.e., $m \leq 4$, since $(12/7)^{2/3} - 1 + 4\delta > 1$. Therefore W(P) ≤ 3/4 + 3 • 1/4 < 5/3.

If $(12/7)^{1/3} - 1 < u_2 \leq 2^{1/3} - 1$ and $u_3 \leq (12/7)^{1/3} - 1$, then at most one more task with a utilization greater than δ can be assigned to the processor, i.e., $m \leq 4$, since $(12/7)^{2/3} - 1 + (12/7)^{1/3} - 1 + 3\delta > 1$. Then W(P) ≤ 3/4 + 1/3 + 2 • 1/4 < 5/3. If $u_2 \leq 2^{1/3} - 1$ and $(12/7)^{1/3} - 1 < u_3 \leq 2^{1/3} - 1$, then at most one more task with a utilization greater than δ and less than $(12/7)^{1/3} - 1$ can be assigned to the processor, i.e., $m \leq 4$, since $(12/7)^{2/3} - 1 + 3[(12/7)^{1/3} - 1] > 1$. Then W(P) ≤ 3/4 + 2 • 1/3 + 1/4 = 5/3.

If $2^{1/3} - 1 < u_2 \leq \sqrt{12/7} - 1$ and $u_3 \leq (12/7)^{1/3} - 1$, then at most one more task with a utilization greater than δ can be assigned to the processor, i.e., $m \leq 4$. Therefore W(P) ≤ 3/4 + 3/8 + 2 • 1/4 < 5/3. If $2^{1/3} - 1 < u_2 \leq \sqrt{12/7} - 1$ and $(12/7)^{1/3} - 1 < u_3 \leq 2^{1/3} - 1$, then no more task with a utilization greater than δ can be assigned to the processor, i.e., $m \leq 3$. Therefore W(P) ≤ 3/4 + 3/8 + 1/3 < 5/3. If $2^{1/3} - 1 < u_3 \leq u_2 \leq \sqrt{12/7} - 1$, then no more task with a utilization greater than δ and can be assigned to the processor, i.e., $m \leq 3$. Therefore W(P) ≤ 3/4 + 2 • 3/8 < 5/3.

If $\sqrt{12/7} - 1 < u_2 \leq 2^{1/2} - 1$ and $u_3 \leq 2^{1/3} - 1$, then no more task with a utilization greater than δ can be assigned to the processor, i.e., $m \leq 3$, since $(12/7)^{2/3} - 1 + \sqrt{12/7} - 1 + 2\delta > 1$, and $(12/7)^{2/3} - 1 + \sqrt{12/7} - 1 + 2^{1/3} - 1 > 1$. Therefore W(P) ≤ 3/4 + 1/2 + 1/3 < 5/3.

If $2^{1/2} - 1 < u_2 \leq (12/7)^{2/3} - 1$, then at most one more task with a utilization greater than δ and less than $(12/7)^{1/3} - 1$ can be assigned to the processor, i.e., $m \leq 3$. Therefore W(P) ≤ 3/4 + 2/3 + 1/4 < 5/3.

If $(12/7)^{2/3} - 1 < u_2 \leq 5/7$, then no more task with a utilization greater than δ can be assigned to the processor, i.e., $m \leq 2$, since $2[(12/7)^{2/3} - 1] + \delta > 1$. Therefore W(P) ≤ 3/4 + 3/4 < 5/3.

Case VI: $5/7 < u_1 \leq 1$. Since $5/7 < u_1$, the total utilization of the rest of the tasks is less than 1 – 5/7 = 2/7 < $\sqrt{12/7} - 1$. Furthermore, since $5/7 + 2\delta > 1$, at most one more task with a utilization less than $\sqrt{12/7} - 1$ can be assigned to the processor. Therefore W(P) ≤ 1+ 3/8 < 5/3.

Let N and N_0 be number of processors required by RM-FFDU and the minimum number of processors required to schedule a given set Σ of n tasks, respectively. Then the total weight of the task set is given by $\sum_{i=1}^{n} W(u_i)$. Since W(P) ≥ 1 for every processor in the RM-FFDU schedule, then $\sum_{i=1}^{n} W(u_i) \geq N - 4$. Since W(P) ≤ 5/3 for every processor in the optimal schedule, $5N_0/3 \geq \sum_{i=1}^{n} W(u_i)$. Therefore, $\Re^{\infty}_{RM-FFDU} \leq 5/3$. ■

Proof of Theorem 3.17: We first claim that $\Re^{\infty}_{RM-FFDU} \leq 5/3$ under $x \in (0, 0.13477]$. For $x \leq 0.13477 = 6(2^{1/6} - 1) - 0.6$, if a processor is assigned n tasks, then $(n + 1)(2^{1/(n+1)} - 1) - x > 0.6$ for $n \geq 5$. If a processor is assigned six or more tasks, then $U \geq 6u > 0.6$.

For $0.13477 \leq x < 0.143492 = 5(2^{1/5} - 1) - 0.6$, if a processor is assigned n tasks, then $(n + 1)(2^{1/(n+1)} - 1) - x > 0.6$ for $n \geq 4$. If a processor is assigned five tasks or more, then U ≥ 5x > 0.6. Now we need only to consider $\Re^{\infty}_{FFDU}$ under x ∈ [0.14349, 1/2]

From Lemma 3.11 to Lemma 3.16, we conclude that $\Re^{\infty}_{RM-FFDU} \leq 5/3$. ■

Theorem 3.18: $\Re^{\infty}_{RM-FFDU} = \frac{5}{3}$.

Proof: In order to prove that the bound is tight. We need to show that the upper bounded number of processors is indeed required for some large task sets if they are scheduled by the RM-FFDU algorithm.

Let $n = 15k$, where $k > 0$ is a natural number.

Then we can construct the following task set:

$u_i = 0.2$, for $i = 1, 2, \ldots, n$.

In the completed RM-FFDU schedule, each processor is assigned three tasks since $0.2 > 2/\prod_{j=1}^{3}(1+0.2) - 1$. Therefore, a total of $n/3$ processors is used to schedule the task set, i.e., $N = n/3$.

In the optimal schedule, each processor is assigned five tasks since $5 \cdot 0.2 = 1$. Hence, a total of $n/5$ processors is used to schedule the task set, i.e., $N_0 = n/5$.

$N / N_0 = 5/3$.

Together with Theorem 3.17, we conclude that $\Re^{\infty}_{RM-FFDU} = \frac{5}{3}$. ■

In this section, we propose a new heuristic algorithm for scheduling a set of fixed-priority periodic tasks on a multiprocessor system. The worst case performance bound of the algorithm is shown to be significantly lower than those in the literature. In fact, this bound of 1.6667 is the lowest bound ever obtained for the RMMS problem.

3.8. Heuristic Algorithms Using the Necessary and Sufficient Condition

In the previous sections, we have developed and analyzed several scheduling algorithms based on various schedulability conditions. While the algorithms are efficient with either linear or $O(n \log n)$ time complexity, their performance may suffer because those conditions are not necessary. Even though using the IFF condition may require more than exponential time in some cases, we would like to know the performance of algorithms that use the IFF condition. In the following we will study two algorithms, one on-line and the other off-line, that use the IFF condition.

The first algorithm is an on-line one and it is called the Rate-Monotonic-First-Fit-IFF (RM-FF-IFF). It is almost the same as the RM-FF algorithm, except that instead of the UO condition, the IFF condition is used to schedule tasks. It is given in Figure 3.10.

All the theorems that are valid for RM-FF are valid for RM-FF-IFF because the IFF condition is not only sufficient but also necessary. For this algorithm, we prove that its per-

Rate-Monotonic-First-Fit-IFF (**RM-FF-IFF**) (Input: task set Σ; Output: *m*)

```
Let the processors be indexed as $P_1$, $P_2$, ......, with each initially
idle. The tasks $\tau_1, \tau_2, \ldots\ldots, \tau_n$ will be assigned in that order. To
assign $\tau_i$, find the least j such that task $\tau_i$ can be feasibly sched-
uled on $P_j$ according to the IFF condition and assign $\tau_i$ to $P_j$. Now
processor $P_j$ has one more task assigned on it.
```

Figure 3.10: Algorithm RM-FF-IFF

formance is upper bounded by 1.96, as stated in the following theorem.

Theorem 3.19: *Let N and N_0 be the number of processors required by RM-FF-IFF and the minimum number of processors required to feasibly schedule a given set of tasks, respectively. Then $N \le 1.96N_0 + 1$.*

Proof: Let $\Sigma = \{\tau_1, \tau_2, \ldots, \tau_m\}$ be a set of m tasks, with their utilizations $u_1, u_2, \ldots, u_m$. Then the total utilization of the tasks is given by $\sum_{i=1}^{m} u_i$. Suppose that among the N processors in the RM-FF-IFF schedule, n_i is the number of processors to each of which exactly $i \ge 1$ tasks are assigned. Then $N = \sum_{i=1}^{\infty} n_i$. For convenience, we let $a = 2\left(2^{1/2} - 1\right)$.

Among the N processors in the RM-FF-IFF schedule, for the n_1 processors to each of which one task is assigned, we have by Theorem 3.3 that

$$\sum_{i=1}^{n_1} u_i > n_1 / \left(1 + 2^{1/n_1}\right) > n_1 / 2 - \ln 2 / 4. \qquad \text{(Eq.3.42)}$$

According to Lemma 3.2, among all processors on each of which at least two tasks are assigned, there are at most one processor whose utilization is not greater than $2(2^{1/3} - 1) = a$. Hence,

$$\sum_{i=1}^{N-n_1} U_i \ge 2a\,(N - n_1 - 1) \qquad \text{(Eq.3.43)}$$

Since the total utilization of the task set must be equal to the total utilization of the processors to which the tasks are assigned, we have

$$\sum_{i=1}^{m} u_i = \sum_{i=1}^{n_1} u_i + \sum_{i=1}^{N-n_1} U_i .$$

According to inequalities (3.42) and (3.43), it is immediate that

$$\sum_{i=1}^{m} u_i \geq n_1 / 2 - \ln 2 / 4 + 2a(N - n_1 - 1) .$$

Since $N_0 \geq \sum_{i=1}^{m} u_i$, we have

$$N_0 \geq 2aN - \ln 2 / 4 - 1 - (2a - 0.5)n_1.$$

Because any two of the tasks that are assigned to the n_1 processors cannot be scheduled on a single processor in the optimal schedule, we have $N_0 \geq n_1$.

Then $N_0 \geq 2aN - \ln 2 / 4 - 1 - (2a - 0.5)n_1 \geq 2aN - \ln 2 / 4 - (2a - 0.5)N_0$

$$\frac{N}{N_0} \leq \frac{2a + 0.5}{2a} - \frac{((\ln 2)/4 + 1)}{2a}\frac{1}{N_0} \qquad \text{(Eq.3.44)}$$

Hence, $\Re^{\infty}_{RM-FF-IFF} \leq \frac{2a + 0.5}{2a} = 1.96$.

For $\alpha = \max_{1 \leq i \leq n}(C_i/T_i) \leq a$, by arguments similar to the above one and the one in the proof of Theorem 3.9, we obtain the rest of the results that are the same as listed in Table 3.2. ■

The following theorem shows that the worst case bound for RM-FF-IFF cannot be better than 1.72.

Theorem 3.20: $\Re^{\infty}_{RM-FF-IFF} \geq 1 + 1/(2\ln 2) = 1.72$.

Proof: For any given N_0, we construct a task set such that in the optimal schedule, N_0 processors are required while in the RM-FF-IFF schedule, $N = N_0 + \lfloor N_0/(2\ln 2) \rfloor$ processors are required to schedule.

Recall that the worst case utilization bound of $n\left(2^{1/n} - 1\right)$ can be achieved by the following set of n tasks:

$C_i = 2^{i/n}(2^{1/n} - 1)$, $T_i = 2^{i/n}$, for $i = 0, 1, \ldots, n - 1$. These n tasks totally utilize the processor in the time period of [0, 2]. The total utilization of the n tasks is therefore given by $U = n(2^{1/n} - 1)$.

Next, we want to decide the number of tasks needed to have a total utilization of close to but no greater than 1/2 for each of the n tasks above. This number is given by

$$m = \left\lfloor 0.5 / \left(2^{1/n} - 1 \right) \right\rfloor \qquad \text{(Eq.3.45)}$$

The task set which achieves the desired bound is given as follows:

It consists of $m \bullet n$ tasks each with a utilization of $2^{1/n} - 1$ and n tasks each with a utilization of $1/2 + \delta$ for any arbitrary small number $\delta > 0$. The tasks with larger utilizations follow those with smaller utilizations.

For the tasks with smaller utilizations, they are divided into m groups each with n tasks. A group of n tasks is given by

$$C_i = 2^{i/n}(2^{1/n} - 1),\ T_i = 2^{i/n},\ \text{for } i = 0, 1, \ldots, n-1.$$

Following these tasks are n tasks as given by

$$C_i = 2^{i/n} / (2^{1/n} - 1) + 2^{i/n}\delta,\ T_i = 2^{i/n},\ \text{for } i = 0, \ldots, n-1.$$

In the RM-FF-IFF schedule, the first $m \bullet n$ tasks will fill i processors, while the last n tasks will fill n processors. Hence $N = n + m$.

In the optimal schedule, n processors are needed. The task assignment is arranged in such a way that all the tasks with the same period are assigned to one processors, since $m(2^{1/n} - 1) + 1/2 + \delta \le 1$ for sufficiently small δ. Hence $N_0 = n$.

$N / N_0 = (n + m) / n = 1 + \left\lfloor 0.5 / \left(2^{1/n} - 1 \right) \right\rfloor / n \to 1 + 1/(2\ln 2)$ when $n \to \infty$.

Therefore we have proven that $\Re^{\infty}_{RM-FF-IFF} \ge 1 + 1/(2\ln 2)$. ■

The second algorithm is an off-line one and it is called the Rate-Monotonic-First-Fit-Decreasing-Utilization-IFF (RM-FFDU-IFF). Before it schedules the tasks, RM-FFDU-IFF first sorts the tasks in the order of non-increasing task utilization. It is almost the same as the RM-FFDU algorithm, except that instead of the UO condition, the IFF condition is used to schedule tasks. It is given in Figure 3.11.

Rate-Monotonic-First-Fit-IFF (**RM-FF-IFF**) (Input: task set Σ; Output: m)

```
Sort the tasks in the order of non-increasing task utilization.
Call RM-FF-IFF
```

Figure 3.11: Algorithm RM-FFDU-IFF

Theorem 3.21: $\Re^{\infty}_{RM-FFDU-IFF} \le 5/3$.

This is true by Theorem 3.17.

The following theorem shows that the worst case bound for RM-FF-IFF cannot be better than 1.44.

Theorem 3.22: $\Re^{\infty}_{RM-FFDU-IFF} \ge 1/(\ln 2) = 1.44$.

Proof: For any given N_0, we construct a task set such that in the optimal schedule, N_0 processors are required while in the RM-FFDU-IFF schedule, $N = N_0/(\ln 2)$ processors are required to schedule it.

Let $n > 100$ is an integer, $k > 0$ is an integer, and m is determined by

$$m = \left\lfloor 1/\left(2^{1/n} - 1\right)\right\rfloor \qquad \text{(Eq.3.46)}$$

A set of $m \bullet n \bullet k$ tasks which achieves the desired bound is given as follows:

It consists of $m \bullet k$ groups of tasks. Each group has n tasks in it. All the $m \bullet k$ groups are identical. A group of n tasks is given by

$C_i = 2^{i/n}(2^{1/n} - 1)$, $T_i = 2^{i/n}$, for $i = 0, 1, \ldots, n - 1$.

Since all tasks have the same utilization, we can assume that the order of the tasks after sorting is the same as it is given.

In the RM-FFDU-IFF schedule, each group of tasks is assigned to one processors. A total of $m \bullet k$ processors is used in the RM-FFDU-IFF schedule, i.e., $N = m \bullet k$.

In the optimal schedule, a total of $n \bullet k$ processors is used, i.e., $N_0 = n \bullet k$.

Therefore, $N / N_0 = m / n = \left\lfloor 1/\left(2^{1/n} - 1\right)\right\rfloor / n \to 1/\ln 2$.

$\Re^{\infty}_{RM-FFDU-IFF} \ge 1/(\ln 2)$. ■

We have reason to believe that the worst case performance bounds for RM-FF-IFF and RM-FFDU-IFF can be further improved. We conjecture that the upper bound for RM-FF-IFF can be lowered to 1.86 and the upper bound for RM-FFDU-IFF can be further reduced to 1.5.

3.9. Average Case Performance Evaluation

In the previous sections, the performance bounds of the new algorithms were derived under worst case assumptions. While a worst case analysis assures that the performance bounds are satisfied for any task set, it does not provide insight into the average case behavior of the algorithms. Do the algorithms perform on the average close to its worst case performance? Do the worst case performance rankings of the algorithms stand as they are in the average case? To answer these questions, one can analyze the algorithms with probabilistic assumptions, or conduct simulation experiments. Since a probabilistic analysis of the algorithms is beyond the scope of this thesis, we resort to simulation.

The simulation is conducted by running the algorithms on a large number of computer generated sample task sets and averaging the results over a number of runs. The task sets are generated by using one of the random number generators, which can generate numbers with very good approximation to the uniform distribution. The number of runs for each data point is chosen mostly to be 20 or more, since for our experiments, 20 runs is large enough to counter the effect of “randomness”.

The simulation consists of three stages. In the first stage, we will compare the performance of all on-line algorithms. In the second stage, we will compare the performance of all off-line algorithms. Finally, a different approach is used to evaluate the performance of various algorithms.

3.9.1. Performance Comparison of New and Existing On-line Algorithms

We present simulation experiments for large task sets with $100 \le n \le 1000$ tasks. In each experiment, we vary the value of parameter α — the maximal load factor of any task in the set, i.e., $\alpha = max_i (C_i / T_i)$. The task periods are assumed to be uniformly distributed with values $1 \le T_i \le 500$. The run-times of the tasks are also taken from a uniform distribution with range $1 \le C_i \le \alpha T_i$. The performance metric in all experiments is the number of processors required to execute a given task set. We first compare the performance of the following on-line algorithms:

- RMNF [20]
- NF-M [16]
- RMGT-M (Section 3.6)
- RM-FF (Section 3.3)
- RM-FFF-IFF (Section 3.8)

Since an optimal schedule cannot be calculated for large task sets, we use the total utilization (or load) $U = \sum_{i=1}^{n} C_i / T_i$ of a task set as the lower bound for the number of processors required. Except for certain figures, each data point in all figures in this section represents the average value of 20 runs of an algorithm on independently generated task sets with identical parameters. All algorithms are executed on identical task sets. The results are plotted in Figure 3.12 for α = 0.2, Figure 3.13 for α = 0.5, Figure 3.14 for α = 0.7, and Figure 3.15 for α = 1.0.

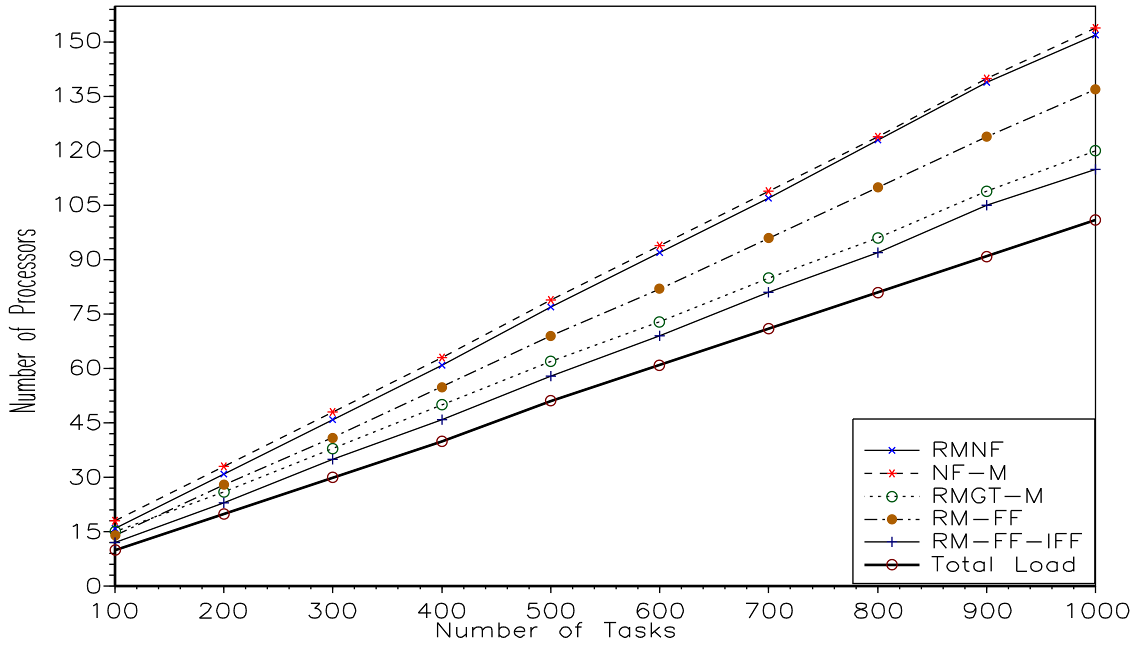


Figure 3.12: Performance of Some On-line Algorithms (α = 0.2)

From the experiments, we conclude that

- All the new on-line algorithms outperform the existing ones.

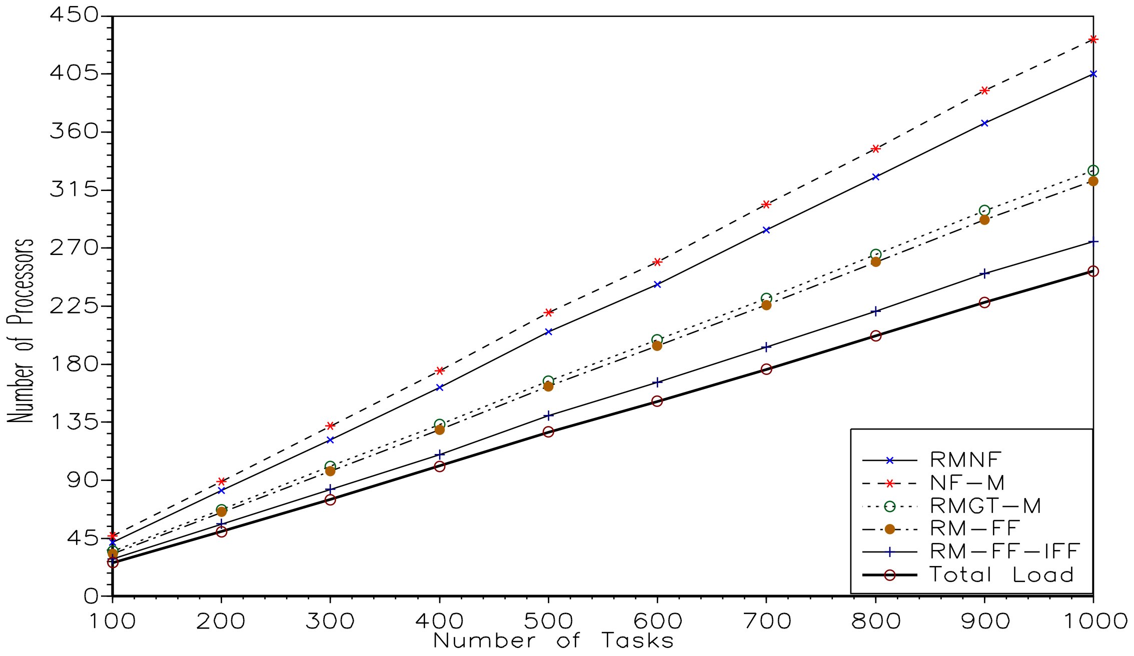


Figure 3.13: Performance of Some On-line Algorithms ($\alpha = 0.5$)

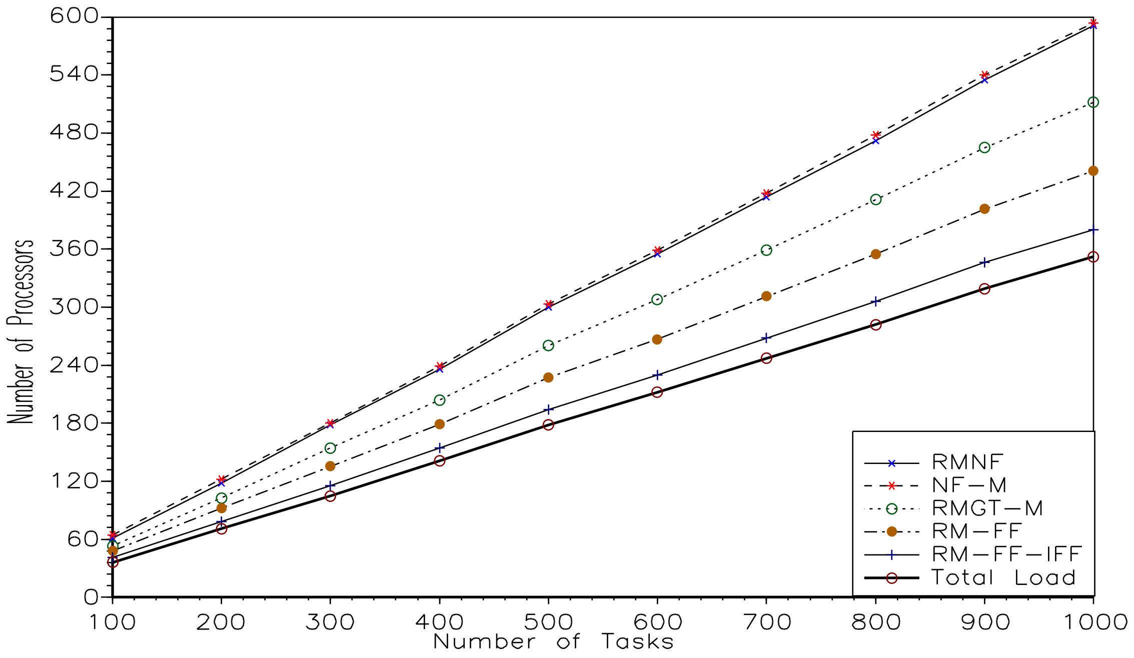


Figure 3.14: Performance of Some On-line Algorithms ($\alpha = 0.7$)

- RM-FF-IFF outperforms all other algorithms, though it takes a considerably large amount of time to compute the results. The larger the ratio between any two task periods, the longer time RM-FF-IFF takes to compute the results.

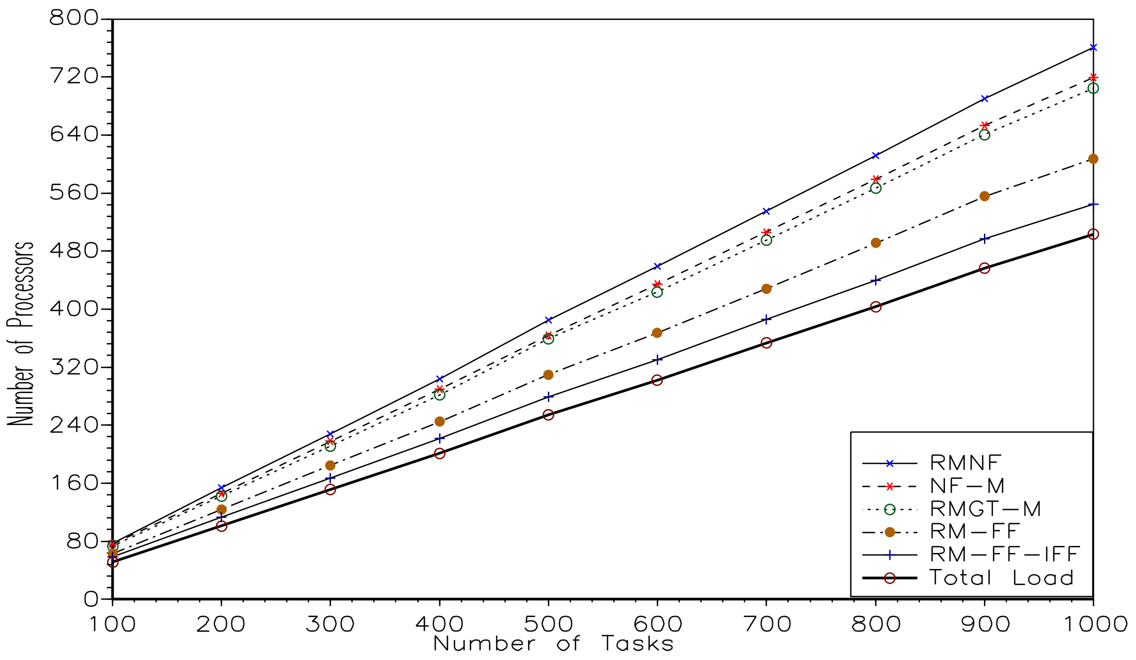


Figure 3.15: Performance of Some On-line Algorithms (α = 1.0)

- The performance of RMGT-M gets better as α becomes smaller, while the opposite holds for RM-FF even though the improvement in RM-FF seems small.
- The number of processors required to schedule a given task set grows proportionally with the number of tasks in the set. The number of processors required to schedule a set of tasks also grows proportionally with the value of α.

Since we only show the performance of three new algorithms in the previous experiment, there are some others that need to be considered. We chose to compare the performance of RM-FF against that of others for good reason. As it will be shown, the performance of RM-FF is quite representative of the several algorithms considered below:

- RM-FF (Section 3.3)
- RM-BF (Section 3.4)
- RRM-FF (Section 3.5)
- RRM-BF (Section 3.5)

All these algorithms are executed on the same task sets as previously used. The results are plotted in Figure 3.16 for α = 0.3, Figure 3.17 for α = 0.7, and Figure 3.18 for α

= 1.0.

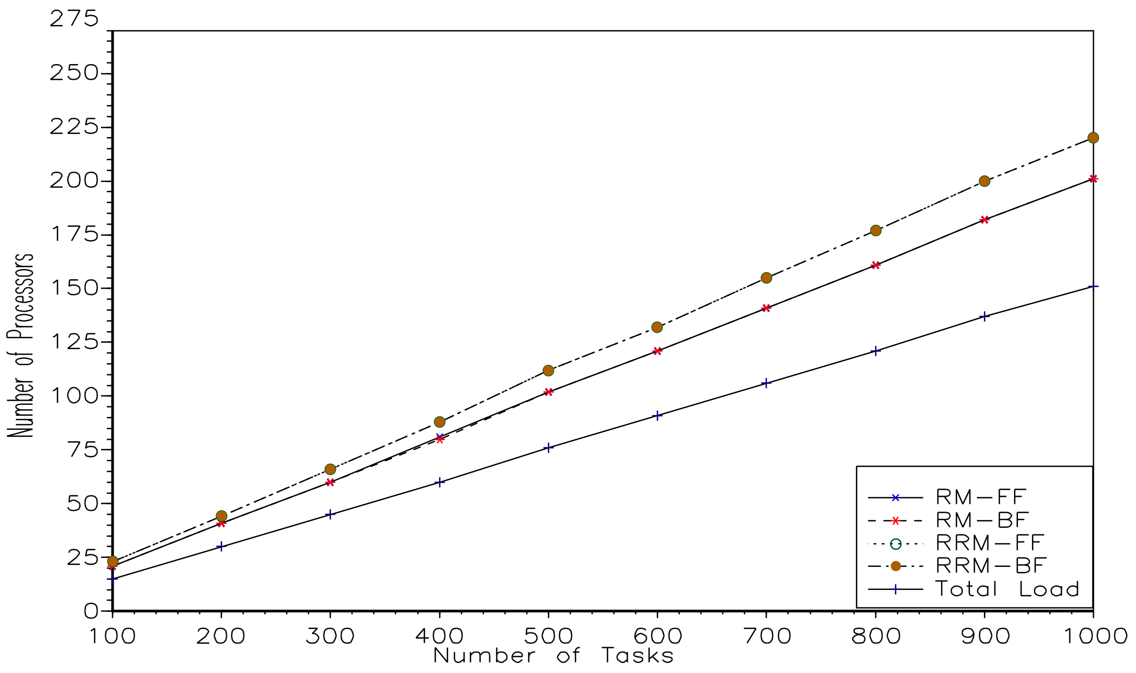


Figure 3.16: Performance of RM-FF, RM-BF, RRM-FF, and RRM-BF ($\alpha = 0.3$)

The following conclusion is accordingly made

- RM-FF and RM-BF perform almost identically, and so do RRM-FF and RRM-BF. RM-BF performs a little bit better than RM-FF in some cases.
- RRM-FF and RRM-BF outperform RM-FF and RM-BF when α is large.
- When α is small (but not smaller), RM-FF and RM-BF performs better than RRM-FF and RRM-BF. The performance of RRM-FF(-BF) and that of RM-FF(-BF) becomes identical when α reaches the threshold value, which is $(2^{1/3} - 1)$ in our experiments.

3.9.2. Performance Comparison of New and Existing Off-line Algorithms

Just as we have done for the on-line algorithms in the previous sub-section, we simulate the performance of the new and existing off-line algorithms following the same strategy. We compare the performance of the following algorithms:

- RMFF [20]

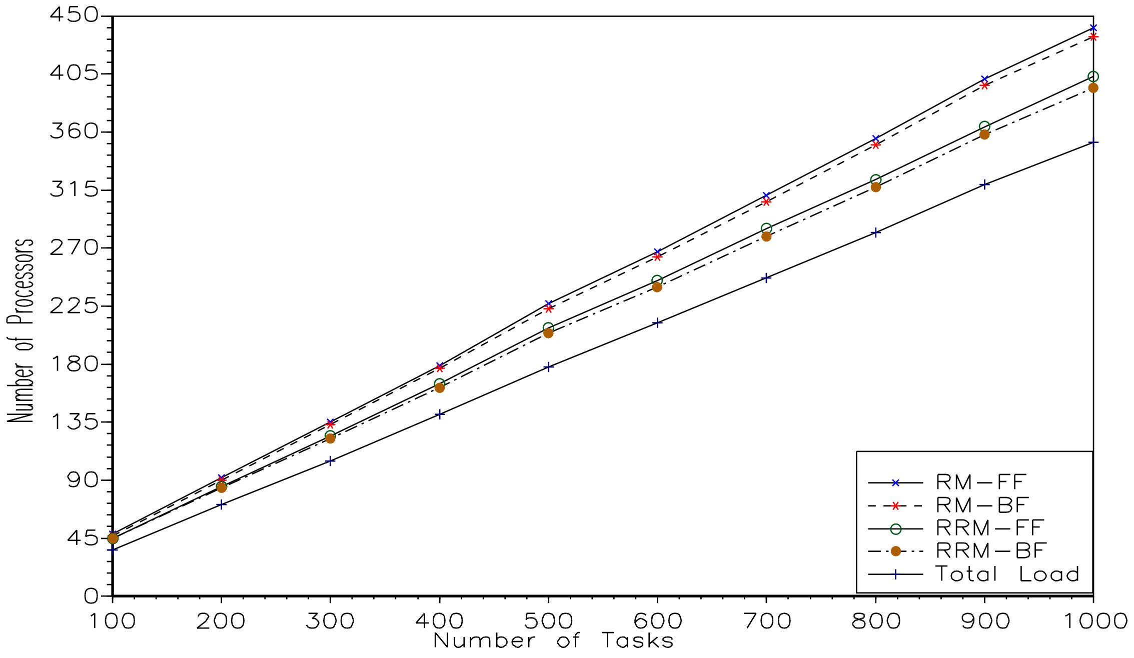


Figure 3.17: Performance of RM-FF, RM-BF, RRM-FF, and RRM-BF (α = 0.7)

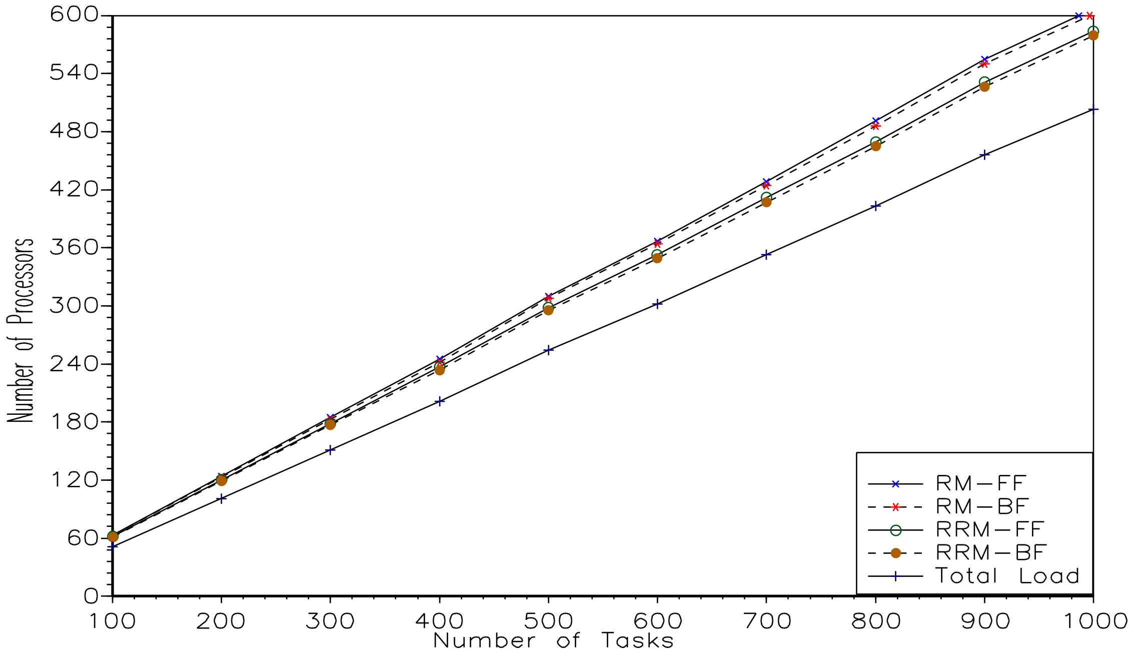


Figure 3.18: Performance of RM-FF, RM-BF, RRM-FF, and RRM-BF (α = 1.0)

- FFDUF [17]
- RMST (Section 3.6)

- RMGT (Section 3.6)
- RM-FFDU (Section 3.7)
- RM-FFDU-IFF (Section 3.8)

The same task sets as used in the previous experiments are run through these off-line algorithms. The results are plotted in Figure 3.19 for $\alpha = 0.2$, Figure 3.20 for $\alpha = 0.5$, Figure 3.21 for $\alpha = 0.7$, Figure 3.22 for $\alpha = 1.0$.

From the experiments, we conclude that

– Except for RMST and RMGT when α is large, all the new algorithms outperform those existing ones.

– Except when α is small, i.e, $\alpha < 0.25$, RM-FFDU-IFF outperforms all other algorithms. Again, RM-FFDU-IFF takes considerably more time to compute the results.

– The performance of RM-FFDU-IFF and RM-FFDU improves as α becomes larger, while the performance of RMST and RMGT degrades.

– RMST and RMGT performs the best when α is small, i.e, $\alpha < 0.25$. Of course, the performance of RMST and RMGT is identical when $\alpha < 1/3$.

– Though the performance of RM-FFDU and FFDUF is quite close, RM-FFDU performs consistently better than FFDUF.

– RMGT still performs quite well when $\alpha < 0.7$.

3.9.3. Yet Another Performance Evaluation of the Algorithms

The total utilization (load) of a task set is given by $\sum_{i=1}^{n} C_i / T_i$, which can be considered as the minimum number of processors required to execute the task set. It is a lower bound on the number of processors to be computed. The number of processors used to execute a task set is more than twice its total utilization in some cases for some algorithms. This comparison may be overly pessimistic, since the optimal number of processors may

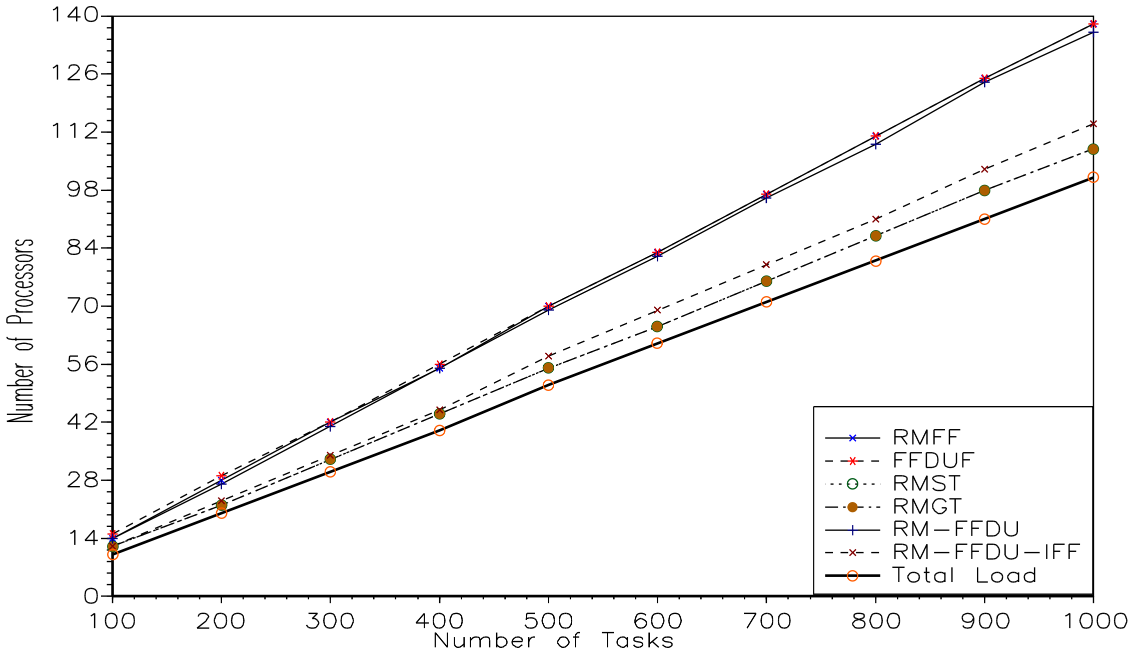


Figure 3.19: Performance of Off-line Algorithms (α = 0.2)

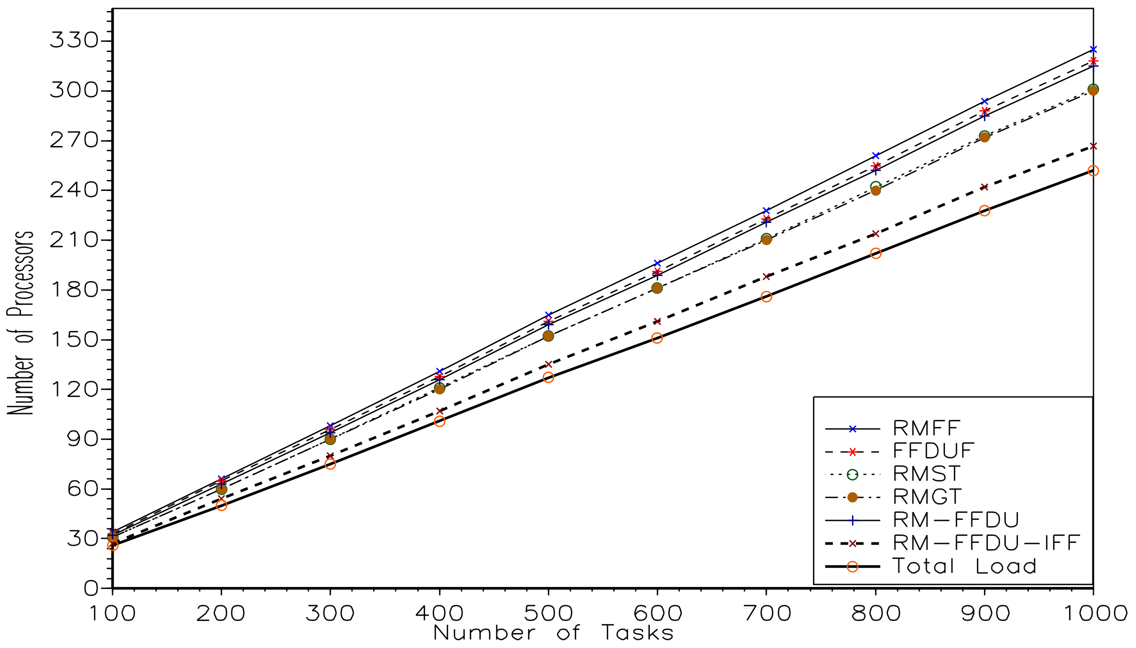


Figure 3.20: Performance of Off-line Algorithms (α = 0.5)

differ from the total utilization greatly in some cases, and little in other cases. Therefore, using the total utilization of the task set as a baseline for performance comparison may not capture the whole picture. The ideal solution would be to find the optimal number of pro-

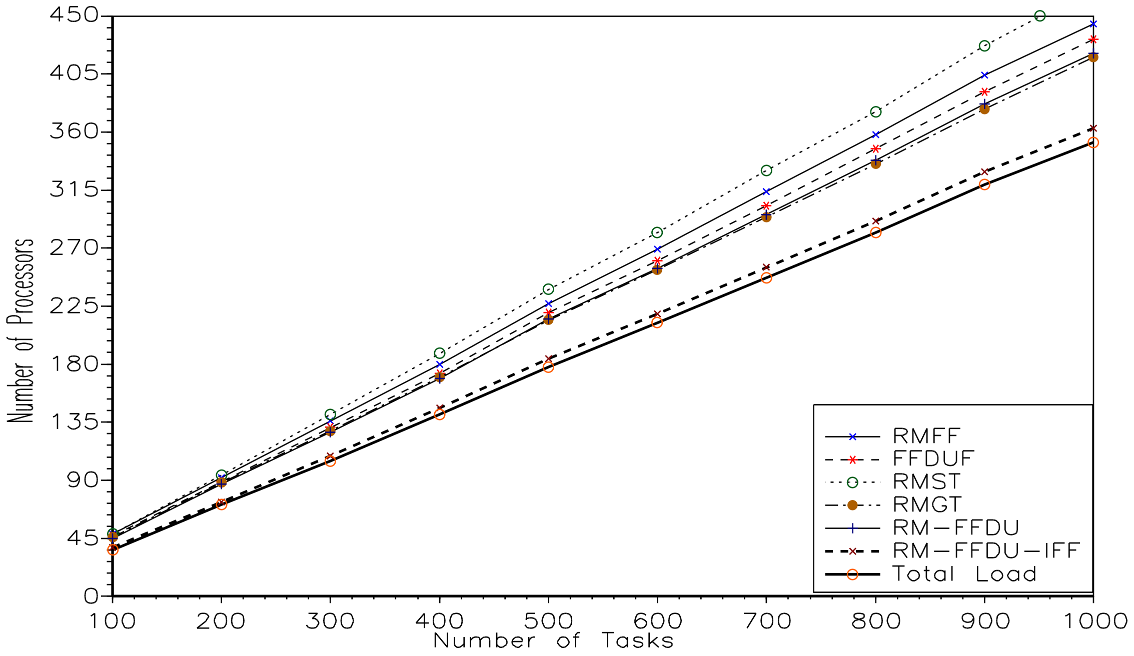


Figure 3.21: Performance of Off-line Algorithms (α = 0.7)

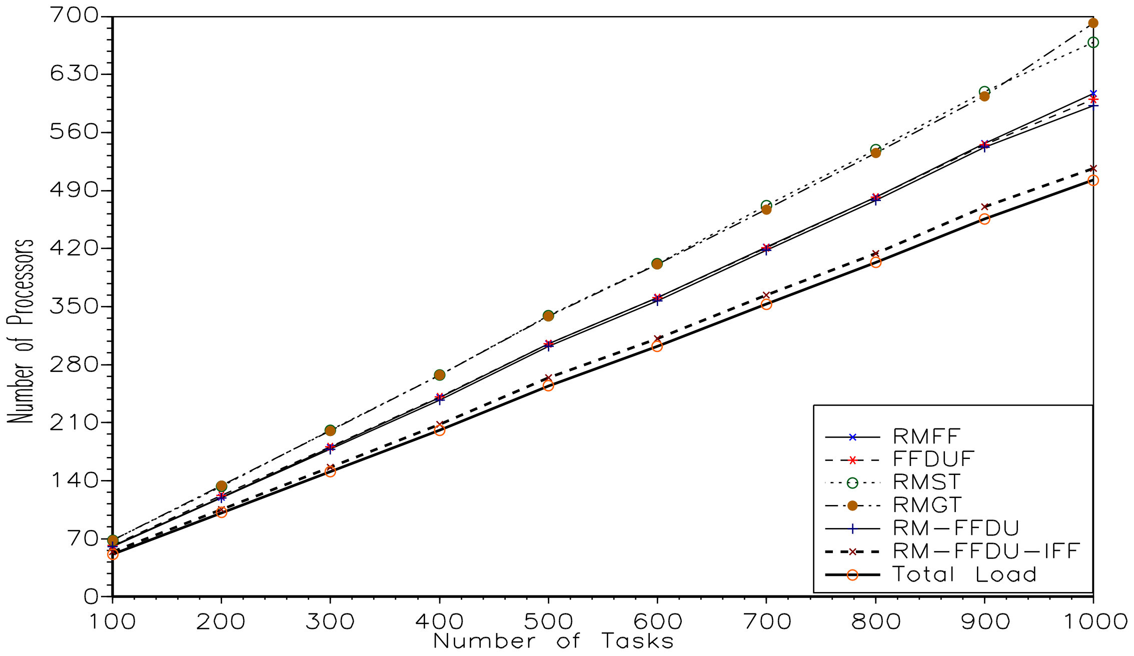


Figure 3.22: Performance of Off-line Algorithms (α = 1.0)

cessors for any given task set. This will, however, require at least exponential time with respect to the number of tasks using existing techniques, since the scheduling problem is

NP-complete.

This observation leads us to the employment of a different methodology. Under this methodology, a task set is generated randomly with the constraint that in the optimal scheduling, it fully utilizes a known number of processors. In other words, given *m* processors and the average number of tasks to be run on each processor, we generate a set of tasks that fully utilizes *m* processors in the optimal schedule. This is accomplished in the following steps:

(1) *M* arrays of random numbers are generated. The sizes of the arrays are uniformly generated, with a mean value corresponding to the average number of tasks on a processor.

(2) Each item in an array is divided by the sum of all items in its array to obtain a number between 0 and 1, which corresponds to the utilization of a task.

(3) For each of the *m* arrays, a number is generated as the period of all the tasks in that array. The numbers are randomly generated between 1 and 100.

(4) A number is randomly selected from the *m* arrays of numbers and then output as the utilization of a task. The computation time of the task is the product of its utilization and its period. This process of random outputting of tasks is repeated until all numbers in the *m* arrays are picked.

Using this methodology to generate task sets, the performance of some of the algorithms is plotted in Figure 3.23 and Figure 3.24. The average number of tasks assigned on a processor in the optimal schedule is selected to be 3 in Figure 3.23 and 6 in Figure 3.24. Each data point is the average value of 20 independently generated task sets with identical parameter. On the x-axis, the number of processors is the optimal number N_0 of processors required to execute a task set. On the y-axis, the extra percentage of processors is defined as $(N_A - N_0) / N_0$, where N_A is the number of processors required by a scheduling algorithm A to schedule the same task set. Note that the performance of the algorithms is consistent with that previously shown. With these figures, we have a better idea of what percentage of extra processors is needed for each algorithm for a given task set.

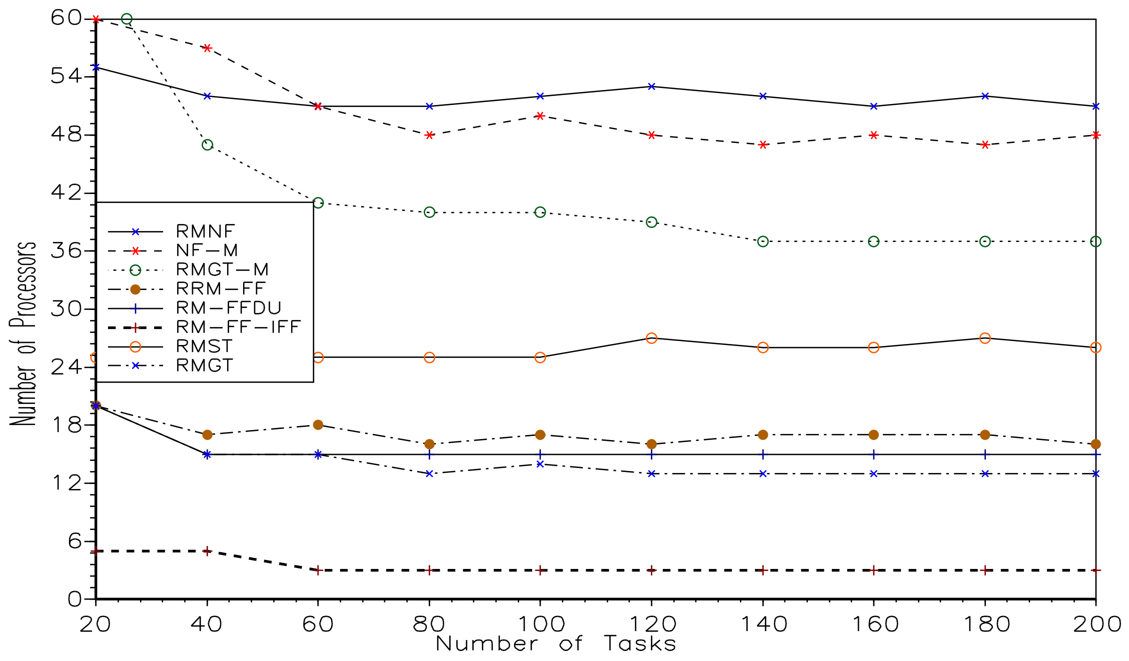


Figure 3.23: Performance of Some Algorithms (Tasks/processor = 3)

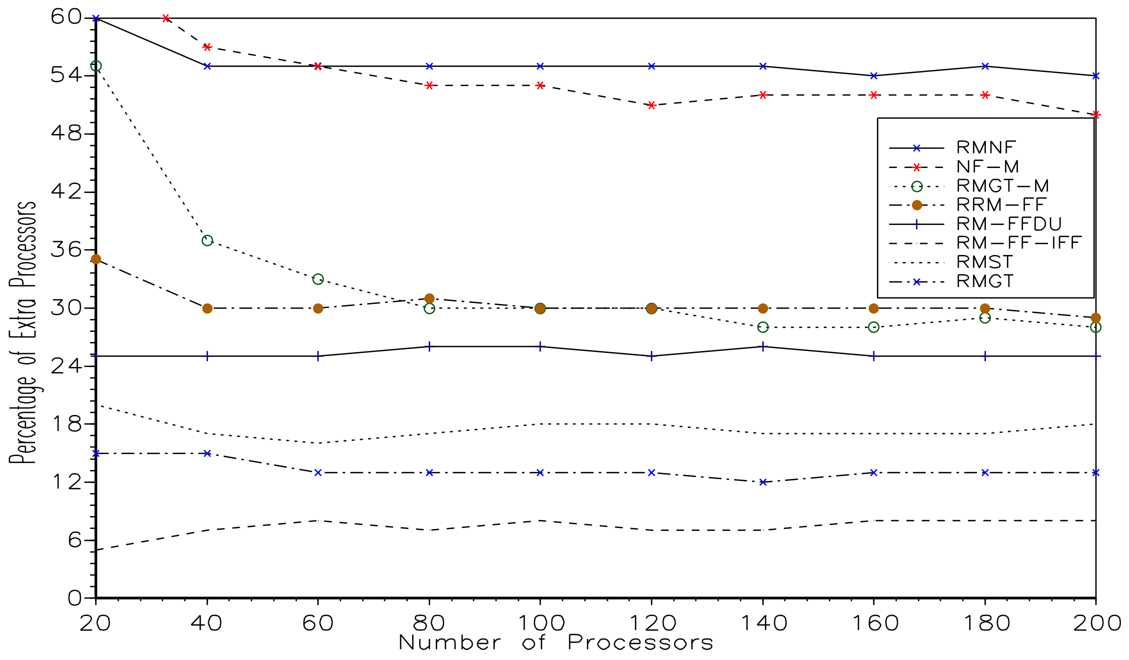


Figure 3.24: Performance of Some Algorithms (Tasks/processor = 6)

From the experiments, we conclude that

– The performance of most of the algorithms is quite good on the average, using less than 70% extra percentage of processors.

– The performance of RM-FFDU, RRM-FF, and RM-FFDU-IFF degrades a little bit as the number of tasks assigned on each processor becomes larger, i.e., the average task utilization decreases.

In the above experiments, we only consider the performance of the algorithms for large number of processors. In many practical applications, the total number of processors used may be less than 20 processors. In order to assess how some of the algorithms perform under this condition, we conduct similar experiments, the results of which are given in Figure 3.25 and Figure 3.26. Note that each data point represents the average value of 40 independent runs with $M = 4$. It is apparent that except for RMGT-M and NF-M whose performance depends on the value of M, the performance of the rest of the algorithms is quite consistent. Therefore, we can conclude that most of the algorithms are suitable for applications requiring few processors as well.

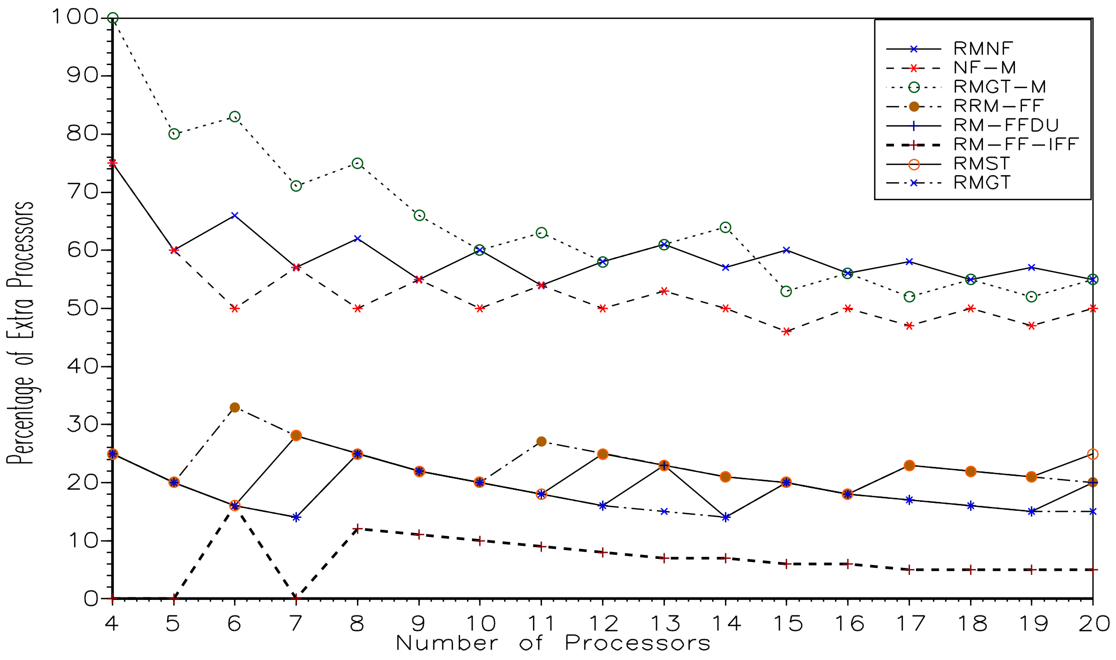


Figure 3.25: Performance of Some Algorithms (Tasks/processor = 3)

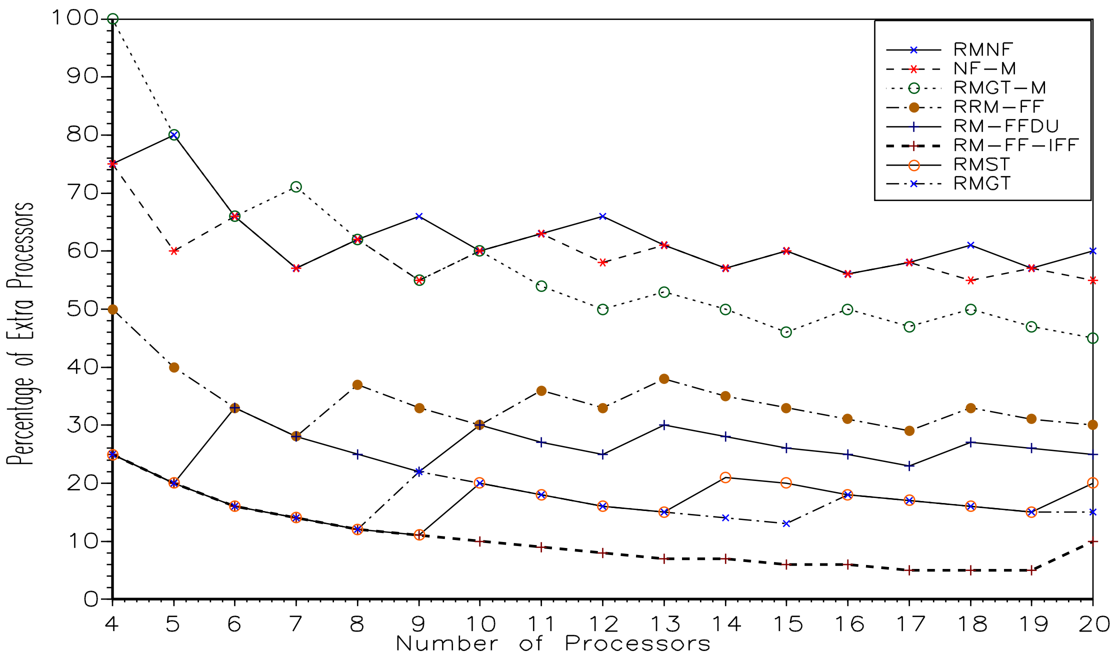


Figure 3.26: Performance of Some Algorithms (Tasks/processor = 6)

3.10. Summary

In this chapter, we have proposed a number of scheduling algorithms for the RMMS problem. All the algorithms are analyzed with respect to the worst case performance, and simulated for average case performance. Though some algorithms are better than others in the worst case, each of them has advantages that others do not.

- The algorithms based on the IFF condition outperform others in most of the cases. Yet it is time-consuming to execute these algorithms.
- The algorithms based on the UO condition have very good and consistent performance. Furthermore, they are simple and fast in execution.
- RMGT and RMGT perform best when the utilization of each task is small.

Chapter 4 Supporting Fault-Tolerance in Rate-Monotonic Scheduling

“Do not venture all your eggs in one basket.”
-- Spanish Proverb

In this chapter, we address the problem of supporting timeliness and dependability at the level of task scheduling. We consider the problem of scheduling a set of tasks, each of which, for fault-tolerance purposes, has multiple versions, onto the minimum number of processors. On each individual processor, the task deadlines are guaranteed by the rate-monotonic algorithm. A simple on-line allocation heuristic is proposed. It is proven that $N \leq 2.33\, N_0 + \kappa$, where N is the number of processors required to feasibly schedule a set of tasks by the heuristic, N_0 is the minimum number of processors required to feasibly schedule the same set of tasks, and κ is the maximum redundancy degree of a task. The bound is also shown to be nearly tight. The average case behavior of the heuristic is studied through simulation. Experimental data show that the heuristic performs surprisingly well on the average.

There are two general approaches to achieve fault-tolerance under RM Scheduling (RMS). The first approach [21] considers processor failures only; a set of periodic tasks is assigned to a processor such that their deadlines are met with the assumption that the processor is fault-tolerant. The fault-tolerant processor may be implemented by such hardware redundancy techniques as the TMR, where triple processors are used to execute the same tasks and the final results are decided upon through voting mechanisms. The major drawbacks of this approach are: (1) tasks are treated uniformly. In practise, however, some tasks may be more critical than others, and their correctness should be ensured by all means, while others may be allowed to miss their deadline occasionally. (2) Resources (e.g., processors) are under-utilized. (3) Possible task errors cannot be prevented from causing total system failures. The second approach can, in theory, tolerate processor failures as well as task errors. Under this approach, a task is replicated or implemented using several versions. The copies or versions of a task are executed on different processors in order to tolerate processor failures. Multiple versions of a task are executed so that possible task errors can be tolerated if they are not the same [2]. Since this approach is more general in the sense that the degree of redundancy on the task level and the processor level is allowed to be different and the tolerance of task errors is taken into account, we focus on this approach in the rest of the chapter and study its effectiveness when it is combined with the RMS.

For this approach to work, it is apparent that copies or versions of a task should be assigned to different processors and the total number of processors used should be minimized. The importance of minimizing the number of processors used to accommodate a set of tasks should not be under-estimated. First, more processors will introduce more processor failures, under probability. Second, more processors will affect the cost, weight, size, and power consumption of the whole system, the increase of any of which may jeopardize the success of the whole application. Therefore, we want to minimize the number of processors required to schedule a set of replicated, periodic tasks such that the timeliness and reliability of the system is guaranteed.

Although it is quite straightforward to assign copies or versions of a task to different processors is quite straightforward, it is non-trivial to minimize the number of processors used to schedule a set of tasks. In fact, the minimization problem has been proven to be NP-complete even in the case where each task has only one copy [43]. However, this fact does not make the problem go away; rather it requires that heuristic algorithms must be developed to solve it. In the following, we propose a simple scheduling algorithm to solve the problem. We then analyze the performance of the algorithm under worst case assumption, and show that in the worst cases, the number of processors used by the heuristic algorithm is no more than 2.33 times that of an optimal algorithm. This is, to our knowledge, the first nearly tight bound obtained for this particular problem. We are also interested in the average case behavior of the algorithm. Simulation results show that the algorithm performs very well on the average. We believe that this is an important step towards building fault-tolerant real-time systems based on the RMS theory.

4.1. Task Model

The Rate-Monotonic Scheduling theory was developed under a set of assumptions. These assumptions, along with the new requirement of task redundancy, are stated as follows:

(A) Each task has κ versions, where κ is a natural number. The κ versions of a task may have different computation time requirements, and the κ versions may be merely copies of one implementation or truly versions of different implementations.

(B) All versions of each task must be executed on different processors.

(C) The requests of all tasks are periodic with constant intervals between requests. The request of a task consists of the requests of all its versions, i.e., all versions of a task are ready for execution when its request arrives.

(D) Each task must be completed before the next request for it arrives, i.e., all its versions must be completed at the end of each request period.

(E) The tasks are independent in the sense that the requests of a task do not depend on the initiation or the completion of requests for other tasks.

Assumptions (A) and (B) make a rather general statement about the redundancy schemes used by each task. The term “version” has been used in *N*-version programming [2] to denote multiple implementations of a task. However, for the sake of convenience, it is used here to denote both true versions of a task and mere copies of a single task version. In the case of using merely duplicated copies, the errors produced by a task cannot be tolerated, since all the versions, or merely duplicated copies, produce the same results. But processor failures can be tolerated by using mere copies of a task. Here we are not concerned with details about what faults are to be tolerated or how faults are tolerated, rather we make the general statement that for fault-tolerance purposes, each task has a number of versions that must be executed on different processors. Note that the number of versions used by each task may be different, i.e., each κ_i may assume a different value.

Assumptions (C), (D), and (E) represent a simplified model of most practical real-time applications. This basic model may not be of much practical relevance if it cannot be extended to accommodate other requirements. Recently this model has been adapted and extended in many aspects by researchers in solving practical problems [8, 69, 72].

FT-RMMS Scheduling Problem: a set of n tasks $\Sigma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ is given with $\tau_i = ((C_{i1}, C_{i2}, \ldots, C_{i\kappa_i}), R_i, D_i, T_i)$ for $i = 1, 2, \ldots, n$, where $C_{i1}, C_{i2}, \ldots, C_{i\kappa_i}$ are the computation times of the κ_i versions of task τ_i. R_i, D_i, and T_i are the release time, deadline, and period of task τ_i, respectively. The question is to schedule the task set Σ using the minimum number of processors such that all the task deadlines are met and all versions of a task execute on different processors.

An optimal algorithm is the one that always uses the minimum number of processors to execute any given task set. According to Assumption (D), the deadline of each task coincides with its next arrival. For periodic task scheduling, it has been proven [46] that the release times of tasks do not affect the schedulability of the tasks. Therefore, release time R_i and deadline D_i can be safely omitted when we consider solutions to the problem. $u_{ij} = C_{ij} / T_i$ is the utilization (or load) of the jth version of task τ_i. $u_i = \sum_{j=1}^{\kappa_i} C_{ij}/T_i$ is the utilization (or load) of task τ_i.

4.2. The Design and Analysis of FT-Rate-Monotonic-First-Fit

Since the result of $(9m)/(8(m-r+1))$ by Bannister and Trivedi [5] is very attractive in view of workload distribution among the processors, it is quite tempting to expect that based on their algorithm, a good heuristic could be developed to solve the FT-RMMS problem. Even though no schedulability test is introduced in their algorithm, it can be added. The restriction that all tasks have the same number of clones and all versions of a task have the same computation time can be relaxed. The major problem left is to minimize the number of processors. We accomplish this by using a binary search technique.

The design of the heuristic consists of two steps: first, assuming *m* number of processors is sufficient for the execution of the task set Σ, Algorithm 0 is used to assign versions of tasks to different processors such that versions of a task are assigned to different processors, and the set of assigned tasks on each processor is schedulable under the RM algorithm. Second, a binary search technique, Algorithm 1, is used to find the minimum number of processors that is possible under Algorithm 0.

Algorithm 0 (Input: task set Σ, *m*; Output: *success*)

```
(1) Initialize $U_i = 0$ for $1 \le i \le m$, and $t = 1$.
(2) Assign the $\kappa_i$ versions of task t simultaneously to the $\kappa_i$ least
    utilized processors, and increment the utilization for each
    processor i to which a version of task t has been assigned by
    $C_{tj}/T_t$, where $j \in \{1, 2, \ldots, \kappa_t\}$. If $U_i > l(2^{1/l} - 1)$, where l is the
    number of versions having been assigned to processor i, Then
    success = FALSE, return. Otherwise, t = t + 1.
(3) If t > n Then success = TRUE, return. Otherwise, go to (2).
```

Figure 4.1: Algorithm 0

The lower bound for the number of processors that is sufficient to execute the task set is given by $\sum_{i=1}^{n}\left(\sum_{j=1}^{\kappa_i} C_{ij}\right)/T_i$, which is the total load of the task set, without considering the fault-tolerant constraint that versions of a task be assigned on different processors. The upper bound is given by $n \times max_{(1 \le i \le n)}\{\kappa_i\}$, which is equal to the total number of tasks times the maximum number of versions of a task. The correctness of the

Algorithm 1 (Input: task set Σ; Output: *m*)

```
(1) LowerBound = $\sum_{i=1}^{n}\left(\sum_{j=1}^{\kappa_i} C_{ij}\right)/T_i$; UpperBound = $n \times max_{\{1 \le i \le n\}}\{\kappa_i\}$;
(2) m = $\lfloor$(LowerBound + UpperBound) / 2$\rfloor$; If (LowerBound = m) Then {m
     = m + 1; EXIT};
(3) Invoke Algorithm 0(Σ,m,success); If success Then UpperBound = m
     Else LowerBound = m.Goto (2).
```

Figure 4.2: Algorithm 1

algorithm is self-evident.

Theorem 4.1: *Let N and N_0 be the number of processors required by Algorithm 1 and the minimum number of processors required to feasibly schedule a set of tasks. Then $N / N_0 \ge C$, where C is any given number.*

Proof: This theorem is proven by constructing task sets that can achieve the bound. Let the maximum number of versions of a task be $\kappa \ge 1$. Then for any given $C \ge 2$, the following task set is constructed:

The task set has a total of *C* tasks, each having κ versions. Versions of a task have the same utilization. For the first $(C - 1)$ tasks, the utilization is given by a very small number $\varepsilon > 0$. The utilization of the *C*th task is given by $2\left(2^{1/2} - 1\right)$. Since $\varepsilon + 2\left(2^{1/2} - 1\right) > 2\left(2^{1/2} - 1\right)$, none of the first $(C - 1)$ tasks can be scheduled together with the *C*th task. Therefore, a total of $C \bullet \kappa$ processors are used by Algorithm 1, while the optimal algorithm uses only κ processors, if $\varepsilon < [1 - 2\left(2^{1/2} - 1\right)] / (C - 1)$. Therefore, $N = C \bullet \kappa$, $N_0 = \kappa$, and $N/N_0 \ge C$. ■

The worst case performance of Algorithm 1 is very poor; it may be caused by the incompatibility of the allocation algorithm and the binary search strategy. A new algorithm based on a bin-packing heuristic is thus developed to obtain better solution. This new algorithm allocates versions of tasks to processors in the similar manner as bin-packing heuristics pack items into bins, with the exception that versions of a task cannot be assigned on a processor.

Bin-packing algorithms [15] are a class of well-studied heuristic algorithms, which perform well for the assignment of variable-size items into fixed-size bins. However, bin-packing heuristics cannot be directly applied to solving the FT-RMMS problem, since there are two major differences involved. First, the full utilization of a processor cannot be always achieved for a set of periodic tasks scheduled by the RM algorithm. In other words, the allocation of tasks to processors is equivalent to packing items into bins with dynamic sizes. Second, the fault-tolerant requirement that versions of a task cannot be assigned onto a processor further complicates the problem. Some modifications of the bin-packing heuristics are necessary. Here we choose to study the following heuristic, which is based on the First-Fit bin-packing heuristic for its simplicity and effectiveness.

Let the processors be indexed as P_1, P_2, …, with each initially in the idle state, i.e., with zero utilization. The tasks $\{\tau_1, \tau_2, \ldots, \tau_n\}$ will be scheduled in that order. κ is the maximum number of versions of a task, i.e., $\kappa = max_{\{1 \le i \le n\}} \kappa_i$. To schedule a version ν of task τ_i, find the least j such that ν, together with all the tasks (versions) that have been assigned to processor P_j, can be feasibly scheduled according to the RM condition for a single processor, and assign task version ν to P_j. When there is no confusion, we sometimes refer to a version belonging to a task τ_i simply as task τ_i.

When the algorithm returns, the value of m is the number of processors required to execute a given set of tasks. Since an idle processor will not be used until all the processors with some utilizations cannot accommodate a new task, it is therefore expected that FT-RM-FF would have better performance than that of Algorithm 1, which is indeed the case as shown by Theorem 4.2. Before proving the upper bound, however, a number of lemmas need to be established.

For clarity purposes, we use a slightly different notations in the proofs below. The versions belonging to a task are referred to as tasks belonging to a task group. Just as versions belonging to the same task should be allocated on different processors, so should tasks belonging to a task group. According, κ is the maximum number of tasks in a task

FT-Rate-Monotonic-First-Fit (FT-RM-FF) (Input: task set Σ; Output: m)

```
(1) Set i = 1 and m = 1. /* i denotes the ith task, m the number of
    processors allocated */
(2) (a) Set l = 1. /* l denotes the lth version of task $\tau_i$ */
    (b) Set j = 1. If the lth version of task i together with the
        versions that have been assigned to processor $P_j$ can be fea-
        sibly scheduled on $P_j$ according to the RM condition for a
        single processor and no version of task i has been previously
        assigned to processor $P_j$, assign the lth version of task i
        to $P_j$. Otherwise, j = j + 1 and go to step 2(b).
    (c) If l > $\kappa_i$, i.e., all versions of task i have been scheduled,
        then go to Step 3. Otherwise, increment l = l + 1, and go to
        Step 2(b).
(3) If j > m, then set m = j. If i > n, i.e., all tasks have been
    assigned, then return. Otherwise i = i + 1 and go to Step 2(a).
```

Figure 4.3: Algorithm FT-RM-FF

group, i.e., $\kappa = max_{\{1 \le i \le n\}} \kappa_i$.

Lemma 4.1: *Suppose the maximum number of tasks in a group is κ. Among all the processors on which $n \ge c \ge 1$ tasks are assigned, there are at most κ processors, each of which has a utilization less than or equal to $c\left(2^{1/(c+1)} - 1\right)$.*

Proof: The lemma is proven by contradiction. Suppose there are $\kappa + 1$ processors each of which has a utilization less than or equal to $c\left(2^{1/(c+1)} - 1\right)$, and let $P_1, P_2, \ldots, P_{\kappa+1}$ be the $\kappa + 1$ such processors, and n_i be the number of tasks assigned to processor P_i with $n_i \ge c$. Let $u_{i,j}$ be the utilization of the jth task that is assigned to processor P_i, for $1 \le i \le \kappa + 1$ and $1 \le j \le n_i$. Then $\sum_{j=1}^{n_i} u_{i,j} \le \left(2^{1/(c+1)} - 1\right)$, for $1 \le i \le \kappa + 1$.

For $1 \le x \le n_{\kappa+1}$, if $u_{\kappa+1,x} \ge \left(2^{1/(c+1)} - 1\right)$, then there exists a task with a utilization $u_{\kappa+1,y} \le \left(2^{1/(c+1)} - 1\right)$, since there are totally $n_{\kappa+1} \ge c$ tasks on each processor and $\sum_{j=1}^{n_{\kappa+1}} u_{\kappa+1,j} \le c\left(2^{1/(c+1)} - 1\right)$, where $x \ne y$ and $1 \le y \le n_{\kappa+1}$. In other words, there exists a task $\tau_{\kappa+1,z}$ on processor $P_{\kappa+1}$ satisfying $u_{\kappa+1,z} \le \left(2^{1/(c+1)} - 1\right)$ with $z \in \{1, 2, \ldots, n_{\kappa+1}\}$.

Since $\sum_{j=1}^{n_i} u_{i,j} + u_{\kappa+1,z} \le c\left(2^{1/(c+1)} - 1\right) + \left(2^{1/(c+1)} - 1\right) = (c+1)\left(2^{1/(c+1)} - 1\right)$ for $1 \le i \le \kappa$, and $u_{\kappa+1,z}$ cannot be assigned on processor P_i, there must exist one and only one task $\tau_{i,j}$ among $\{\tau_{i,j} | j = 1, 2, \ldots, n_i\}$ that belongs to the same group as $u_{\kappa+1,z}$ does, for all $i = 1, 2, \ldots, \kappa$. In other words, the task group that contains task $\tau_{\kappa+1,z}$ has $\kappa + 1$ tasks. This is a contradiction to the assumption that the maximum number of tasks in any group is κ. Therefore the lemma must be true. ■

Lemma 4.2: *If m tasks cannot be feasibly scheduled on* $m - 1$ *processors according to FT-RM-FF, and* $\kappa > 1$*, then the utilization factor of the m tasks is greater than* $2(m - \kappa)\left(2^{1/2} - 1\right)$ *for* $\kappa < m \le 2\kappa$*; or* $m\left(2^{1/2} - 1\right)$ *for* $m > 2\kappa$*.*

Proof: For $\kappa = 1$, the lemma is true by Lemma 3.1. For $\kappa \ge 2$, there are two cases to consider:

Case 1: $\kappa < m < 2\kappa$. Then $\sum_{i=1}^{m} u_i > 2(m-\kappa)\left(2^{1/2} - 1\right)$, where u_i is the utilization of task τ_i.

Since there are $m > \kappa$ tasks in total, these tasks must belong to at least two different task groups. Suppose that the least number of tasks in a group among these m tasks is p. Since these p tasks cannot be scheduled together with any other tasks on a single processor, we have

$$u_i + u_j' > 2\left(2^{1/2} - 1\right), \tag{Eq.4.1}$$

for $i = 1, 2, \ldots, p$ and $j = 1, 2, \ldots, m - p$, where u_is are the utilizations of the p tasks, and u_j's the utilizations of the rest of the tasks (see Figure 4.4).

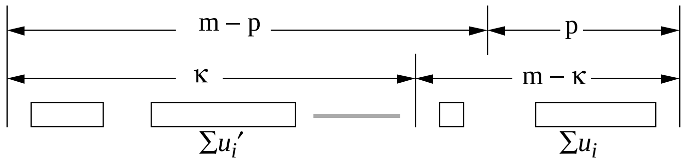


Figure 4.4: Task Configuration when $\kappa < m \le 2\kappa$

Apparently, $\sum_{i=1}^{m} u_i = \sum_{i=1}^{m-p} u_i' + \sum_{i=1}^{p} u_i$. Summing up the $p(m-p)$ inequalities in (4.1) yields

$$p\sum_{i=1}^{m-p} u_i' + (m-p)\sum_{i=1}^{p} u_i > 2p(m-p)\left(2^{1/2} - 1\right). \tag{Eq.4.2}$$

If $p \geq m - p$, then

$$p(\sum_{i=1}^{m-p} u_i' + \sum_{i=1}^{p} u_i) \geq p \sum_{i=1}^{n-p} u_i' + (m-p) \sum_{i=1}^{p} u_i > 2p(m-p)\left(2^{1/2} - 1\right).$$

Therefore, $\sum_{i=1}^{m} u_i = \sum_{i=1}^{m-p} u_i' + \sum_{i=1}^{p} u_i > 2\,(m-p)\left(2^{1/2} - 1\right)$. Since $p \leq \kappa$, we have $m - p \geq m - \kappa$. $\sum_{i=1}^{m} u_i > 2(m-\kappa)\left(2^{1/2} - 1\right)$.

If $p < m - p$, then there are two sub-cases to consider. If $p \geq m - \kappa$, then from inequality (4.2), we have $(m-p)(\sum_{i=1}^{m-p} u_i' + \sum_{i=1}^{p} u_i) \geq p \sum_{i=1}^{m-p} u_i' + (m-p) \sum_{i=1}^{p} u_i > 2\,p\,(m-p)\left(2^{1/2} - 1\right)$. In other words, $\sum_{i=1}^{m} u_i = \sum_{i=1}^{m-p} u_i' + \sum_{i=1}^{p} u_i > 2p\left(2^{1/2} - 1\right) \geq 2(m-\kappa)\left(2^{1/2} - 1\right)$, since $p \geq m - \kappa$ by assumption. If $p < m - \kappa$, then there must exist $(m - \kappa - \mathrm{p})$ tasks, which belong to some task groups that are different from the task groups the rest of the κ tasks belong to. For each of the $(m - \kappa)$ tasks as shown in Figure 4.5, it can

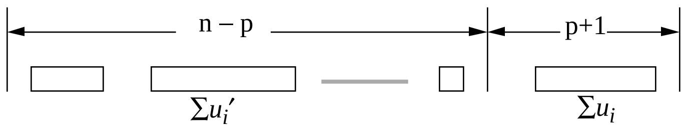


Figure 4.5: Task Configuration when $2\kappa < m$

be paired with some of the $(m - \kappa)$ distinctive tasks among the rest of the κ tasks, such that we have

$$u_i + u_j' > 2\left(2^{1/2} - 1\right), \text{ for } i = 1, 2, \ldots, m-\kappa \text{ and } j = 1, 2, \ldots, m-\kappa.$$

$$\sum_{i=1}^{m} u_i = \sum_{\kappa} u' + \sum_{m-\kappa} u \geq \sum_{m-\kappa} u' + \sum_{m-\kappa} u > 2(m-\kappa)\left(2^{1/2} - 1\right).$$

Case 2: $m > 2\kappa$. Then $\sum_{i=1}^{m} u_i > m\left(2^{1/2} - 1\right)$. Proving this claim is equivalent to proving the following:

Suppose that the total number of task groups is γ with $\gamma > 2$, and the maximum number of tasks in a group is κ, then for $m > 2\kappa$, $\sum_{i=1}^{m} u_i > m\left(2^{1/2} - 1\right)$.

We prove this claim by using induction on the number of tasks in a group among the γ task groups. First, $m = \gamma$, i.e., $\gamma > 2\kappa$. Since each task belongs to a different group, $\sum_{i=1}^{m} u_i > m\left(2^{1/2} - 1\right)$ according to Lemma 3.1.

Suppose that $\sum_{i=1}^{m} u_i > m\left(2^{1/2} - 1\right)$ is true for $q_i \leq p_i$, where q_i is the number of tasks belonging to task group i, $p_i \geq 1$ is a constant number, for $1 \leq i \leq \gamma$, and $\sum_{i=1}^{\gamma} p_i = n \geq m$. Then for a newly added task belonging to task group j, it is equivalent to saying that $q_j = p_j + 1$, or $m = n + 1$. The newly added task cannot be scheduled on any of the m processors. Let τ_{n+1} denote the newly added task with utilization u_{n+1}. Since $m \leq n$, $m > 2\kappa$, and $\kappa > p_j$, then $n > 2p_j$. For convenience, we let $p = p_j$.

If $\sum_{i=1}^{p+1} u_i \leq (p+1)\left(2^{1/2} - 1\right)$, then

$$(p+1)\sum_{i=1}^{n-p} u_i' + (n-p)\sum_{i=1}^{p+1} u_i > 2(p+1)(n-p)\left(2^{1/2} - 1\right).$$

$$(p+1)\left(\sum_{i=1}^{n-p} u_i' + \sum_{i=1}^{p+1} u_i\right) + (n-2p-1)\sum_{i=1}^{p+1} u_i$$
$$> (p+1)(n+1)\left(2^{1/2} - 1\right) + (p+1)(n-2p-1)\left(2^{1/2} - 1\right).$$

$$\sum_{i=1}^{n-p} u_i' + \sum_{i=1}^{p+1} u_i > (n+1)\left(2^{1/2} - 1\right) + (n-2p-1)\left[(p+1)\left(2^{1/2} - 1\right) - \sum_{i=1}^{p+1} u_i\right].$$

Since $n - 2p - 1 \geq 0$,

$$\sum_{i=1}^{m} u_i = \sum_{i=1}^{n-p} u_i' + \sum_{i=1}^{p+1} u_i > (n+1)\left(2^{1/2} - 1\right).$$

If $\sum_{i=1}^{p+1} u_i > (p+1)\left(2^{1/2} - 1\right)$, there are two sub-cases to consider.

If $u_{n+1} \geq \left(2^{1/2} - 1\right)$, then $\sum_{i=1}^{m} u_i = \sum_{i=1}^{n} u_i + u_{n+1} > n\left(2^{1/2} - 1\right) + \left(2^{1/2} - 1\right) = (n+1)\left(2^{1/2} - 1\right)$.

If $u_{n+1} < \left(2^{1/2} - 1\right)$, then $\sum_{i=1}^{p+1} u_i = \sum_{i=1}^{p} u_i + u_{n+1} > (p+1)\left(2^{1/2} - 1\right)$ (we assume $u_{n+1} = u_{p+1}$ for convenience). $\sum_{i=1}^{p} u_i > p\left(2^{1/2} - 1\right)$. Since τ_{n+1} cannot be scheduled together with any of the $n - p$ tasks on any processor, $\sum_{i=1}^{n-p} u_i' + (n-p)u_{n+1} > 2(n-p)\left(2^{1/2} - 1\right)$.

$$\sum_{i=1}^{m} u_i = \sum_{i=1}^{n-p} u_i' + \sum_{i=1}^{p+1} u_i = \sum_{i=1}^{n-p} u_i' + \sum_{i=1}^{p} u_i + u_{n+1}$$
$$> 2(n-p)\left(2^{1/2} - 1\right) + p\left(2^{1/2} - 1\right) - (n-p-1)u_{n+1}$$
$$= (n+1)\left(2^{1/2} - 1\right) + (n-p-1)\left(\left(2^{1/2} - 1\right) - u_{n+1}\right) \geq (n+1)\left(2^{1/2} - 1\right),$$ since $n > p + 1$. ■

Theorem 4.2: *Let N and N_0 be the number of processors required by FT-RM-FF and the minimum number of processors required to feasibly schedule a set of tasks, respectively. Then* $N \leq \left(2 + \left(3 - 2^{3/2}\right) / \left(2\left(2^{1/3} - 1\right)\right)\right) N_0 + \kappa$, *where* κ *is the maximum number of tasks in a task group. Hence* $2.2833 \leq \Re_{FT-RM-FF}^{\infty} \leq 2.33$.

In order to prove the above bound, we define a weighting function that maps the utilization of tasks into the real interval [0, 1] as follows:

$$W(u) = \begin{cases} u/a & 0 \le u < a \\ 1 & a \le u \le 1 \end{cases}, \text{ where } a = 2\left(2^{1/3} - 1\right).$$

Lemma 4.3: *If a processor is assigned a number of tasks* $\tau_1, \tau_2, \ldots, \tau_n$ *with utilizations* $u_1, u_2, \ldots, u_n$*, then* $\sum_{i=1}^{n} W(u_i) \le 1/a$*, where* $a = 2\left(2^{1/3} - 1\right)$.

This lemma is true according to Lemma 3.1

Lemma 4.4: *In the completed FT-RM-FF schedule. If a processor is assigned m* ≥ 2 *tasks and* $\sum_{i=1}^{m} u_i \ge 2\left(2^{1/3} - 1\right)$*, then* $\sum_{i=1}^{m} W(u_i) \ge 1$.

Proof: Since $\sum_{i=1}^{m} u_i \ge 2\left(2^{1/3} - 1\right)$, $\sum_{i=1}^{m} W(u_i) \ge 1$ by the definition of weighting function. ■

Proof of Theorem 4.2: Let $\Sigma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ be a set of *m* tasks, with their utilizations $u_1, u_2, \ldots, u_n$, respectively, and $\varpi = \sum_{i=1}^{n} W(u_i)$.

Suppose that among the *N* processors that are used by FT-RM-FF to schedule a given set Σ of tasks, *L* of them have $\sum_j W(u_j) = 1 - \beta_i$ with $\beta_i > 0$, where *j* ranges over all tasks in processor *i* among the *L* processors. Let us divide these processors into two different classes:

(1) Processors to each of which only one task is assigned. Let n_1 denote the number of processors in this class.

(2) Processors to each of which two or more tasks are assigned. Let n_2 denote the number of processors in this class. According to Lemma 4.1, there are at most κ processors whose utilization in the schedule is less than or equal to $a = 2\left(2^{1/3} - 1\right)$. Therefore $n_2 = \kappa$.

Obviously, $L = n_1 + n_2$. For each of the rest $N - L$ processors, $\sum_j W(u_j) \ge 1$, where *j* ranges over all tasks in a processor. There are two cases to consider with regard to n_1.

Case 1: $n_1 > 2\kappa$. For the processors in class (1), $\sum_{i=1}^{n_1} u_i > n_1(2^{1/2} - 1)$ according to Lemma 4.1. Since $W(u_i) < 1$ by assumption, we have $u_i < a$. Therefore $\sum_{i=1}^{n_1} W(u_i) > n_1(2^{1/2} - 1)/a$. Moreover, according to Lemma 4.1, there are at most κ tasks, the utilization of each of which is less than or equal to $(2^{1/2} - 1)$. In the optimal assignment of these tasks, the optimal number N_0 of processors used cannot be less than $n_1 / 2$, i.e., $N_0 \geq n_1 / 2$, since possibly with κ exceptions, any three tasks among these tasks cannot be scheduled on one processor. Note that in the optimal schedule, the necessary and sufficient condition must be used, i.e., both computation time and period of a task are taken into account.

Now we are ready to determine the relationship between N and N_0.

$$\varpi = \sum_{i=1}^{m} W(u_i) \geq (N - L) + n_1 (2^{1/2} - 1) / a = N - n_1 - n_2 + n_1 (2^{1/2} - 1) / a$$

$$= N - n_1 \left[1 - \left(2^{1/2} - 1\right)/a\right] - n_2$$

$$\geq N - 2N_0 \left[1 - \left(2^{1/2} - 1\right)/a\right] - n_2, \text{ since } N_0 \geq n_1 / 2.$$

Since $\varpi \leq N_0 / a$ by Lemma 4.3,

$$N_0 / a \geq N - 2N_0 \left[1 - \left(2^{1/2} - 1\right)/a\right] - n_2 \geq N - 2N_0 \left[1 - \left(2^{1/2} - 1\right)/a\right] - \kappa$$

Therefore, $N \leq \left[2a + 1 - 2\left(2^{1/2} - 1\right)\right] N_0/a + \kappa$, where $a = 2\left(2^{1/3} - 1\right)$.

Case 2: $n_1 \leq 2\kappa$. Then $L = n_1 + n_2 \leq 3\kappa$, i.e., the number of processors with $W(u_i) < 1$ is at most 3κ. Since $\varpi \leq N_0 / a$ and $\varpi = \sum_{i=1}^{m} W(u_i) \geq N - L$,

$$N \leq N_0/a + L \leq N_0/a + 3\kappa.$$

If $n_1 \leq 2\kappa$, then the upper bound is given by $1 / a \approx 1.92$. This implies that if the number of processors on each of which one task is assigned is small, i.e., $n_1 \leq 2\kappa$, then the upper bound can be improved significantly, from 2.33 to 1.92.

Since the term κ is constant with respect to N in $N \leq \left[2a + 1 - 2\left(2^{1/2} - 1\right)\right] N_0/a + \kappa$, it becomes negligible when N_0 becomes large. Therefore, the worst case performance of FT-RM-FF is upper bounded by 2.33. According to Theorem 3.7, it is lower bounded by 2.283 when $\kappa = 1$. Hence $2.283 \leq \Re^{\infty}_{FT-RM-FF} \leq 2.33$, the bound of 2.33 is nearly tight.■

4.3. Average Case Performance Evaluation

In order to gain some insight into the average case behavior of the new algorithm, we use the same approach as we did in Section 3.9 to evaluate its performance.

Our simulation studies consist of two steps: (1) generate task sets with random distributions; (2) run the task sets through the algorithms to produce results. The output parameter for each algorithm is the number of processors used to accommodate a given set of tasks.

The input data of all parameters for a task set are generated according to uniform distribution. The periods of tasks are generated in the range of $1 \le T_i \le 500$. The number of versions for each task is uniformly distributed in the range of $1 \le \kappa_i \le 5$. The computation time of each version is in the range of $1 \le C_{i,j} \le \alpha T_i$, where α is the maximum allowable utilization of any version, i.e., $\alpha = max_{i,j}(C_{i,j}/T_i)$.

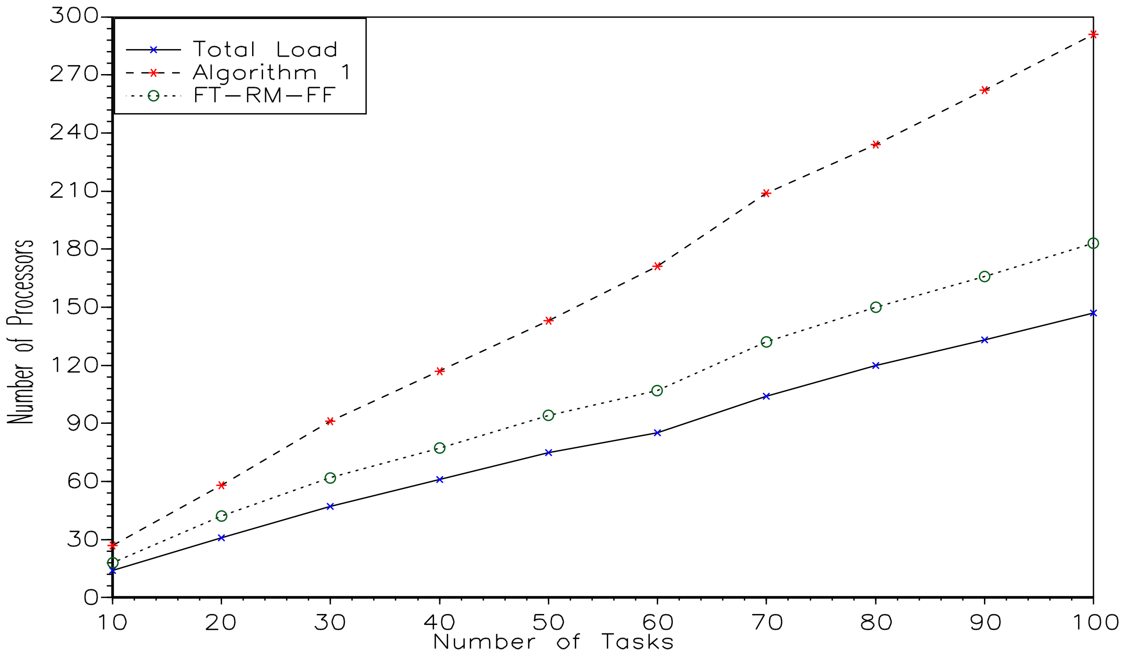


Figure 4.6: Performance of FT-RM-FF and Algorithm 1 (α = 1.0)

The result is plotted in Figures 4.6 and 4.7 with two values of α. Each data point depicts the average value of 10 independently generated task sets with identical parameters. In order to make comparisons, we also ran the same data through Algorithm 1, and the results are plotted in the same figure.

The total utilization (load) of a task set is given by $\sum_{i=1}^{n}\left(\sum_{j=1}^{\kappa_i} C_{i,j}\right)/T_i$, which can be considered as the minimum number of processors needed to execute the task set. It is a lower bound on the number of processors to be computed. FT-RM-FF outperforms

Algorithm 1 in all the experiments we have carried out. On the average, Algorithm 1 uses 130% extra processors compared to the lower bound, and FT-RM-FF uses 40% extra processors, which is a lot better than the 133% extra processors needed in the worst case.

Since using $\sum_{i=1}^{n}\left(\sum_{j=1}^{\kappa_i} C_{i,j}\right)/T_i$ as the lower bound for a scheduling algorithm may be too pessimistic, we are interested in finding out the extra percent of processors that is used by the FT-RM-FF algorithm to schedule any given task set. We will present some more simulation data about FT-RM-FF along with the algorithm presented in the next chapter.

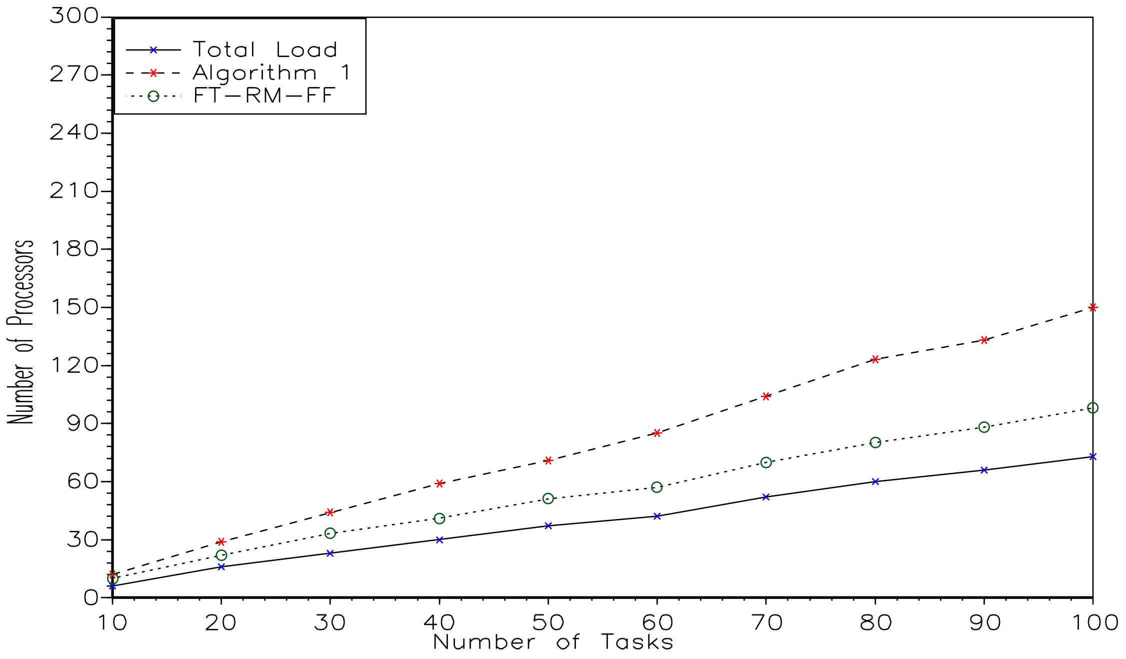


Figure 4.7: Performance of FT-RM-FF and Algorithm 1 (α = 0.5)

Chapter 5 Supporting Fault-Tolerance in Earliest-Deadline-First Scheduling

"Nothing ventured, nothing gained."
-- Anonymous

We now turn to the problem of scheduling a set of multiple-version periodic tasks using the minimum number of processors such that the task deadlines are met by the EDF algorithm on each individual processor. We state the problem as follows:

FT-EDFMS Problem: A set of n tasks $\Sigma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ is given, where $\tau_i = ((C_{i1}, C_{i2}, \ldots, C_{i\kappa_i}), R_i, D_i, T_i)$ for $i = 1, 2, \ldots, n.$, $C_{i1}, C_{i2}, \ldots, C_{i\kappa_i}$ are the computation times of the κ_i versions of task τ_i. R_i, D_i, and T_i are the release time, deadline, and period of task τ_i, respectively. What is the minimum number of processors required to execute the task set such that the versions of each task are executed on different processors and all the task deadlines are met by EDF?

Liu and Layland prove that a set of periodic tasks $\Sigma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ with the deadline of each task coinciding with its next arrival can be feasibly scheduled by EDF if and only if $\sum_{i=1}^{n} C_i/T_i \leq 1$ [46]. Note that the release time of each task, R_i, does not affect the schedulability of a set of periodic tasks. Therefore, R_i and D_i can be safely omitted in scheduling tasks to processors.

Since $0 < C_i/T_i \leq 1$ and $\sum_{i=1}^{n} C_i/T_i \leq 1$, we can treat the assignment of a set of tasks to a single processor as packing a list of items into a bin with a unit size. The quantity $u_i = C_i/T_i$ for a task (version) corresponds to the size of an item. In order to distinguish task versions belonging to one task from those belonging to another, we assign colors to them such that versions belonging to one task share the same color. Versions belonging to different tasks have different colors. Then items that have the same color cannot be assigned to the same bin and the maximum number of items having the same color is κ. The number of colors is therefore equal to the number of tasks in a task set. The problem of scheduling a set of multiple-version periodic tasks to processors can thus be reduced to the following ***bin-packing problem***:

An item is associated with a color, and its size is no more than 1. There are an infinite number of colors available. At most κ items have the same color. No items with the same color are assigned to the same bin. Then given a number of colorful items, what is the best way to pack the items into bins, such that the minimum number of bins is used?

With the exception of $\kappa = 1$, this bin-packing problem has not been studied. When $\kappa = 1$, the above bin-packing problem is reduced to the classical one-dimensional bin-packing problem, which has been extensively studied by a number of researchers for years. The classical one-dimensional bin-packing problem arises in a wide range of applications, such as computer memory allocation, packing trucks with a given weight limit, and assigning commercials to stations breaks on television.

The above bin-packing problem also occurs in a number of applications, besides the scheduling of tasks for fault-tolerance. For example, the problem of allocating a set of parallelized tasks to the minimum number of processors such that the makespan of the whole schedule is bounded can also be reduced to this bin-packing problem.

5.1. The Design and Analysis of FT-EDF-First-Fit

Since this bin-packing problem is an NP-complete problem, we resort to heuristic approach to solve it. We choose the First-Fit scheme for similar reason as outlined in the previous chapters. The new algorithm is called FT-EDF-FF.

FT-EDF-FF: Let the bins be indexed as B_1, B_2, …, with each initially filled to level zero. Given a list of colorful items, where the size of each item is no more than 1 and the maximum number of items having the same color is κ, the items are assigned to bins in the order they are given. In assigning an item to a bin, the smallest-indexed bin that does not contain an item with the same color as the item being assigned and in which the item can be fit, is selected to contain the item. An item is assigned to a new bin if it cannot be assigned to any non-empty bin.

The main result is stated in the following theorem. Where there is no confusion, we refer an item with a size of b simply as item b.

Theorem 5.1: *For any list L of items b_1, b_2, …, b_n, FT-EDF-FF(L) $\leq 1.7L^* + 2.19\kappa$, where κ is the maximum number of items having the same color, FT-EDF-FF(L) is the number of bins used by FT-EDF-FF to pack the list L, and L^* is the minimum number of bins used to pack the same list.*

Before proving the theorem, we need to establish several lemmas.

Lemma 5.1: *Suppose the maximum number of items having the same color is κ. Among all the bins to each of which $n \geq c \geq 1$ items are assigned, there are at most κ of them, each of which is no more than $c / (c + 1)$ full.*

Proof: The lemma is proven by contradiction. Suppose that there are $\kappa + 1$ bins each of which is no more than $c / (c + 1)$ full. Let $B_1, B_2, \ldots, B_{\kappa + 1}$ be such $\kappa + 1$ bins and $b_{i,j}$ be the jth item that is assigned to bin B_i, for $1 \leq i \leq \kappa + 1$ and $1 \leq j \leq n$. Then $\sum_{j=1}^{n} b_{i,j} \leq c / (c + 1)$, for $1 \leq i \leq \kappa + 1$.

Let us look at the sizes of items assigned to the last bin, $B_{\kappa + 1}$, among the $\kappa + 1$ bins. Since there are $n \geq c$ items in the bin $B_{\kappa + 1}$ and $\sum_{j=1}^{n} b_{\kappa+1,j} \leq c / (c + 1)$, there must exist an item $b_{\kappa+1,z}$ in the bin $B_{\kappa + 1}$ such that $b_{\kappa+1,z} \leq 1 / (c + 1)$ and $z \in \{1, 2, \ldots, n\}$. If not, then $\sum_{j=1}^{n} b_{\kappa+1,j} > c / (c + 1)$.

Since $\sum_{j=1}^{n} b_{i,j} + b_{\kappa+1,z} \leq 1 / (c + 1) + c / (c + 1) = 1$ for $1 \leq i \leq \kappa$, and $b_{\kappa+1,z}$ cannot be assigned to the bin B_i, there must exist one and only one item $b_{i,j}$ among the items $\{b_{i,j} | j = 1, 2, \ldots, n\}$, that has the same color as $b_{\kappa+1,z}$ does, for all $i = 1, 2, \ldots, \kappa$. In other words, there are a total of $\kappa + 1$ items having the same color as item $b_{\kappa+1,z}$. This is a contradiction to the assumption that the maximum number of items having the same color is κ. Therefore the lemma must be true. ■

In the following, we define a weighting function W(α) that maps the size of an item, α, to a number between zero and one, i.e., $W(\alpha)$: (0, 1] $\rightarrow$ (0, 1], as given in Figure 5.1. We call the value of $W(\alpha)$ the weight of item α, and the sum of the weights of the items assigned to a bin the weight of the bin. The weighting function is defined in such a way that with a few exceptions, the weight of a bin in the completed FT-EDF-FF packing is equal to or greater than 1, and the weight of a bin in the optimal packing is no greater than 1.7.

$$W(\alpha) = \begin{cases} (6\alpha)/5 & 0 < \alpha \le 1/6 \\ (9\alpha)/5 - 1/10 & 1/6 < \alpha \le 1/3 \\ (6\alpha)/5 + 1/10 & 1/3 < \alpha \le 1/2 \\ 1 & 1/2 < \alpha \le 1 \end{cases}$$

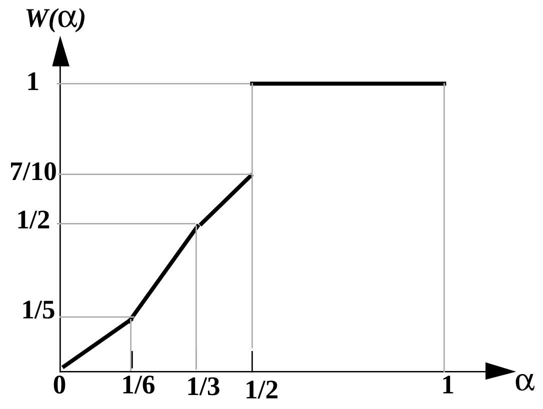


Figure 5.1: Weighting Function $W(\alpha)$

We first claim that for any bin in the optimal packing, the total weight of the bin is no greater than 1.7, i.e., $\sum_{i=1}^{m} W(b_i) \le 1.7$.

Lemma 5.2: *Let a bin be filled with items b_1, b_2, …, b_m. Then* $\sum_{i=1}^{m} W(b_i) \le$ 1.7.

This lemma was proven by Johnson in [30].

In order to prove that, with a limited number of bins, the weight of each bin in the completed FT-EDF-FF packing is no less than one, we divide the bins into several groups according to the levels they are filled to. Since a bin can be filled to a level from zero to one, we instead divide the bins into groups according to the regions their levels fall into. A total number of seven regions is defined: (0, 1/2], (1/2, 2/3], (2/3, 2/3 + 1/18), [2/3 + 1/18, 3/4), [3/4, 4/5), [4/5, 5/6), and [5/6, 1]. For each region, the result is stated in a lemma. The proof of the theorem is given at the end.

Lemma 5.3: *Let a bin be filled with items $b_1 \geq b_2 \geq \ldots \geq b_m$. If $\sum_{i=1}^{m} b_i \leq 1/2$, then there are at most κ bins with $\sum_{i=1}^{m} W(b_i) < 1$ and $m \geq 1$.*

Proof: According to Lemma 5.1, among all bins to each of which $m \geq 1$ items are assigned, there are at most κ of them, each of which is no more than 1/2 full. Therefore, there are at most κ bins with $\sum_{i=1}^{m} W(b_i) < 1$. ■

Lemma 5.4: *Let a bin be filled with items $b_1 \geq b_2 \geq \ldots \geq b_m$. If $1/2 < \sum_{i=1}^{m} b_i \leq 2/3$, then there are at most κ bins with $\sum_{i=1}^{m} W(b_i) < 1$ and $m \geq 2$.*

Proof: For $1/2 < \sum_{i=1}^{m} b_i \leq 2/3$, the bins with $\sum_{i=1}^{m} W(b_i) < 1$ must be assigned at least two items, i.e., $m \geq 2$. If $m = 1$, then $b_1 > 1/2$ and $\sum_{i=1}^{m} W(b_i) \geq 1$.

According to Lemma 5.1, among all bins to each of which $m \geq 2$ items are assigned, there are at most κ of them, each of which is no more than 2/3 full. Therefore, there are at most κ bins with $\sum_{i=1}^{m} W(b_i) < 1$. ■

For the region of (2/3, 2/3 + 1/18), there may be an infinite number of bins with $\sum_{i=1}^{m} W(b_i) < 1$. However, the deficiency of weights created by these bins can be bounded, as shown by the next lemma. However, in order to show that this deficiency can indeed be bounded, we need a few definitions.

Definition 5.1: Let a bin B_i be filled with items $b_1, b_2, \ldots, b_m$. The color of an item b_j is denoted by $\chi(b_j)$, and the set of colors of the items in a bin B_i is denoted by $\chi(B_i)$. The deficiency δ_i of a bin B_i is defined as $\delta_i = 1 - \sum_{i=1}^{m} b_i$, i.e., where the bin is filled up to the level of $1 - \delta_i$ in the completed FT-EDF-FF packing. For convenience in defining the coarseness of a bin, we introduce an imaginary bin with a zero index, such that its coarseness is zero, and its color set is empty. Then the coarseness of a bin with an index larger than zero is defined as

$$\alpha_i = max_{\{0 \le j < i \wedge (\chi(B_j) \cap \chi(B_i)) = 0)\}} \delta_j, \text{ for } i \ge 1.$$

Specifically, the coarseness of a bin is equal to the maximum deficiency, among all the bins that are ahead of the current bin and that do not share any color with the current bin. Intuitively, the size of each item in a bin must be larger than the coarseness of the bin. If a bin has a coarseness of zero, then either it is the first one or, most likely, every bin ahead of it shares at least one color with it.

Lemma 5.5: *Let a bin B_i with coarseness α_i be filled with items $b_1 \ge b_2 \ge \ldots \ge b_m$ and* $2/3 < \sum_{i=1}^{m} b_i < 3/4$. *Then there are at most* κ *bins with* $\sum_{i=1}^{m} W(b_i) < 1$ *and* $m \ge 3$. *If l is the number of bins with* $2/3 < \sum_{i=1}^{m} b_i < 2/3 + 1/18$, $\sum_{i=1}^{m} W(b_i) = 1 - \beta_i$, $\beta_i > 0$, *and* $m = 2$, *then* $\sum_{i=1}^{l} W(B_i) > l - 9\kappa/20$.

Proof: According to Lemma 5.1, among all bins to each of which $m \ge 3$ items are assigned, there are at most κ bins of them, each of which is no more than 3/4 full. For those bins with $m \ge 3$, there are at most κ bins with $2/3 < \sum_{i=1}^{m} b_i < 3/4$. Therefore, there are at most κ bins with $\sum_{i=1}^{m} W(b_i) < 1$.

Accordingly, we need only to focus our attention on the bins each of which is assigned two items, i.e., $m = 2$. Furthermore, $1/2 > b_1 \ge 1/3$ and $b_2 < 1/3$, since $2/3 < \sum_{i=1}^{2} b_i < 2/3 + 1/18$ and $\sum_{i=1}^{m} W(b_i) < 1$.

Claim 1: There are at most $(3\kappa)/2$ such bins that have a coarseness of zero.

Let us consider the worst case configuration of the FT-EDF-FF bin-packing where the maximum number of bins with zero coarseness is achieved. Note that for these bins, a bin with zero coarseness implies that all the bins ahead of it contain one of the two colors it contains. This is because each of these bins has a deficiency of at least 1 – (2/3 + 1/18).

Recall that for the first of these bins, it contains exactly two colors. For the bins that follows it, every one of them must contain at least one of its colors. Now we want to find out the maximum number of bins that can possibly satisfy this constraint. Let n be the number to be derived. Then it is apparent that $n < 2\kappa$, because the maximum number of items with the same color is κ.

Let c_1 and c_2 be the two colors in the first bin. Let i_1 be the number of bins that immediately follow the first bin and share the same color c_1 and i_2 be the number of bins that immediately follow the first bin and share the color c_2. If $i_1 = i_2$, then $n \le i_1 \le \kappa$. Let us assume that $i_1 > i_2$. Let $j > 0$ be the number of bins that immediately follow the i_1th bin and have one color c_2. Then $i_2 + j \le \kappa$, since the number of bins containing color c_2 must be no more than κ. Furthermore, $i_1 - i_2 + j \le \kappa$. This is because the $(i_1 - i_2)$ bins that immediately follow the first i_2 bins must share one color with the j bins that immediately follow the i_1th bin with the other color being c_2. This is illustrated in Figure 5.2.

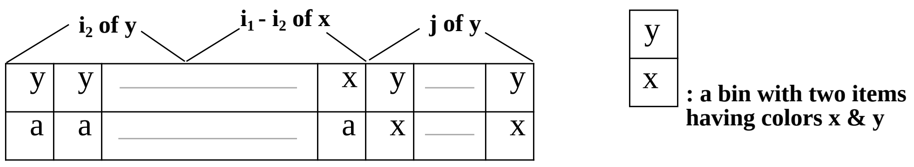


Figure 5.2: Worst Case Configuration of Zero Coarseness

Since $i_1 \le \kappa$, $i_1 - i_2 + j \le \kappa$, and $i_2 + j \le \kappa$, we conclude that $n \le i_1 + j \le (3\kappa)/2$.

Claim 2: $\sum_{i=1}^{2} W(b_i) \ge 1$ if $\sum_{i=1}^{2} b_i \ge 1 - \alpha_i$.

For any such bin with coarseness $\alpha_i > 0$, α_i must be larger than 1/3 – 1/18 (since $\sum_{i=1}^{2} b_i < 2/3 + 1/18$).

Let b_1 and b_2 be the two items assigned to a bin B_i and $b_1 \ge b_2$. Then $b_1 > \alpha_i \ge 1/3 - 1/18$ and $b_2 > \alpha_i \ge 1/3 - 1/18$, according to the definition of coarseness. If $\alpha_i \ge 1/3$, then $b_2 \ge 1/3$ and $\sum_{i=1}^{2} W(b_i) \ge 1/2 + 1/2 = 1$.

If $\alpha_i < 1/3$, then $b_1 \ge 1/3$, and $b_2 < 1/3$. Otherwise, $b_1 + b_2 < 2/3$, which contradicts the assumption that $\sum_{i=1}^{2} b_i > 2/3$. Then $\sum_{i=1}^{2} W(b_i) = 6b_1/5 + 1/10 + 9b_2/5 - 1/10 > 6\sum_{i=1}^{2} b_i/5 + 3b_2/5 > (6/5) \bullet (1 - \alpha_i) + 3\alpha_i/5 = 1 + 1/5 - 3\alpha_i/5 > 1$, since $b_2 > \alpha_i$ and $\alpha_i < 1/3$.

For future reference, if $\sum_{i=1}^{2} W(b_i) = 1 - \beta_i$ and $\beta_i > 0$, then we must have

$\sum_{i=1}^{2} b_i < 1 - \alpha_i$, $1/3 \le b_1 \le 1/2$, and $1/6 < b_2 < 1/3$. $\sum_{i=1}^{2} W(b_i) = 6b_1/5 + 1/10 + 9b_2/5 - 1/10 > 6b_1/5 + 9(2/3 - b_1)/5 = 6/5 - 3b_1/5 \ge 9/10$ since $b_1 \le 1/2$. In other words, $\beta_i \le 1/10$.

Claim 3: $\sum_{i=1}^{m} b_i \le 1 - \alpha_i - 5\beta_i/9$ if $\sum_{i=1}^{m} W(b_i) = 1 - \beta_i$ with $\beta_i > 0$.

To prove this claim, let b_1 and b_2 be the two items assigned to a bin B_i with $b_1 \ge b_2$. Suppose $\sum_{i=1}^{m} b_i = 1 - \alpha_i - \gamma$ with $\gamma > 0$. Then we can construct a bin filled with two items σ_1 and σ_2 such that $\sigma_1 + \sigma_2 = b_1 + b_2 + \gamma$, and $\sigma_1 \le 1/2$ and $\sigma_2 \le 1/2$. Then $W(\sigma_1) + W(\sigma_2) \ge 1$. Since the slope of the weighting function W in the range of (0, 1/2] does not exceed 9/5, therefore $W(\sigma_1) + W(\sigma_2) \le \sum_{i=1}^{m} W(b_i) + 9\gamma/5$. In other words, $1 \le 1 - \beta_i + 9\gamma/5$. $5\beta_i/9 \le \gamma$. $\sum_{i=1}^{m} b_i \le 1 - \alpha_i - 5\beta_i/9$.

Suppose that in the completed FT-EDF-FF packing, let l be the number of bins with $\sum_{i=1}^{m} W(b_i) < 1$. Among the l bins, let B_1', B_2', …, B_h' be the bins that have non-zero coarseness. If we group these bins according to $\chi(B_i') \cap \chi(B_j') = 0$ for any pair of bins in a group, then there are at most $(3\kappa)/2$ different groups, according to Claim 1. Within each group, let n be the number of bins in such group. Then $\alpha_i < \alpha_j$ if $i < j$. Since $\alpha_i \ge \alpha_{i-1} + 5\beta_{i-1}/9$, for $1 < i \le n$, then $\sum_{i=1}^{n-1} \beta_i \le 9/5 \bullet \sum_{i=2}^{n} (\alpha_i - \alpha_{i-1}) = 9/5 \bullet (\alpha_n - \alpha_1) \le 9/5 \bullet (2/3 + 1/18 - 2/3) = 1/10$. Since $\beta_n \le 1/10$, we have $\sum_{i=1}^{n} \beta_i \le 2/10$. Therefore $\sum_{i=1}^{h} \beta_i \le (3\kappa)/2 \bullet 2/10 = 3\kappa/10$.

For the $(3\kappa)/2$ bins with zero coarseness, suppose that there are $g \le (3\kappa)/2$ of them, each with $\sum_{i=1}^{m} W(b_i) = 1 - \beta_i$ where $\beta_i > 0$. According to the reasoning above, $\sum_{i=1}^{g} \beta_i \le (3\kappa)/2 \bullet 1/10 = 3\kappa/20$.

Therefore, $\sum_{i=1}^{l} \beta_i \le 3\kappa/10 + 3\kappa/20 = 9\kappa/20$, where $l = h + g$.

$\sum_{i=1}^{l} W(B_i) > l - 9\kappa/20$. ■

Lemma 5.6: *Among all the bins filled to the level of* $2/3 + 1/18 \le \sum_{i=1}^{m} b_i < 3/4$, *there are at most* κ *of them with* $\sum_{i=1}^{m} W(b_i) < 1$ *and* $m \ge 3$.

Proof: Let a bin B_i be filled with items $b_1 \ge b_2 \ge \ldots \ge b_m$ and $2/3 + 1/18 \le \sum_{i=1}^{m} b_i \le 3/4$.

If $m = 1$, then $b_1 \geq 2/3 + 1/18 > 1/2$. $\sum_{i=1}^{m} W(b_i) \geq 1$.

If $m = 2$, there are three cases to consider:

(1) If $b_1 > 1/2$, then $\sum_{i=1}^{m} W(b_i) \geq 1$.

(2) If $1/3 < b_1 \leq 1/2$ and $1/3 < b_2 \leq 1/2$, then $\sum_{i=1}^{m} W(b_i) \geq 1/2 + 1/2 = 1$.

(3) If $1/3 < b_1 \leq 1/2$ and $1/6 < b_2 \leq 1/3$, then $\sum_{i=1}^{m} W(b_i) \geq 6b_1/5 + 1/10 + 9(2/3 + 1/18 - b_1)/5 - 1/10 = 13/10 - 3b_1/5 \geq 1$.

Obviously, the bins with $\sum_{i=1}^{m} W(b_i) < 1$ must be assigned at least three items, i.e., $m \geq 3$. According to Lemma 5.1, among all bins to each of which $m \geq 3$ items are assigned, there are at most κ bins of them, each of which is no more than 3/4 full. Therefore, there are at most κ bins with $\sum_{i=1}^{m} W(b_i) < 1$. ■

Lemma 5.7: *Among all the bins filled to the level* $3/4 \leq \sum_{i=1}^{m} b_i < 4/5$, *there are at most* κ *of them with* $\sum_{i=1}^{m} W(b_i) < 1$ *and* $m \geq 4$.

Proof: Let a bin B_i be filled with items $b_1 \geq b_2 \geq \ldots \geq b_m$ and $3/4 \leq \sum_{i=1}^{m} b_i < 4/5$.

If m is equal to 1 and 2, then we can prove, similarly to the proof of Lemma 5.6, that $\sum_{i=1}^{m} W(b_i) \geq 1$.

If $m = 3$, there are seven cases to consider:

(1) If $b_1 > 1/2$, then $\sum_{i=1}^{m} W(b_i) \geq 1$.

(2) If $1/3 < b_1 \leq 1/2$ and $1/3 < b_2 \leq 1/2$, then $\sum_{i=1}^{m} W(b_i) \geq 1/2 + 1/2 = 1$.

(3) If $1/3 < b_1 \leq 1/2$, $1/6 < b_2 \leq 1/3$, and $1/6 < b_3 \leq 1/3$, then $\sum_{i=1}^{m} W(b_i) = 6b_1/5 + 1/10 + 9b_2/5 - 1/10 + 9b_3/5 - 1/10 \geq 6[3/4 - (b_2 + b_3)]/5 + 9(b_2 + b_3)/5 - 1/10 = 3(b_2 + b_3)/5 + 4/5 > 1$, since $b_2 + b_3 > 1/3$.

(4) If $1/3 < b_1 \leq 1/2$, $1/6 < b_2 \leq 1/3$, and $b_3 \leq 1/6$, then $\sum_{i=1}^{m} W(b_i) = 6b_1/5 + 1/10 + 9b_2/5 - 1/10 + 6b_3/5 \geq 9b_2/5 + 6(3/4 - b_2)/5 = 3b_2/5 + 9/10 > 1$.

(5) If $1/3 < b_1 \leq 1/2$ and $b_2 \leq 1/6$, then $\sum_{i=1}^{m} W(b_i) = 6b_1/5 + 1/10 + 6b_2/5 + 6b_3/5 = 6(\sum_{i=1}^{m} b_i)/5 + 1/10 \geq (6/5) \cdot (3/4) + 1/10 = 1$.

(6) If $1/6 < b_1 \leq 1/3$, $1/6 < b_2 \leq 1/3$, and $1/6 < b_3 \leq 1/3$, then $\sum_{i=1}^{m} W(b_i) = 9(\sum_{i=1}^{m} b_i)/5 - 3/10 \geq (9/5) \cdot (3/4) - 3/10 > 1$.

(7) If $1/6 < b_1 \le 1/3$, $1/6 < b_2 \le 1/3$, and $b_3 \le 1/6$, then $\sum_{i=1}^{m} W(b_i) = 9b_1/5 - 1/10 + 9b_2/5 - 1/10 + 6b_3/5 \ge 9(3/4 - b_3)/5 + 6b_3/5 - 2/10 > 23/20 - 3b_3/5 > 1$.

Obviously, the bins with $\sum_{i=1}^{m} W(b_i) < 1$ must be assigned at least four items, i.e., $m \ge 4$. According to Lemma 5.1, among all bins to each of which $m \ge 4$ items are assigned, there are at most κ bins of them, each of which is no more than 4/5 full. Therefore, there are at most κ bins with $\sum_{i=1}^{m} W(b_i) < 1$. ■

Lemma 5.8: *Among all the bins filled to the level $4/5 \le \sum_{i=1}^{m} b_i < 5/6$, there are at most κ of them with $\sum_{i=1}^{m} W(b_i) < 1$ and $m \ge 5$.*

Proof: Let a bin B_j be filled with items $b_1 \ge b_2 \ge \ldots \ge b_m$, and $4/5 \le \sum_{i=1}^{m} b_i < 5/6$.

If m is equal to 1, 2, and 3, then we can prove, similarly to the proof of Lemma 5.7, that $\sum_{i=1}^{m} W(b_i) \ge 1$.

If $m = 4$, there are eight cases to consider:

(1) If $b_1 > 1/2$, then $\sum_{i=1}^{m} W(b_i) \ge 1$.

(2) If $1/3 < b_1 \le 1/2$ and $1/3 < b_2 \le 1/2$, then $\sum_{i=1}^{m} W(b_i) \ge 1/2 + 1/2 = 1$.

(3) If $1/3 < b_1 \le 1/2$, $1/6 < b_2 \le 1/3$, and $1/6 < b_3 \le 1/3$, then $\sum_{i=1}^{m} W(b_i) \ge 6b_1/5 + 1/10 + 9b_2/5 - 1/10 + 9b_3/5 - 1/10 \ge 6[4/5 - (b_2 + b_3)]/5 + 9(b_2 + b_3)/5 - 1/10 = 3(b_2 + b_3)/5 + 43/50 > 1$, since $b_2 + b_3 > 1/3$.

(4) If $1/3 < b_1 \le 1/2$, $1/6 < b_2 \le 1/3$, and $b_3 \le 1/6$, then $\sum_{i=1}^{m} W(b_i) \ge 6b_1/5 + 1/10 + 9b_2/5 - 1/10 + 6(b_3 + b_4)/5 \ge 9b_2/5 + 6(4/5 - b_2)/5 = 3b_2/5 + 24/25 > 1$.

(5) If $1/3 < b_1 \le 1/2$ and $b_2 \le 1/6$, then $\sum_{i=1}^{m} W(b_i) = 6b_1/5 + 1/10 + 6(b_2 + b_3 + b_4)/5 = 6(\sum_{i=1}^{m} b_i)/5 + 1/10 \ge (6/5) \cdot (4/5) + 1/10 > 1$.

(6) If $1/6 < b_1 \le 1/3$, $1/6 < b_2 \le 1/3$, $1/6 < b_3 \le 1/3$, and $1/6 < b_4 \le 1/3$, then $\sum_{i=1}^{m} W(b_i) = 9(\sum_{i=1}^{m} b_i)/5 - 4/10 \ge (9/5) \cdot (4/5) - 4/10 > 1$.

(7) If $1/6 < b_1 \le 1/3$, $1/6 < b_2 \le 1/3$ and $b_3 \le 1/6$, then $\sum_{i=1}^{m} W(b_i) = 9b_1/5 - 1/10 + 9b_2/5 - 1/10 + 6(b_3 + b_4)/5 \ge 9[4/5 - (b_3 + b_4)]/5 + 6(b_3 + b_4)/5 - 2/10 > 31/25 - 3(b_3 + b_4)/5 > 1$.

(8) If $1/6 < b_1 \le 1/3$ and $b_2 \le 1/6$, then $\sum_{i=1}^{m} W(b_i) = 9b_1/5 - 1/10 + 6(b_2 + b_3 +$

$b_4)/5 \geq 9[4/5 - (b_2 + b_3 + b_4)]/5 + 6(b_2 + b_3 + b_4)/5 - 1/10 > 67/50 - 3(b_2 + b_3 + b_4)/5 > 1$, since $b_2 + b_3 + b_4 \leq 1/2$.

Obviously, the bins with $\sum_{i=1}^{m} W(b_i) < 1$ must be assigned at least five items, i.e., $m \geq 5$. According to Lemma 5.1, among all bins to each of which $m \geq 5$ items are assigned, there are at most κ of them, each of which is no more than 5/6 full. Therefore, there are at most κ bins with $\sum_{i=1}^{m} W(b_i) < 1$. ■

Lemma 5.9: *Let a bin B_i be filled with items $b_1 \geq b_2 \geq \ldots \geq b_m$. If $\sum_{i=1}^{m} b_i \geq 5/6$, then $\sum_{i=1}^{m} W(b_i) \geq 1$.*

Proof: Since $W(\beta) / \beta \geq 6/5$ in the range of $0 \leq \beta \leq 1/2$ and $W(\beta) = 1$ when $\beta > 1/2$, we have $\sum_{i=1}^{m} W(b_i) \geq 5/6 \bullet 6/5 = 1$. ■

Proof of Theorem 5.1: Suppose that in the final FT-EDF-FF-packing, there are m bins $B_1, B_2, \ldots, B_m$, each of which receives at least one item, and $\sum_j W(B_j) < 1$. Let $\sum_j W(B_j) = 1 - \beta_j$, with $\beta_j > 0$ for $1 \leq j \leq m$.

Since our goal is to prove that $1.7L^* \geq W \geq$ FT-EDF-FF$(L) - \sum_{i=1}^{m} \beta_i$, we need to bound the quantity $\sum_{i=1}^{m} \beta_i$.

According to Lemma 5.8, if $\sum_{i=1}^{m} b_i \in [4/5, 5/6)$, there are at most κ bins with $m \geq 5$ and $\sum_{i=1}^{m} W(b_i) < 1$. Let l be the number of bins with $\sum_j W(B_j) = 1 - \beta_j$ and $\beta_j > 0$ for $1 \leq l \leq \kappa$. $\sum_{i=1}^{l} \beta_i \leq \kappa(1 - 4/5 \bullet 6/5) = \kappa/25$.

According to Lemma 5.7, if $\sum_{i=1}^{m} b_i \in [3/4, 4/5)$, there are at most κ bins with $m \geq 4$ and $\sum_{i=1}^{m} W(b_i) < 1$. Let l be the number of bins with $\sum_j W(B_j) = 1 - \beta_j$ and $\beta_j > 0$ for $1 \leq l \leq \kappa$. $\sum_{i=1}^{l} \beta_i \leq \kappa(1 - 3/4 \bullet 6/5) = \kappa/10$.

According to Lemma 5.5 and Lemma 5.6, if $\sum_{i=1}^{m} b_i \in [2/3, 3/4)$, then there are at most κ bins with $m \geq 3$ and $\sum_{i=1}^{m} W(b_i) < 1$. Let l be the number of bins with $\sum_j W(B_j) = 1 - \beta_j$ and $\beta_j > 0$ for $1 \leq l \leq \kappa$. $\sum_{i=1}^{l} \beta_i \leq \kappa(1 - 2/3 \bullet 6/5) = \kappa/5$.

If $\sum_{i=1}^{m} b_i \in (2/3, 2/3 + 1/18)$, then let l be the number of bins with $m = 2$ and $\sum_{i=1}^{2} W(b_i) < 1$. Let l be the number of such bins with $\sum_j W(B_j) = 1 - \beta_j$ and $\beta_j > 0$. According to Lemma 5.5, $\sum_{i=1}^{l} \beta_i \leq 9\kappa/20$.

According to Lemma 5.4, if $\sum_{i=1}^{m} b_i \in (1/2, 2/3]$, then there are at most κ bins with $m \geq 2$ and $\sum_{i=1}^{m} W(b_i) < 1$. Let l be the number of bins with $\sum_j W(B_j) = 1 - \beta_j$ and $\beta_j > 0$ for $1 \leq l \leq \kappa$. $\sum_{i=1}^{l} \beta_i \leq \kappa(1 - 1/2 \bullet 6/5) = 2\kappa/5$.

According to Lemma 5.3, if $\sum_{i=1}^{m} b_i \in (0, 1/2]$, then there are at most κ bins with $m \geq 1$ and $\sum_{i=1}^{m} W(b_i) < 1$. Let l be the number of bins with $\sum_j W(B_j) = 1 - \beta_j$ and $\beta_j > 0$ for $1 \leq l \leq \kappa$. Then $\sum_{i=1}^{l} \beta_i \leq \kappa$.

Therefore, $\sum_{i=1}^{m} \beta_i \leq \kappa(1 + 2/5 + 9/20 + 1/5 + 1/10 + 1/25) = 2.19\kappa$.

In summary, FT-EDF-FF*(L)* $\leq 1.7L^* + 2.19\kappa$. ■

We conjecture that the constant can be further lowered from 2.19 to 1, if a better weighting function can be found. When $\kappa = 1$, the problem becomes the well-known classical bin-packing problem. Since the ratio 1.7 is not affected by the value of κ, our result therefore subsumes the previous known result [30]. Also, when $\kappa = 1$, examples that achieve the bound of 1.7 has been given in [30]. Since the term 2.19κ is a constant, it disappears when the optimal number of bins L^* approaches infinity. Therefore, we conclude that the bound is asymptotically tight.

5.2. The Average Case Performance Evaluation

In order to gain some insight into the average case behavior of the new algorithm, we use the same approach as we did in Section 3.9 to evaluate its performance.

Our simulation studies consist of two steps: (1) generate task sets with random distributions; (2) run the task sets through the algorithms to produce results. For reasons soon to be made clear, we use two methodologies to generate task sets with random characteristics. The output parameter for each algorithm is the number of processors used to accommodate a given set of tasks.

In the first study, the input data of all parameters for a task set are generated according to uniform distribution, except the number of tasks a task set has. The periods of tasks are generated in the range of $1 \leq T_i \leq 500$. The number of versions for each task is uniformly distributed in the range of $1 \le \kappa_i \le 5$. The computation time of each version is in the range of $1 \le C_{i,j} \le \alpha T_i$, where α is the maximum allowable utilization of any version, i.e., $\alpha = max_{i,j}(C_{i,j}/T_i)$.

The result is plotted in Figures 5.3 and 5.4 with two values of α. Each data point depicts the average value of 10 independently generated task sets with identical parameters. In order to make comparisons, we also ran the same data through Algorithm 1, and the results are plotted in the same figure. Observe that Algorithm 1 can be readily modified to allocate multiple-version periodic tasks for EDF scheduling by replacing the RM condition with the EDF condition. We hereafter refer to the latter algorithm as *EDF-Algorithm 1* and the former as *RM-Algorithm 1*. The performance of both allocation schemes under EDF condition is consistently better than that under RM condition. This is expected since the total utilization for each processor is bounded by $n\left(2^{1/n} - 1\right)$ under RM condition, and 1 under the EDF condition. For FT-RM-FF and FT-EDF-FF, their performance is consistently better than that of Algorithm 1 under RM and EDF conditions, respectively.

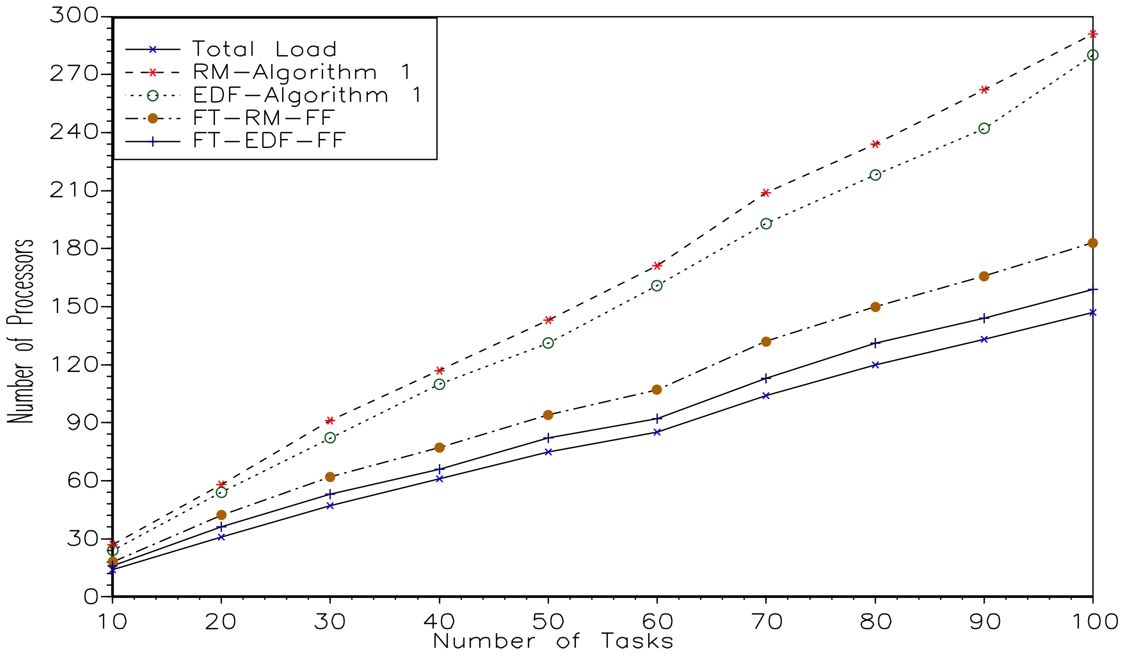


Figure 5.3: Performance Comparison of the Four Algorithm (α = 1.0)

The total utilization (load) of a task set is given by $\sum_{i=1}^{n}\left(\sum_{j=1}^{\kappa_i} C_{i,j}\right)/T_i$, which

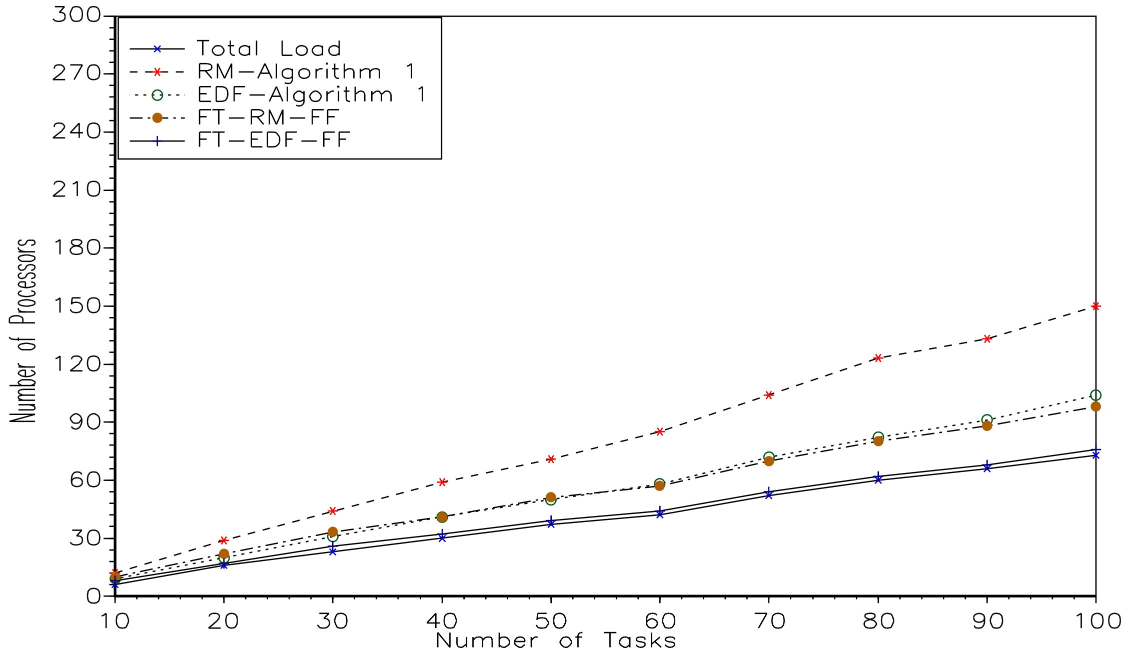


Figure 5.4: Performance Comparison of the Four Algorithm (α = 0.5)

can be considered as the minimum number of processors needed to execute the task set. It is a lower bound on the number of processors to be computed. For RM-Algorithm 1, the number of processors used to execute a task set is more than twice its total utilization in some cases. This comparison may be overly pessimistic, since the optimal number of processors may differ from the total utilization greatly in some cases, and little in other cases. Therefore, using the total utilization of the task set as a baseline performance may not capture the whole picture. The ideal solution would be to find the optimal number of processors for any given task set. However, this is usually deemed to be likely requiring exponential time with respect to the number of tasks using existing techniques, since the scheduling problem is NP-complete.

This observation leads us to the employment of a new methodology. Under this methodology, a task set is generated randomly with the constraint that in the optimal scheduling, it fully utilizes a known number of processors, using either the RM or EDF algorithm. In other words, given *m* processors, and the average number of task versions to be run on each processor, we generate a set of tasks that fully utilizes *m* processors, and at the same time, satisfies the timing and fault-tolerant constraints of the tasks. This is accomplished in the following steps:

(1) *M* arrays of random numbers are generated. The sizes of the arrays are uniformly generated, with a mean value corresponding to the average number of versions to be run on a processor.

(2) Each item in an array is divided by the sum of all items in its array to obtain a number between 0 and 1, which corresponds to the utilization of a version.

(3) For each task, the numbers of the versions *v* it has are generated, confirming to uniform distribution, with a mean corresponding to the average number of versions per task. Then *v* numbers are randomly selected from the *m* arrays of numbers to be the utilizations of the versions. This process is repeated until all the items in the *m* arrays are picked.

Using this methodology to generate task sets, the performance of the four algorithms is plotted in Figure 5.5. Each data point in this figure (and subsequent figures) is the

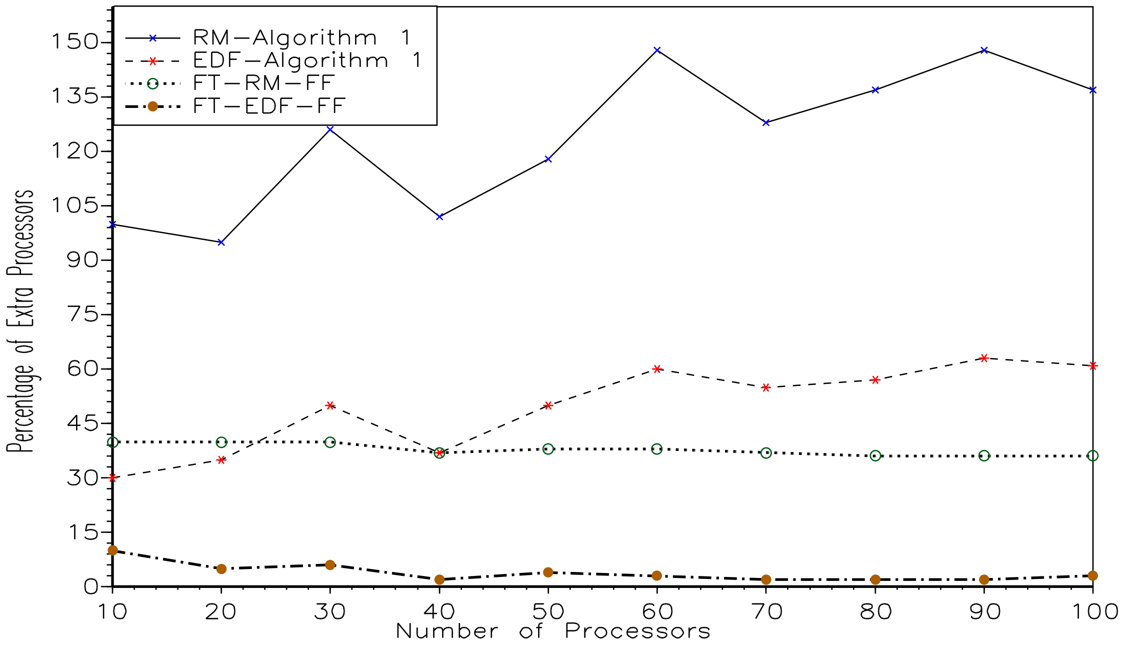


Figure 5.5: Performance Comparison of the Four Algorithm (α = 1.0)

average value of 10 independently generated task sets with identical parameter. On the x-axis, the number of processors is the optimal number N_0 of processors required to execute a task set. On the y-axis, the extra percentage of processors is defined as $(N_A - N_0) / N_0$, where N_A is the number of processors required by a heuristic algorithm A to allocate the same task set. Though the performance of the algorithms is consistent as shown in Figures 5.3 and 5.4, we have a better idea of what percentage of extra processors is needed for each algorithm for a given task set.

For some heuristics, their performance is highly sensitive to the order of input data, and hence they are referred to as having unstable performance. We are also interested in the stability of our heuristics. We consider two options: (1) For each task, its versions are assigned to processors according to the largest computation time first strategy. (2) The tasks are assigned to processors in the order of decreasing utilization. For this to work, the task set must be sorted first. Apparently, there are four ways to arrange the input data:

US: The task set is UnSorted as it is randomly generated.

VD: Versions of each task are sorted in Decreasing order of computation time.

TD: Tasks are sorted in the order of Decreasing utilization (TD).

VD-TD: Versions of each task are sorted in Decreasing order of computation time (VD), and Tasks are sorted in the order of Decreasing utilization (TD). Note that the utilization of a task is the sum of the utilizations of all its versions

For the same set of inputs that produced the results shown in Figure 5.5, the performance of RM-Algorithm 1 is shown in Figure 5.6. The improvement of performance is quite significant, when tasks are assigned to processors in the order of decreasing task utilization. What is a little bit surprising is that the order in which the versions of a task is assigned to processors does not affect the performance very much.

For the same set of inputs that produced the results shown in Figure 5.5, the performance of FT-RM-FF and FT-EDF-FF is shown in Figures 5.7 and 5.8. Note that both FT-RM-FF and FT-EDF-FF are not much sensitive to the order in which tasks are assigned to

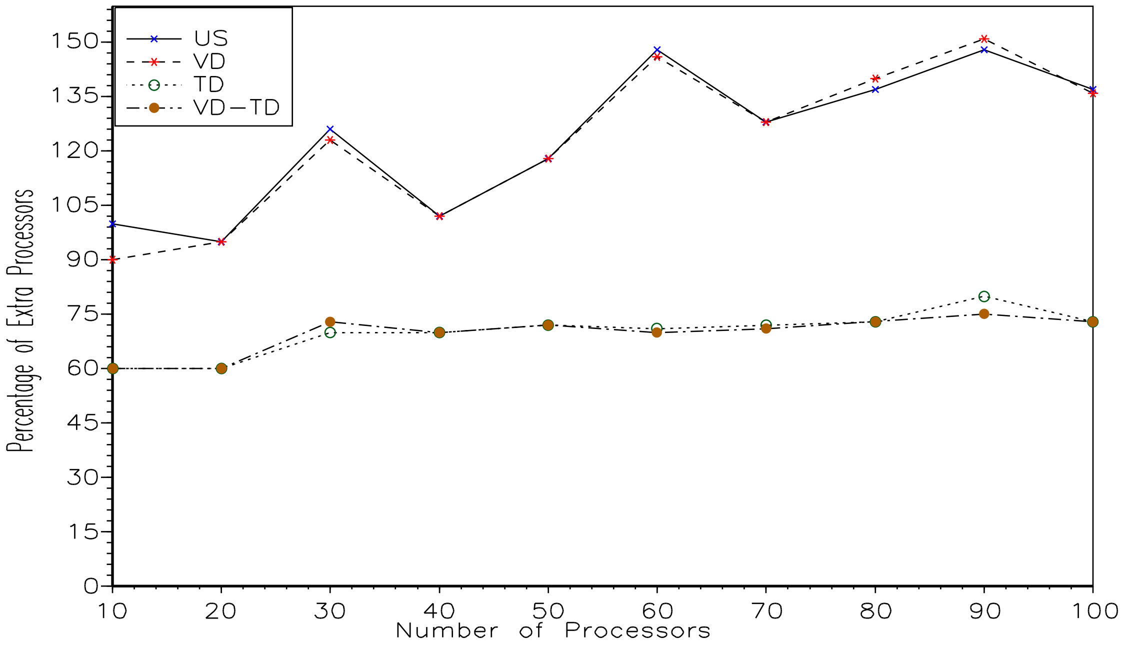


Figure 5.6: Performance of RM-Algorithm 1

processors and the order in which versions of a task are assigned to processors.

In summary, our simulation studies show that FT-RM-FF and FT-EDF-FF are insensitive to the order of assigning tasks to processors and to the order of assigning versions of a task to processors. FT-RM-FF and FT-EDF-FF outperform RM-Algorithm 1 and EDF-Algorithm 1. The performance of FT-RM-FF is quite good. The 40% extra processors is almost inevitable because of the schedulability condition $n\left(2^{1/n}-1\right)$. The performance of FT-EDF-FF is near-optimal, since it usually requires less than 10% extra processors. The order of assigning tasks to processors affects the performance of Algorithm 1 significantly. The superiority of FT-RM-FF and FT-EDF-FF is reflected not only in their out-performance over Algorithm 1, but most importantly, they can be used as on-line algorithms.

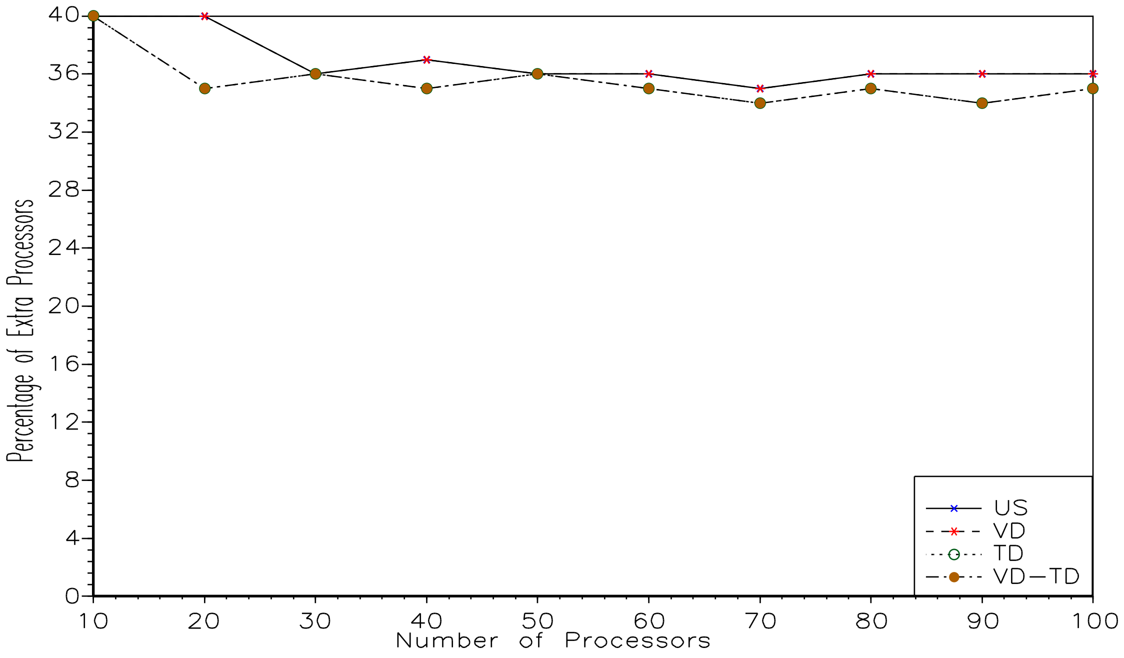


Figure 5.7: Performance of FT-RM-FF with Sorted Input

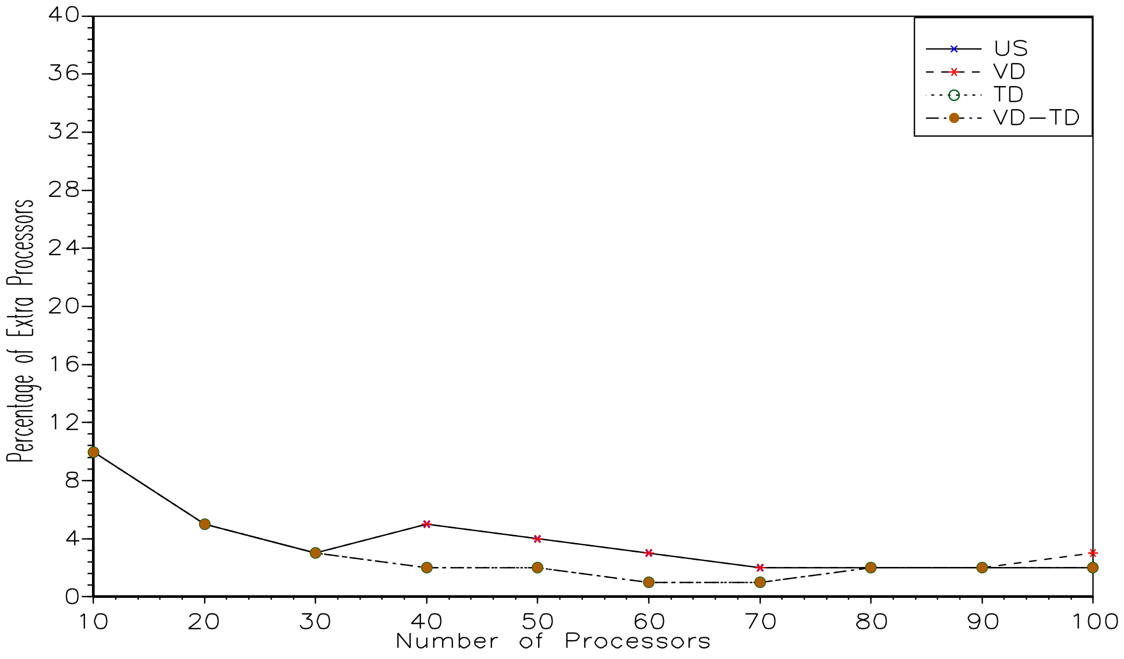


Figure 5.8: Performance of FT-EDF-FF with Sorted Input

Chapter 6 Non-preemptive Scheduling of Periodic Tasks for Fault-Tolerance

“Don’t find fault, find a remedy.”
-- Henry Ford, 1863-1947

In this chapter, we study the problem of scheduling a set of periodic tasks non-preemptively on a multiprocessor for fault-tolerance. The scheduling problem is formulated in the similar manner as its preemptive counterpart. However, the problem of non-preemptively scheduling a set of periodic tasks on a multiprocessor is a much difficult one. Jeffay, Stanat and Martel [28] have shown that a set of periodic tasks may not be schedulable non-preemptively on a single processor, even if its total CPU utilization is very small, i.e., close to zero. It is currently not at all clear whether a reasonably efficient heuristic exists for scheduling a set of periodic tasks non-preemptively on a single processor system. The fact that multiple processors are involved further complicates the scheduling problem, let alone the additional requirement of guaranteeing task deadlines even in the presence of processor failures. In this thesis, we will focus on a special case of the scheduling problem.

In the following, we consider the tolerance of processor failures using a simple software redundance scheme. This is a special case of the TFT scheduling problem. The tasks are independent and non-preemptive. Each task has a primary copy and a backup copy, and the scheduling goal is to achieve 1-TFT for processor failure, i.e., the tolerance of one arbitrary processor failure. This case of the TFT problem is chosen to be studied, because it is the simplest one.

The task redundancy scheme specified in the above case actually corresponds to the primary-backup copy approach or recovery block approach. Primary-backup copy approach requires the multiple implementation of a specification [66]. The first implementation is called the primary copy, and the other implementations are called the backup copies. The primary and if necessary, the backup copies, execute in series. If the primary copy fails, one of the backup copies is switched in to perform the computation again. This process is repeated until that either correct results are produced or all the backup copies are exhausted. Here we consider a special case of the primary-backup copy approach where each task has one backup copy only. The following Lemmas [51] guarantee that having one backup copy for each task is sufficient for the tolerance of one arbitrary processor failure.

Lemma 6.1: *In order to tolerate one or more processor failures and guarantee that the deadline of a task is met using the primary-backup copy approach, the computation time of the task must be less than or equal to half of the period of the task, assuming that the deadline coincides with the period.*

Lemma 6.2: *One arbitrary processor failure is tolerated and the deadlines of tasks are met, if and only if the primary copy and the backup copy of each task is scheduled on two different processors and there is no overlapping in time between their executions.*

An obvious implication of Lemma 6.1 is that for each task, if the computation time of the task is larger than half of its period, it is impossible to find a schedule which is 1-TFT. This is due to the observation that if the primary copy fails at the very end, there will not be enough time left to complete a backup copy, assuming that the backup copy has the same computation time requirement as the primary copy. This fact is used implicitly in many situations throughout this chapter.

In scheduling the backup copies, we have the options of allowing them to be overlapped or forbidding them from overlapping. Here we first consider the case where the backup copies are not allowed to be overlapped with each other, and then the case where the backup copies are allowed to be overlapped.

6.1. Non-overlapping of Backup Copies

What we mean by disallowing them to be overlapped is that backup copies of the tasks whose primary copies are scheduled on different processors are not allowed to overlap in time of their executions on a processor. For obvious reasons, backup copies of the tasks whose primary copies are scheduled on the same processor must not be scheduled to overlap in time of their executions on a processor. When the given number of processors is two, there apparently exists an optimal algorithm to schedule a set of tasks having a common deadline so as to tolerate one arbitrary processor failure. However, for more than two processors, the scheduling problem is NP-complete, even when the tasks have the same deadline.

6.1.1. Complexity of the Scheduling Problem

We first define the problem and then prove that it is NP-complete.

Task Sequencing Using Primary-Backup with a Common Deadline.

Instance: Set Σ of tasks, number of processors $m \geq 3$, for each task $t \in \Sigma$, one primary copy $P(t)$ and one backup copy $G(t)$, a length $l(t) \in Z^{+}$ (the set of natural numbers), a common release time $r \in Z^{+}$, a common deadline $d(t) = D \in Z^{+}$, and $l(P(t)) = l(G(t)) = l(t)$. No overlapping of backup copies is allowed.

Question: Is there an *m*-processor schedule σ for Σ that is 1-TFT, i.e., for each task $t \in \Sigma$, $\sigma_i(P(t)) + l(P(t)) \leq \sigma_j(G(t))$, and $\sigma_i(G(t)) + l(G(t)) \leq D$, where $i \neq j$, *i* and *j* designate the index of processors.

Theorem 6.1: *Task Sequencing Using Primary-Backup with a Common Deadline is NP-complete.*

Proof: It is sufficient to prove that this scheduling problem is NP-complete even in the case of $m = 3$. It is easy to verify that this problem is in NP. We next transform the PARTITION problem, an NP-complete problem, to the scheduling problem.

The PARTITION problem [23] is stated as follows: given a finite set A and a “size”

$s(a) \in Z^+$ for each $a \in A$, is there a subset $A' \subseteq A$ such that $\sum_{a \in A'} s(a) = \sum_{a \in A - A'} s(a)$?

Given an instance of A = $\{a_1, a_2, \ldots, a_n\}$ of the PARTITION problem, we construct a task set Σ using the primary-backup copy approach to run on three processors for the tolerance of a single arbitrary processor failure, such that Σ can be scheduled if and only if there is a solution to the PARTITION problem. Σ consists of $n+1$ tasks as follows:

$$r(t) = 0, l(t) = a_t, d(t) = 2B,$$

where $t \in \{\tau_1, \tau_2, \ldots, \tau_n\}$, $\sum_{1 \le i \le n} a_i = 2B$ (this can be assumed without loss of generality); and one other task β:

$$r(\beta) = 0, l(\beta) = B, d(\beta) = 2B.$$

It is easy to see that this transformation can be constructed in polynomial time. What we need to show is that the set *A* can be partitioned into two sets S_1 and S_2 such that $\sum_{a \in S_1} s(a) = \sum_{a \in S_2} s(a)$ and $S_1 + S_2 = A$, if and only if the task set can be scheduled.

First, suppose that A can be partitioned into two sets S_1 and S_2 such that $\sum_{a \in S_1} s(a) = \sum_{a \in S_2} s(a) = B$ and $S_1 + S_2 = A$. Then we schedule, for each $a \in S_1$, the primary copy of the task α with $l(a) = a$ on processor 2 anywhere between time interval [0, *B*), and its backup copy on processor 3 anywhere between time interval [*B*, 2*B*). For each task $a \in S_2$ with $l(a) = a$, the primary copy of task α is scheduled on processor 3 anywhere between time interval [0, *B*) and its backup copy on processor 1 anywhere between time interval [*B*, 2*B*). Therefore, the 2*n* copies of the *n* tasks can be scheduled on processors satisfying the condition set in Lemma 6.2. For task β, its primary copy is scheduled on processor 1 during time period [0, *B*), and its backup copy is scheduled on processor 2 between time period [*B*, 2*B*), as shown in Figure 6.1. Thus, the task set Σ can be scheduled on three processors such that the schedule is 1-TFT.

Conversely, if the task set Σ is scheduled on three processors such that the schedule is 1-TFT, we claim that for all tasks scheduled between the time interval [0, *B*) on processor 2, the sum of the tasks' lengths is *B*, i.e., $\sum_{a \in S_1} s(a) = B$. To be able to tolerate one arbi-

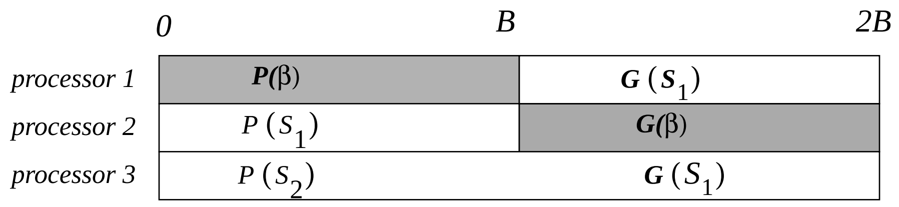


Figure 6.1: Mapping from PARTITION to Task Sequencing

trary processor failure, the primary copy of a task and its backup copy must be scheduled on two different processors and their execution time must not be overlapped. This later requirement is guaranteed by the primary-backup copy approach. Since the common deadline is 2*B* and the total task execution time is $2(2B + B) = 6B$, any 1-TFT schedule should have no idle time during the time interval [0, 2*B*) on all three processors. Therefore, any 1-TFT schedule must be equivalent to the schedule shown in Figure 6.2, if processors are properly renamed and the primary copies are moved in front of all the backup copies for each processor. Shuffling the primary copies in front of all the backup copies will not violate any scheduling constraint, since primary copies can start earlier than scheduled and backup copies can start later than scheduled, as long as the release time and the deadline constraints are not violated. For processor 3, exactly one copy, either primary or backup, of any task among the *n* tasks must be scheduled on it. This is because any 1-TFT schedule for the three processor requires that no idle time exists on any processor, and the primary copy of a task and its backup copy must never be scheduled on the same processor. Therefore, we let all the tasks scheduled on processor 2 between time interval [0, *B*) be the set S_1, and the tasks on processor 1 between time interval [*B*, 2*B*) be the set S_2. We then have $\sum_{a \in S_1} s(a) = \sum_{a \in S_2} s(a)$ and $S_1 + S_2 = A$. We have solved the PARTITION problem. The scheduling problem is therefore NP-complete. ■

6.1.2. A *1-Timely-Fault-Tolerant* Scheduling Algorithm

Since the scheduling problem is NP-complete, a heuristic scheduling algorithm is presented in this section to obtain approximate solution.

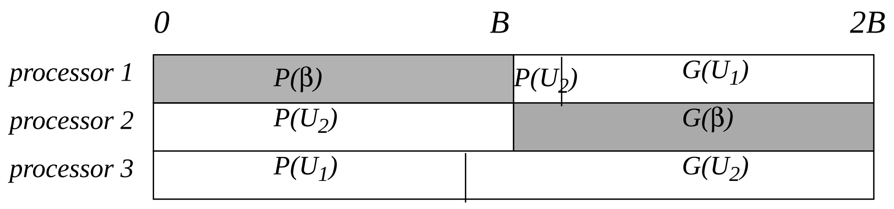


Figure 6.2: Mapping from Task Sequencing to PARTITION

In scheduling a set of tasks on *m* processors, the algorithm must be designed to minimize the schedule length on each processor such that the task set can be successfully scheduled, and at the meantime, to prevent the overlapping of the primary copy of a task and its backup copy. This scheduling problem, at a first glance, seems very much to resemble the scheduling problem of minimizing the makespan of a schedule in a multiprocessor system. Since the scheduling to minimize the makespan of a schedule is NP-complete, several scheduling heuristics have been developed, among which LPT [25] and MULTIFIT [14] are notable ones. However, there are two key issues that set this scheduling problem apart from the one to minimize the makespan: the requirement of scheduling primary copies as well as backup copies, and the requirement that the primary copy of a task cannot overlap its backup copy, and backup copies of different tasks cannot overlap each other in execution either. The MULTIFIT algorithm, though out-performing LPT in the worst cases, is not easily adapted to solve the 1-TFT scheduling problem. The LPT algorithm is therefore adopted here to serve as the base algorithm upon which a scheduling heuristic is developed.

The algorithm starts by first scheduling the primary copies on the *m* processors using the LPT algorithm. It then schedules the backup copies, by following several rules described below, such that the primary copy of a task and its backup copy are scheduled on different processors, and the backup copies of those tasks, whose primary copies are scheduled on a processor, are also scheduled on one processor. The algorithm is given in Figure 6.3. Note that *D* is the common deadline of the tasks.

NOV (Input: Task Set Σ, *m*, *1-TFT*; Output: *success, schedule*)

```
(1) Sort the tasks in order of non-increasing computation times and
    rename them $T_1, T_2, \ldots, T_n$. Compute $\Omega = \sum_{i=1}^{n} l(T_i)$. If
    $\Omega > (mD)/2$ or $l(T_1) > D/2$, then report that the task set can-
    not be scheduled on m processors by this algorithm such that a
    1-TFT schedule can be produced. Otherwise, go to Step 2.
(2) Apply the LPT algorithm to schedule the task set on m processors.
(3) Sort the primary schedules for the m processors in order of non-
    increasing schedule lengths. Duplicate the primary schedules to
    form m backup schedules and append them at the end of the pri-
    mary schedules (Figure 6.4a).
(4) Swap the backup schedules according to the swapping rules
    defined below (Figure 6.4b). Shift the backup schedules to
    obtain the mixed schedules according to the shifting rules
    defined below (Figure 6.4c).
(5) Find the maximum length among the mixed schedules and compare
    it to D. If it is longer than D, the task set cannot be sched-
    uled. Otherwise, the mixed schedules generated in Step 4 are
    the schedules which are 1-TFT as a whole.
```

Figure 6.3: Algorithm NOV

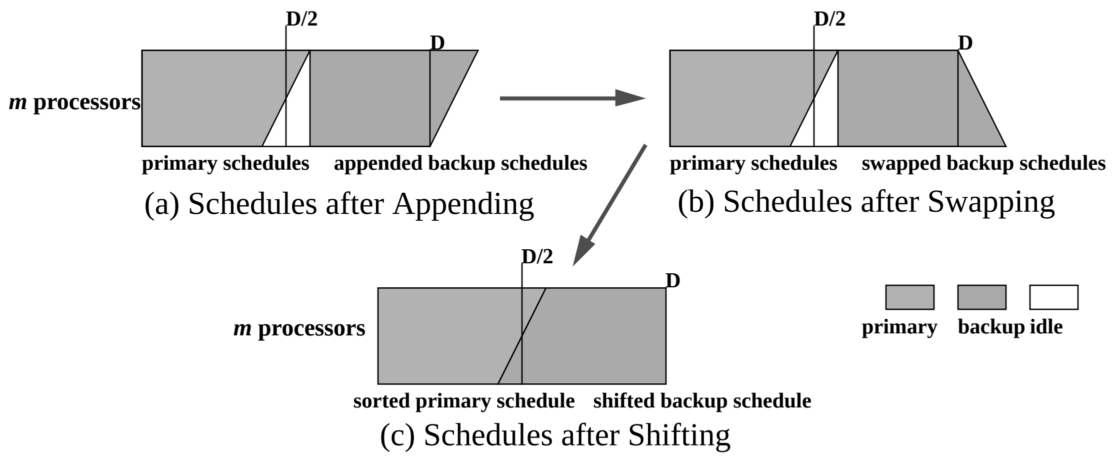


Figure 6.4: Scheduling Process of NOV

The functioning of the algorithm is illustrated by the following simple example.

Example: Using NOV to schedule the following task set on four processors: $\Sigma = \{\tau_1, \tau_2, \ldots, \tau_7\}$, $\{l(\tau_i) \mid i = 1, \ldots, 7\} = \{10, 8, 8, 7, 6, 6, 3\}$, $r = 0$, and $D = 25$. First, the LPT algorithm is used to schedule the primary copies of the tasks on four processors,

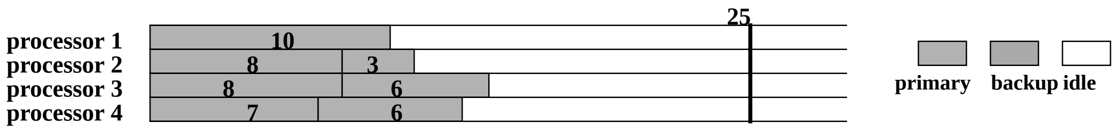


Figure 6.5: Schedule Generated by LPT

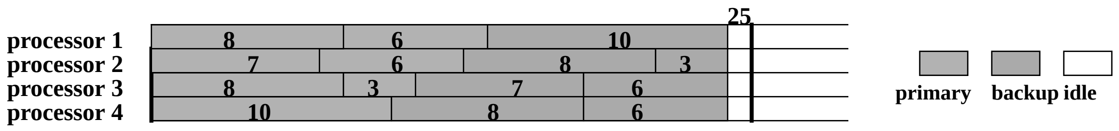


Figure 6.6: Schedule Generated after Swapping and Appending

as shown by Figure 6.5. Second, the four primary schedules are sorted in non-increasing order. Third, the primary schedules are duplicated to form the backup schedules, which are then appended to the back of the primary schedules. Lastly, the backup schedules are swapped and shifted appropriately. The final result is shown in Figure 6.6. Note that if the number of processors available is three, the task set cannot be scheduled by this algorithm.

The reason to sort the primary schedules before appending is to minimize the maximum length of the mixed schedule along with the swapping and shifting processes in the later stages. The swapping process makes sure that the backup copy of a task is not scheduled on the same processor as its primary copy. The purpose of shifting is to minimize the finishing time of the mixed schedule as well as to avoid the overlapping of backup copies among different tasks. To elaborate on the swapping and shifting processes, we formally define the swapping and shifting rules.

Swapping Rules:

(1) If the number of processors *m* is even, the longest backup schedule is appended behind the shortest primary schedule, and the second longest backup schedule is appended behind the second shortest primary schedule, and so forth.

(2) If *m* is odd, then the backup schedules of the three central processors are appended in acyclic fashion. The three central processors are the ones whose positions are in the middle. The backup schedules of the rest of the processors are swapped by following swapping rule (1).

To define the shifting rules, we need the following definitions.

Definition 6.1: Two processors are called twin processors if backup copies of the tasks in the primary schedule on a processor are appended after the primary schedule of the other processor. The two schedules on twin processors are called twin schedules. For example, in Figure 6.6, processors 1 and 4 are twin processors, so are processors 2 and 3.

Definition 6.2: For the primary schedule of a processor i, $l_p(i)$ is defined as its primary schedule length. $l_q(i)$ is defined as the computation time of the first task in the primary schedule. Obviously, $l_p(i) \geq l_q(i)$. Though $l_p(i)$ denotes the length of a schedule, it will also be used to denote the corresponding time interval whose length is $l_p(i)$.

Shifting Rules:

Suppose the backup schedule of processor j is appended behind the primary schedule of processor i.

(1) If $l_p(i) \leq D/2$ and $l_p(j) \leq D/2$, then the tasks in $l_p(j)$ are shifted together ahead of time such that the starting time of the first task in $l_p(j)$ is $max\{l_p(i), l_p(j)\}$. If $l_p(j) \neq l_q(j)$, the starting time of the first task in $l_p(j)$ can be moved to $l_p(i)$ and the rest of the backup copies can be moved ahead accordingly.

(2) Otherwise, the tasks in $l_p(j)$ are shifted together ahead of time such that the starting time of the first task in $l_p(j)$ is $l_p(i)$.

(3) Apply the above rules to every schedule on the processors.

The schedule thus generated by NOV is 1-TFT, as shown by the following theorem.

Theorem 6.2: *NOV produces an 1-TFT schedule.*

Proof: Since any primary copy of a task and its backup copy are scheduled on two different processors, as guaranteed by the Swapping Rules, we need only show that there is no overlapping between the primary copy of a task and its backup copy. Obviously, there is no overlapping between the primary copy of a task and its backup copy after the swapping process, but before the shifting process. What we need to show is that no overlapping occurs when the shifting is carried out. There are four cases to consider.

Case 1: $l_p(i) \le D/2$ and $l_p(j) \le D/2$. Since the starting time for the first task in $l_p(j)$ is $max\{l_p(i), l_p(j)\}$, there is no overlapping between the primary copies of the task scheduled on processor *j* and their corresponding backup copies on processor *i*. If $l_p(j) \neq l_q(j)$, there must be at least two tasks in the primary schedule on processor *j*. Also, the inequality $l_p(i) > l_q(j)$ must hold. If not, the second task on processor *j* should be scheduled on processor *i* according to the LPT algorithm. Since $l_p(i) > l_q(j)$, no overlapping can occur between any primary copy and its corresponding backup copy.

Case 2: $l_p(i) \le D/2$ and $l_p(j) > D/2$. Since $l_p(j) > l_p(i)$, there must be at least two tasks in the primary schedule on processor *j*. Following similar argument used in Case 1 yields that no overlapping can occur between any primary copy and its corresponding backup copy.

Case 3: $l_p(i) > D/2$ and $l_p(j) \le D/2$. No overlapping can possibly occur between any primary copy and its corresponding backup copy in this case.

Case 4: $l_p(i) > D/2$ and $l_p(j) > D/2$. Obviously, no overlapping can possibly occur between any primary copy and its corresponding backup copy in this case. If this case occurs, no 1-TFT schedule can be generated.

Since Step 5 in the scheduling algorithm ensures that any backup copy finishes before the deadline *D*, the schedule thus generated is 1-TFT. The theorem holds. ■

Observation: The schedule generated by NOV is 1-TFT in the worst case and $\lfloor m/2 \rfloor$-TFT in the best case, where *m* is the number of processors. The schedule is 1-TFT by Theorem 6.2. The schedule is $\lfloor m/2 \rfloor$-TFT, because the failure of up to $\lfloor m/2 \rfloor$ number processors can be sustained, if none of the $\lfloor m/2 \rfloor$ processors that fail has its twin among them. In the schedule generated by NOV as shown in Figure 6.6, if processors 1 and 2 fail, their twin processors, processors 3 and 4, can execute the backup copies such that none of the task deadline is missed.

6.1.3. Analysis and Performance Evaluation

In order to evaluate the performance of the scheduling algorithm, we develop another heuristic algorithm that calls the above algorithm to solve its corresponding optimization problem. In other words, we assume that the number of processors is not known and the scheduling goal is to find the minimum number of processors required to execute a set of tasks. Then this is the optimization problem corresponding to the schedule problem described above. We use the typical binary search technique to find the minimum number of processors required to schedule a given set of tasks such that the schedule generated is 1-TFT. The algorithm is given in Figure 6.7.

NOV-Test (Input: Task Set Σ, 1-TFT; Output: *m* and *schedule*);

```
(1) LowerB := $\lfloor \lceil \sum_{i=1}^{n} l(T_i) \rceil / D \rfloor$; UpperB := n;
(2) m := $\lfloor (\text{LowerB} + \text{UpperB}) / 2 \rfloor$; IF (LowerB = m) THEN {m := m + 1;
     EXIT};
(3) Invoke NOV ($\Sigma$, m, 1-TFT, success, schedule);
(4) IF success THEN UpperB := m ELSE LowerB := m; Goto Step 2.
```

Figure 6.7: Algorithm NOV-Test

Example: Suppose that a task set is given as the one in Section 6.1.2., and the question is to find the minimum number of processors necessary to execute the task set, allowing for one processor failure. The number of processors returned by executing NOV-Test is four, which is in fact equal to the optimal number of processors required.

The time complexity of NOV is $O(n\log n + n\log m)$, where n is the number of tasks, and m is the number of processors. The sorting process takes $O(n\log n)$ time. The LPT in Step 2 takes $O(n\log m)$ time. Since the binary search is bounded by $O(\log n)$, NOV-Test takes $O((n\log n + n\log m)\log n)$ time.

To evaluate the performance of the NOV algorithm, we generate task sets randomly, and run NOV-Test. Since the scheduling problem is NP-complete, it is hopeless in practice to use enumeration techniques to find the optimal solution even when the number of tasks is small. However, to find out how well the algorithms perform, we consider the lowest bounds possible for each schedule. Since backup copies are allowed to overlap, the minimum number of processors required to schedule the task set is $\lceil 2Sum/D \rceil$, where *Sum* is the total computation time of the tasks, and *D* is the deadline or period. The factor of 2 comes from the fact that no overlapping of backup copies is allowed. Therefore, we use $\lceil 2Sum/D \rceil$ as the lowest bound possible for each schedule.

Our simulation is carried out in the following fashion: First, a common deadline *D* is chosen. Then a range of values is chosen, from which the computation times of the tasks are randomly generated. NOV-Test is run for each set of tasks. The ratio between the common deadline *D* and the maximum computation time of the tasks, i.e., $r = D / \max_i (C_i)$, is kept between 2 and 10. For each different value *r*, we run NOV-Test for a wide range of task sets. Because of space limit, we only show the result of a typical set of experiments, where $r = 3$ and each data point represents the average value of the number of processors obtained by running 20 independently generated task sets. The result is plotted in Figure 6.8. It is evident from our extensive simulation that the difference between the number of processors computed by this algorithm and the lowest bound possible is only a few. Thus it is concluded that the performance of the algorithm is near-optimal.

The performance of NOV may seem surprisingly good at the first glance. However, it is not surprising at all if we take a closer look at the performance of the heuristic. Graham [25] proved that the worst case performance of LPT was tightly bounded by $4/3 - 1/3m$, where *m* is the number of processors. However, that bound is only achievable by a pathological example, where, with the exception of one processor, the number of tasks scheduled on each processor is only two. Coffman and Sethi [13] later generalized Graham’s bound to be $(k + 1)/k - 1/(km)$, where *m* is the number of processors, and *k* is the least number of tasks on any processor, or *k* is the number of tasks on a processor whose last task terminates the schedule. This result shows that the worst case performance bound for LPT approaches unity approximately as $1 + 1/k$. The worst case performance of NOV is therefore expected to be better than $1 + 1/k$.

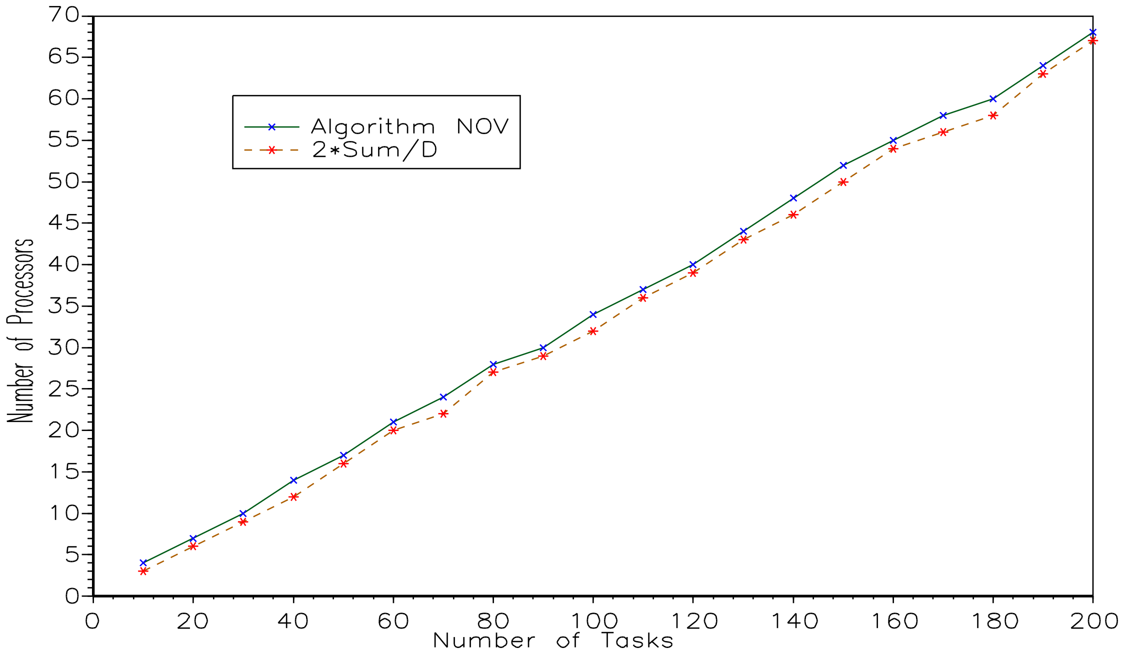


Figure 6.8: Performance of NOV

In our experiments, each processor is approximately assigned five tasks, and thus the worst case performance bounds for both heuristics are expected to be less than 1 + 1/5 = 1.2, according to the above analysis. Also, it is quite unlikely to randomly generate a task set, which can coincide with the worst cases for the heuristic.

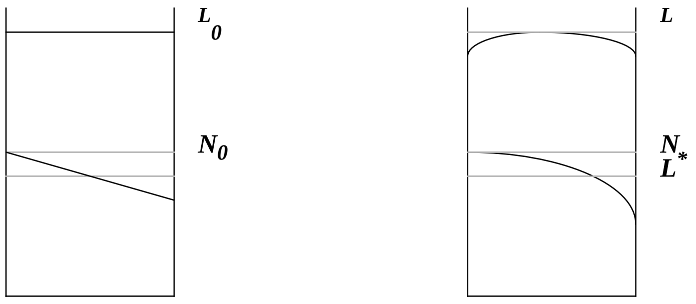
To analyze the performance of the algorithm, we first analyze the performance of the following algorithm:

NOV_1 (Input: Task Set Σ, *m*, *1-TFT*; Output: *Length, schedule*)

```
(1)-(4) The same as Steps 1-4 of Algorithm NOV.
(5) Return the maximum length among the mixed schedules as the length
      of the overall schedule.
```

The only difference between NOV_1 and NOV is that there is no deadline constraint in NOV_1.

If $\Re_A = \max_{L_o}\left(\frac{L(A)}{L_0}\right)$, where *A* represents NOV_1, L(A) the length of schedule generated by heuristic A, and L_0 the length of the optimal schedule, then we can conclude that $\Re_B = \max_{L_o}\left(\frac{L(B)}{L_0}\right)$, where *B* represents NOV-Test and L_0 the length of the corresponding optimal schedule. The conclusion is based on the result by Coffman, Garey, and Johnson [14].



Optimal Schedule Schedule by NOV_1

Figure 6.9: Relationship Between Performance Parameters

Let L and L_0 denote the schedule lengths obtained by NOV_1 and the optimal algorithm, respectively. Let N and N_0 denote the schedule lengths obtained by LPT and the optimal algorithm, respectively. $L^* = \left(\sum_{i=1}^{n} l(T_i)\right)/n$. The relationship among these parameters is given in Figure 6.9. By definition, $L \geq L_0$, $N \geq N_0$, $N_0 \geq L^*$, and $L_0 \geq 2L^*$.

Suppose k is the minimum number of tasks assigned to a processor in a schedule.

Lemma 6.3: *If $k \leq 1$ in the schedule produced by NOV_1, the schedule is optimal. In other words, $\Re_A = 1$.*

This lemma is trivially true since each processor is assigned at most one task.

Theorem 6.3: $\Re_A \leq 7/6 - 1/(6m)$*, where A represents NOV_1 and m is the number of processors.*

Proof: We prove the theorem by contradiction.

If $L = 2L^*$, then the schedule is optimal and hence $\Re_A = 1$. Therefore we need only consider the case where $L > 2L^*$.

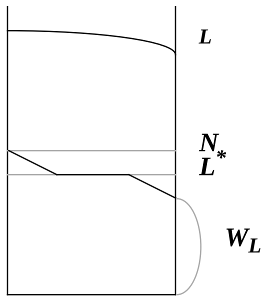
Suppose that $\Re_A > 7/6 - 1/(6m)$, i.e., $\frac{L}{L_0} > \frac{7}{6} - \frac{1}{(6m)}$. Since $L_0 \geq 2L^*$, we have $\frac{L}{2L^*} \geq \frac{L}{L_0} > \frac{7}{6} - \frac{1}{(6m)}$. In other words,

$$\frac{L}{L^*} > \frac{7}{3} - \frac{1}{(3m)} . \qquad \text{(Eq.6.1)}$$

Since $N \neq L^*$ (otherwise $L = 2L^*$), the schedule generated by LPT is shown in Figure 6.10, where W_L is the earliest time a processor finishes executing its primary task copies. There are two cases to consider: $N_0 = L^*$ and $N_0 > L^*$.

Case 1: $N_0 = L^*$. Since $L_0 = 2\,N_0$ from the assumed condition, it follows from inequality (6.1) that $L / N_0 > 7/3 - 1/(3m)$. Since $L \geq N + W_L$, we have $N/N_0 > 7/3 - 1/(3m) - W_L/L^*$. Since $W_L/L^* < 1$, we have $N/N_0 > 7/3 - 1/(3m) - W_L/L^* > 4/3 - 1/(3m)$, which contradicts the result that $N/N_0 \leq 4/3 - 1/(3m)$ by Graham [25].

Case 2: $N_0 > L^*$. Since $L_0 = 2L^*$, we have $L/L_0 = L/(2L^*)$, i.e, $L/L^* > 7/3 - 1/(3m)$. Since $L = N + W_L$, $L/L^* = N/L^* + W_L/L^*$. Since $W_L/L^* < 1$, we have $N/N_0 \geq N/L^* > 4/3 - 1/(3m)$, which results in a contradiction again. ■



m processors

Figure 6.10: A Schedule Generated by LPT and NOV_1

Theorem 6.4: *$\Re_A \leq (k+1)/k$, where A represents NOV_1, m is the number of processors, and k is the minimum number of tasks assigned on each processor in the primary schedule.*

Proof: Let m be the number of processors required to schedule a given task set Σ.

Obviously, $L \leq 2N$. Let τ_l be the task with the largest index whose finishing time in the primary schedule is L. Then $N \leq L(\tau_l) + \sum_{i=1, i \neq l}^{n} \tau_i / m = (m-1) L(\tau_l) / m + \sum_{i=1}^{n} \tau_i / m \leq mL(\tau_l) / m + \sum_{i=1}^{n} \tau_i / m$. Since $mL(\tau_l) \leq \sum_{i=1}^{n} \tau_i / k$, where k is the minimum number of tasks assigned to each processor in the primary schedule, $N \leq (1 + 1/k) \sum_{i=1}^{n} \tau_i / m$.

Since $L_0 \geq 2 \sum_{i=1}^{n} \tau_i / m$, we have

$$L \le 2N \le 2\,(1 + 1/\mathrm{k}) \sum_{i=1}^{n} \tau_i \le (1 + 1/k)L_0.$$

Therefore, $\Re_A \le (1 + 1/k)$. ■

6.2. Overlapping of Backup Copies

What we mean by allowing backup copies to be overlapped is that backup copies of the tasks whose primary copies are scheduled on different processors are allowed to be overlapped in time of their executions on a processor, since, in the worst case, only one processor failure is tolerated by assumption. However, backup copies of the tasks whose primary copies are scheduled on the same processor should not be scheduled to overlap each other in time of their executions on a processor. If the given number of processors is two, there apparently exists an optimal algorithm to schedule a set of tasks having a common deadline so as to tolerate one arbitrary processor failure. However, for more than two processors, the scheduling problem is NP-complete, even if the tasks have the same deadline.

6.2.1. Complexity of the Scheduling Problem

Task Sequencing Using Primary-Backup with a Common Deadline

Instance: Set Σ of tasks, number of processors $m \ge 3$, for each task $t \in \Sigma$, one primary copy $P(t)$ and one backup copy $G(t)$, a length $l(t) \in Z^+$ (i.e., computation time), a common release time $r \in Z^+$, a common deadline $d(t) = D \in Z^+$, and $l(t) = l(P(t)) = l(G(t))$. Note that overlapping among backup copies of the tasks on different processors is allowed.

Question: Is there an m-processor schedule σ for Σ that is 1-TFT, i.e., for each task $t \in \Sigma$, $\sigma_i(P(t)) + l(P(t)) \le \sigma_j(G(t))$, and $\sigma_i(G(t)) + l(G(t)) \le D$, where $i \ne j$, i and j designate the index of processors.

Theorem 6.5: *The Task Sequencing Problem is NP-complete.*

Proof: It is easy to verify that the scheduling problem belongs to NP. We now transform the PARTITION problem [23] to the scheduling problem when the number of processors is 3, i.e., $m = 3$.

Given an instance of A = $\{a_1, a_2, \ldots, a_n\}$ of the PARTITION problem, we construct a task set Σ using the primary-backup copy approach to run on three processors for the tolerance of one arbitrary processor failure, such that Σ can be scheduled, if and only if there is a solution to the PARTITION problem. Σ consists of $n + 4$ tasks as follows:

$$r(t) = 0, l(t) = a_t, d(t) = 3B,$$

for $t = \tau_1, \tau_2, \ldots, \tau_n$, where $\sum_{1 \le i \le n} a_i = 2B$ (This can be assumed without lose of generality). These n tasks are referred to as α-type tasks.

The other four tasks β_1, β_2, β_3, and β_4 are defined as

$$r(\beta) = 0, l(\beta) = B, d(\beta) = 3B,$$

where $\beta = \beta_1, \beta_2, \beta_3, \beta_4$. These four tasks are referred to as β-type tasks.

It is easy to see that this transformation can be constructed in polynomial time. What we will show in the following is that the set A can be partitioned into two sets S_1 and S_2 such that $\sum_{a \in S_1} s(a) = \sum_{a \in S_2} s(a)$ and $S_1 + S_2 = A$, if and only if the task set can be scheduled to produce an 1-TFT schedule.

First, suppose that A can be partitioned into two sets S_1 and S_2 such that $\sum_{a \in S_1} s(a) = \sum_{a \in S_2} s(a)$ and $S_1 + S_2 = A$. Then for each task $\alpha \in S_1$ whose length is $l(\alpha) = a$, its primary copy is scheduled on processor 2 anywhere during time interval [B, $2B$), and its backup copy on processor 3 anywhere during time interval [2B, 3B). For each task $\alpha \in S_2$ whose length is $l(\alpha) = a$, its primary copy is scheduled on processor 3 anywhere during time interval [B, 2B), and its backup copy on processor 2 anywhere during time interval [2B, 3B). For the tasks β_1, β_2, β_3, and β_4, they are scheduled in the manner as shown in Figure 6.11. The schedule thus generated is 1-TFT according to Lemma 6.2. Therefore, the task set Σ is scheduled on the three processors such that the schedule is 1-TFT.

Conversely, if the task set Σ can be scheduled on three processors such that the schedule is 1-TFT, then the schedule has one of the two forms as given in Figures 6.12 and 6.13, if the processors are properly renamed and the tasks properly adjusted. Note that for

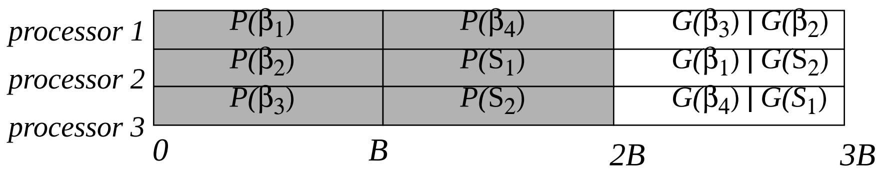


Figure 6.11: Mapping from PARTITION to Task Sequencing

each processor schedule, shuffling the primary copies in front of all the backup copies will not violate any scheduling constraint, since primary copies can start earlier than scheduled and backup copies can start later than scheduled, as long as the release time and the deadline constraints are not violated.

Case 1 (Figure 6.12): The primary copies of the four tasks β_1, β_2, β_3, and β_4 are scheduled on three processors. Let us assume, with lose of generality, that the primary copies of β_1 and β_2 are scheduled on processor 1. Then one of their backup copies must start at time *2B* and complete at the deadline 3*B*, either on processor 2 or on processor 3. It is further assumed that backup copy is scheduled on processor 2. For processor 3, exactly one copy, either primary or backup, of any task among the *n* α-type tasks must be scheduled on it. This is because any 1-TFT schedule for the three processor requires that no idle time exists on any processor, and the primary copy of a task and its backup copy must not be scheduled on the same processor. Therefore, let all the tasks scheduled on processor 2 during time interval [*B*, 2*B*) be the set S_1 (U_2 during [*B*, 2*B*] in the Figure 6.12), and the tasks on processor 1 during time interval [2*B*, 3*B*) be the set S_2, we have $\sum_{a \in S_1} s(a) = \sum_{a \in S_2} s(a)$ and $S_1 + S_2 = A$. We have solved the PARTITION problem.

Case 2 (Figure 6.13): The primary copies of the four tasks β_1, β_2, β_3, and β_4 are scheduled on two processors. For the backup copies of the tasks β_1, β_2, β_3, and β_4, there are two cases in which they can be scheduled:

Case 2.1: The four backup copies are scheduled on processor 3 during time interval [*B*, 3*B*). Then during time interval [0, *B*) for processor 3, only primary copies can be scheduled if any 1-TFT schedule exists. Let all the tasks scheduled on processor 3 during time

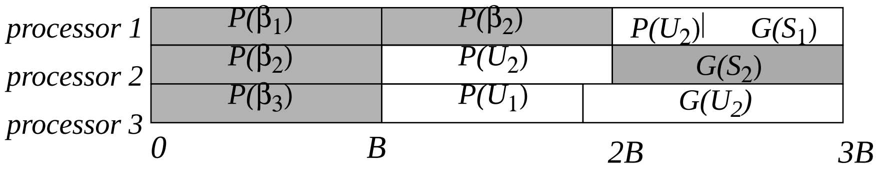


Figure 6.12: Mapping from Task Sequencing to PARTITION

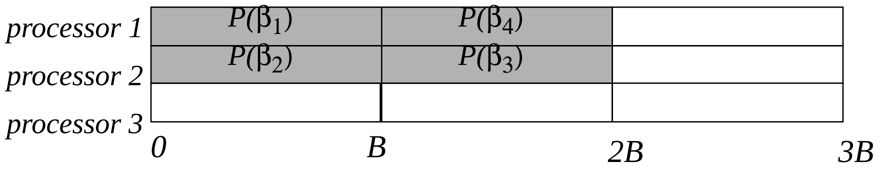


Figure 6.13: Mapping from Task Sequencing to PARTITION

interval [0, *B)* be the set S_1, and the rest of the *n* tasks be the set S_2, we again have $\sum_{a \in S_1} s(a) = \sum_{a \in S_2} s(a)$ and $S_1 + S_2 = A$.

Case 2.2: Two of the four backup copies are scheduled on processor 1 and processor 2 during time interval [2*B*, 3*B*) respectively. This implies that all the primary copies of the *n* tasks are scheduled on processor 3 (if any of the *n* tasks is scheduled on processor 1 or 2, then there is not enough time for any of the backup copy of β-type tasks to finish). The backup copies of the *n* α-type tasks must be scheduled on processor 1 and 2 during time interval [2*B,* 3*B*). Let all the tasks whose backup copies are scheduled on processor 1 during time interval [2*B*, 3*B*) be the set S_1, and the tasks whose backup copies are scheduled on processor 2 during time interval [2*B*, 3*B*) be the set S_2, we have $\sum_{a \in S_1} s(a) = \sum_{a \in S_2} s(a)$ and $S_1 + S_2 = A$. We have solved the PARTITION problem.

The scheduling problem is therefore NP-complete. ■

6.2.2. A *1-Timely-Fault-Tolerant* Scheduling Algorithm

In the following, we will first develope a heuristic to solve the scheduling problem as formulated above, i.e., the special case one, and then evaluate its performance. Though the requirement that all tasks share a common deadline may seem restrictive, the analytic results obtained below can be quite useful. In fact, our results answer the following question as well: given a set of tasks each with a primary copy and a backup copy (but with no real-time constraints), and the requirement that the failure of any one processor be tolerated, how to schedule the task set, such that the length of the fault-tolerant schedule is minimized, i.e., all the tasks complete execution as early as possible even in the presence of one arbitrary processor failure?

In scheduling a set of tasks on *m* processors, the algorithm must be designed to minimize the schedule length on each processor such that the task set can be successfully scheduled, and at the meantime, to prevent the overlapping of the primary copy of a task and its backup copy. Again, for reasons similar to develop NOV, the LPT algorithm is adopted here to serve as the base algorithm upon which a new algorithm is developed.

The scheduling algorithm starts by sorting the set of tasks in order of non-increasing computation times, and invokes the LPT algorithm to schedule the set of primary copies on the *m* processors. After all primary copies have been scheduled, all the tasks scheduled on any processor are in order of non-increasing computation time, since the LPT algorithm schedules tasks in the same order. Starting from the first processor schedule, we repeatedly apply the ALPT (Adapted Largest Processing Time first) algorithm to the backup copies of the tasks, whose primary copies are scheduled on the same processor, until either the inability of the heuristic to schedule the task set is reported, or all the *m* processor schedules are exhausted. In the later case, the task set can be scheduled by the heuristic to produce an 1-TFT schedule on *m* processors. The ALPT algorithm schedules tasks like LPT, except that the tasks (backup copies) may be scheduled a little bit later than they should be in LPT. This modification is to avoid the overlapping of the primary copy of a task and its backup copy.

We use pseudo-code to describe the scheduling algorithms in Figure 6.14. Note that we sometimes refer to the *m* schedules for *m* processors as one schedule as a whole. Let $s_i(\tau)$ and $f_i(\tau)$ denote the starting time of task τ and its finishing time on processor *i*, respectively. The processors are numbered from one to *m*. The function ρ is defined as $\rho(L_y) = y$ for the schedule L_y, or $\rho(\upsilon) = y$ for task υ, where y is the index of the processor

```
OV (Input: Task Set Σ, m, 1-TFT; Output: success, schedule)

(1) Sort the tasks in the order of non-increasing computation time
    and rename them $T_1, T_2, \ldots, T_n$. Compute $\Omega = \sum_{i=1}^{n} l(T_i)$. IF
    $\Omega \geq mD$ or $l(T_1) > D/2$, THEN success := FALSE and report that
    the task set is not schedulable on m processors such that a 1-
    TFT schedule be produced. Otherwise, go to Step 2.
(2) Apply LPT algorithm to schedule the task set on m processors.
(3) Let $L_1, L_2, \ldots, L_m$ denote the lengths of the schedules on m pro-
    cessors (initially equal to the lengths of primary schedules).
    IF $max\{L_i | (1 \leq i \leq m)\}$ > D THEN success := FALSE; EXIT (the task
    set can't be scheduled); ELSE go to Step 4.
(4)
(line 1)  FOR processor $i \leftarrow 1$ TO m DO
            Let $\upsilon_1, \upsilon_2, \ldots, \upsilon_{k_i}$ be the $k_i$ tasks (primary copies) sched-
            uled on processor i.
(line 2)    FOR task $j \leftarrow 1$ TO $k_i$ DO /* ALPT Algorithm */
(line 3)      x := $\rho( min\{L_h | (h \neq i \wedge 1 \leq h \leq m)\} )$;
(line 4)      z := $max\{f_i(\upsilon_j), L_x\}$;
(line 5)      $L_x$ := z + $l(\upsilon_j)$ ; $s_x(G(\upsilon_j))$ := z;
(line 6)      IF $L_x$ > D Then success := FALSE; EXIT (The task set
              is infeasible);
(line 7)success := TRUE; EXIT.
```

Figure 6.14: Algorithm OV

on which task copy υ is scheduled. L_y denotes the length of schedule or the schedule itself (understood by context) for the processor *y*. The process of scheduling can be illustrated by the following simple example.

Example: The following set of tasks is given to be scheduled on three processors such that one processor failure can be tolerated: $\Sigma = \{T_1, T_2, \ldots, T_7\}$, $\{l(T_i) \mid i = 1, 2, \ldots, 7\} = \{10, 8, 8, 7, 6, 6, 3\}$, $r = 0$, and $D = 25$. First, the LPT algorithm is used to schedule the primary copies of the tasks on three processors, as shown by Figure 6.15. For a processor *i*, the backup copies of the tasks whose primary copies are scheduled on processor *i* are scheduled on all the other processors except processor *i*. The scheduling process is illustrated by Figures 6.16a, 6.16b, and 6.16c. Note that if the number of processors available is 2, then this set cannot be scheduled on 2 processors to produce a 1-TFT schedule. The correctness of the schedule generated by OV is guaranteed by the following theorem.

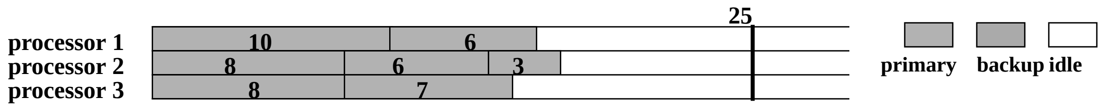
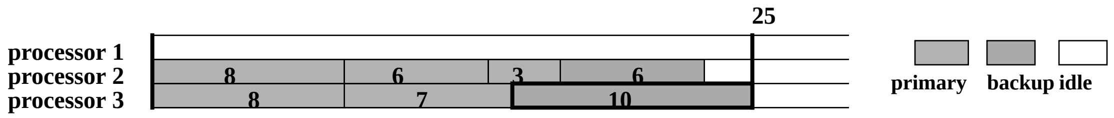
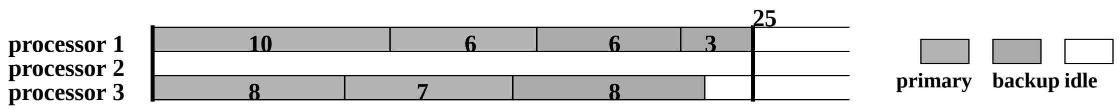


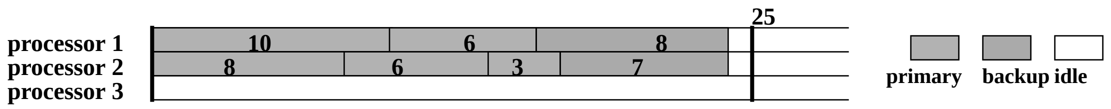
Figure 6.15: Schedule created by LPT



(a) Schedule created by OV for the backup task copies on processor 1



(b) Schedule created by OV for the backup task copies on processor 2



(c) Schedule created by OV for the backup task copies on processor 3

Figure 6.16: Scheduling Process of OV

Theorem 6.6: *Algorithm OV generates an 1-TFT schedule.*

Proof: According to Lemma 6.2, what we need to show is that for each task, its primary copy and its backup copy are scheduled on two different processors, such that the starting time for the backup copy is no earlier than the completion time of the primary copy, and its finishing time is no later than the deadline, and that the backup copies of the tasks whose primary copies are scheduled on a processor cannot be overlapped in time for their execution in the same processor.

Formally, following the notations used above, we need to show that

$\forall i\ (1 \le i \le m \wedge \forall j\ (1 \le j \le k_i \wedge f_i(\upsilon_j) \le s_x(G(\upsilon_j)) \wedge f_x(G(\upsilon_j)) \le D \wedge i \ne x\ \wedge$ $\forall j_1 (j_1 < j \wedge ((\rho(G(\upsilon_{j_1})) = x) \rightarrow (f_x(G(\upsilon_{j_1})) \le s_x(G(\upsilon_j)))))))$ holds, where m is the number of processors, and k_i is the number of primary copies scheduled on processor $i \in [1, m]$.

For each $i \in [1, m]$ and $j \in [1, k_i]$, since $x = \rho\left(min\{L_h | (h \neq i \wedge 1 \leq h \leq m)\} \right)$ from line 3, $i \neq x$. Since $s_x(G(\upsilon_j)) = z = max\{f_i(\upsilon_j), L_x\}$, $s_x(G(\upsilon_j)) \geq f_i(\upsilon_j)$. $f_x(G(\upsilon_j)) \leq$ D from line 6.

Since L_i is initialized to be the length of the primary schedule on processor *i*, $f_x(G(\upsilon_{j_1})) \leq s_x(G(\upsilon_j))$ since $L_x = z + l(\upsilon_j)$ and $s_x(G(\upsilon_j)) = z = max\{f_i(\upsilon_j), L_x\}$ $\geq L_x$ from lines 4 & 5, for $j_1 < j$.

Therefore, the schedule thus generated is 1-TFT. ■

6.2.3. Analysis and Simulation Results

Before we analyze the performance of the heuristic OV, let us define what we mean by being optimal for a fault-tolerant schedule. A fault-tolerant schedule is optimal if for all possible processor failure as assumed, its schedule length is the minimum possible. More specifically, let *m* denote the number of processors in the system, and $W_L(i)$ the length of the fault-tolerant schedule (schedule for primary and backup copies) on the other $m - 1$ processors, assuming that processor P_i has failed, then the length of the overall fault-tolerant schedule is defined as $W_L = max_{\{1 \leq i \leq m\}} W_L(i)$. If W_L is the minimum possible, then the schedule is optimal. The algorithm that generates the optimal schedule is called the optimal algorithm.

In order to analyze the performance of OV**,** we need to distinguish between two types of task sets: those that are feasible and those that are not. Given a task set Σ and a common period *D*, the task set is called infeasible if its optimal schedule length exceeds its given period *D*. In other words, for an infeasible task set, no matter which algorithm is used, it cannot be scheduled to produce a feasible schedule. We are, therefore, only interested in the task sets that are feasible, i.e., their optimal schedule lengths do not exceed their given periods. A good measurement of the performance of OV will be the frequency of successes it offers in scheduling feasible task sets. However, this measurement depends heavily on the parameter of period *D*. If the given period *D* is very large with regards to a given task set, then the algorithm will always find a feasible schedule for it. For example, if the given period D is twice as long as the optimal schedule length, then according to the results by Graham [25], any algorithm, as long as it does not leave any processor idle when there are tasks ready for execution, can generate a feasible schedule.

On the other hand, if the period D is very close to the optimal schedule length, then OV may or may not find the feasible schedule, neither may other heuristics. In general, we have no sure way of knowing whether a set of tasks is feasible or not, unless we run the algorithm to find it out. In either case, it does not make much sense to analyze the performance of OV directly, since it involves the given period D, which can be arbitrary.

However, if we disregard the given period D and focus on the ratio between the length of the schedule generated by a heuristic and the optimal schedule length, we have a better idea of how well the heuristic performs. For example, if the ratio is 1.2 for a heuristic, then as long as the given period D is equal to or more than 1.2 of the optimal schedule length, it can always find a feasible schedule. A heuristic with a ratio of 1.2 will always perform no worse than a heuristic with a ratio of 1.8. Note that the ratio is obtained under worst-case conditions. This analysis leads us to the study of a heuristic (hereinafter referred to as heuristic A) slightly different from OV. Heuristic A schedules tasks in exactly the same manner as OV except that line 6 in Step 4 is omitted, i.e., it treats each task set as having a period D of infinity. Note that this heuristic is exactly an solution to the question raised at the beginning of this section.

Let $L(A)$ and L_0 denote the length of the overall fault-tolerant schedule generated by heuristic A and the optimal schedule length, respectively. Then the ratio $\Re_A$ = $\max_{L_0}\left(\frac{L(A)}{L_0}\right)$ measures how close a schedule is to an optimal schedule, with respect to the completion time of the tasks. This metric is an indicator of how good a scheduling heuristic is. In the following we seek $\Re_A$ for the heuristic A.

Let us define PS_i as the schedule of primary copies on processor P_i, for $1 \le i \le m$, Σ_i as the set of tasks whose primary copies are assigned on processor P_i, and $\Sigma_i^- = \Sigma - \Sigma_i$.

We further define PM_i as the primary schedule on the other $m - 1$ processors where processor P_i has failed. We assert that the primary schedule PM_i is equivalent to the schedule generated by LPT on the task set Σ_i^- for $m - 1$ processors. A schedule is equivalent to another schedule if both schedules have the same set of tasks and the starting time of each task (and hence its completion time) is the same in both schedules. The possible difference between two equivalent schedules is that some tasks may be assigned on different processors. The relationship between PS_i and PM_i is illustrated in Figure 6.17.

Lemma 6.4: *The primary schedule PM_i is equivalent to the schedule generated by LPT on the task set Σ_i^- for $m - 1$ processors, i.e., $LPT(\Sigma, m) - PS_i \cong LPT(\Sigma - \Sigma_i, m - 1)$, for $1 \le i \le m$, where $LPT(\Sigma, m)$ denotes the primary schedule generated by LPT from task set Σ on m processors.*

Proof: For any task $T_j \in \Sigma - \Sigma_i$ with $j \in [1, 2, \ldots, m - 1]$, it starts on time zero in both schedules LPT(Σ, m) and LPT($\Sigma - \Sigma_i$, $m - 1$). For LPT($\Sigma - \Sigma_i$, $m - 1$), the first $m - 1$ tasks are assigned to the $m - 1$ processors with a starting time of zero. For LPT(Σ, m), the first m tasks are assigned to the m processors with a starting time of zero. Since one of the first m tasks is deleted, the rest $m - 1$ tasks are the first $m - 1$ tasks in $\Sigma - \Sigma_i$.

Let $|\Sigma_i| = n_i$, then $|\Sigma_i^-| = |\Sigma - \Sigma_i| = n - n_i$. For any task T_j in LPT(Σ, m) $- PS_i$ with a starting time of $s(T_j)$ for $m - 1 \le j$, it must be scheduled on a processor other than processor P_i and $s(T_j)$ be the earliest idle time among the $m - 1$ processors. This implies that the starting time for task T_j in the schedule LPT($\Sigma - \Sigma_i$, $m - 1$) is the same as it is in LPT(Σ, m).

On the other hand, for any task T_j in LPT($\Sigma - \Sigma_i$, $m - 1$) with a starting time of $s(T_j)$ for $m - 1 \le j \le n - n_i$, it cannot be scheduled on processor P_i in the schedule LPT(Σ, m), otherwise it would have been deleted in through $\Sigma - \Sigma_i$. Therefore, the earliest idle time among the $m - 1$ processors other than processor P_i is exactly the same as the starting time $s(T_j)$ for task T_j in LPT($\Sigma - \Sigma_i$, $m - 1$). Therefore, the two schedules are equivalent. ■

What Lemma 6.4 tells us is that every schedule PM_i is equivalent to the schedule generated by LPT on the task set $\Sigma - \Sigma_i$. Since OV first schedules the primary copies using

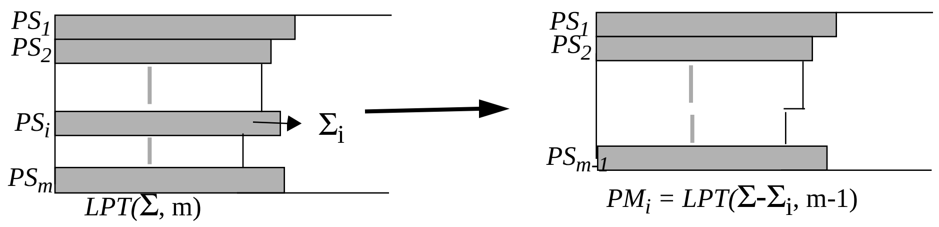


Figure 6.17: Relationship Between Schedules

LPT and then the backup copies using ALPT, the worst case performance bound is therefore expected to be around $1 + 1/k$ for $k > m$ according to the result by Coffman and Sethi [13]. This is due to the observation that for $k > m$, all the backup copies of the tasks are scheduled immediately after the primary schedule on each processor. In the following, we show that our heuristic A has an upper bound which is similar to that for LPT. But it turns out to be non-trivial to show that the upper bounds are tight for heuristic A.

Lemma 6.5: *Let k denote the least number of tasks (primary copies) on any processor or the number of tasks on a processor whose last task terminates the schedule. If* $k = 1$, *then the schedule is optimal.*

Proof: The backup copy of a task will be assigned a starting time no earlier than its primary copy's finishing time. Let T^* be the task with the minimum computation time requirement τ^*, and P^* be the processor on which T^* is assigned.

For any task T other than T^*, its backup copy will be scheduled on processor P^* or any idle processor, with a starting time at τ, which is the computation time requirement of task T. For task T^*, its backup copy will be assigned to an idle processor with a starting time of τ^*, if there is any idle processor, or to the processor on which the task with the second smallest computation time requirement is assigned, with a starting time equal to the finishing time of the task.

Since all the backup copies of the tasks are assigned the earliest starting times as possible, the schedule is therefore optimal. ■

Let τ_{max} be the largest computation time in a task set and $L(i)$ be the length of the schedule on processor P_i. Then we have the following result.

Theorem 6.7: $\Re_A \le \frac{3}{2} - \frac{1}{2(m-1)}$, *where m is the number of processors. If* $k\tau_{max} = \lceil \sum_T \tau_i \rceil / (m-1)$ *with* $k \ge 2$, *then* $\Re_A \le 1 + 1/k - 1/(k(m-1))$.

Proof: Since the backup copy of any task τ must be assigned a starting time no earlier than its primary copy's finishing time, $L_0 \ge 2\tau$. Let T^* be the smallest backup copy that finishes last in the fault-tolerant schedule where the processor P_i has failed, and τ^* be its computation time requirement. Let P_j be the processor on which T^* is assigned. Since the primary schedule can be taken as generated by LPT according to Lemma 6.4, and the backup schedule by ALPT, we have $L(j) = \tau^* + s(T^*)$, where $s(T^*)$ is the starting time of task T^*. Furthermore, $s(T^*) \le \lceil \sum_{T \ne T^*} \tau_i \rceil / (m-1)$.

$$L(j) = \tau^* + s(T^*) \le \tau^* + \lceil \textstyle\sum_{T \ne T^*} \tau_j \rceil / (m-1)$$
$$\le (m-2)\tau^*/(m-1) + \lceil \textstyle\sum_T \tau_j \rceil / (m-1)$$
$$\le (m-2)L_0/(2(m-1)) + L_0,$$

since $L_0 \ge \max\{2\tau^*, \lceil \sum_T \tau_i \rceil / (m-1)\}$.

Since $\Re_A = \max_{\{1 \le i \le m\}} L(i) / L_0 = \max_{\{1 \le i \le m\}} \left(\frac{L(i)}{L_0} \right)$, we have $\Re_A \le 3/2 - 1/(2(m-1))$.

If $k\tau_{max} = \lceil \sum_T \tau_j \rceil / (m-1)$ with $k \ge 2$, then $k\tau^* \le k\tau_{max} \le \lceil \sum_T \tau_j \rceil / (m-1) \le L_0$, where τ_{max} is the largest computation time in the task set.

Since $L(i) \le (m-2)\tau^*/(m-1) + \lceil \sum_T \tau_j \rceil / (m-1) \le (m-2)L_0 / (k(m-1)) + L_0$, we have $\Re_A \le 1 + 1/k - 1/(k(m-1))$. ■

The bounds given in Lemma 6.7 are the worst-case bounds. It is interesting to find out the performance of the scheduling algorithm on the average cases. Ideally, we want to find out the success ratio of OV with regards to feasible task sets. Since it is hard to verify whether a task set is feasible or not (the scheduling problem is NP-complete), we again develop another heuristic algorithm, which calls the above algorithm to solve its corresponding optimization problem. In other words, we assume that the number of processors is not known and the scheduling goal is to find the minimum number of processors required to execute a set of tasks. Then this is the optimization problem corresponding to the schedule problem described above. We use the typical binary search technique to find the minimum number of processors required to schedule a given set of tasks such that the schedule generated is 1-TFT, as we have done in Section 6.1.3. The algorithm is given as follows:

OV-Test (Input: Task Set Σ, 1-TFT; Output: *m* and *schedule*);

```
(1) LowerB := $\lfloor \lceil \sum_{i=1}^{n} l(T_i) \rceil / D \rfloor$; UpperB := n;
(2) m := $\lfloor (\text{LowerB} + \text{UpperB}) / 2 \rfloor$; IF (LowerB = m) THEN {m := m + 1;
      EXIT};
(3) Invoke OV (Σ, m, 1-TFT, success, schedule);
(4) IF success THEN UpperB := m ELSE LowerB := m; Goto Step 2.
```

Example: Suppose that a task set is given as the one in Section 6.2.2., and the question is to find the minimum number of processors necessary to execute the task set, allowing for one processor failure. The number of processors returned by executing OV-Test is three, which is in fact equal to the optimal number of processors required.

To evaluate the performance of OV, we generate task sets randomly, and run OV-Test. Since the scheduling problem is NP-complete when the number of processors is three, we use $\lceil Sum/D \rceil$ as the minimum number of processors required to schedule the task set, where *Sum* is the total computation time of the tasks and *D* is the deadline or period.

Our simulation is carried out in the following fashion: first, a common deadline *D* is chosen. Then a range of values is chosen, from which the computation times of the tasks are randomly generated according to the uniform distribution. OV-Test is run for each set of tasks. The ratio between the common deadline *D* and the maximum computation time of the tasks, i.e., $r = D / \max_i (C_i)$, is kept between 2 and 10. For each different value *r*, we run OV-Test for a wide range of task sets. Because of space limit, we only show the result of a typical set of experiments, where *r* = 3 and each data point represents the average value of the number of processors obtained by running 20 independently generated task sets. The result is plotted in Figure 6.18. It is evident from our extensive simulation that there is only one or two processor difference between the number computed by this algorithm and the lowest bounds possible. Thus it is concluded that the performance of the algorithm is near-optimal.

In our experiments, each processor is approximately assigned six tasks, and thus the worst case performance bounds for both heuristics are expected to be less than 1 + 1/6 = 1.1667, according to the above analysis. The performance of OV is therefore consistent with the analysis. Since the lower bound, *Sum/D*, is the lowest possible, the algorithm may in fact find the optimal schedules in many cases. In any case, the algorithms find schedules that are near optimal.

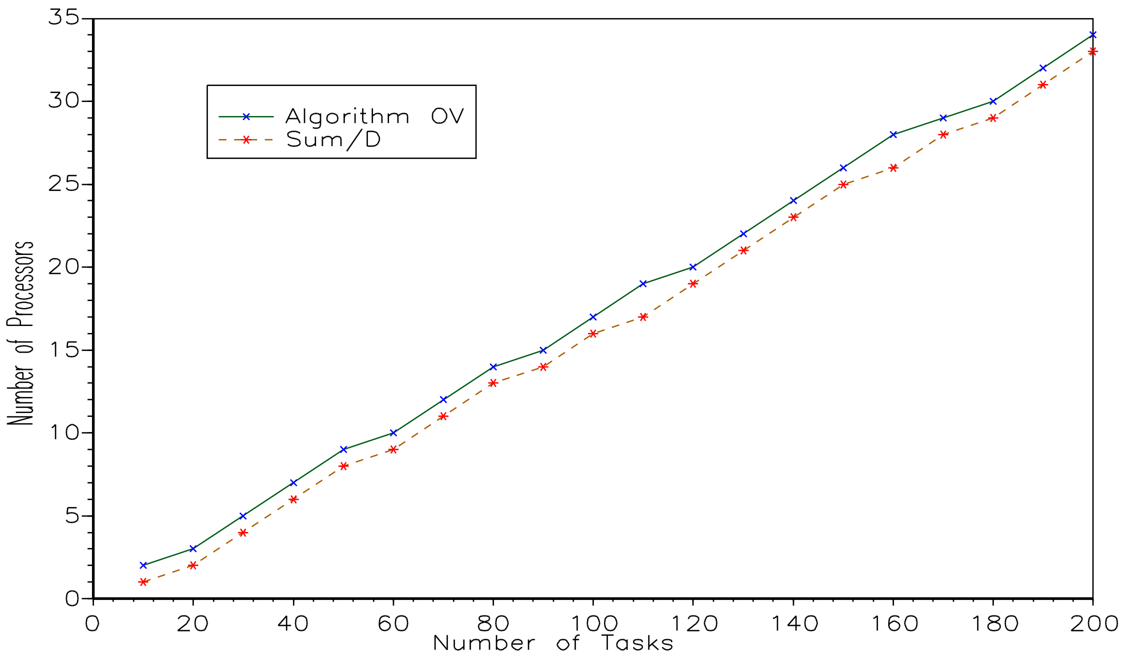


Figure 6.18: Performance of OV

Chapter 7 Conclusion

“That which is achieved the most, still has the whole
of its future yet to be achieved.”
-- Lao Zi, Dao De Jing

7.1. Contributions

In this thesis, we study four scheduling problems that are fundamental to support timeliness and dependability in a computer system. We solve the problems by developing a number of algorithms that are provably effective. Simulation results also reveal that they have good average case performance.

(1) The worst case performance bounds of various algorithms for the RMMS problem are given in Table 7.1. Comparing the bounds in Table 1.1 and in Table 7.1, it is apparent that we have the best on-line and off-line algorithms to date with regard to worst case performance and average case performance. By the “best” algorithm in the worst case performance we mean that the algorithm has the lowest worst case performance bound among all the heuristic algorithms for the same problem, whose bounds are known. In the analysis of algorithms, we not only obtain the upper bounds but also provide examples that show the upper bounds are tight or nearly tight. We derive the worst case performance of the algorithms with respect to the maximum allowable utilization of a task as well. Simulation results show that we also have the best algorithms for the RMMS problem with regards to average case performance.

(2) In solving the RMMS problem, we discover a number of schedulability conditions for the RM scheduling. These conditions are sufficient conditions, but they can deliver

Table 7.1: Performance Bounds of New Algorithms for RMMS

Algorithm A	$\Re_A^\infty$	Complexity	Type
RM-FF	[2.283, 2.33]	$O(n\log n)$	On-line
RM-BF	[2.283, 2.33]	$O(n\log n)$	On-line
RRM-FF	≤ 1.96	$O(n\log n)$	On-line
RRM-BF	≤ 1.96	$O(n\log n)$	On-line
RMGT-M	≤ 2	$O(n)$	On-line
RMST	$2.0; \leq 1/(1-\alpha)$	$O(n\log n)$	Off-line
RMGT	1.75	$O(n\log n)$	Off-line
RM-FFDU	1.667	$O(n\log n)$	Off-line
RM-FF-IFF	[1.72, 1.96]	unbounded	On-line
RM-FFDU-IFF	[1.44, 1.667]	unbounded	Off-line

better performance than Liu and Layland's condition. These conditions offer us a much broader view on the RM scheduling.

(3) For the FT-RMMS problem, we solve it by proposing an algorithm called FT-RM-FF. FT-RM-FF is shown to have a nearly tight bound of 2.33. This is the first theoretical result ever obtained for the fault-tolerance of periodic task systems.

(4) For the FT-EDFMS problem, we solve it by proposing an algorithm called FT-EDF-FF, whose performance is shown to be tightly bounded by 1.7.

(5) For the fourth problem, we prove that the problem of scheduling a set of periodic tasks on as few as three processors and with a common task deadline such that one arbitrary processor failure can be tolerated is intractable. Two algorithms are proposed to solve the problem with respect to the overlapping of backup tasks. Analytical bounds are also derived for the two algorithms. Simulation shows that they have near-optimal performance on the average.

(6) Though some of the above problems are studied in the context of real-time and fault-tolerant computer systems, they are in fact equivalent to some unsolved problems in other contexts. By solving these problems, we are actually solving other problems as well.

For example, the FT-EDFMS problem is equivalent to a constrained bin-packing problem, where some items should not be assigned to the same bin.

Although it is not difficult to adapt any of the existing bin-packing heuristics to solve the RMMS, it is difficult to analyze their worst case performance; to obtain the tight bounds for these heuristics is even more so. We are aware that the number of steps of our proof may seem daunting. But because of the importance of the final result, clarity and rigor are of prime concern. Worst case analysis is necessary for real-time applications, since the missing of hard deadlines can result in a catastrophic consequence. Once proven, the algorithm and its performance results can be used by practitioners without worrying about its worst case behavior.

The results presented in this thesis are fundamental, since allocating a set of tasks on the least number of processors is a natural extension to the problem of scheduling a set of tasks on a single processor. The two analytic results for FT-RM-First-Fit and FT-EDF-First-Fit are the generalization of the previous well-known results.

The Rate-Monotonic scheduling was first discovered around 1972-1973, and made known to the world through Liu and Layland's 1973 paper. It took about 15 years until 1988 when RM scheduling was used as a scheduling algorithm for a real-time operating system. Now the RM algorithm has been used in a number of applications [7]. The first result on RM scheduling heuristic for multiprocessor was derived in 1978 and was presented in Dhall and Liu's 1978 paper. Interests in task scheduling on multiprocessors have rapidly increased only recently, because of the inevitable employment of multiprocessors in many real-time systems. We believe that the results presented in this thesis are timely results for the research community and for practitioners at large.

In summary, we believe that by solving these problems, we have contributed to the establishment of a firm theoretical foundation for guaranteeing task deadlines in a real-time and fault-tolerant environment.

7.2. Future Work

(1) There are several interesting problems remaining for the RMMS problem. What are the tight bounds for the RM-FF-IFF and RM-FFDU-IFF algorithms? What is the low bound for any on-line algorithm for the RMMS problem? For bin-packing, it is proven that the low bound for any optimal on-line algorithm cannot be smaller than 1.533... [44]. Does the other variation of the Best-Fit heuristic other than the one we investigated in this thesis have a better performance in the worst case?

(2) Another potentially fruitful area for research is the scheduling of periodic tasks with resource sharing on a multiprocessor system such that task deadlines are guaranteed by the RM algorithm. We very much intend to study problems in this area.

(3) Our future work for the FT-RMMS problem will focus on designing algorithms with lower worst case performance bounds. We believe that algorithms with better performance can be found, although it may not be easy to obtain the tight bounds for these algorithms.

(4) Many problems remain open for non-preemptive scheduling of tasks. The tolerance of more than one arbitrary processor failures requires that the number of primary copies or backup copies be more than one for each task. Also, the requirement that all tasks share a common deadline seems restrictive. It will be of great interest to solve the scheduling problem under the condition that tasks have different deadlines. However, we believe that finding reasonably efficient heuristics to solve the scheduling problem with only real-time constraints is prerequisite to solving the scheduling problem with real-time and fault-tolerant constraints. Another question remains where for some task systems, the mapping of tasks to processors is not arbitrary because of access to peripheral devices.

Appendix A

In this appendix, we show that some errors exist in the proof for the upper bound of RMFF by Dhall and Liu in [20]. Their RMFF is almost the same as our RM-FF except that in their RMFF, the IP condition is used with the tasks being sorted in the order of increasing period. They obtained the following results:

Lemma I: If m tasks can not be feasibly scheduled on $m - 1$ processors according to RMFF, then the utilization factor of the set of tasks is greater than $m/\left(1 + 2^{1/3}\right)$.

Lemma II: If tasks are assigned to the processors according to RMFF, among all processors to each of which two tasks are assigned, there is at most one processor for which the utilization factor of the set of the two tasks is less than 1/2.

Theorem I: Let N be the number of processors required to feasibly schedule a set of tasks by RMFF, and N_0 the minimum number of processors required to feasibly schedule the same set of tasks. Then as N_0 approaches infinity, $2 \le N / N_0 \le 4 \times 2^{1/3} / (1 + 2^{1/3})$ ($\cong$ 2.23).

Unfortunately, Lemma I is incorrect, as shown by the following counter example. Lemma II gives a weak result for RMFF. These two errors led the authors to arrive at the wrong upper bound. In the following, we first show the incorrectness of Lemma I, and then give a strong version of Lemma II.

Example: Consider the case where $m = 2$ and the two tasks are given as follows:

$\tau_1 = (2^{1/2} - 1, 1)$

$\tau_2 = (2 - 2^{1/2} + \varepsilon, 2^{1/2})$, where ε is a small number and $\varepsilon > 0$.

According to RMFF, τ_1 is first assigned to a processor. Since $u_1 = 2^{1/2} - 1$ and $2(1 + u_1)^{-1} - 1 = 2^{1/2} - 1 < 2^{1/2} - 1 + \varepsilon / 2^{1/2} = u_2$, τ_2 can not be scheduled together with task τ_1 on one processor, according to Condition IP. Since τ_1 and τ_2 can not be scheduled on one processor, $u_1 + u_2$ must be greater than $2/(1 + 2^{1/3}) \cong 0.88$ according to Lemma I. But $u_1 + u_2 = 2(2^{1/2} - 1) + \varepsilon / 2^{1/2} = 0.8284 + \varepsilon / 2^{1/2}$, which is less than 0.88 for small ε.

A stronger (tight) version of their Lemma II can be given in the following lemma.

The proof is similar to that of Lemma 3.2.

Lemma II (Revised): If tasks are assigned to the processors according to RMFF, among all processors to each of which two tasks are assigned, there is at most one processor for which the utilization factor of the set of the two tasks is less than $2(2^{1/3} - 1)$.

Note that Lemma 1 is true if their RMFF used instead the necessary and sufficient condition given by Lehoczky et al in their 1989 paper [40] for $m > 2$, and our new result as stated in Theorem 3.3. It may be the case that Dhall and Liu indeed considered the problem of scheduling a set of n tasks on n processors (with one task on each processor), and obtained the result in Lemma I. However, for their upper bound to hold, the necessary and sufficient condition must be used in their RMFF scheme instead. In either case, the upper bound of 2.23 does not fit.

Bibliography

[1] N.C. Audsley, *Deadline Monotonic Scheduling*, Ph.D. Thesis, Dept. Computer Science, University of York, 1990.

[2] A. Avizienis, The N-version Approach to Fault-Tolerant Software, *IEEE Transactions on Software Engineering 11*, 1985, 1491-1501.

[3] B.S. Baker, A New Proof for the First Fit Decreasing Bin Packing Algorithm, *J. Algorithms 6*, 1985, 49-70.

[4] S. Balaji et al, Workload Redistribution for Fault-Tolerance in a Hard Real-Time Distributed Computing System, *FTCS-19*, Chicago, Illinois, June 1989, 366-373.

[5] J.A. Bannister and K. S. Trivedi, Task Allocation in Fault-Tolerant Distributed Systems, *Acta Informatica 20*, Springer-Verlag, 1983, 261-281.

[6] A. Burchard, J. Liebeherr, Y. Oh, and S.H. Son, Assigning Real-Time Tasks to Homogeneous Multiprocessor Systems, submitted to *IEEE Transactions on Computers*, January 1994.

[7] A. Burchard, Y. Oh, J. Liebeherr, and S. H. Son, A Linear Time On-line Task Assignment Scheme for Multiprocessor Systems, *IEEE 11th Workshop on Real-Time Operating Systems and Software,* Seattle, Washington, May 1994.

[8] R.W. Butler, An Assessment of the Real-Time Application Capabilities of the SIFT Computer System, *NASA Technical Memorandum 84432*, April 1982.

[9] S. Cheng, J.A. Stankovic, and K. Ramamritham, Scheduling Algorithms for Hard Real-Time Systems: A Brief Survey, Tutorial: Hard Real-Time Systems, EFF Press, 1988, 150-173.

[10] H. Chetto and M. Chetto, Some Results of the Earliest Deadline Scheduling Algorithm, *IEEE Transactions on Software Engineering 15(10)*, 1989, 466-473.

[11] R.W. Conway, W.L. Maxwell, and L.W. Miller, *Theory of Scheduling*, Addison-Wesley, Reading, MA, 1967.

[12] E.G. Coffman, Jr. (ed.), *Computer and Job Shop Scheduling Theory*, New York:

Wiley, 1975.

[13] E.G. Coffman, Jr. and R. Sethi, A Generalized Bound on LPT Sequencing, *Revue Francaise d'Automatique Informatique Recherche Operationelle 10 (5)*, 1976, Suppl., 17-25.

[14] E.G. Coffman, Jr., M.R. Garey, and D.S. Johnson, An Application of Bin-Packing to Multiprocessor Scheduling, *SIAM J. Computing 7*, 1978, 1-17.

[15] E.G. Coffman, Jr., M.R. Garey, and D.S. Johnson, Approximate Algorithms for Bin Packing - An Updated Survey, In *Algorithm Design for Computer System Design*, 49-106, G. Ausiello, M. Lucertinit, and P. Serafini (eds), Springer-Verlag, New York, 1985.

[16] S. Davari and S.K. Dhall, An On Line Algorithm for Real-Time Tasks Allocation, *IEEE Real-Time Systems Symposium*, 1986, 194-200.

[17] S. Davari and S.K. Dhall, On a Periodic Real-Time Task Allocation Problem, *Proc. of 19th Annual International Conference on System Sciences*, 1986, 133-141.

[18] R.I. Davis, K.W. Tindell, and A. Burns, Scheduling Slack Time in Fixed Priority Preemptive Systems, *IEEE Real-Time Systems Symposium*, 1993222-231.

[19] S.K. Dhall, *Scheduling Periodic-Time-Critical Jobs on Single Processor and Multiprocessor Computing Systems*, Ph.D. Thesis, University of Illinois, Urbana, 1977.

[20] S.K. Dhall and C.L. Liu, On a Real-Time Scheduling Problem, *Operations Research 26*, 1978, 127-140.

[21] B.L. Di Vito and R.W. Butler, Provable Transient Recovery for Frame-Based, Fault-Tolerant Computing Systems, *IEEE Real-Time Systems Symposium*, 1992, 275-279.

[22] J.D. Gafford, Rate-Monotonic Scheduling, *IEEE Micro*, June 1991, 34-39.

[23] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the The-*

ory of NP-completeness, W.H. Freeman and Company, NY, 1978.

[24] M.J. Gonzalez and J.W. Soh, Periodic Job Scheduling in a Distributed Processor System, *IEEE Transactions on Aerospace and Electronic Systems 12(5),* September 1976, 530-535.

[25] R. L. Graham, Bounds on Multiprocessing Timing Anomalies, *SIAM J. Appl. Math. 17*, 1969, 416-429.

[26] R.L. Graham et al. Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey, *Annals of Discrete Mathematics 5*, 1979, 287-326.

[27] A.L. Hopkins et al, FTMP-A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft, *Proc. of the IEEE 66 (10),* October, 1978.

[28] K. Jeffay, D.F. Stanat, and C.U. Martel, On non-preemptive scheduling of periodic and sporadic tasks," *IEEE Real-Time Systems Symposium*, 1991, 129-139.

[29] B.W. Johnson, *Design and Analysis of Fault Tolerant Digital Systems*, Addison-Wesley, 1989.

[30] D.S. Johnson, *Near-Optimal Bin Packing Algorithms*, Ph.D. Thesis, MIT, 1973.

[31] D.S. Johnson, A. Bemers, J.D. Ullman, M.R. Garey, and R.L. Graham, Worst-Case Performance Bounds for Simple One-dimensional Packing Algorithms, *SIAM J. Comput. 3*, 1974, 299-326.

[32] D.S. Johnson, Fast Algorithms for Bin Packing, *J. Comput. Syst. Sci. 8*, 1974, 272-314.

[33] D.S. Johnson, and M.R. Garey, A 71/60 Theorem for Bin Packing, *J. Complexity 1*, 1985, 65-106.

[34] R.M. Kieckhafer, C.J. Walter, A.M. Finn, and P.M. Thambidurai, The MAFT Architecture for Distributed Fault Tolerance, *IEEE Transactions on Computers 37 (4),* April 1988, 398-405.

[35] J.C. Knight and P.E. Ammann, Design Fault Tolerance, *Reliability Engineering and System Safety 32*, 1991, 25-49.

[36] C.M. Krishna and K.C Shin, On Scheduling Tasks with a Quick Recovery from Failure, *IEEE Transactions on Computers 35(5)*, May 1986, 448-454.

[37] J. Labetoulle, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. *Preemptive Scheduling of Uniform Machines Subject to Release Dates*, Report BW 99, Mathematisch Centrum, Amsterdam, 1979.

[38] C.C. Lee and D.T. Lee, A Simple On-line Bin-Packing Algorithm, *JACM 32 (3)*, July 1985, 562-572.

[39] J.P. Lehoczky, L. Sha, and J.K. Strosnider, Enhanced Aperiodic Responsiveness in Hard Real-time Environments, *IEEE Real-Time Systems Symposium*, 1987, 261-270.

[40] J.P. Lehoczky, L. Sha, and Y. Ding, The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior, *IEEE Real-Time Systems Symposium*, 1989, 166-171.

[41] J.P. Lehoczky, Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines, *IEEE Real-Time Systems Symposium*, 1990, 201-209.

[42] J.P. Lehoczky and S. Ramos-Thuel, An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems, *IEEE Real-Time Systems Symposium*, 1992, 110-123.

[43] J.Y.T. Leung and J. Whitehead, On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks, *Performance Evaluation 2*, 1982, 237-250.

[44] M.F. Liang, A Lower Bound for On-line Bin Packing, *Information Processing Letters 10 (2)*, March 1982, 76-79

[45] A.L. Liestman and R.H. Campbell, A Fault Tolerant Scheduling Problem, *IEEE Transactions on Software Engineering 12(11)*, November 1986, 1089-1095.

[46] C.L. Liu and J. Layland, Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment, *JACM 10(1)*, 1973, 174-189.

[47] J.W.S. Liu, K.-J. Lin, and S. Natarajan, Scheduling Real-time, Periodic Jobs Using

Imprecise Results, *IEEE Real-Time Systems Symposium*, 1987, 252-260.

[48] J.W.S. Liu, K.-J. Lin, W.K. Shih, A.C. Yu, J.Y. Chung and W. Zhao, Algorithms for Scheduling Imprecise Computations, *Computer*, May 1989, 58-68.

[49] A.K. Mok, *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*, Ph.D. Thesis, M.I.T., 1993.

[50] Y. Oh and S.H. Son, Multiprocessor Support for Real-Time Fault-Tolerant Scheduling, *IEEE Workshop on Architectural Aspects of Real-Time Systems*, San Antonio, Texas, December 1991, 76-80.

[51] Y. Oh and S.H. Son, An Algorithm for Real-Time Fault-Tolerant Scheduling in Multiprocessor Systems, *4th Euromicro Workshop on Real-Time Systems*, Athens, Greece, June 1992.

[52] Y. Oh and S.H. Son, Preemptive Scheduling of Periodic Tasks on Multiprocessor: Dynamic Algorithms and Their Performance, *TR-CS-93-26, Department of Computer Science, University of Virginia*, May 1993.

[53] Y. Oh and S.H. Son, Allocating Fixed-Priority Periodic Tasks on Multiprocessor Systems, re-submitted to *Real-Time Systems Journal*, February 1994.

[54] Y. Oh and S. H. Son, Rate-Monotonic Scheduling on Multiprocessor Systems, submitted to *Informatica*, February 1994.

[55] Y. Oh and S.H. Son, Scheduling Hard Real-Time Tasks with Tolerance of Multiple Processor Failures, *Euromicro Journal, Special Issue on Parallel Processing in Embedded Real-Time Systems*, 1994, to appear.

[56] Y. Oh and S. H. Son, Enhancing Fault-Tolerance in Rate-Monotonic Scheduling, *Real-Time Systems Journal, Special Issue on Responsive Computer Systems*, May 1994, to appear.

[57] Y. Oh and S.H. Son, Task Allocation Algorithms for Fault-tolerance in Hard Real-time Systems, submitted to *IEEE Trans. on Parallel and Distributed Systems*, February 1994.

[58] Y. Oh and S.H. Son, Scheduling Hard Real-Time Tasks with Reliability Constraint, revised and re-submitted to *Journal of Operational Research Society*, May 1994.

[59] D.K. Pradhan, *Fault-Tolerant Computing -- Theory and Techniques*, Volumes I and II, Prentice-Hall, Englewood Cliffs, N.J., 1986.

[60] R. Rajkumar, *Task Synchronization in Real-Time Systems*, Ph.D. Thesis, Carnegie-Melon University, August 1989.

[61] K. Ramamritham, Allocation and Scheduling of Complex Periodic Tasks, *International Conference on Distributed Computing Systems*, May 1990.

[62] K. Ramamritham and J.A. Stankovic, Scheduling Strategies Adopted in Spring: A Overview, a chapter in *Foundations of Real-Time Computing: Scheduling and Resource Allocation* (ed.) by A.M. van Tilborg and G.M. Koob, 1991, 277-307.

[63] S. Ramos-Thuel and J.K. Strosnider, The Transient Server Approach to Scheduling Time-Critical Recovery Operations, *IEEE Real-Time Systems Symposium*, 1991, 286-295.

[64] S. Ramos-Thuel, *Enhancing Fault Tolerance of Real-Time Systems through Time Redundancy*, Ph.D. Thesis, Carnegie Mellon University, May 1993.

[65] S. Ramos-Thuel and J.P. Lehoczky, On-line Scheduling of Hard Deadline Aperiodic Tasks in Fixed-Priority Systems, *IEEE Real-Time Systems Symposium*, 1993, 160-171.

[66] B. Randell, System Structure for Software Fault Tolerance, *IEEE Transactions on Software Engineering 1*, 1975, 220-232.

[67] K. Schwan and H. Zhou, Dynamic Scheduling of Hard Real-time Tasks and Real-time Threads, *IEEE Transactions on Software Engineering 18(8)*, 1992, 736-748.

[68] P. Serlin, Scheduling of Time Critical Processes, *Proc. of the Spring Joint Computers Conference 40*, 1972, 925-932.

[69] L. Sha, J.P. Lehoczky, and R. Rajkumar, Solutions for Some Practical Problems in Prioritized Preemptive Scheduling, *IEEE Real-Time Systems Symposium*, 1986,

181-191.

[70] L. Sha, R. Rajkumar, J.P. Lehoczky, and K. Ramamritham, Mode Change Protocols for Priority-Driven Preemptive Scheduling, *Journal of Real-Time Systems 1(3)*, 1989, 244-264.

[71] L. Sha, R. Rajkumar, and J.P. Lehoczky, Priority Inheritance Protocols: An Approach to Real-Time Synchronization, *IEEE Transactions on Computers 39(9)*, 1990, 1175-1185.

[72] L. Sha and J.B. Goodenough, Real-Time Scheduling Theory and Ada, *Computer*, April 1990, 53-65.

[73] W-K. Shih, J.W.S. Liu, and J-Y Chung, Fast Algorithms for Scheduling Imprecise Computations, *IEEE Real-Time Systems Symposium*, 1989, 12-19.

[74] K.G. Shin, G. Koob, and F. Jahanian, Fault-Tolerance in Real-Time Systems, *IEEE Real-Time Systems Newsletter 7 (3)*, 1991, 28-34.

[75] T.B. Smith, Fault-Tolerant Processor Concepts and Operation, *Proc. of 14th IEE Fault-Tolerant Computing Symposium*, June 1984.

[76] A. Spector and D. Gifford, The Space Shuttle Primary Computer System, *CACM*, September 1984, 874-900.

[77] B. Sprunt, J.P. Lehoczky, and L. Sha, Exploiting Unused Periodic Time for Aperiodic Service Using the Extended Priority Exchange Algorithm, *IEEE Real-Time Systems Symposium*, 1988, 251-258.

[78] B. Sprunt, L. Sha, and J.P. Lehoczky, Aperiodic Task Scheduling for Hard Real-time Systems, *Journal of Real-Time Systems 1*, 1989, 27-60.

[79] B. Sprunt, *Aperiodic Task Scheduling for Real-Time Systems*, Ph.D. Thesis, Carnegie Melon University, 1990.

[80] J.A. Stankovic, Misconception of Real-Time Computing, *IEEE Computer 21 (10)*, 1988, 10-19.

[81] K.W. Tindell, A. Burns, and A.J. Wellings, Mode Change in Priority Pre-emp-

tively Scheduled Systems, *IEEE Real-Time Systems Symposium*, 1992, 100-109.

[82] J.H. Wensley et.al, SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control, *Proc. of the IEEE 66 (10)*, October 1978, 1240-1255.

[83] A. C. Yao, New Algorithms for Bin Packing, *JACM 27*, 1980, 207-227.