

Student Work

---

8-1-2004

## An Evolutionary-Based Approach for Real-Time Fault-Tolerant Multiprocessor Scheduling.

Yohitsugu Hashimoto

Follow this and additional works at: <https://digitalcommons.unomaha.edu/studentwork>

---

### Recommended Citation

Hashimoto, Yohitsugu, "An Evolutionary-Based Approach for Real-Time Fault-Tolerant Multiprocessor Scheduling." (2004). *Student Work*. 3572.

<https://digitalcommons.unomaha.edu/studentwork/3572>

This Thesis is brought to you for free and open access by DigitalCommons@UNO. It has been accepted for inclusion in Student Work by an authorized administrator of DigitalCommons@UNO. For more information, please contact [unodigitalcommons@unomaha.edu](mailto:unodigitalcommons@unomaha.edu).



**An Evolutionary-Based Approach for Real-Time Fault-Tolerant Multiprocessor  
Scheduling**

**A Thesis**

**Presented to the**

**Department of Computer Science**

**and the**

**Faculty of the Graduate College**

**University of Nebraska**

**In Partial Fulfillment**

**of the Requirements for the Degree**

**Masters of Science**

**University of Nebraska at Omaha**

**by**

**Yoshitsugu Hashimoto**

**August 2004**

UMI Number: EP74770

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



UMI EP74770

Published by ProQuest LLC (2015). Copyright in the Dissertation held by the Author.

Microform Edition © ProQuest LLC.

All rights reserved. This work is protected against unauthorized copying under Title 17, United States Code

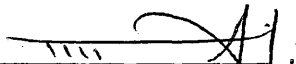
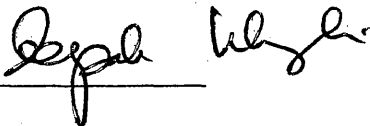
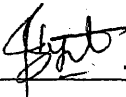


ProQuest LLC.  
789 East Eisenhower Parkway  
P.O. Box 1346  
Ann Arbor, MI 48106 - 1346

THESIS ACCEPTANCE

Acceptance for the faculty of the Graduate College,  
University of Nebraska, in partial fulfillment of the  
requirements for the degree of Master of Science,  
University of Nebraska at Omaha.

Committee

Name	Signature
Hesham Ali	
Deepak Khazanchi	
PRITHVIRAJ DASGUPTA	

Chairperson (signature)  Date 7/26/04

Co-Chairperson (signature) \_\_\_\_\_ Date \_\_\_\_\_

(if applicable)

## **Acknowledgements**

I would like to give special thanks to my thesis advisor, Dr. Hesham Ali. I have learned many precious things from him through his lectures, working with him as a teaching assistant, and throughout this thesis process. I greatly appreciate his help and time.

Also, I would like to give thanks to my family. They let me stay and study in the United States and helped me in many ways.

Finally, I would like to say “thank you” to all my friends in Omaha. They constantly helped and encouraged me.

## **Abstract**

### **An Evolutionary-Based Approach for Real-Time Fault-Tolerant Multiprocessor Scheduling**

Yoshitsugu Hashimoto, MS

University of Nebraska, 2004

Advisor: Dr. Hesham Ali

Recently, the number of applications demanding real-time performance from their multiprocessor systems has been significantly increasing. At the same time, due to the possible catastrophic consequences from missing deadlines, fault-tolerance has become a critical issue in real-time systems.

A scheduling problem is a classical problem that involves finding a feasible allocation and order of tasks into processors while optimizing a given objective function. In addition, in a fault-tolerant multiprocessor scheduling system, all performance requirements must be satisfied even if a processor failure occurs.

If a given problem requires time that is exponential in the problem size, the problem is called NP-complete. In this sense, a scheduling problem is known to be NP-complete in its general form as well as in many restricted cases. Therefore, many researchers have proposed heuristic algorithms to solve various scheduling problems.

As an alternative to traditional heuristic algorithms, several optimization methods such as simulated annealing, tabu search, and genetic algorithms have been adapted to solve

various NP-complete problems and proven their effectiveness. Nonetheless, almost none of these methods have been used for fault-tolerant scheduling problems.

In this thesis, we first introduce a variety set of existing heuristic algorithms and optimization methods specifically designed to solve scheduling problems. Then our focus moves toward genetic algorithms. We present a genetic algorithm and take a new approach to address real-time fault-tolerant scheduling. We also modify the existing heuristic algorithm to fit into our problem and compare these two algorithms to analyze the effectiveness of the genetic algorithm.

# Table of Contents

1	Introduction .....	1
1.1	Real-Time Systems .....	1
1.2	The Scheduling Problem .....	2
1.3	NP-Completeness .....	3
1.4	NP-Complete Scheduling Problems .....	4
2	Basic Terminology and Problem Definition .....	7
2.1	Basic Terminology .....	7
2.2	Problem Definition .....	13
3	Previous Work .....	16
3.1	Schedulability Analysis .....	17
3.2	Uniprocessor Systems .....	20
3.2.1	Rate-Monotonic (RM) Algorithm .....	20
3.2.2	Deadline-Monotonic (DM) Algorithm .....	23
3.2.3	Earliest-Deadline-First (EDF) Algorithm .....	24
3.3	Multiprocessor Systems .....	26
3.3.1	Partitioned Scheduling .....	26
3.3.2	Guided & Non-guided Search .....	30
3.3.2.1	Guided Search .....	30
3.3.2.2	Non-guided Search .....	32
3.4	Fault-Tolerant Scheduling .....	37
4	Fault-Tolerant Scheduling Algorithms .....	38
4.1	Introduction .....	38
4.2	Fault-Tolerance .....	39
4.3	Triple Modular Redundancy (TMR) Approach .....	40
4.4	Primary/Backup (PB) Approach .....	41
4.5	Imprecise Computational (IC) Model .....	42
4.6	$(m,k)$ -firm Deadline Model .....	42
4.7	Prototype Scheduler .....	43
4.7.1	Overview .....	43
4.7.2	Task Allocation .....	48
4.7.3	The RFS Algorithm in Pseudocode .....	53
4.7.4	Performance Measures .....	54
4.7.5	Simulator Interface .....	56
4.7.6	Simulation Results .....	57
4.7.7	Summary .....	60
5	Genetic Algorithms .....	61
5.1	Introduction .....	61



5.2	Overview	62
5.3	Chromosome Encoding	64
5.4	Initial Population	65
5.5	Fitness Function	66
5.6	Selection Schemes	66
5.6.1	Roulette Wheel Selection	66
5.6.2	Rank Selection	67
5.6.3	Tournament Selection	68
5.6.4	Elitism Selection	69
5.6.5	Steady-State Selection	69
5.7	Genetic Operators	70
5.7.1	Crossover	70
5.7.2	Mutation	72
5.7.3	Inversion	73
5.7.4	Rotation	74
6	Genetic Scheduling Algorithms	75
6.1	Existing Genetic Algorithms	75
6.2	Partitioned Genetic Algorithm	76
6.3	The HAR Algorithm	79
6.3.1	Chromosome Encoding	79
6.3.2	Initial Population	81
6.3.3	Genetic Operators	81
6.3.3.1	Reproduction	81
6.3.3.2	Crossover	82
6.3.3.3	Mutation	82
6.3.3.4	Shortfalls	83
6.4	The Combined-Genetic List (CGL) Algorithm	84
6.4.1	Initial Population	84
6.4.2	Genetic Operators	84
6.4.2.1	Reproduction	85
6.4.2.2	Crossover	85
6.4.2.3	Mutation	86
6.5	Other Genetic Algorithms Related to CGL	86
6.6	Observations	87
6.7	Proposed Model	89
6.7.1	Requirements for Real-Time Fault-Tolerant Systems	90
6.7.2	Priority Assignment	91
6.7.3	Chromosome Encoding	92
6.7.4	Hybrid Task Allocation Method (HTAM)	94
6.7.5	Processor Search	96
6.7.6	Earliest Empty Slot Search	97
6.7.7	Initial Population	100
6.7.8	Fitness Function	102

6.7.9 Genetic Operators .....	102
6.7.9.1 Reproduction .....	102
6.7.9.2 Crossover .....	104
6.7.9.3 Mutation .....	108
6.7.10 Backup Overloading .....	110
6.7.11 GRFS in Pseudocode .....	112
<b>7 Simulation Results and Analysis .....</b>	<b>113</b>
7.1 The Best Depth First (BDF) Algorithm .....	113
7.2 Input DAG .....	114
7.3 BDF vs. GRFS .....	114
7.4 Maximum Degree .....	119
7.5 Number of Generations .....	122
7.6 Initial Population Size .....	124
7.7 Maximum Population Size .....	126
7.8 Backup Overloading .....	128
<b>8 Conclusion and Future Work .....</b>	<b>132</b>
8.1 Conclusion .....	132
8.2 Future Work .....	134
<b>References .....</b>	<b>135</b>

## List of Figures

Figure 2.1	A task graph .....	8
Figure 3.1	The RMNF algorithm .....	27
Figure 3.2	The RMFF algorithm .....	28
Figure 3.3	The branch-and-bound algorithm .....	31
Figure 3.4	The search tree for the branch-and-bound algorithm .....	32
Figure 3.5	The simulated annealing algorithm .....	34
Figure 3.6	The tabu search algorithm .....	36
Figure 4.1	Triple Modular Redundancy (TMR) .....	40
Figure 4.2	Replication only .....	44
Figure 4.3	De-allocation only .....	45
Figure 4.4	Backup overloading .....	46
Figure 4.5	Replication and backup overloading .....	47
Figure 4.6	Replication only (task allocation) .....	48
Figure 4.7	De-allocation only (task allocation) .....	49
Figure 4.8	Backup overloading (task allocation) .....	50
Figure 4.9	Replication and backup overloading (task allocation) .....	52
Figure 4.10	The RFS algorithm .....	53
Figure 4.11	The RFS interface .....	56
Figure 4.12	Processor usage .....	57
Figure 4.13	Total number of tasks missed deadlines .....	58
Figure 4.14	Total communication cost .....	59
Figure 5.1	A traditional genetic algorithm .....	62
Figure 5.2	Binary encoding .....	64
Figure 5.3	Permutation encoding .....	64
Figure 5.4	Value encoding .....	64
Figure 5.5	Tree encoding .....	65
Figure 5.6	One-point crossover .....	70
Figure 5.7	Two-point crossover .....	71
Figure 5.8	$N$ -point crossover ( $N=3$ ) .....	71
Figure 5.9	Uniform crossover .....	72
Figure 5.10	Mutation (flip) .....	73
Figure 5.11	Mutation (swap) .....	73
Figure 5.12	Inversion .....	73
Figure 5.13	Upward rotation .....	74
Figure 5.14	Downward rotation .....	74
Figure 6.1	A clustered graph .....	77
Figure 6.2	A Gantt chart .....	77
Figure 6.3	The DSC algorithm .....	78
Figure 6.4	A task graph and its representation .....	80
Figure 6.5	Crossover (CGL) .....	86
Figure 6.6	Priority in GRFS .....	92

Figure 6.7	String representation .....	93
Figure 6.8	Hybrid Task Allocation Method (HTAM) .....	95
Figure 6.9	EST (Case 1) .....	99
Figure 6.10	EST (Case 2-1) .....	99
Figure 6.11	EST (Case 2-2) .....	99
Figure 6.12	The algorithm to find EST .....	100
Figure 6.13	Initial population .....	101
Figure 6.14	Selection .....	103
Figure 6.15	Reproduction .....	104
Figure 6.16	Two-point crossover .....	106
Figure 6.17	Crossover .....	107
Figure 6.18	Upward rotation .....	108
Figure 6.19	Mutation .....	109
Figure 6.20	Natural backup overloading .....	110
Figure 6.21	Compulsive backup overloading .....	111
Figure 6.22	The GRFS algorithm .....	112
Figure 7.1	Schedule in the BDF algorithm .....	114
Figure 7.2	BDF vs. GRFS (maximum degree=2) .....	116
Figure 7.3	BDF vs. GRFS (maximum degree=3) .....	116
Figure 7.4	BDF vs. GRFS (maximum degree=4) .....	117
Figure 7.5	BDF vs. GRFS (maximum degree=5) .....	117
Figure 7.6	Precedence constraints and performance .....	118
Figure 7.7	Maximum degree (processors) .....	120
Figure 7.8	Maximum degree (tasks) .....	121
Figure 7.9	Number of generations .....	123
Figure 7.10	Initial population size (processors) .....	125
Figure 7.11	Initial population size (tasks) .....	125
Figure 7.12	Maximum population size (processors) .....	127
Figure 7.13	Maximum population size (tasks) .....	128
Figure 7.14	Backup overloading (processors) .....	129
Figure 7.15	Backup overloading (tasks) .....	130
Figure 7.16	Processor usage (processors) .....	131
Figure 7.17	Processor usage (tasks) .....	131

## List of Tables

Table 1.1	Complexity of various scheduling problems .....	6
Table 2.1	Scheduling algorithm and priority assignment .....	12
Table 3.1	Notation and description .....	16
Table 3.2	Feasibility test .....	19
Table 3.3	Schedulability analysis .....	25
Table 3.4	Upper bounds of RM-based scheduling algorithms .....	29
Table 4.1	Summary for the RFS algorithm .....	60
Table 6.1	Clusterization function .....	79
Table 6.2	Information in genes .....	94
Table 7.1	Parameter list (BDF vs. GRFS) .....	115
Table 7.2	Overall result (BDF vs. GRFS) .....	115
Table 7.3	Parameter list (maximum degree) .....	119
Table 7.4	Overall result (maximum degree) .....	119
Table 7.5	Parameter list (generations) .....	122
Table 7.6	Overall result (generations) .....	122
Table 7.7	Parameter list (initial population size) .....	124
Table 7.8	Overall result (initial population size) .....	124
Table 7.9	Parameter list (maximum population size) .....	126
Table 7.10	Overall result (maximum population size) .....	126
Table 7.11	Parameter list (backup overloading) .....	128
Table 7.12	Overall result1 (backup overloading) .....	129
Table 7.13	Overall result2 (backup overloading) .....	130

# Chapter 1

## Introduction

### 1.1 Real-Time Systems

In recent years, an increasing number of applications are demanding real-time performance from their multiprocessor systems. Ramamritham and Stankovic [Ramamritham & Stankovic 94] defined real-time systems as “those systems in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced.” A real-time system has to fulfill these properties: (1) timeliness: it is required for the real-time system to complete certain tasks within the time boundaries; (2) simultaneous processing: if there is more than one event to be executed simultaneously, the real-time system must execute them concurrently so that all deadlines are met; (3) predictability: the real-time system has to deal with all possible events in a predictable way; and (4) dependability: it is necessary for all tasks to be completed before their deadlines even in the presence of processor failures.

Further, real-time systems can be classified into three categories based on their properties. Manimaran and Murthy defined these three categories as:

1. *hard* real-time systems, in which the consequences of not executing a task before its deadline may be catastrophic,
2. *firm* real-time systems, in which the result produced by the corresponding task ceases to be useful as soon as the deadline expires, but the consequences of not meeting the deadline are not very severe, and

3. *soft* real-time systems, in which the utility of results produced by a task with a soft deadline decreases over time after the deadline expires.

Taken from [Manimaran & Murthy 98]

Examples of hard real-time systems include a car's engine management system, medical systems such as pacemakers, and industrial process controllers. Examples of soft real-time systems include airline reservation systems, banking, and live audio-video systems.

## 1.2 The Scheduling Problem

A scheduling problem involves finding an optimal sequence of jobs which maximizes evaluation functions. Examples of scheduling problems include job-shop problem, flow-shop problem, timetabling problem, and traveling salesman problem. In a multiprocessor scheduling problem, a set of tasks and a set of processors are given. Then the problem is to how to assign each task to a processor so that all performance requirements are met. Ramamritham and Stankovic presented four different scheduling paradigms depending on (1) whether schedulability analysis is available, (2) whether schedulability analysis is done statically or dynamically, and (3) whether schedulability analysis itself can be used as a schedule [Ramamritham & Stankovic 94]. These four paradigms are described below:

- *Static table-driven approaches*: These approaches produce static schedulability analysis which results in a schedule used to determine the execution time for each task.
- *Static priority-driven preemptive approaches*: These approaches produce static schedulability analysis but no schedule.

- *Dynamic planning-based approaches*: In these approaches, schedulability analysis is done dynamically and results in a schedule which is used to determine the execution time for each task.
- *Dynamic best-effort approaches*: In these approaches, no schedulability analysis is done. The system does its best to meet deadlines.

### 1.3 NP-Completeness

An algorithm is a polynomial-time algorithm if on inputs of size  $n$ , its worst-case running time is  $O(n^k)$  for some constant  $k$ . Further, an algorithm can be placed into one of three classes: P, NP, and NP-complete. The class P consists of problems that are solvable in polynomial time, and the class NP consists of problems that can be verified in polynomial time. If a problem is in the class P, it also can be verified in polynomial time; therefore, we can say that  $P \subseteq NP$  [Cormen et al. 90].

A problem is in the class NP-complete if it is in NP and is as hard as any problem in NP. Garey and Johnson [Garey & Johnson 79] presented four steps to prove that a given language  $L$  is NP-complete:

1. Show that  $L$  is in NP.
2. Select a known NP-complete language  $L'$ .
3. Construct a transformation  $f$  from  $L'$  to  $L$ .
4. Prove that  $f$  is a polynomial transformation.

They also listed six basic NP-complete problems: 3-Satisfiability, 3-Dimensional Matching, Vertex Cover, Clique, Hamiltonian Circuit, and Partition.



## 1.4 NP-Complete Scheduling Problems

A scheduling problem is known to be NP-complete in its general form as well as in many restricted cases [Ullman 75]. In this section, some NP-complete scheduling problems are introduced.

### 1. Uniprocessor Scheduling with Release Times and Deadlines

Each task is independent and has a release time and deadline. This problem is a transformation from 3-PARTITION [Garey & Johnson 79]. It is NP-complete in the strong sense. It is solvable in polynomial time if the lengths of all tasks are identical, preemptions are allowed, or all release times are 0.

### 2. Multiprocessor Scheduling

Each task is independent and has an overall deadline. This problem is a transformation from PARTITION [Garey & Johnson 79]. It is NP-complete in the strong sense for arbitrary number of processors and in the normal sense for two processors. It is solvable in polynomial time if the lengths of all tasks are identical.

### 3. Precedence-Constrained Multiprocessor Scheduling

Each task is precedence-constrained and has identical length and an overall deadline. This problem is a transformation from 3-SATISFIABILITY [Ullman 75]. It is NP-complete in the normal sense for arbitrary number of processors. It is solvable in polynomial time if the number of processors is two or if the number of processors is arbitrary and precedence constraints are “forest”.

#### 4. Multiprocessor Scheduling with Individual Deadlines

Each task is precedence-constrained and has identical length and individual deadline. This problem is a transformation from VERTEX COVER [Garey & Johnson 79]. It is NP-complete in the normal sense for arbitrary number of processors. It is solvable in polynomial time if the number of processors is two or precedence constraints are “in-tree” (no task has more than one immediate successor).

Table 1.1 summarizes the complexity of scheduling problems with varying conditions. In this table, the communication cost is ignored and we assume that the target machine is fully connected.

<b>Uniprocessor Scheduling with Release Times and Deadlines</b>						
Problem	Graph	Execution Time	Number of Processors	Precedence Constraints	Deadline	Complexity
	Arbitrary	Identical	1	Yes / No	Arbitrary	Polynomial
	Arbitrary	Arbitrary	1	No	Arbitrary	NP-complete
<b>Uniprocessor Scheduling to Minimize Tardy Tasks</b>						
Problem	Graph	Execution Time	Number of Processors	Precedence Constraints	Deadline	Complexity
	Arbitrary	Arbitrary	1	No	Arbitrary	Polynomial
	Arbitrary	Arbitrary	1	Yes	Arbitrary	NP-complete
<b>Uniprocessor Scheduling to Minimize Weighted Completion Time</b>						
Problem	Graph	Execution Time	Number of Processors	Precedence Constraints	Deadline	Complexity
	Arbitrary	Arbitrary	1	“forest”	None	Polynomial
	Arbitrary	Arbitrary	1	Yes	None	NP-complete
<b>Multiprocessor Scheduling</b>						
Problem	Graph	Execution Time	Number of Processors	Precedence Constraints	Deadline	Complexity
	Arbitrary	Identical	2	No	Overall	Polynomial
	Arbitrary	Arbitrary	$\geq 2$	No	Overall	NP-complete
<b>Multiprocessor Scheduling with Precedence Constraints</b>						
Problem	Graph	Execution Time	Number of Processors	Precedence Constraints	Deadline	Complexity
	Arbitrary	Identical	2	Yes	Overall	Polynomial
	Arbitrary	Identical	$\geq 3$	Yes	Overall	NP-complete
<b>Multiprocessor Scheduling with Individual Deadlines</b>						
Problem	Graph	Execution Time	Number of Processors	Precedence Constraints	Deadline	Complexity
	Arbitrary	Identical	2	Arbitrary	Arbitrary	Polynomial
	Arbitrary	Identical	$\geq 3$	Yes	Arbitrary	NP-complete
<b>Multiprocessor Scheduling to Minimize Weighted Completion Time</b>						
Problem	Graph	Execution Time	Number of Processors	Precedence Constraints	Deadline	Complexity
	Arbitrary	Identical	2	No	None	Polynomial
	Arbitrary	Arbitrary	$\geq 3$	No	None	NP-complete

**Table 1.1 Complexity of various scheduling problems**

## Chapter 2

### Basic Terminology and Problem Definition

In this chapter, we first define basic terminology for modeling various scheduling problems. In the latter half of the chapter, we formally define our problem using the defined terminology.

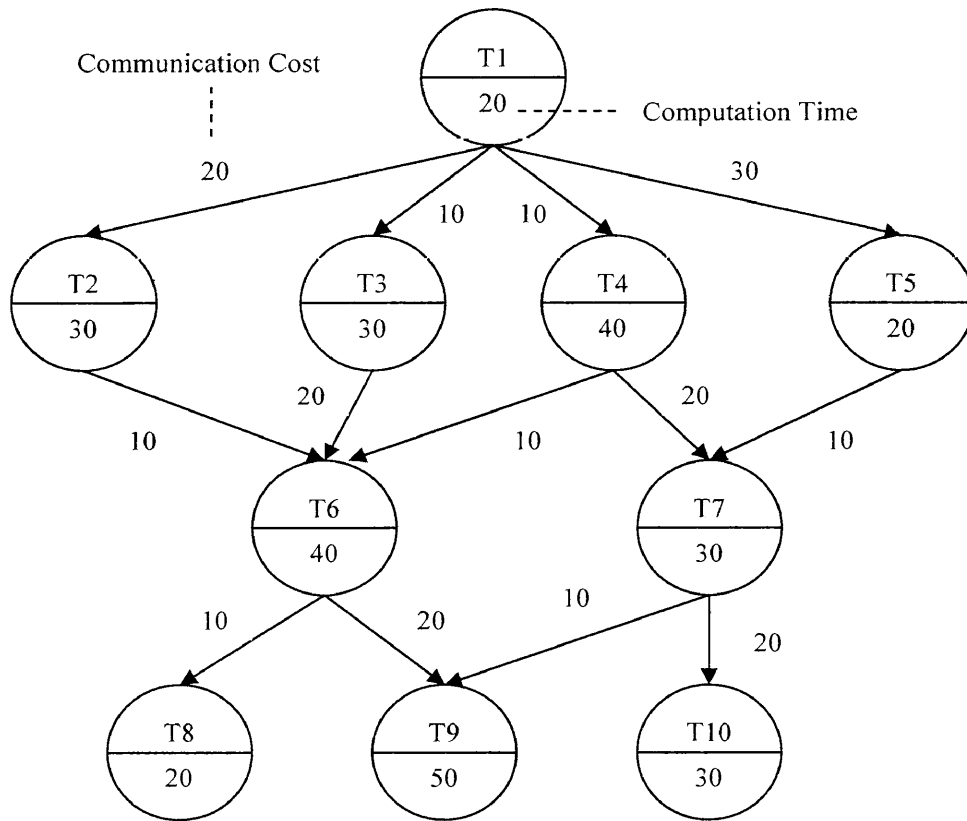
#### 2.1 Basic Terminology

##### Task Graph

A task graph is a directed acyclic graph (DAG) which is defined as  $DAG = \{V, E\}$  where  $V$  is a set of nodes representing the tasks, and  $E$  is a set of directed edges representing the communication message. Since edges in a DAG are directed, they represent the precedence constraints among the tasks. An example of DAG is shown in Figure 2.1.

##### Graph Density

The density of a graph is represented as a ratio of the number of edges and the possible maximum number of edges in the graph. The possible maximum number of edges in a graph can be defined as  $max | E | = \frac{n(n-1)}{2}$ . If the number of edges in the graph is close to  $max | E |$ , the graph is said to be dense. If the number of edges in the graph is much less than  $max | E |$ , the graph is said to be sparse.



**Figure 2.1 A task graph**

### Precedence Constraints

A precedence relation refers to the execution order among tasks. The relation  $T_1 < T_2$  implies that the computation of task  $T_2$  depends on the results of the computation of task  $T_1$ . In other words, task  $T_1$  must be completed before task  $T_2$  can be initiated, and the results of the computation of task  $T_1$  must be sent to the processor which is computing task  $T_2$ .

### **Predecessor and Successor**

If there exists an edge  $e_{ij} \in E$  between task  $T_i$  and task  $T_j$ , then  $T_i$  is a *predecessor* of  $T_j$  and  $T_j$  is a *successor* of  $T_i$ .

### **Communication Cost**

Tasks communicate with each other to achieve the overall control mission. When a communication between two tasks occurs, it also implies a precedence relation between these tasks. For example, if task  $T_i$  needs to send a message to task  $T_j$ , then the precedence relation  $T_i < T_j$  exists because task  $T_j$  cannot start running until it receives the message from task  $T_i$ .

Further, if task  $T_i$  and task  $T_j$  are assigned to the same processor, they can send and receive a message through shared memory; thus, overheads of such communication are very small. In contrast, if these two tasks are assigned to different processors, they will incur interprocessor communication which requires extra processing such as packetization and depacketization [Hou & Shin 94].

### **Task Types**

*Periodic tasks* are invoked at fixed time intervals. A periodic task is characterized by its worst-case execution time, period, and relative deadline. These attributes are usually known a priori. *Aperiodic tasks* are invoked randomly in response to environmental stimuli. Their arrival times are unknown at design time. A firm aperiodic task is characterized by its arrival time, worst-case execution time, and relative deadline. Soft

aperiodic tasks have no deadline constraints. *Sporadic tasks* are hard real-time aperiodic tasks with defined minimum inter-arrival times between two consecutive invocations. A sporadic task is characterized by its relative deadline, minimum inter-arrival time, and worst-case execution time [Isović & Fohler 00].

### **Preemptive and Non-Preemptive**

*Preemptive* means that the system will interrupt a running task to run another task, if that other task is higher priority and suddenly becomes ready to run. Advantages of preemptive scheduling include: (1) scheduling decisions can take effect as soon as the system state changes and (2) the system can take full advantage of task priorities. Disadvantages include: (1) the system requires a means to guarantee mutual exclusion and (2) Worst-Case Execution Time (WCET) analysis is very complicated.

*Non-preemptive* tasks are those that cannot be interrupted during execution by other tasks; mutual exclusion is automatically guaranteed and WCET analysis is relatively easy. A major disadvantage of non-preemptive scheduling is that the scheduling decision takes effect after a task has executed, and once a task starts executing, all other tasks will be blocked until execution is complete.

### **Schedule Types**

According to Stankovic *et al.*, “a *static scheduling* algorithm has complete knowledge regarding the task set and its constraints such as deadlines, computation times, precedence constraints, and future release times.” In a static scheduling system, all tasks

are predetermined and remain fixed while the system is being operated. A *dynamic scheduling* algorithm has complete knowledge of the currently active set of tasks but is not aware of future tasks. In a dynamic scheduling system, the scheduling of tasks is done as they arrive. As a result, the schedule changes over time. Regardless of whether the final runtime algorithm is static or dynamic, off-line scheduling should always be done in real-time systems. An example of benefits obtained from off-line scheduling is a static set of priorities to be used at run time [Stankovic *et al.* 94].

### **Homogeneous and Heterogeneous**

If a system is *homogeneous*, it means that all processors in the system are identical. Therefore, the capacity and the reliability among the processors are all the same. If a system is *heterogeneous*, each processor may have different capacity and reliability. In such a system, the total length of the resultant schedule and the total reliability of the system vary depending on how tasks are assigned to each processor.

In addition, if a task needs to receive a message from its predecessor which has been assigned to the different processor, communication cost occurs. The communication cost could be homogeneous (meaning the cost is fixed among any pair of processors) or heterogeneous (the cost varies depending on the pair).

### **Priority**

In non-real-time systems, the priority of a job changes depending on whether it is CPU-bound or I/O-bound. In real-time systems, priority assignment is related to the time



constraints associated with a task and this assignment can be either static or dynamic [Ramamritham & Stankovic 94]. In Table 2.1, we show several known scheduling algorithms and their priority assignment types.

Scheduling Algorithm	Priority Assignment
Rate-Monotonic	Static
Deadline-Monotonic	Static
Weight-Monotonic	Static
Slack-Monotonic	Static
Earliest-Deadline-First	Dynamic
Least-Laxity-First	Dynamic

**Table 2.1 Scheduling algorithm and priority assignment**

### Feasible Schedule

A schedule is said to be *feasible* if it fulfills all application constraints for a given set of tasks.

### Optimal Schedule

A scheduling algorithm is said to be *optimal* with respect to schedulability if it can always find a feasible schedule whenever any other scheduling algorithms can do so. We list two examples of optimal scheduling algorithms for uniprocessor systems.

- Liu and Layland showed that the Rate-Monotonic (RM) algorithm is optimal among all scheduling algorithms that use static priorities if the deadlines of all tasks are equal to their request periods [Liu & Layland 73].
- Leung and Whitehead showed that the Deadline-Monotonic (DM) algorithm is optimal among all scheduling algorithms that use static priorities if the deadlines of all tasks are equal to or less than their request periods [Leung & Whitehead 82].

## 2.2 Problem Definition

In this section, we describe our scheduling model which consists: (1) target machine, (2) tasks, (3) computation time and communication cost, and (4) performance metrics. There exist many kinds of scheduling problems in the literature depending on the combination of scheduling attributes. We first describe assumptions in our scheduling system.

### Assumptions

1. Tasks are aperiodic, non-preemptive, and precedence constrained.
2. Processors are computationally heterogeneous (each processor may have distinct processing speed).
3. Scheduling is static, meaning that our scheduling algorithm has complete knowledge about the tasks such as deadlines, computation times, and precedence constraints.
4. Communication costs are heterogeneous, meaning that the communication cost between any two processes may be distinct.

### Target Machine

The target machine is modeled as a set  $P = \{P_1, P_2, \dots, P_m\}$  of heterogeneous processors connected by a real-time communication network. This real-time network allows our scheduling system to remain unconcerned with the complexities of fault-tolerant message passing protocols [Sabine & Sha 94]. Processor  $P_i$  is defined as  $P_i = (T_i, S_i)$  where  $T_i$  is a set of tasks allocated on  $P_i$  and  $S_i$  is the speed of  $P_i$ . Each processor can execute one task at a time and all tasks can be executed on any processor.

## Tasks

A set of real-time tasks is represented as  $T = \{T_1, T_2, \dots, T_n\}$ . Task  $T_i$  is defined as  $T_i = (A_i, D_i, PRE_i, SUC_i, PR_i, BK_i)$ , where:

1.  $A_i$  is the *computation amount* of task  $T_i$ ;
2.  $D_i$  is the *deadline* of task  $T_i$ ;
3.  $PRE_i$  is a set of *predecessors* of task  $T_i$ , where  $T_i$  is a predecessor of  $T_j$  if there exists a precedence constraint such that  $T_i < T_j$ ;
4.  $SUC_i$  is a set of *successors* of task  $T_i$ , where  $T_j$  is a successor of  $T_i$  if there exists a precedence constraint such that  $T_i < T_j$ ;
5.  $PR_i$  is the *primary copy* of task  $T_i$ ;
6.  $BK_i$  is the *backup copy* of task  $T_i$ .

## Computation Time and Communication Cost

We assume that each task may have a distinct computation amount and each processor may have a distinct processing speed. The *computation time* of task  $T_i$  on processor  $P_j$  is calculated as

$$C_{ij} = \left\lceil \frac{A_i}{S_j} \right\rceil.$$

The *communication cost* between processor  $P_i$  and processor  $P_j$  is calculated as

$$CC_{ij} = \left\lceil \frac{M}{R_{ij}} \right\rceil, \text{ where } M \text{ is the amount of message to be sent, and } R_{ij} \text{ is the}$$

transmission rate between processor  $P_i$  and processor  $P_j$ .

## Performance Metrics

Performance metrics used to evaluate a schedule include; balancing the load, utilization of processors, minimizing the sum of completion times, minimizing schedule length, minimizing the number of processors required, and minimizing the maximum lateness [Stankovic *et al.* 94]. The need to minimize the schedule length pervades static non-real-time systems, and minimizing response times and increasing the throughput are the primary metrics in dynamic non-real-time systems [Ramamritham & Stankovic 94]. We introduce two commonly used performance metrics.

## Processor Utilization

Liu and Layland [Liu & Layland 73] defined the *processor utilization factor* to be the fraction of processor time spent in the execution of the task set.

$$U = \sum_{i=1}^n \frac{C_i}{T_i}, \text{ where } C_i \text{ is the computation time and } T_i \text{ is the request period.}$$

## Finishing Time

Hou *et al.* defined the finishing time of a schedule as

$$FT(S) = \max_{P_j \in P} \{ ftp(P_j) \},$$

where  $ftp(P_j)$  is the finishing time for the final task in processor  $P_j$  [Hou *et al.* 94].

## Chapter 3

### Previous Work

In this chapter, we introduce previous work performed by various researchers. Before delving into a variety set of scheduling algorithms, we first show the tool used to evaluate the schedulability of a given algorithm.

Scheduling algorithms are designed for two different systems: uniprocessor systems and multiprocessor systems. As a first step of introducing various algorithms, we describe three well-known scheduling algorithms originally designed for uniprocessor systems: the Rate-Monotonic (RM) algorithm, the Deadline-Monotonic (DM) algorithm, and the Earliest-Deadline-First (EDF) algorithm. Following these classic algorithms, we introduce more complicated scheduling algorithms designed for multiprocessor systems and fault-tolerant systems.

To facilitate our definition process, we use notation in Table 3.1 throughout this chapter.

Notation	Description
$C_i$	Worst-case computation time of task $\tau_i$
$T_i$	Period of task $\tau_i$
$D_i$	Deadline of task $\tau_i$
$R_i$	Worst-case response time of task $\tau_i$

**Table 3.1 Notation and description**

### 3.1 Schedulability Analysis

*Schedulability analysis* is the process of determining whether a task set can be scheduled by a given algorithm in a manner that no task will miss its deadline. Since the consequences of missing a deadline may be catastrophic in hard real-time systems, schedulability analysis plays an extremely important role. In general, schedulability analysis involves a *feasibility test* that is customized for the actual algorithm used, and it is possible to have a different type of feasibility test depending on the characteristics of the schedule. Three well-known feasibility tests are listed below:

- *Processor Utilization Analysis*: The fraction of processor time that is used for executing the task set may not exceed a given bound. This analysis is mainly used for RM and EDF scheduling on a uniprocessor.
- *Response Time Analysis*: Worst-case response time for each task is calculated and compared against the deadline of the task. This analysis is mainly used for DM scheduling on a uniprocessor.
- *Processor Demand Analysis*: The accumulated computation demand for the task set under a given time interval must not exceed the length of the interval. This analysis is mainly used for EDF scheduling on a uniprocessor.

#### Processor Utilization Analysis

Liu and Layland [Liu & Layland 73] defined the *processor utilization factor* to be the fraction of processor time spent in the execution of the task set:

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

### Response Time Analysis

The response time  $R_i$  for a task  $\tau_i$  represents the worst-case completion time of the task when execution interference from other tasks are accounted for. Joseph and Pandya [Joseph & Pandya 86] defined  $R_i$  as

$$R_i = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j, \text{ where } hp(i) \text{ is the set of tasks with higher priority than}$$

task  $\tau_i$ . Note that the equation above has  $R_i$  on both sides. To solve the equation for  $R_i$ , we need to use a recurrence relation

$$R_i^{n+1} = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j.$$

This equation either: (1) converges or (2) will grow to exceed  $D_i$  in which case the task has failed the feasibility test.

### Processor Demand Analysis

The processor demand for a task  $\tau_i$  in a given time interval  $[0, L]$  is the amount of processor time that the task needs in the interval in order to meet the deadlines that fall within the interval. Jeffay and Stone [Jeffay & Stone 93] defined the total processor demand as

$$C_p(0, L) = \sum_{i=1}^n \left\lceil \frac{L}{T_i} \right\rceil C_i.$$

If  $C_p(0, L)$  exceeds  $L$ , it is not schedulable.

## Necessary and Sufficient Conditions

To understand how the feasibility test works, we need to know two conditions: *necessary* and *sufficient*.

- A condition  $A$  is said to be *necessary* for a condition  $B$ , if (and only if) the falsity of  $A$  guarantees the falsity of  $B$ .
- A condition  $A$  is said to be *sufficient* for a condition  $B$ , if (and only if) the truth of  $A$  guarantees the truth of  $B$ .

If the feasibility test is necessary, it means that if the scheduling algorithm fails the test, it is guaranteed that a set of tasks is not schedulable. However, even if the scheduling algorithm passes the test, it does not guarantee that a set of tasks is schedulable. If the feasibility test is sufficient, it means that if the scheduling algorithm passes the test, it is guaranteed that a set of tasks is schedulable. However, even if the scheduling algorithm fails the test, it is still possible for a set of tasks to be schedulable. We summarize these relationships in Table 3.2.

Condition	Feasibility Test	
	Pass	Fail
Necessary	Undefined	Not Schedulable
Sufficient	Schedulable	Undefined
Necessary & Sufficient	Schedulable	Not Schedulable

**Table 3.2 Feasibility test**



## 3.2 Uniprocessor Systems

In this section, we introduce three scheduling algorithms designed for uniprocessor systems: the Rate-Monotonic (RM) algorithm, the Deadline-Monotonic (DM) algorithm, and the Earliest-Deadline-First (EDF) algorithm.

### 3.2.1 Rate-Monotonic (RM) Algorithm

Liu and Layland proposed the Rate-Monotonic (RM) algorithm, which is priority-driven and preemptive. They studied the scheduling problem under the following assumptions:

1. The requests for all tasks are periodic, with constant interval between requests.
2. Each task must be completed before the next request for it occurs. The deadline of a task equals the period of the task.
3. Each task is independent and given a unique priority.
4. The computation time for each task is constant and does not vary with time.
5. Any non-periodic tasks in the system are special. They are initialization or failure-recovery routines and do not have hard deadlines.

In the RM algorithm, priorities assigned to tasks are inversely proportional to the length of period. Therefore, the task with the shortest period is assigned the highest priority. A scheduling algorithm is said to be *optimal* with respect to schedulability if it can always find a feasible schedule whenever any other scheduling algorithm can do so. In this sense, the RM algorithm is optimal among all fixed priority algorithms [Liu & Layland 73].

### RM Schedulability Analysis

Liu and Layland presented a least upper bound to processor utilization in fixed priority systems. This feasible test is sufficient but not necessary.

**Theorem 3.1** For a set of  $n$  independent periodic tasks, if  $\sum_{i=1}^n \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$  holds then all tasks can be scheduled by the RM policy.

For large  $n$ , the least upper bound approaches  $\ln 2$  (0.693) meaning that as long as the CPU utilization is less than 69%, all deadlines are guaranteed to be met [Liu & Layland 73].

### Extended RM Schedulability Analysis

Lehoczky *et al.* [Lehoczky *et al.* 89] presented a necessary and sufficient feasibility test for fixed priority periodic task sets. They considered a set of  $n$  periodic tasks  $\tau_1, \tau_2, \dots, \tau_n$ . Task  $\tau_i$  has a phasing relative to 0,  $I_i$ , with  $0 \leq I_i < T_i$ . Then the expression

$$W_i(t) = \sum_{j=1}^i C_j \left\lceil \frac{t}{T_j} \right\rceil,$$

gives the cumulative demands on the processor made by these tasks over  $[0, t]$  where 0 is the worse-case phasing. They also defined three other notations to present the theorem:

$$(1) L_i(t) = \frac{W_i(t)}{t}, (2) L_i = \min_{\{0 < t \leq T_i\}} L_i(t), \text{ and } (3) L = \max_{\{1 \leq i \leq n\}} L_i.$$

**Theorem 3.2**

1. A periodic task  $\tau_i$  can be scheduled for all task phasings using the RM algorithm if and only if  $L_i \leq 1$ .
2. The entire task set can be scheduled for all task phasings using the RM algorithm if and only if  $L \leq 1$ .

Joseph and Pandya [Joseph & Pandya 86] also presented a sufficient and necessary feasibility test in terms of response-time analysis.

**Theorem 3.3** If the condition  $\forall i: R_i \leq D_i$  is true, then all tasks can be scheduled by the RM algorithm.

**Extended RM Algorithm**

The RM algorithm was first formally presented by Liu and Layland. After a long period of time, it was picked up again by Lehoczky, Sha and their colleagues [Lehoczky *et al.* 89] and extended by researchers in a variety of ways to deal with shared resources, aperiodic tasks, tasks with different importance levels, and mode changes [Ramamritham & Stankovic 94]. As a result of this work and its simplicity and flexibility, the RM algorithm has been used in a variety of critical applications such as navigation of satellites, avionics, and industrial process controls [Ghosh *et al.* 98]. One of the most prominent usages of the RM algorithm was carried out by NASA in its software for the Apollo space missions.

### 3.2.2 Deadline-Monotonic (DM) Algorithm

If deadlines of periodic tasks are less than periods, the RM algorithm is no longer optimal. Leung and Whitehead [Leung & Whitehead 82] presented the Deadline-Monotonic (DM) algorithm which generalizes the RM algorithm allowing deadlines less than periods. In the DM algorithm, priorities assigned to tasks are inversely proportional to the length of the deadline. Thus, the task with the shortest deadline is assigned the highest priority. This scheme is optimal in the sense that if any static priority scheduling algorithm can schedule a task set where task deadlines are unequal to their periods, the DM algorithm can also schedule that task set.

#### DM Schedulability Analysis

The processor utilization analysis is the same as the RM algorithm except  $T_i$  is replaced by  $D_i$ . Hence, we have the condition

$$\sum_{i=1}^n \frac{C_i}{D_i} \leq n (2^{1/n} - 1).$$

The response time analysis is also the same as the RM algorithm. We have

$$\forall i : R_i \leq D_i .$$

Audsley *et al.* [Audsley *et al.* 91] presented two sufficient but not necessary schedulability tests and a complex, sufficient and necessary schedulability test for DM task sets.

### 3.2.3 Earliest-Deadline-First (EDF) Algorithm

As Liu and Layland [Liu & Layland 73] introduced the RM algorithm, they also presented the Earliest-Deadline-First (EDF) algorithm in the same paper. In the EDF algorithm, priority is determined by how critical the process is at a given instant of time. The task whose absolute deadline is closest in time receives the highest priority. This scheme is optimal in the sense that if any dynamic priority scheduling algorithm can schedule a task set where task deadlines are equal to their periods, the EDF algorithm can also schedule that task set.

#### EDF Schedulability Analysis

Liu and Layland [Liu & Layland 73] gave a sufficient and necessary processor utilization condition.

**Theorem 3.4** If  $\sum_{i=1}^n \frac{C_i}{T_i} \leq 1$  holds then all tasks are schedulable by the EDF algorithm.

Baruah *et al.* presented another sufficient and necessary condition in terms of processor demand analysis [Baruah *et al.* 90].

**Theorem 3.5**

We define total demand in  $[0, L]$  as  $W(L) = \sum_{i=1}^n \left\lceil \frac{L}{T_i} \right\rceil C_i$ , busy period  $B_p = \min\{L \mid W(L) = L\}$ , and  $H = \text{lcm}(T_1, \dots, T_n)$ . If  $D = \{d_{i,k} \mid d_{i,k} = kT_i + D_i, d_{i,k} \leq \min(B_p, H), 1 \leq i \leq n, k \geq 0\}$ , then a set of periodic tasks with deadline less than periods is schedulable by the EDF algorithm if and only if

$$\forall L \in D, L \geq \sum_{i=1}^n \left( \left\lceil \frac{L - D_i}{T_i} \right\rceil + 1 \right) C_i.$$

We summarize periodic task scheduling algorithms and schedulability analysis in Table 3.3.

	$D_i = T_i$	$D_i \leq T_i$
<b>Static Priority</b>	<b>RM</b> Processor Utilization Approach $\sum_{i=1}^n \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$	<b>DM</b> Response Time Approach $\forall i, R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \leq D_i$
<b>Dynamic Priority</b>	<b>EDF</b> Processor Utilization Approach $\sum_{i=1}^n \frac{C_i}{T_i} \leq 1$	<b>EDF</b> Processor Demand Approach $\forall L > 0, L \geq \sum_{i=1}^n \left( \left\lceil \frac{L - D_i}{T_i} \right\rceil + 1 \right) C_i$

**Table 3.3 Schedulability analysis**

### 3.3 Multiprocessor Systems

In designing scheduling algorithms for multiprocessor systems, we can distinguish between at least two distinct approaches. Baruah and Goossens defined these approaches as (1) *partitioned scheduling* and (2) *global scheduling*. We first describe these two approaches and an additional one as follows:

- *Partitioned Scheduling*: All jobs generated by a task are required to execute on the same processor.
- *Global Scheduling*: Different jobs of an individual task may execute upon different processors.
- *Guided Search & Non-guided Search*: Depending on migration constraints, jobs generated by a task may or may not execute on more than one processor.

[Baruah & Goossens 03]

#### 3.3.1 Partitioned Scheduling

In partitioned scheduling, each processor has its own queue for ready tasks. All tasks are organized in groups, and each task group is assigned to a specific processor. Leung and Whitehead showed that the problem of deciding whether a task set is schedulable on  $m$  processors with respect to partitioned scheduling is NP-complete in the strong sense [Leung & Whitehead 82].

#### Rate-Monotonic-First-Fit (RMFF) Algorithm

Dhall and Liu generalized the RM algorithm for multiprocessor systems and presented the two partitioned scheduling heuristics called Rate-Monotonic-First-Fit (RMFF)

algorithm and Rate-Monotonic-Next-Fit (RMNF) algorithm. As mentioned earlier, Liu and Layland used  $n(2^{1/n} - 1)$  in their feasibility test, but Dhall and Liu used a different condition as below [Dhall & Liu 78]:

**Theorem 3.6** We have a set of  $n$  tasks  $\tau_1, \tau_2, \dots, \tau_n$  with periods  $T_1 \leq T_2 \leq \dots \leq T_n$ .

Let  $U = \sum_{i=1}^{n-1} \frac{C_i}{T_i} \leq (n-1)(2^{1/(n-1)} - 1)$ . Then if  $\frac{C_n}{T_n} \leq 2(1 + \frac{U}{n-1})^{-(n-1)} - 1$  holds, the set of

tasks can be scheduled by the RM algorithm.

We show the RMNF and RMFF algorithms in Figure 3.1 and Figure 3.2 respectively.

```

Sort tasks by period in non-decreasing order

i = 1 /* ith task */
j = 1 /* number of processors */

While i < n
  If task  $\tau_i$  can be scheduled on processor  $P_j$ 
    Assign  $\tau_i$  to  $P_j$ 
  Else
    Assign  $\tau_i$  to  $P_{j+1}$ 
    j = j + 1
  End-If

  i = i + 1
End-While

```

**Figure 3.1 The RMNF algorithm**



```

Sort tasks by period in non-decreasing order

i = 1      /* ith task */
m = 1      /* number of processors */

While i < n
    j = 1  /* jth processor */

    While task  $\tau_i$  is not assigned
        If  $u_i \leq 2(1 + \frac{U_j}{k_j})^{-k_j} - 1$ 
            Assign task  $\tau_i$  to  $P_j$ 
             $k_j = k_j + 1$ 
             $U_j = U_j + u_i$ 
        Else
             $j = j + 1$ 
        End-If
    End-While

     $i = i + 1$ 
End-While

```

**Figure 3.2 The RMFF algorithm**

As we can observe from Figure 3.1, the RMNF algorithm checks only the current processor to examine if a task (together with other tasks that have already been assigned to that processor) can be scheduled or not. This approach is inefficient in that it ignores the processors used earlier. The RMFF algorithm overcomes this problem by first checking those processors to which some tasks have been assigned [Oh & Son 93].

### Other RM-Based Multiprocessor Scheduling Algorithms

After the RMNF and RMFF algorithms were introduced, the RM-based scheduling algorithm was further improved by researchers. For instance, Oh and Son [Oh & Son 93] presented the Rate-Monotonic-Best-Fit (RMBF) algorithm in which the full processors are examined in a specific order. They also introduced the Rate-Monotonic-Small-Tasks (RMST) algorithm and the Rate-Monotonic-General-Tasks (RMGT) algorithm and compared the upper bounds among these algorithms as shown in Table 3.4 [Burchard *et al.* 95].

Scheme	RMNF	RMFF	RMBF	RMST	RMGT
$\lim_{N_{opt} \rightarrow \infty} \frac{N}{N_{opt}}$	2.67	2.33	2.33	$\frac{1}{1-\alpha}$	1.75

**Table 3.4 Upper bounds of RM-based scheduling algorithms**  
( $N$  is the number of processors required and  $\alpha$  is the maximal load factor)

### 3.3.2 Guided & Non-guided Search

#### 3.3.2.1 Guided Search

##### Branch-and-bound Algorithms

*Branch-and-bound* (B&B) algorithms are search-based enumeration techniques that enumerate the entire solution space implicitly. An early survey of B&B algorithms is given by Lawler and Wood [Lawler & Wood 66]. A set of solutions with a B&B algorithm is represented as a search tree. In this tree, the root node corresponds to the original problem to be solved, and each other node corresponds to a subproblem of the original problem. For each node, a set of child nodes is generated by modifying the current node, and a *lower-bound function* is used to assign a real number (called the *bound*) to each child node. If the bound is no better than the current best solution (called the *incumbent*), that entire branch is pruned; otherwise, the node will be used for further branching. A typical B&B algorithm is shown in Figure 3.3.

##### Existing B&B Algorithms for Scheduling Systems

In the literature, several researchers employed B&B algorithm for real-time scheduling systems. For instance, Xu and Parnas [Xu and Parnas 90] used it for uniprocessor real-time systems. Later, Xu provided another algorithm to deal with multiprocessor systems that are subject to precedence and exclusion constraint [Xu 93]. Peng *et al.* presented a solution to the problem of allocating periodic tasks to heterogeneous processing nodes in a distributed real-time system [Peng *et al.* 97]. For fault-tolerant real-time systems, Hou

and Shin used B&B algorithm to implicitly enumerate all possible allocations while effectively pruning unnecessary search paths [Hou & Shin 94].

```

incumbent = ∞    /* best solution so far */
S = {P0}      /* intermediate solution set */

While S ≠ Empty

    Select and remove node P from S
    /* branching */
    Branch on P to generate P1, P2, ..., Pk

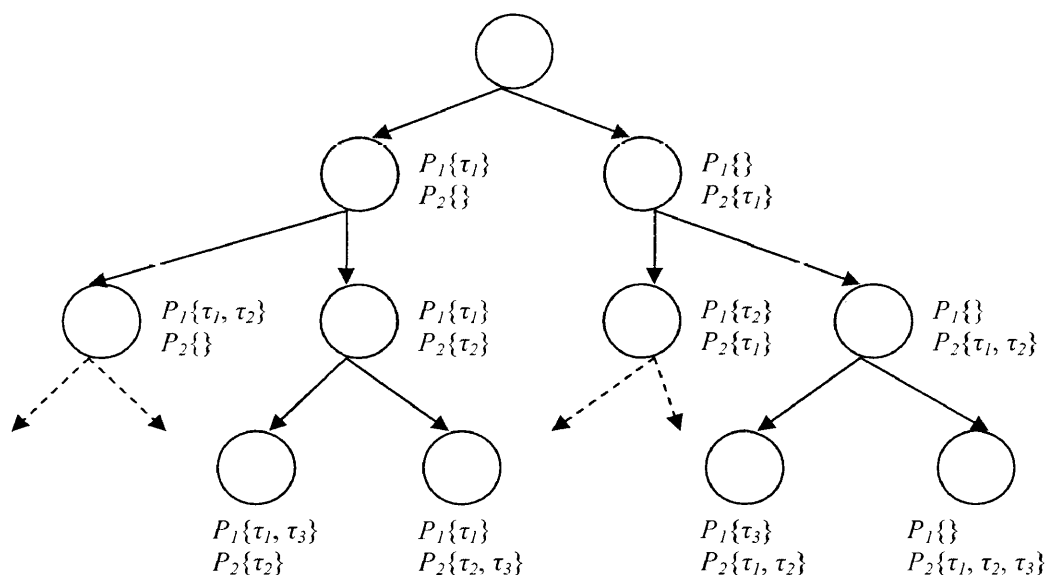
    For each Pi
        If LB(Pi) < incumbent and Pi is a feasible solution
            incumbent = LB(Pi)
            solution = Pi
        Else
            If LB(Pi) ≥ incumbent
                Prune Pi    /* bounding */
            Else
                Add Pi to S
            End-If
        End-If
    End-For

End-While

```

**Figure 3.3 The branch-and-bound algorithm**

In Figure 3.4, we show an example of the search tree. In this tree, we assume that there are two processors ( $P_1$  and  $P_2$ ) in a system and three tasks ( $\tau_1$ ,  $\tau_2$ , and  $\tau_3$ ) to be assigned.



**Figure 3.4 The search tree for the branch-and-bound algorithm**

### 3.3.2.2 Non-guided Search

In this section, we introduce three optimization methods recognized as most promising approaches to practical applications; (1) simulated annealing, (2) tabu search, and (3) genetic algorithms [Dorn 95].

#### Simulated Annealing

*Simulated annealing* (SA) is an optimization method based on ideas from statistical physics where the annealing of a solid to its ground state is simulated. Kirkpatrick *et al.* were the first who applied SA to solve combinatorial optimization problems [Kirkpatrick *et al.* 83]. In SA, each point in the search space has an *energy* associated with it, and the goal of SA is to find the point of minimum energy. The algorithm starts off at an arbitrary

point and at each point, selects a neighbor of the current point randomly. If the neighbor is better than the current point, the current point always moves to its neighbor. Otherwise, to avoid local optima, a certain probability is used to decide whether the neighbor is accepted as a new point or not. Suppose that a schedule  $S_i$  has a neighbor  $S_j$ , and the energy of these two schedules is  $E_i$  and  $E_j$  respectively. If  $E_i > E_j$ , then  $S_j$  is accepted as a new point. Otherwise, the algorithm accepts  $S_j$  with a probability of

$$P = \min \left\{ 1, e^{-\frac{E_i - E_j}{t}} \right\},$$

where  $t$  (called the *temperature*) is a positive control parameter which is decreased during the execution of the algorithm [Dorn 95].

### Existing SA Algorithms for Scheduling Systems

Simulated annealing was used by Tindell *et al.* [Tindell *et al.* 92] and DiNatale and Stankovic [DiNatale & Stankovic 95] for real-time multiprocessor scheduling systems. Tindell *et al.* only dealt with the actual task allocation problem. DiNatale and Stankovic, however, handled both the allocation of the task by SA and the scheduling problem according to local fixed-priority policies. Here, we focus on the work performed by Tindell *et al.* They first made some definitions to make SA suitable for multiprocessor systems as follows:

- The *problem space* is the set of all possible allocations for a given set of tasks and processors, and a *point* in the problem space is an assignment of tasks to processors.
- The *neighbor space* of a point is the set of all points that can be reached by moving any single task to any other processor.
- The *energy* of a point is a measure of the goodness of the allocation of the task represented by that point.

- The *energy function*, with parameters, determines the shape of the problem space.

The algorithm presented by Tindell *et al.* is shown in Figure 3.5

```

Choose starting point  $P_0$  randomly
Choose starting temperature  $C_0$ 

While solution is not found

    While thermal  $\neq$  equilibrium
         $E_p$  = energy at point  $P_n$ 
        Choose neighbor point  $T$ 
         $E_t$  = energy at point  $T$ 

        If  $E_t < E_p$ 
             $P_{n+1} = T$ 
        Else
             $x = \frac{E_p - E_t}{C_n}$ 
            If  $e^x \geq \text{random}(0, 1)$ 
                 $P_{n+1} = T$ 
            Else
                 $P_{n+1} = P_n$ 
            End-If
        End-If
    End-While

     $C_{n+1} = f(C_n)$  /* decrease temperature */

End-While

```

**Figure 3.5** The simulated annealing algorithm

## **Tabu Search**

*Tabu search* (TS) is a search method inspired to a great extent by research in cognitive science [Glover 89]. One of the key ideas is a dynamic memory that stores attributes of past solutions of the search process. In a short-term memory characteristics of recent solutions are stored to avoid reversal or cycles in search. A long-term memory is used to define phases of intensified and diversified search. If a region of the search space was examined intensively without finding better solutions, the long-term memory can switch parameters to guide the search process to different regions [Dorn 95].

### **Existing TS Algorithms for Scheduling Systems**

Thesen applied TS to multiprocessor scheduling algorithms using different tabu, local search and list management strategies [Thesen 98]. They used two types of moves to generate new solutions:

- *Transfer*: A task is assigned to another processor.
- *Interchanges*: Two tasks performed by different processors are swapped.

The major difference between these two techniques is the number of possible moves. For instance, if we make a move between two processors performing task  $\tau_1$  and task  $\tau_2$ . Then the number of transfers available is  $\tau_1 + \tau_2$ , while the number of interchanges available is  $\tau_1 \times \tau_2$ . Thesen combined transfer and interchanges to select the best approach, and their algorithm is shown in Figure 3.6.



```

 $D_{best} = \infty$  /* best duration so far */
 $TL = \{\}$  /* tabu list */

While solution is not found

     $P_i$  = processor with longest finishing time
     $D = D_i$  /* sum of duration of tasks assigned to  $P_i$  */

    If  $D < D_{best}$ 
         $D_{best} = D$  /* best duration so far */
         $S_{best} = S$  /* best solution so far */
    End-If

    Add  $S$  to  $TL$ 

    If length of  $TL >$  maximum length
        Delete oldest entry from  $TL$ 
    End-If

     $P_j$  = processor with shortest finishing time

    Find non-tabu task  $\tau_i$  in  $P_i$  and
    non-tabu task  $\tau_j$  in  $P_j$  for best solution

    If  $\tau_i$  and  $\tau_j$  are found
        Exchange  $\tau_i$  and  $\tau_j$ 
    Else
         $\tau_i$  = non-tabu task selected at random
        Move  $\tau_i$  to  $P_j$ 
    End-If

     $S$  = current solution

End-while

```

**Figure 3.6 The tabu search algorithm**

## Genetic Algorithms

*Genetic algorithms* (GAs) first proposed by Holland [Holland 75] are a class of algorithms that are based on the mechanics of natural selection and natural genetics. In GAs, a set of candidate solutions is called *population*, and the first population is initiated at random. Individuals in the population are *chromosomes* that are represented originally by bit strings. In each generation the population is affected by *genetic operators* such as the *crossover* operator, the *mutation* operator and the *selection mechanism*. The crossover operator selects a pair of individuals at random and exchanges some part of the information. The mutation operator chooses an individual randomly and alters it. A *fitness function* evaluates the goodness of individual chromosomes. The selection mechanism is usually a combination of the fitness function with some probability [Dorn 95]. More details about GAs are described in Chapter 5.

### 3.4 Fault-Tolerant Scheduling

Ghosh *et al.* extended the original Rate Monotonic Scheduling (RMS) and introduced Fault-Tolerant Rate-Monotonic Scheduling (FTRMS). They also transformed the rate-monotonic scheduling schedulability bounds into fault-tolerant bounds. [Ghosh *et al.* 98]. Bertossi *et al.* extended RMFF to a fault-tolerant multiprocessor system by presenting the Fault-Tolerant Deadline-Monotonic (FTDM) algorithm and also the Fault-Tolerant Rate-Monotonic First-Fit (FTRMFF) algorithm [Bertossi *et al.* 99]. More details about fault-tolerant scheduling are described in Chapter 4.

## Chapter 4

### Fault-Tolerant Scheduling Algorithms

In this chapter, we first describe several algorithms presented by various researchers for the real-time fault-tolerant scheduling systems. In the latter half of this chapter, we present our Real-time Fault-tolerant Scheduling (RFS) algorithm. The RFS algorithm is a prototype and is modified in Chapter 6 so that it incorporates the functions used in genetic algorithms.

#### 4.1 Introduction

A system is *fault-tolerant* if it produces correct results even in the presence of faults. Due to the critical nature of tasks in hard real-time systems, it is essential that every task completes its execution before its deadline even in the presence of processor failures. This makes fault tolerance a necessary requirement of hard real-time systems [Ghosh *et al.* 97]. There are three kinds of faults to be tolerated as shown below. A study showed that transient faults are 30 times more frequent than permanent faults. Another study concluded that 83% of all faults are either transient or intermittent [Ghosh *et al.* 98].

- *Permanent Faults*: These are caused by the total failure of a component, which might not ever (or for a long time) work correctly again. Permanent faults are typically tolerated by using hardware redundancy.
- *Transient Faults*: These are temporary malfunctions of the computing unit or any other associated components. Since error caused by transient faults may exist only for a short period of time, it is very hard to detect them.
- *Intermittent Faults*: These are repeated occurrences of transient faults.

## 4.2 Fault-Tolerance

A fault-tolerance is realized by using redundancy. In [Jalote 98], Jalote stated that “Redundancy is the key to supporting fault tolerance: there can be no fault tolerance without redundancy” and defined redundancy as “those parts of the system that are not needed for the correct functioning of the system, if no fault tolerance is to be supported.”

There are three kinds of redundancy in terms of fault-tolerance:

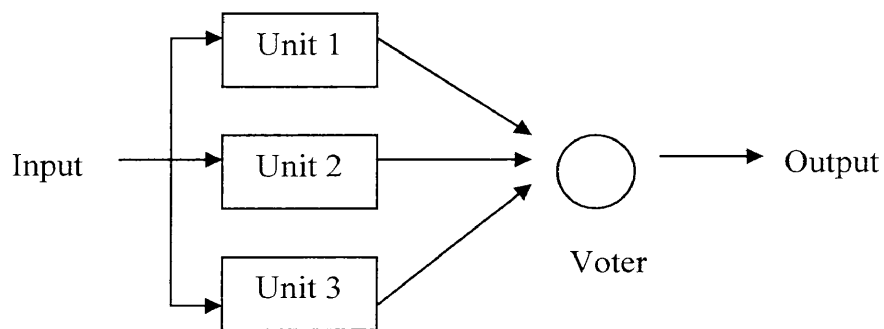
- *Hardware Redundancy*: It contains the hardware components that are added to the system to support fault-tolerance.
- *Software Redundancy*: It includes all programs and instructions that are employed for supporting fault-tolerance.
- *Time Redundancy*: It requires extra time for performing tasks for fault-tolerance.

In a multiprocessor scheduling system, fault-tolerance can be provided by either hardware redundancy or software redundancy. In either case, it is necessary to assign multiple copies of tasks to different processors. Manimaran and Murthy introduced four different techniques that have evolved for fault-tolerant scheduling of real-time tasks as below [Manimaran & Murthy 98]:

- *Triple Modular Redundancy (TMR) Model*
- *Primary Backup (PB) Model*
- *Imprecise Computational (IC) Model*
- *(m, k)-firm Deadline Model*

### 4.3 Triple Modular Redundancy (TMR) Approach

The concept of TMR was originally suggested by Von Neumann. In the TMR approach, three hardware units work in parallel, and three versions of a task are executed on each unit. The outputs from these three units are given to the voting element, and it delivers the majority vote as output. Figure 4.1 depicts this process. As we can infer, the TMR approach can completely mask the failure of one unit at a given time, but if the failure of two units occurs at the same time, the output may not be correct. However, if these errors are compensating in nature, the TMR mechanism may be able to handle “double failure” [Jalote 98].



**Figure 4.1 Triple Modular Redundancy (TMR)**

#### 4.4 Primary/Backup (PB) Approach

##### Replica (Active Replication) and De-allocating Backup (Passive Replication)

Sabine and Sha [Sabine & Sha 94] defined two types of backups as below:

1. *Replica*: A replica is a copy of a task which is scheduled in case the primary copy of the task fails to execute correctly. In the event of such a failure, the results of the backup may be used instead of those of the primary.
2. *De-allocating Backup*: A de-allocating backup is a copy of a task scheduled to run after the primary copy finishes which executes only if the primary fails.

The major differences between replicas and de-allocating backups are; (1) replicas are always executed even if no failure occurs, but de-allocating backups are executed only when their corresponding primary copies fail, and (2) replicas can be executed simultaneously with their corresponding primary copies, but de-allocating backups cannot be executed before the completion of their corresponding primary copies.

##### Backup De-allocation and Backup Overloading

In addition to two types of backups, we also have two kinds of fault-tolerant techniques that can take advantage of these two types of backups. With an observation that resources reserved for backup copies could be re-utilized, Ghosh *et al.* defined two techniques used for fault-tolerant systems [Ghosh *et al.* 97]:

- *Backup De-allocation*: This technique is based on the reclamation of resources reserved for backup tasks when the corresponding primaries complete successfully.
- *Backup Overloading*: This technique assigns more than one backup in the same time slot on the same processor (overlapping of multiple backups).

#### 4.5 Imprecise Computational (IC) Model

The IC model has been proposed as a way to handle transient overload and to enhance fault tolerance of real-time systems. In the IC model, a task is divided into *mandatory* and *optional* parts. Although the mandatory part must be completed before the task's deadline, optional part can be skipped without causing intolerable timing faults. Under normal operating conditions, the optional part is completed, and the result of the task has desired quality. We call this result *precise*. If the optional part, or a part of it, is skipped in order to conserve system resources, the result of the task is called *imprecise* [Liu *et al.* 94].

#### 4.6 $(m, k)$ -firm Deadline Model

In  $(m, k)$ -firm deadline model, a periodic task is said to have an  $(m, k)$ -firm guarantee requirement if it is adequate to meet the deadlines of  $m$  out of any  $k$  consecutive instances of the task where  $m$  and  $k$  are two positive integers with  $m \leq k$ . The main advantage of this guarantee model is that one can represent a wide range of tolerance to deadline misses by properly choosing the values of  $m$  and  $k$ . The traditional hard deadline requirement can be represented as  $(1, 1)$ -firm guarantee requirement and a soft deadline requirement of a bound on the fraction of deadline misses can be approximated by picking a large value for  $k$  and choosing  $m$  such that  $\frac{m}{k}$  equals the desired fraction [Ramanathan 97].

## 4.7 Prototype Scheduler

In this section, we present our Real-time Fault-tolerant Scheduling (RFS) algorithm as a prototype. In Chapter 6, we improve this prototype by incorporating the functions from genetic algorithms into the RFS algorithm.

A schedule without fault-tolerance is called a *primary schedule*. As mentioned in this chapter, there are two types of backup copies: replica and de-allocating backup, and also two types of techniques for fault-tolerance: backup de-allocation and backup overloading. This prototype scheduler combines these types of backup copies and techniques and can create five different types of schedule: (1) primary, (2) replication only, (3) de-allocation only, (4) backup overloading, and (5) replication and backup overloading. We explain each of these schedules in the following section.

### 4.7.1 Overview

In this section, we show an overview for each type of schedule. We use the notations  $P_i$ ,  $R_i$ , and  $B_i$  to denote the primary copy, replica, and backup copy of task  $T_i$  respectively in the following sections.



### Replication Only

This technique uses only replicas. A replica, denoted by  $R_i$ , is a copy of  $P_i$ , which executes independently, whether or not  $P_i$  completes successfully. The result from  $R_i$  may be used in place of those from  $P_i$ . This situation is depicted in Figure 4.2.

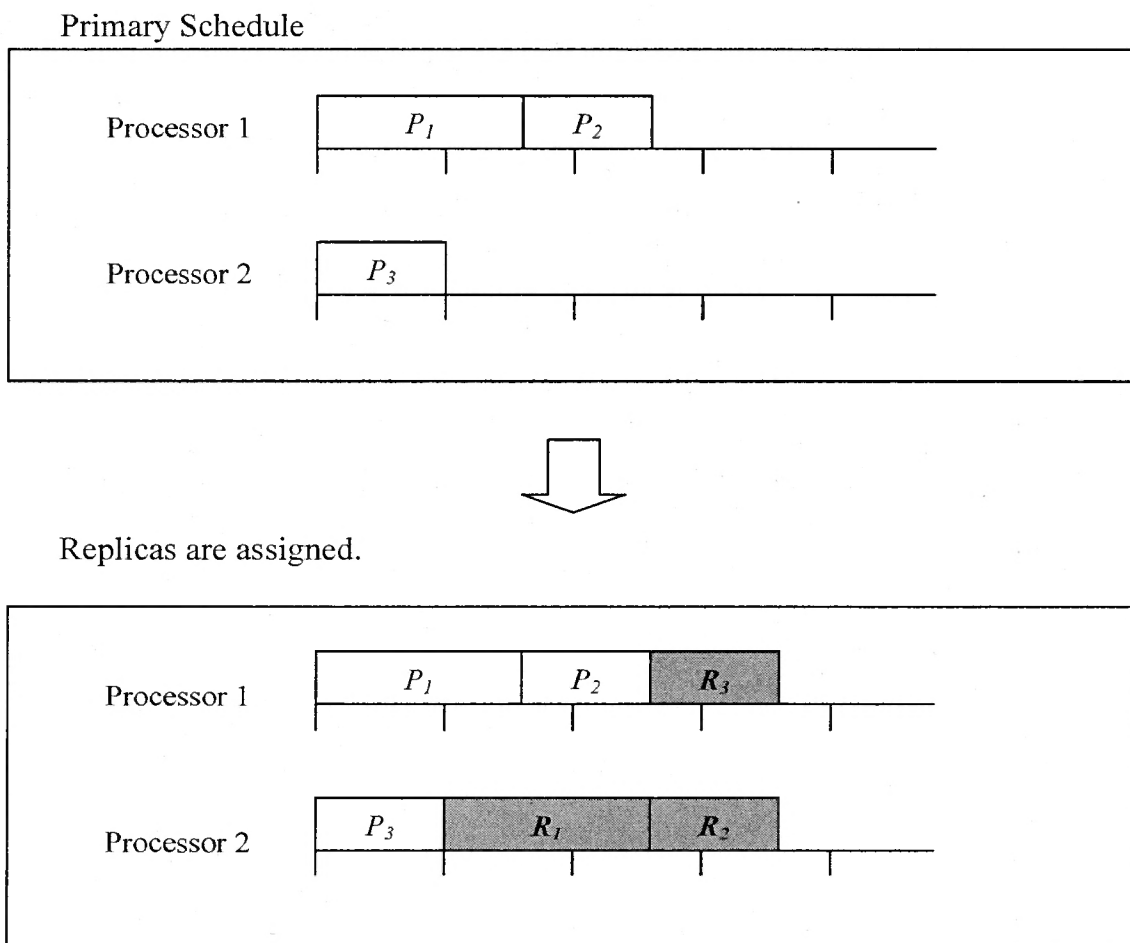


Figure 4.2 Replication only

### De-allocation Only

This technique uses only de-allocating backups. A de-allocating backup, denoted by  $B_i$ , is a copy of  $P_i$ , which is executed only if  $P_i$  fails. For  $B_i$  to know the result of  $P_i$ , there is a message called *de-allocation message*, which is originating from a primary copy  $P_i$  to its de-allocating backup  $B_i$ , indicating whether or not  $P_i$  executed successfully. This process is illustrated in Figure 4.3.

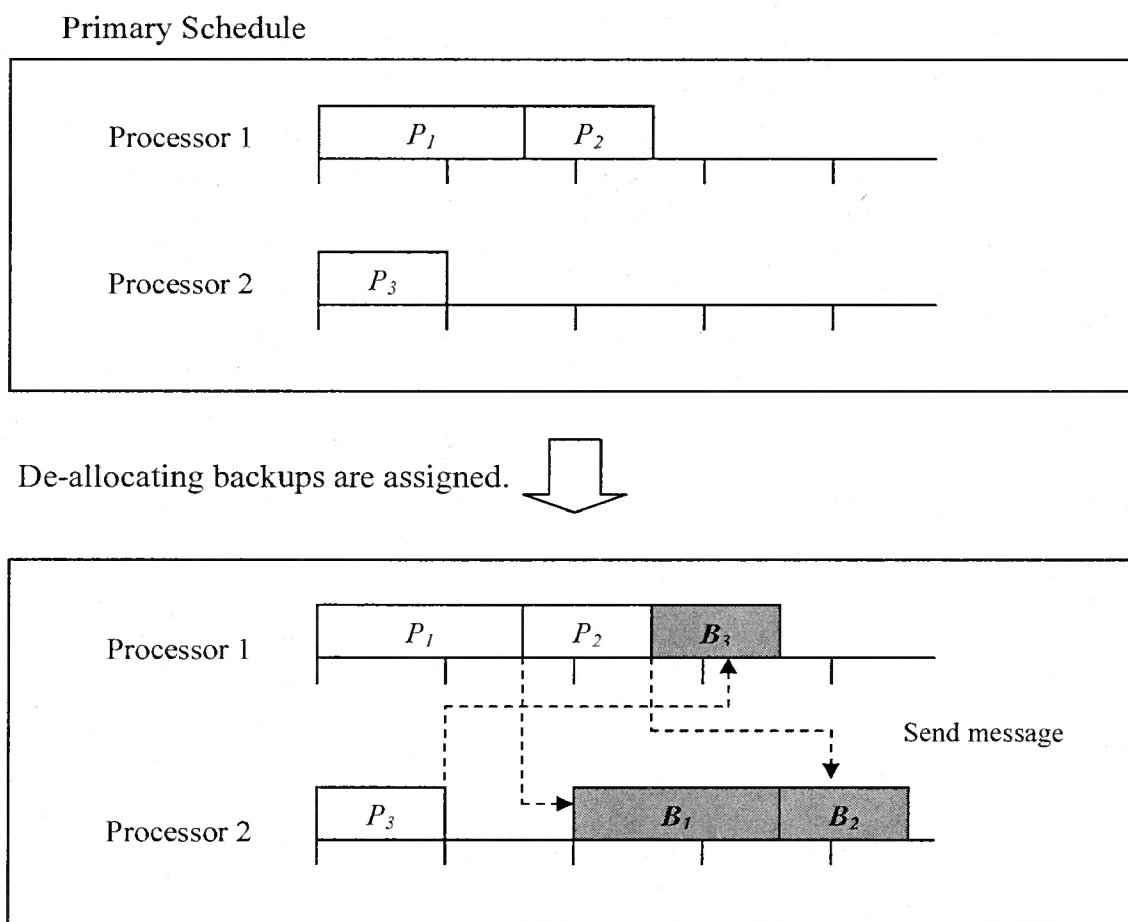
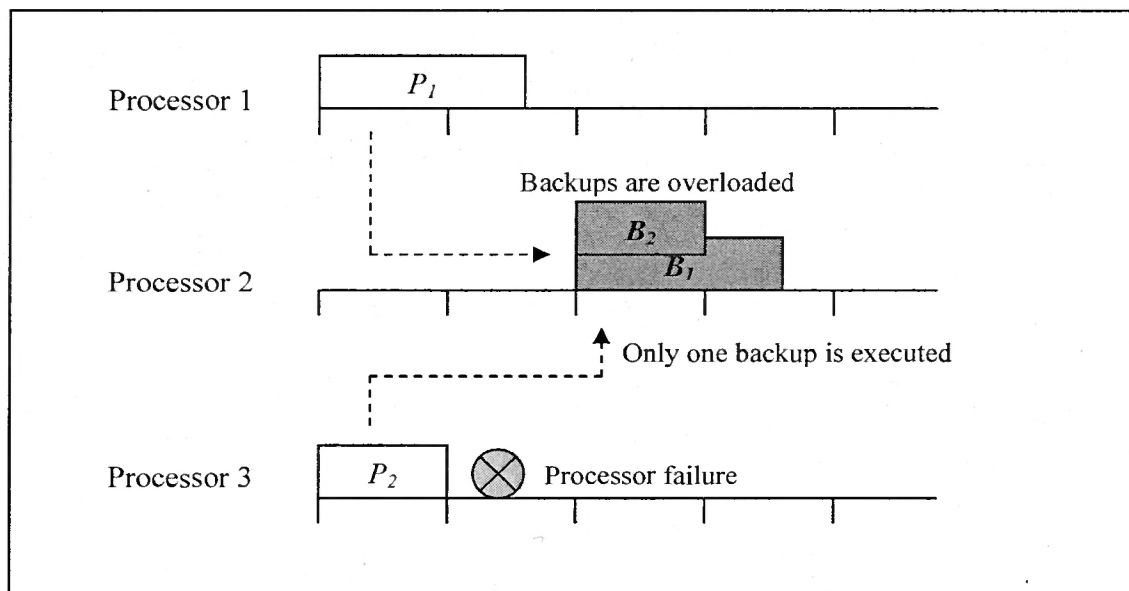


Figure 4.3 De-allocation only

### Backup Overloading

In backup overloading technique, more than one backup copy could be assigned to the same time slot so that it conserves system resources effectively. We can observe how this technique is possible using an example. Suppose that we have two primary copies;  $P_1$  and  $P_2$ , and each primary copy has corresponding backup copy  $B_1$  and  $B_2$ . If  $P_1$  is assigned to the first processor and  $P_2$  is assigned to the second processor, then  $B_1$  and  $B_2$  can be overloaded on the third processor. This is because our objective is to tolerate one processor failure. If two primary copies are assigned to the different processors, only one backup copy may need to be executed in case of processor failure. This is illustrated in Figure 4.4.



**Figure 4.4 Backup overloading**

### Replication and Backup Overloading

This is a mixture of replication and backup overloading. The objective of this technique is to combine the benefits from these two techniques. The algorithm proceeds as follows; (1) it first tries to schedule a task using backup overloading technique, and if it fails, (2) it tries to schedule a task using replication technique as a second attempt. An example is shown in Figure 4.5

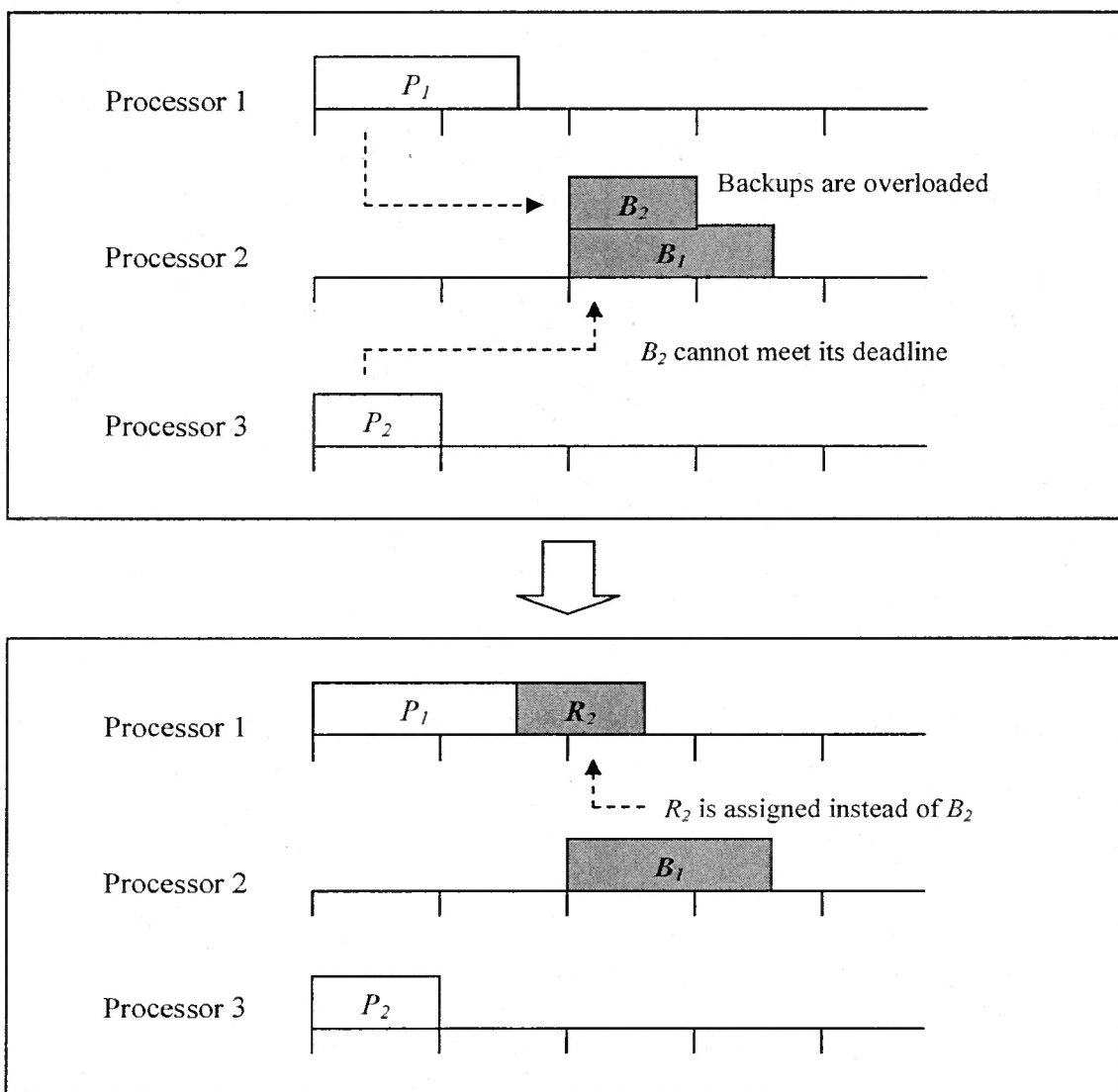


Figure 4.5 Replication and backup overloading

### 4.7.2 Task Allocation

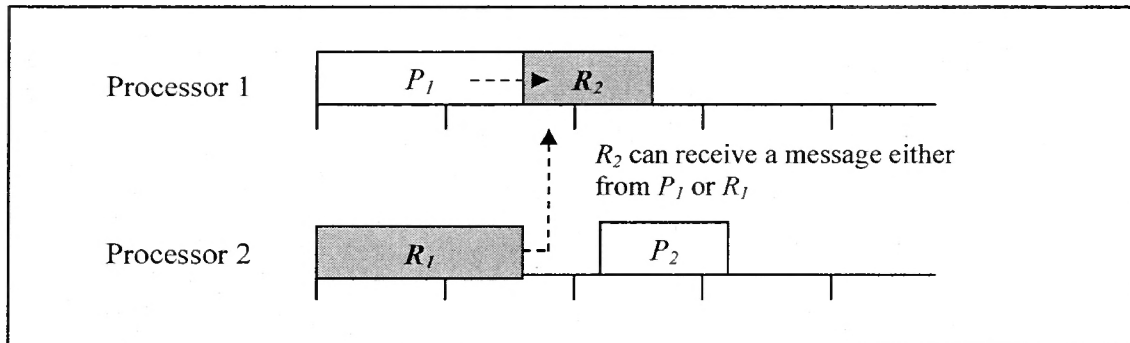
In this section, we describe several restrictions on the task allocation; especially in terms of selecting the processor and finding the earliest time slot in where a task is to be assigned, for each technique. Note that in this simulation, we assume that there is no more than one predecessor for each task.

#### Replication Only

In this technique, there is one restriction in selecting the processor for assigning backup copies.

*A primary copy and its backup copy must be assigned to the different processor.*

One of the goals of the RFS algorithm is to tolerate one processor failure: transient or permanent. If both a primary and a backup copy of the task are assigned to the same processor, they would never be executed if that processor fails.



**Figure 4.6 Replication only (task allocation)**

As shown in Figure 4.6,  $R_2$  can receive a message either from  $P_1$  or  $R_1$ . It implies that if no failure occurs, communication cost can be excluded from consideration. However, we still need to consider the case that either  $P_1$  or  $R_1$  fails, thus the earliest start time for a replica is:

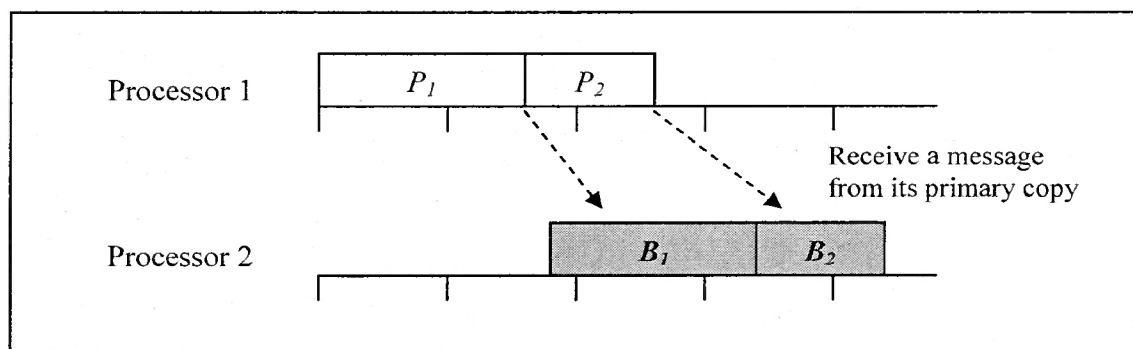
$$\max(\text{Primary predecessor's end time} + \text{communication cost}, \\ \text{Replica predecessor's end time} + \text{communication cost}).$$

### De-allocation Only

In this technique, in addition to the one introduced in the replication only technique, there is one additional restriction in finding the earliest start time.

*A backup copy must be assigned after the completion of its primary copy.*

A backup copy is executed only when its primary copy fails. If the backup copy is assigned prior to its primary copy, it would never know whether its primary copy succeeded or failed.



**Figure 4.7 De-allocation only (task allocation)**

A de-allocating backup always needs to receive a message from its primary copy, indicating if the primary copy is successful or not. The earliest start time for a de-allocating backup is:

Primary copy's end time + communication cost.

### Backup Overloading

In the RFS algorithm, almost the same approach to the backup de-allocation technique is used to find the earliest start time for backup copies in the backup overloading technique. However, there is one more additional constraint.

*The backup copies of task  $T_i$  and  $T_j$  can be overloaded if and only if their primary copies are assigned to the different processors.*

Suppose that both primary copies are assigned to the same processor, and that processor fails. In this case, both of their backup copies must be executed; therefore, they cannot be overloaded.

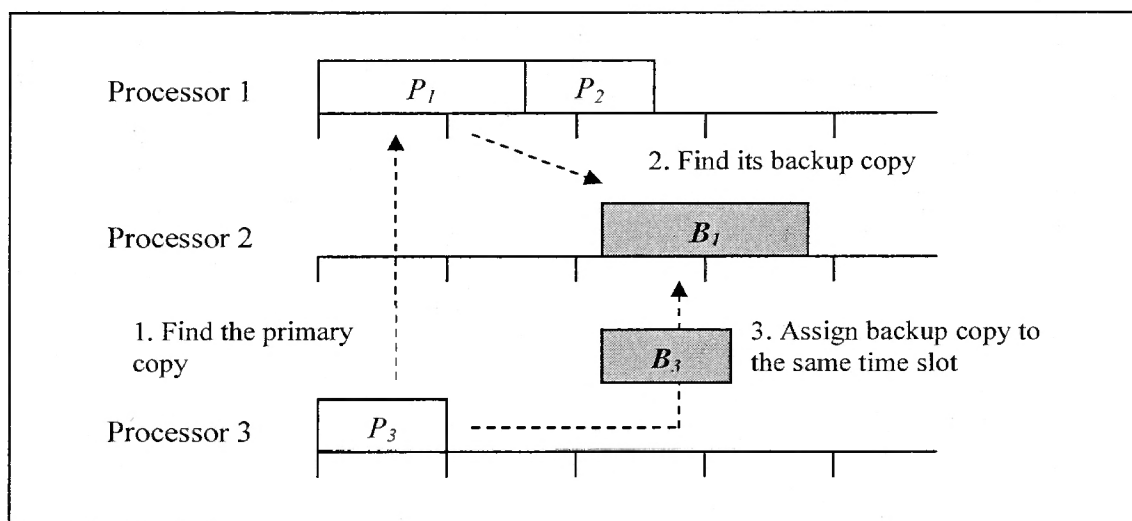


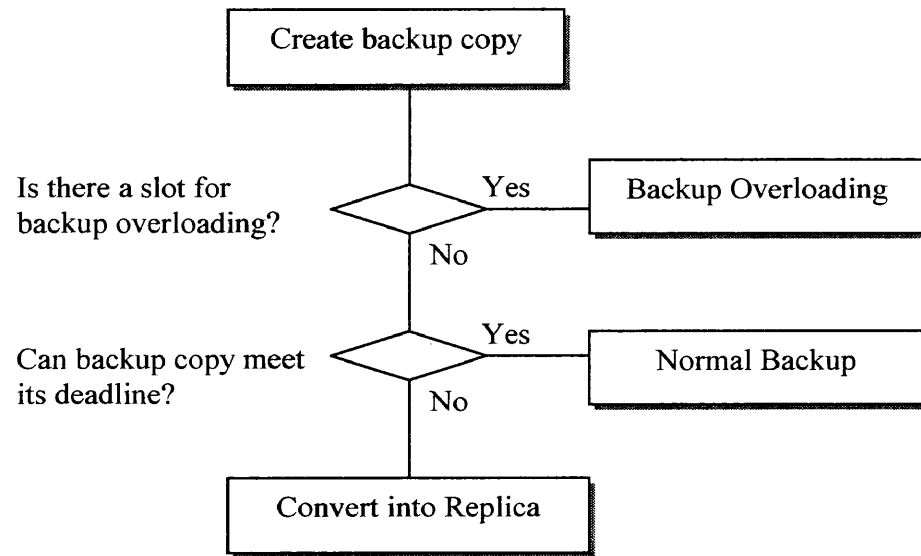
Figure 4.8 Backup overloading (task allocation)

In Figure 4.8, the backup copy of  $P_3$  is to be scheduled. We first search the primary copy which is scheduled at the similar time slot to  $P_3$ , and in this case, it is  $P_1$ . If we fail to find such a primary copy, backup overloading may not be used. Consequently, the earliest start time for the backup copy should be found in the same manner as the de-allocation only technique. After finding the primary copy, we locate its backup copy and schedule the backup copy of  $P_3$  at the same time slot as the backup copy of  $P_1$ , which results in backup overloading.

### **Replication and Backup Overloading**

The replication and backup overloading technique, as its name implies, is a combination of two techniques. First, this algorithm tries to find the slot for backup overloading. If backup overloading is not possible, it tries backup de-allocation as a second attempt. Finally, if the de-allocating backup cannot meet its deadline, it is converted into a replica since the replica has a better chance to meet its deadline than the de-allocating backup. This process is illustrated in Figure 4.9.





**Figure 4.9 Replication and backup overloading (task allocation)**

### 4.7.3 The RFS Algorithm in Pseudocode

The basic algorithms for replication only, de-allocation only, backup overloading, and replication and backup overloading are similar to each other. The major differences among these algorithms come from the search for the earliest empty slot described in the previous section.

```
Put all tasks into the queue
Sort the queue by deadline

Get processor list

/* scheduling starts */

For each task in the queue
    Create a replica/de-allocating backup
    For each processor
        Search the earliest empty slot
        If Not found
            continue
        End-If
        Create the schedule
        Evaluate the schedule
    End-For
    Determine the best schedule
End-For
```

**Figure 4.10** The RFS algorithm

#### 4.7.4 Performance Measures

The RFS algorithm can create four different types of fault-tolerant schedule. To analyze the performance of each schedule, we need some measurements. For this purpose, the RFS algorithm generates some statistical information: processor usage, total number of un-scheduled tasks, total number of tasks missed deadlines, and total communication cost.

##### Processor Usage

The processor usage identifies how much of processor's capability is used, and it is calculated by the formula:

$$PU = \text{Total length of tasks on the processor} / \text{Schedule Length} * 100.$$

The higher the processor usage is, the lesser the idle time for the processor is. We may want to achieve high processor usage to use the processors efficiently.

##### Total Number of Un-scheduled Tasks

The total number of un-scheduled tasks is the number of tasks that could not be scheduled on any processor. There are two reasons to explain why it happens.

1. All processors that can schedule the task have 100% usage. The total number of tasks exceeds the capabilities of all processors. To solve this problem, it is necessary to increase the number of processors.
2. The application is hard real-time, and the task could not meet its deadline. In a hard real time environment, each task must meet its deadline; otherwise, the schedule for the task would not be created. To solve this problem, it may be necessary to increase the number of processors, or the search algorithm should

be carefully designed so that its primary concern is how to make each single task meet its deadline.

### **Total Number of Tasks Missed Deadlines**

The total number of tasks missed deadlines is the number of tasks that could not meet their deadlines. This number depends on the number of processors and also the algorithm being used. To reduce this number, it may be necessary to increase the number of processors, or a better algorithm should be employed.

### **Total Communication Cost**

The total communication cost is the total amount of cost for communications among all tasks. In the RFS algorithm, there are two kinds of communications.

1. Communication between a task and its successor
2. Communication between a primary copy and its backup copy

The communication cost is generated if a sender and a receiver are scheduled on the different processors. There may be the case where the communication cost becomes a critical issue because as the total communication cost increases, the traffic on the communication bus gets busy. In such a case, there is a high probability to have a conflict in the network.

### 4.7.5 Simulator Interface

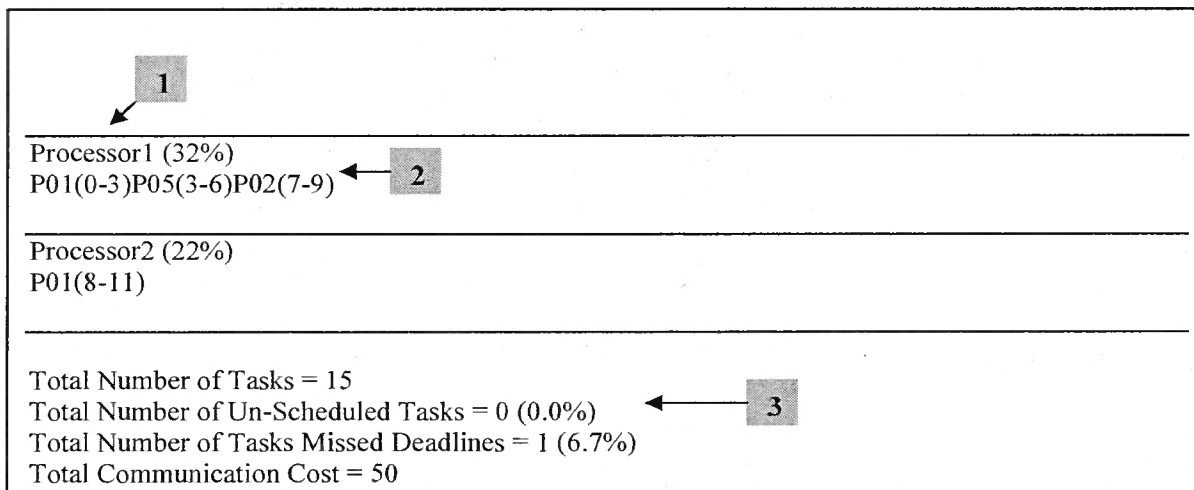
In this section, we describe the interface of the RFS algorithm.

```

C:\Documents and Settings\yhashimoto\Desktop\fault tolerant\newproject\Debug\schedule.exe
*****
* Primary Schedule *
*****
-----
Processor1< 32%>
P01<0-3>P05<3-6>P02<7-9>P04<9-11>P01<16-19>P05<19-22>
-----
Processor2< 22%>
P01<8-11>P05<11-14>P02<14-16>P03<16-19>
-----
Processor3< 18%>
P02<0-2>P04<2-4>P03<8-11>P04<16-18>
-----
Processor4< 6%>
P03<0-3>
-----
Processor5< 0%>
-----

Total Number of Tasks = 15
Total Number of Un-Scheduled Tasks = 0 (0.0%)
Total Number of Tasks Missed Deadlines = 1 (6.7%)
Total Communication Cost = 50

```



**Figure 4.11 The RFS interface**

1. Processor 1 (32%): This part identifies the processor and its usage. For example, in the figure above, *Processor 1* uses 32 percent of its capacity.
2. P01(0-3): This part identifies the task scheduled on the processor and the time interval of the schedule. For example, *P01* is the name of the primary copy, and it is scheduled on the time interval between 0 and 3. Note that there are three primary

copies named *POI*, and this is because three primary copies, namely parent, child, and grandchild are created from a task.

3. The last part shows some statistical information; the total number of tasks created, the total number of tasks that could not be scheduled for some reason, the total number of tasks that could not meet their deadlines, and the total communication cost.

#### 4.7.6 Simulation Results

In this section, we show the simulation results from five different types of schedule, (1) primary, (2) replication only, (3) de-allocation only, (4) backup overloading, and (5) replication and backup overloading using four performance measures, (1) processor usage, (2) total number of un-scheduled tasks, (3) total number of tasks missed deadlines, and (4) total communication cost.

#### Processor Usage

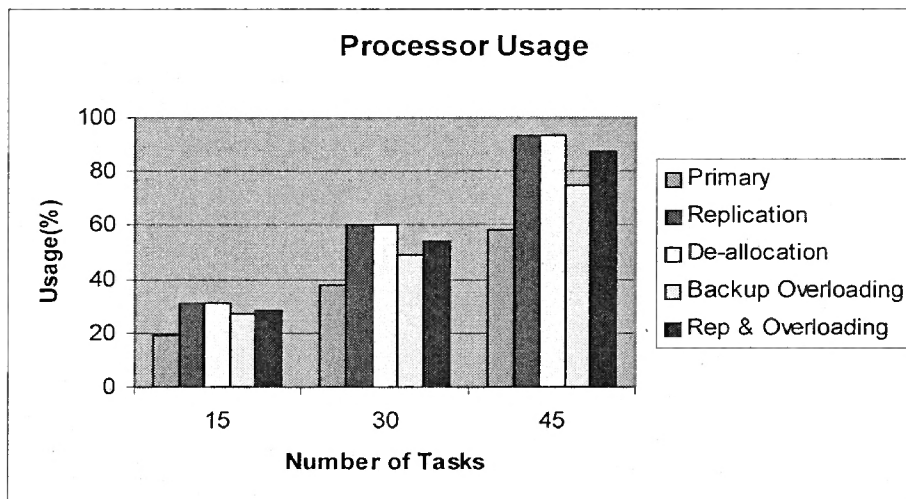


Figure 4.12 Processor usage

As Figure 4.12 shows, *replication only* and *de-allocation only* have the highest processor usage. This is because that these two algorithms schedule replica/backup in the same

manner as the primary copy. In general, they double the processor usage. Although the number of tasks is the same for all fault-tolerant algorithms, *backup overloading* has the lowest processor usage because some backup copies are scheduled at the same time slots. If the number of tasks is relatively large, *backup overloading* could reduce the processor usage. *Replication and backup overloading* is ranked between *replication only* and *backup overloading* because it has both replicas and overloaded backup copies.

### Total Number of Un-scheduled Tasks

Since the number of processors is ample, the total number of un-scheduled tasks is zero for all algorithms.

### Total Number of Tasks Missed Deadlines

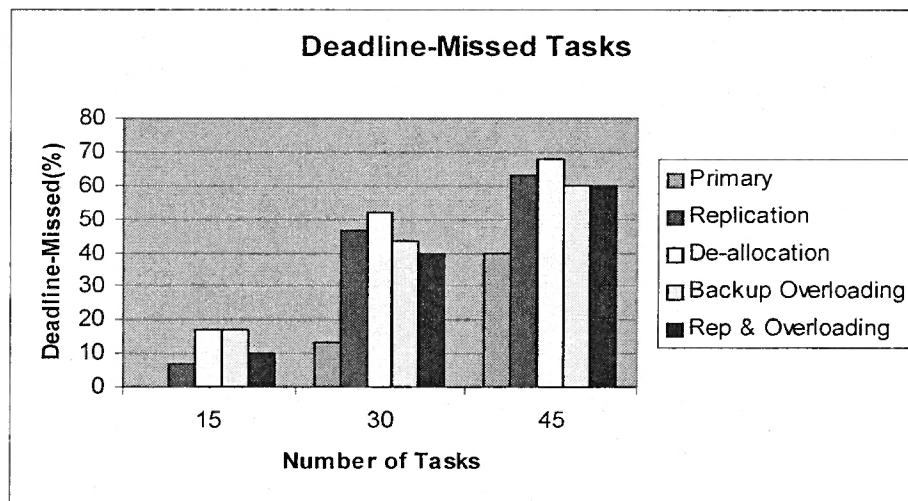
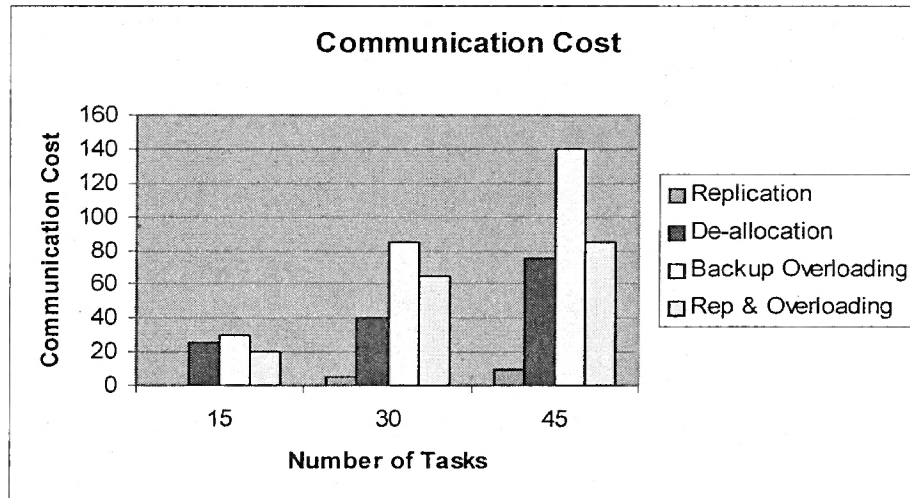


Figure 4.13 Total number of tasks missed deadlines

*De-allocation only* has the highest number of tasks that missed their deadlines. This result comes from its characteristic, which is the earliest start time for a backup copy is a summation of its primary copy's end time and the communication cost. In general, this characteristic has a high probability to cause the missing of deadlines. In contrast, a replica can receive a message either from its primary predecessor or from its replica predecessor. As a result, it has lower number of tasks that missed their deadlines. *Backup overloading* could overcome the drawback of *de-allocation only* by scheduling more than one backup copy at the same time slot. *Replication and backup overloading* has both advantages of these two algorithms, and therefore, it may achieve the best performance.

### Total Communication Cost



**Figure 4.14 Total communication cost**

Since a replica can choose lower communication cost from its primary predecessor or replica predecessor, *replication only* has the lowest communication cost among all fault-tolerant schedules. In contrast, *backup overloading* is quite high in communication cost.



This is because there is a relatively high possibility that the slot to be overloaded could be found on the processor which causes high communication cost.

#### 4.7.7 Summary

Schedule Type	Processor Usage	Un-scheduled Tasks	Deadline-Missed Tasks	Communication Cost
Replication Only	High	High	Middle	Low
De-allocation Only	High	High	High	Middle
Backup Overloading	Low	Middle	Low	High
Replication and Backup Overloading	Middle	Low	Low	Middle

**Table 4.1 Summary for the RFS algorithm**

- *Replication Only*: It can be characterized with a high processor usage and a low communication cost. If the number of tasks is relatively small, this algorithm works well.
- *De-allocation Only*: It can be characterized with a high processor usage and the number of tasks that might miss their deadlines. If the number of tasks is relatively small and deadlines are loose, this algorithm works well.
- *Backup Overloading*: It can be characterized with a low processor usage and a high communication cost. If the number of tasks is relatively large, this algorithm might be chosen instead of de-allocation only.
- *Replication and Backup Overloading*: It has both advantages of *replication only* and *backup overloading*, thus this algorithm may have the highest performance in general cases.

## Chapter 5

# Genetic Algorithms

In this chapter, we introduce the basic ideas behind genetic algorithms (GAs) and several genetic algorithms especially designed for multiprocessor scheduling systems.

### 5.1 Introduction

A genetic algorithm is a search algorithm which is based on the principles of evolution and natural genetics. Before introducing GAs in depth, we first look at the basic ideas from biology used in GAs. Fang listed the essence of *The Origin of Species*, which was written by Charles Darwin in 1859, as below:

1. Each individual tends to pass on its traits to its offspring.
2. Nevertheless, nature produces individuals with different traits.
3. The fittest individuals – those with the most favorable traits – tend to have more offspring than do those with unfavorable traits. This drives the population as a whole towards favorable traits.
4. Over a long period, variation can accumulate, producing entirely new species whose traits make them especially suited to particular ecological niches.

Taken from [Fang 94]

Each entity in the population has genetic information which is called a *chromosome* (also sometimes called a *genome*), and a chromosome is made up of *genes*. A gene is the unit of inheritance. From the list above, we can observe that the parents with favorable genes are more likely to be selected from the population and produce their children. The

children's genes originally come from their parents, and it is likely that the children inherit favorable genes from their parents. As a result of this process, the favorable genes in the new population also have a high probability to be selected and reproduced. This evolutionary cycle continues from generation to generation, and the population will gradually adapt optimally or near optimally to the environment [Fang 94].

## 5.2 Overview

GAs were first introduced by Holland [Holland 75] to study the adaptive process of natural systems and to develop artificial systems. GAs use a direct analogy to natural behavior such as *population*, *chromosome*, *reproduction*, *cross-breeding*, and *mutation* [Amphlett & Bull 96]. A traditional GA is shown in Figure 5.1, which was originally presented by Beasley *et al.* [Beasley *et al.* 93].

```
Generate initial population
Compute fitness of each individual

While population has not converged

    For population_size / 2
        Select two individuals from old generation
        Recombine them to give two offspring
        Compute fitness of the two offspring
        Insert offspring into new generation

    End-For
End-While
```

**Figure 5.1 A traditional genetic algorithm**

A GA consists of a string representation of the nodes in the search space, a set of genetic operators for generating new search nodes, a fitness function to evaluate the search nodes, and a stochastic assignment to control the genetic operators. In general, a GA proceeds in the following way:

1. *Initialization*: A set of initial individuals is randomly generated.
2. *Evaluation*: The fitness of every individual in the current population is calculated according to the predefined problem-specific measure, which is called the *fitness function*.
3. *Selection*: Select pairs of individuals from the current population for cross-breeding. The selection may be performed randomly or stochastically based on the fitness of each solution.
4. *Cross-Breeding*: Produce offspring from the selected individuals using genetic operators known as *crossover* and *mutation*.
  - *Crossover*: A new individual is produced by exchanging genes of parent chromosomes.
  - *Mutation*: A new individual is produced by altering the selected chromosome.
5. *Reproduction*: Some of the individuals in the current population may be replaced with new individuals produced in the cross-breeding step, depending on their fitness. As a result, new population becomes the current generation.
6. Repeat steps 2 through 5 until the algorithm converges.

### **Differences between Genetic Algorithms and Normal Optimization**

GAs differ from normal optimization and search procedures in the following ways [Goldberg 89].

1. GAs work with a coding of the parameter set rather than parameters themselves.

2. GAs search from a population of search nodes instead of from a single one.
3. GAs use probabilistic transition rules rather than deterministic rules.
4. GAs use payoff information instead of derivatives or other auxiliary knowledge.

### 5.3 Chromosome Encoding

Traditionally, chromosomes in GAs are encoded in binary as strings of ones and zeros as shown in Figure 5.2. This encoding is suitable for computer systems since all computers store everything as bit strings, thus any solution would eventually boil down to a string of ones and zeros. The permutation encoding contains a sequence of numbers (Figure 5.3). It can be used in ordering problems, such as the traveling salesmen problem or a task ordering problem. When it is necessary to have more complicated values such as real numbers or characters, the value encoding may be used (Figure 5.4).

Chromosome 1	0	0	1	0	0	1	0
Chromosome 2	1	1	1	1	1	1	1

**Figure 5.2 Binary encoding**

Chromosome 1	5	2	0	9	3	1	7
Chromosome 2	1	8	3	7	2	5	4

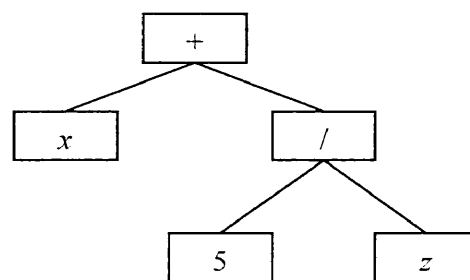
**Figure 5.3 Permutation encoding**

Chromosome 1	C	E	D	V	A	H	G
Chromosome 2	Y	O	R	F	S	T	Q

**Figure 5.4 Value encoding**

The tree encoding is also used in *genetic programming*. Genetic programming (GP) is a branch of GAs invented by Cramer [Cramer 85] and first explored in depth by Koza [Koza 92]. GP allows computer programs to evolve using evolutionary patterns like GAs. A solution in GP is usually represented by LISP expressions as shown in Figure 5.5.

Chromosome:  $( + x ( / 5 z ) )$



**Figure 5.5 Tree encoding**

#### 5.4 Initial Population

Each individual of the initial population is generated through a random list heuristic. For example, if we represent the solutions strings of ones and zeros, then these ones and zeros are chosen at random. However, it is still possible to generate the initial population with a little more intelligence such as knowledge about the characteristics and restrictions of the population.

## 5.5 Fitness Function

A *fitness function* is a predefined problem-specific measure of fitness used to evaluate each individual in the current population. In general, the results from the fitness function are used to select individuals in the current population for cross-breeding. An ideal fitness function correlates closely with the algorithm's goal, and yet may be computed quickly since GAs require many iterations.

## 5.6 Selection Schemes

The purpose of selection is to select pairs of individuals from the current population for cross-breeding. The selection may be performed at random or stochastically. There are many kinds of possible selection schemes, and we describe several of them used frequently.

### 5.6.1 Roulette Wheel Selection

This is the standard scheme for selecting parent chromosomes. In this scheme, each individual is assigned a probability of being selected based on its fitness. The probability of a particular individual  $i$  is defined as:

$$P(i) = \frac{F(i)}{\sum_{j=1}^n F(j)}, \text{ where } F(i) \text{ is the fitness of } i.$$

The individual is then selected by simulating the spinning of a suitably weighted roulette wheel. Thus, each individual has a chance to be selected that is directly proportional to its fitness. The effect of this scheme depends strongly on the range and scaling of fitness values in the current population. For example, early in a search it is possible for a few *super individuals* (solutions with fitness values significantly better than average) to dominate the selection process. One way to avoid this problem is to scale the fitness before selection [Fang 94].

### 5.6.2 Rank Selection

Baker [Baker 85] suggested the rank selection scheme, in which selection probabilities are based on the individual's rank or position in the population. For example, the best individual might be assigned the probability of  $\frac{2}{N}$ . This scheme has been found to be superior to the roulette wheel scheme [Whitley 89].

There are other possible forms of rank selection. One example is to use subjective fitness based on the rank in the population. The subjective fitness of a particular individual  $i$  is defined as:

$$SF(i) = \frac{(P - R_i)(\max - \min)}{(P - 1) + \min}, \text{ where } P \text{ is the population size, } R_i \text{ is the rank of } i,$$

$\max$  is the best fitness, and  $\min$  is the worst fitness. Then the probability of  $i$  is defined as:



$$P(i) = \frac{SF(i)}{\sum_{j=1}^n SF(j)}.$$

Whitley [Whitley 89] proposed a bias scheme to control the selection pressure. In Whitley's scheme, the larger the bias, the stronger the selection pressure; thus using a bias makes it possible for the fitter individuals to have higher probabilities to be selected.

### 5.6.3 Tournament Selection

Brindle [Brindle 81] described the tournament selection scheme as selecting  $K$  individuals from the current population at random and returning the best individuals among them. One of the major benefits of this scheme is that the selection pressure can be easily adjusted by changing the tournament size. If the tournament size is relatively large, weak individuals have lower probabilities to be selected.

Other forms of tournament selection exist. For example, Goldberg [Goldberg 90] proposed *Boltzmann tournament selection*, which uses a Boltzmann distribution to select individuals from the current population. According to a Boltzmann distribution, the probability of a particular individual  $i$  is defined as:

$$P(i) = \frac{e^{\frac{F(i)}{T}}}{\sum_{j=1}^n e^{\frac{F(j)}{T}}}, \text{ where } F(i) \text{ is the fitness of } i \text{ and } T \text{ is the temperature.}$$

#### **5.6.4 Elitism Selection**

When the individuals in the population are selected stochastically, there is a risk to lose one of the best individuals during the process. To avoid this problem, a technique called elitism is often used. In this scheme, some of the best individuals are retained at each generation without alternations. De Jong [De Jong 75] tested an elite scheme and found that on problems with just one maximum (or minimum) the algorithm performance was much improved, but on multimodal problems it was degraded.

#### **5.6.5 Steady-State Selection**

GAs can be categorized into generational GAs or steady-state GAs depending on how they replace the population. In generational GAs, each member of the population is replaced in the subsequent generation. In steady-state GAs, only a single or small number of population members are replaced at any one time. To replace the individuals, there are several strategies such as (1) replace the worst, (2) replace a randomly chosen individual, and (3) replace the oldest.

Whitley's Genitor algorithm [Whitley 89] was the first steady-state GA. In the Genitor algorithm, two parent individuals are selected by ranking selection and produce one offspring, which replaces the worst individual in the population. Syswerda [Syswerda 89] introduced variations of the steady-state GA and various deletion methods. Davis [Davis 91] tested the steady-state GA and showed that a steady-state GA, when combined with a feature that eliminates duplicate chromosomes, is superior to a generational GA.

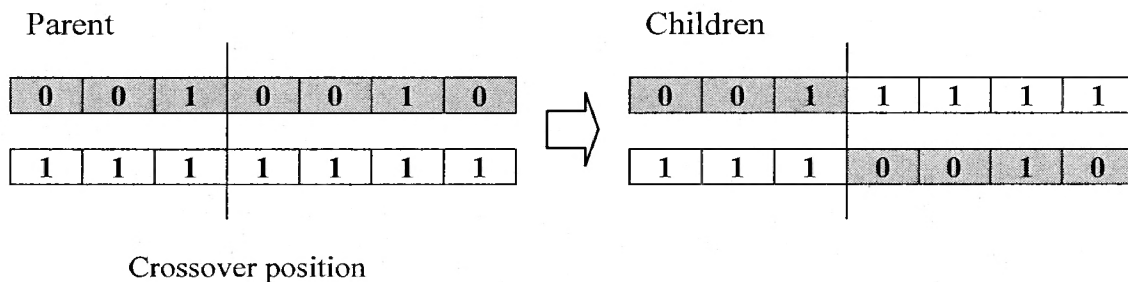
## 5.7 Genetic Operators

Genetic variation is a necessity for the process of evolution. Genetic operators are analogous to those which occur in the natural world: selection, crossover, and mutation.

### 5.7.1 Crossover

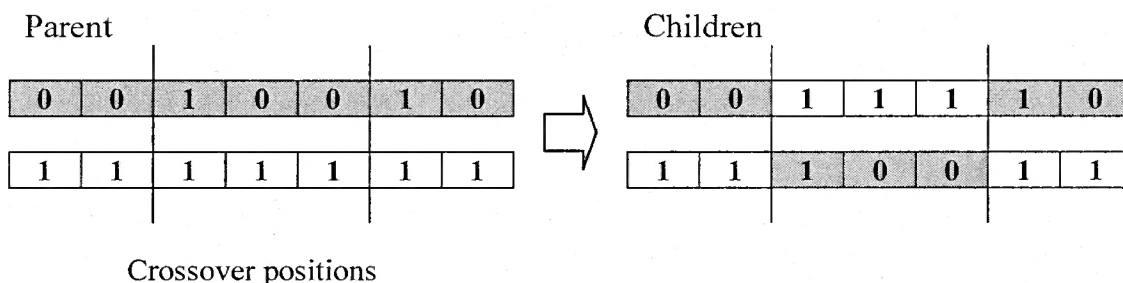
The crossover operator is used to construct better offspring by combining the genes of good existing chromosomes. Many crossover techniques exist for different data structures. Here, we introduce four kinds of well-known crossover operators.

- *One-point Crossover*: One crossover position is chosen at random. Then all genes beyond that point in the chromosome are swapped between the two parents.



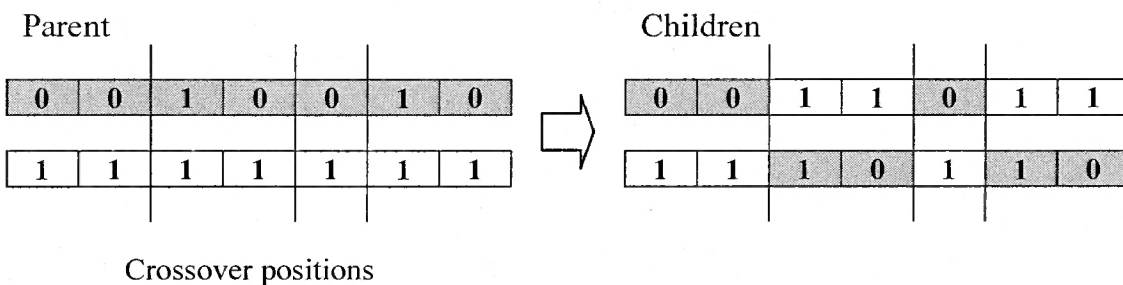
**Figure 5.6 One-point crossover**

- *Two-point Crossover*: Two crossover positions are chosen at random and everything between these two points is swapped between the two parents.



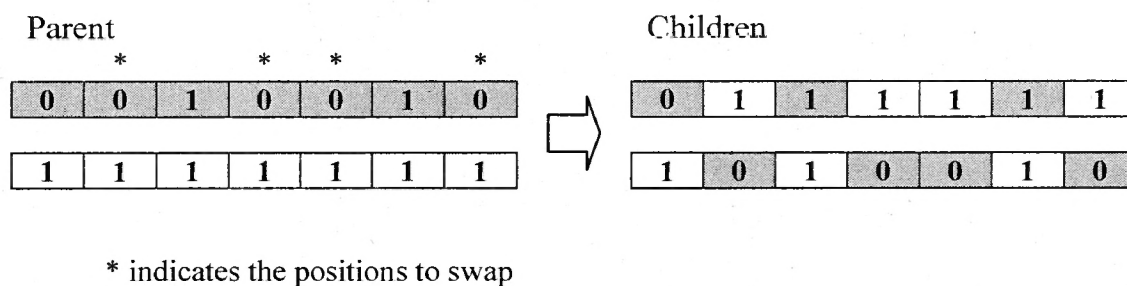
**Figure 5.7 Two-point crossover**

- *N-point Crossover*:  $N$  crossover positions are chosen at random and only the genes between odd and even positions are swapped. De Jong [De Jong 75] tested multiple point crossover operators and found that performance is degraded as more genes are swapped. Although it is necessary to introduce some changes in order to make progress, too many alternations result in the high probability of destroying the good genes of a chromosome.



**Figure 5.8 N-point crossover ( $N=3$ )**

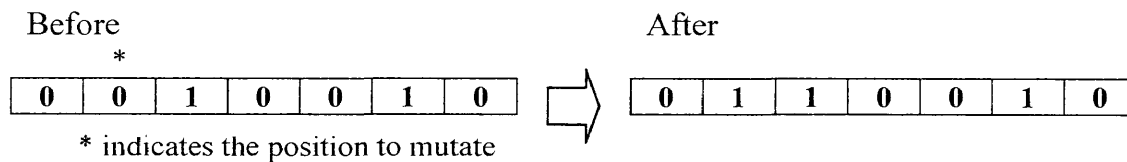
- *Uniform Crossover*: It was introduced by Syswerda [Syswerda 89]. Each gene of the first parent has a 0.5 probability of swapping with the corresponding gene of the second parent.



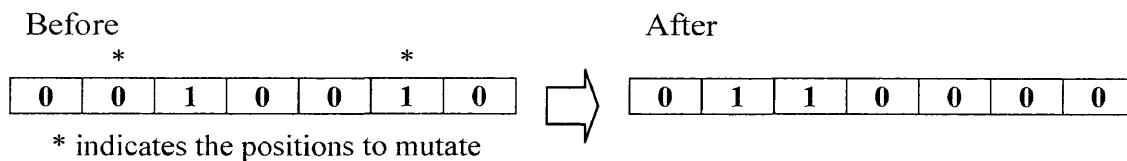
**Figure 5.9 Uniform crossover**

### 5.7.2 Mutation

Though the crossover operator produces a large range of possible solutions, it is still possible to converge prematurely. The mutation operator is used to avoid this problem by maintaining genetic diversity from one generation to the next. Typical GAs have a very small probability of mutation of perhaps 0.01 or less. In this process, the offspring are taken and each bit in the chromosome is mutated in some way, typically by altering it with a given probability. In Figure 5.10, there are total of seven bits in the chromosome. Then a random number is generated at each bit, and if it is lower than the mutation rate, the corresponding bit is altered. Figure 5.11 illustrates another mutation which swaps two bits.



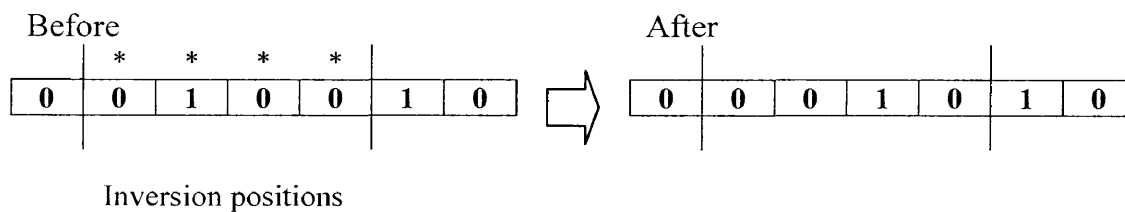
**Figure 5.10 Mutation (flip)**



**Figure 5.11 Mutation (swap)**

### 5.7.3 Inversion

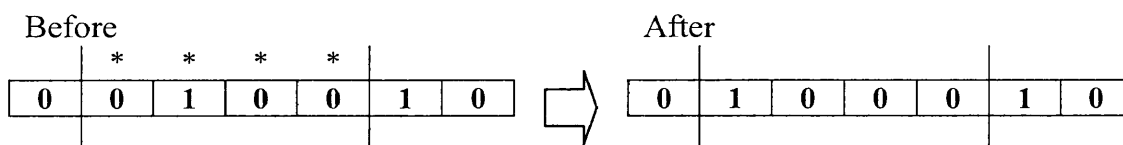
The inversion operator works on a single chromosome. Two inversion positions are chosen at random, and the order of the genes between these two positions is inverted.



**Figure 5.12 Inversion**

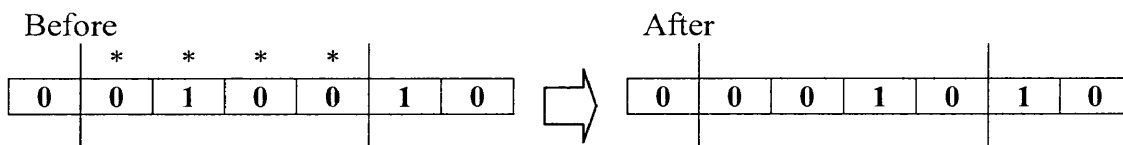
### 5.7.4 Rotation

The rotation operator rotates the genes between two positions chosen at random. Depending on the direction of the rotation, upward and downward rotations are possible [Chung & Dietz 95]. Figure 5.13 and Figure 5.14 illustrate both rotations.



Rotation positions

**Figure 5.13 Upward rotation**



**Figure 5.14 Downward rotation**

## Chapter 6

### Genetic Scheduling Algorithms

In this chapter, we first introduce several existing GAs designed for multiprocessor scheduling problems. Then we present our Genetic Real-time Fault-tolerant Scheduling (GRFS) algorithm, which is intended to improve the performance of the existing algorithms.

#### 6.1 Existing Genetic Algorithms

Several GAs have been developed for multiprocessor task scheduling, and the primary difference among them is how to represent chromosomes that encode scheduling information. Wang and Korfhage [Wang & Korfhage 95] presented a genetic scheduling algorithm using a two-dimensional matrix encoding. A major problem of this algorithm is that after executing the crossover and mutation operators, the resultant schedule may not be valid. Although they provided the repair operators to fix the problem, executing these operators requires extra computational time.

Hou *et al.* [Hou *et al.* 94] made a different approach using the string representation which is based on the schedule of the tasks in each individual processor. Later, Corrêa *et al.* [Corrêa *et al.* 99] proved that the representation used in the algorithm provided by Hou *et al.* cannot express the full range of possible schedules. To overcome this drawback, they proposed the full-search genetic algorithm (FSG) and combined-genetic list (CGL) algorithm. Lee and Chen [Lee and Chen 03] tackled the scheduling problem from a



different perspective, by partitioning tasks into clusters. Several approaches proposed by various researchers are described in depth in the following sections.

## 6.2 Partitioned Genetic Algorithm

Partitioned Genetic Algorithm (PGA), proposed by Lee and Chen [Lee and Chen 03], is based on first clustering tasks into several clusters and assigning the tasks to the processors using these clusters. A well-known clustering method used for scheduling parallel tasks to the processors was introduced by Sarkar [Sarkar 89], and later Yang and Gerasoulis [Yang & Gerasoulis 94] improved it by presenting the Dominant Sequence Clustering (DSC) algorithm. Kianzad and Bhattacharyya [Kianzad and Bhattacharyya 01] also used clustering technique and developed a GA formulation, called Clusterization Function Algorithm (CFA).

These algorithms define *tlevel* and *blevel* as:

$$tlevel(v_i) = \max_{v_j \in pred(v_i)} \{ tlevel(v_j) + t_j + c_{ji} \}$$

$$blevel(v_i) = t_i + \max_{v_j \in succ(v_i)} \{ c_{ij} + blevel(v_j) \}$$

where  $t_i$  is the execution time of  $v_i$ ,  $c_{ji}$  is the communication cost between  $v_i$  and  $v_j$ ,  $pred(v_i)$  is the set of immediate predecessors of  $v_i$ , and  $succ(v_i)$  is the set of immediate successors of  $v_i$ . An example of a clustered graph and its corresponding schedule are illustrated in Figure 6.1 and Figure 6.2 respectively. In Figure 6.1,  $tlevel(v_1) = 0$ ,  $blevel(v_1) = 12$ ,  $tlevel(v_4) = 2$ , and  $blevel(v_4) = 5$ . Note that a communication cost in a

clustered graph becomes zero if the start and end nodes of this edge are in the same cluster.

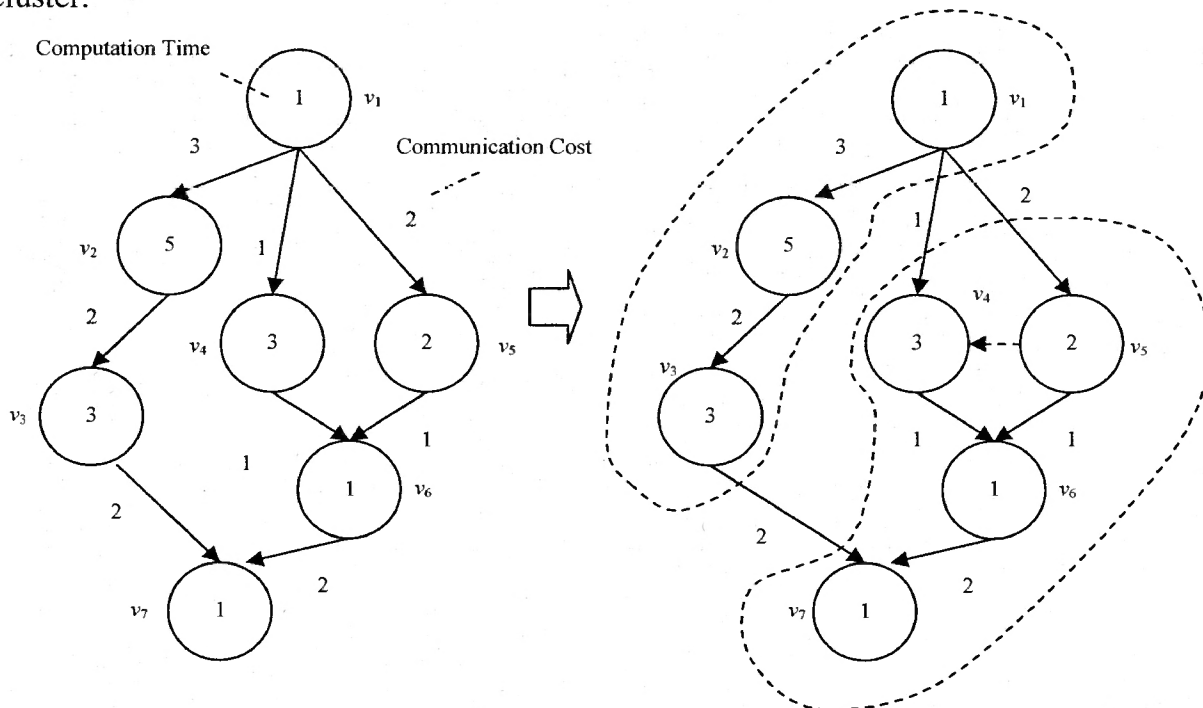


Figure 6.1 A clustered graph

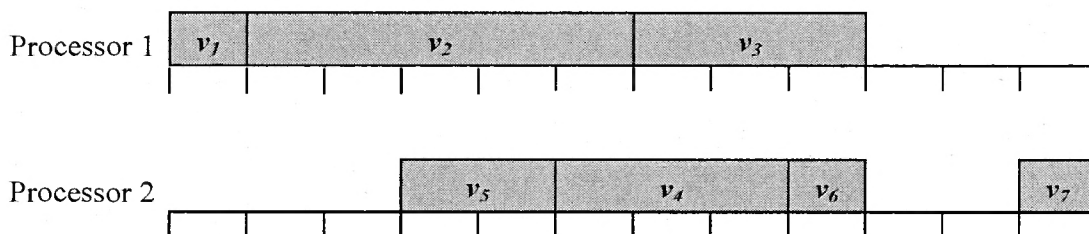


Figure 6.2 A Gantt chart

The main objective of clustering algorithms is to minimize the net execution time, which is defined as:

$$NT = \max_{v_x \in V} \{ tlevel(v_x) + blevel(v_x) \}.$$

Since this problem is known to be NP-complete, heuristics are proposed to solve the problem. Several researchers have used the algorithm known as Dominant Sequence Clustering (DSC), which is shown in Figure 6.3 [Yang & Gerasoulis 94].

```

EG(Examined) = null, UEG(Un Examined) = V
For i < N
    Compute blevel(vi)
    tlevel(vi) = 0
End-For

While UEG ≠ null
    vx = free task with the highest priority
    vy = partial free task with the highest priority

    If priority(vx) ≥ partpriority(vy)
        Reduce tlevel(vx)
        If no zeroing
            vx remains in a unit cluster
        End-If
    Else
        Reduce tlevel(vx) under constraint DSRW
        If no zeroing
            vx remains in a unit cluster
        End-If
    End-If
    Update the priorities of vx's successors
    Put vx into EG
End-While

```

**Figure 6.3 The DSC algorithm**

In Figure 6.3, there are some additional definitions as described below.

$$priority(v_i) = tlevel(v_i) + blevel(v_i)$$

$$partpriority(v_i) = ptlevel(v_i) + blevel(v_i), ptlevel(v_i) = \max\{ tlevel(v_j) + t_j + c_{ji} \}$$

$$v_j \in pred(v_i) \cap EG$$

DSRW: Zeroing incoming edges of a free node should not affect the reduction of  $ptlevel(v_i)$  if it is reducible by zeroing an incoming DS edge of  $v_i$ .

Kianzad and Bhattacharyya [Kianzad and Bhattacharyya 01] employed a GA to solve a clustering problem and defined the clusterization function as:

$$f(e_i) \begin{cases} 0 & \text{if } (e_i \in \beta) \\ 1 & \text{otherwise} \end{cases}$$

where  $\beta$  is a subset of task graph edges. For example, in Figure 6.1,  $\beta = \{\{e_{12}, e_{23}\}, \{e_{46}, e_{56}, e_{67}\}\}$ . The corresponding clusterization function  $f_\beta$  is shown in Table 6.1

$e_{12}$	$e_{14}$	$e_{15}$	$e_{23}$	$e_{37}$	$e_{46}$	$e_{56}$	$e_{67}$
0	1	1	0	1	0	0	0

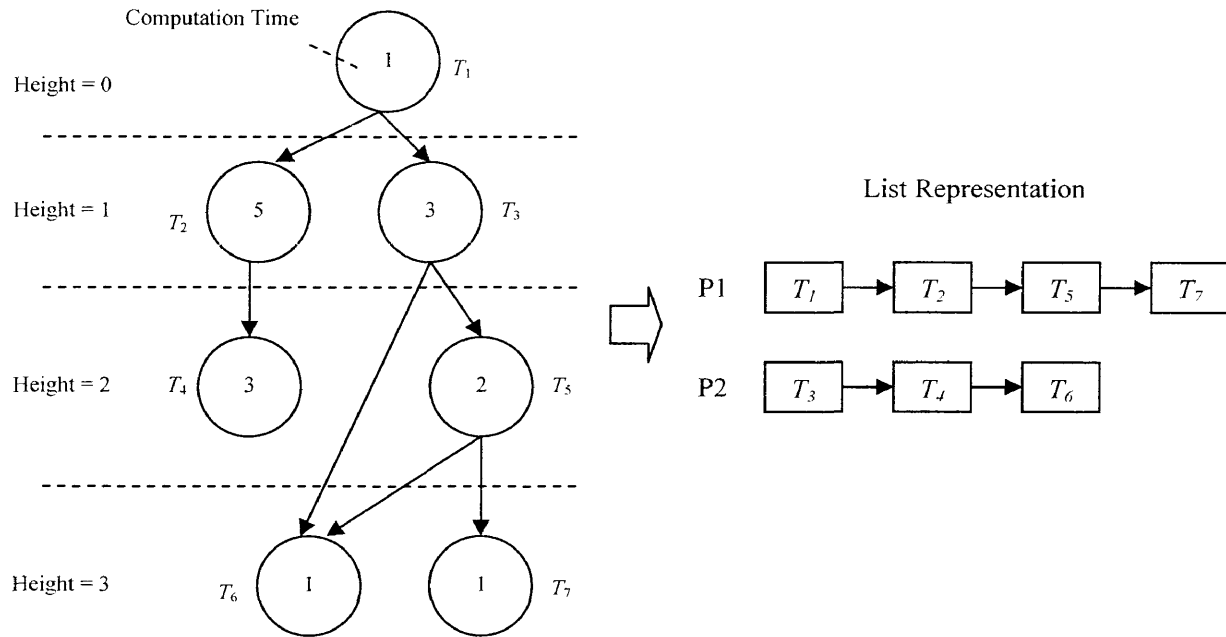
**Table 6.1 Clusterization function**

### 6.3 The HAR Algorithm

Hou *et al.* [Hou *et al.* 94] proposed a GA designed for multiprocessor scheduling using a string representation. We denote this algorithm HAR for short and describe it in this section.

#### 6.3.1 Chromosome Encoding

In HAR, the schedule is represented as several lists of computational tasks, and there is a one-to-one correspondence between processors and lists. Each list corresponds to the computational tasks executed on a processor, and the order of the tasks in the list indicates the order of execution. An example of a task graph and its corresponding list representation are illustrated in Figure 6.4.



**Figure 6.4 A task graph and its representation**

Although this list representation encodes the order of execution effectively, it does not contain information regarding the precedence constraints. For this reason, a method that guarantees the feasibility of all lists produced by the algorithm was devised. This method is based on the concept called *height* of tasks, which is defined below.

$$height(T_j) = \left. \begin{array}{l} 0 \\ 1 + \max \{ height(T_i) \} \\ \quad T_i \in pred(T_j) \end{array} \right\} \begin{array}{l} \text{if } pred(T_j) = \emptyset \\ \text{otherwise} \end{array}$$

For example, in Figure 6.4, we have  $height(T_1) = 0$ ,  $height(T_2) = height(T_3) = 1$ ,  $height(T_4) = height(T_5) = 2$ , and  $height(T_6) = height(T_7) = 3$ . If a task  $T_i$  is a predecessor

of a task  $T_2$ , then the relationship  $height(T_1) < height(T_2)$  always holds. If all tasks are ordered according to their heights, we can guarantee the feasibility of a given schedule.

### 6.3.2 Initial Population

The initial population is randomly generated in a way that all tasks are assigned to the processors according to their heights. Let  $G(h)$  be the set of tasks with height  $h$ . For each processor,  $P_1, P_2, \dots, P_{m-1}$ , choose  $r$  tasks,  $0 \leq r \leq |G(h)|$ , from each set  $G(h)$  at random. Then, remove these  $r$  tasks from  $G(h)$  and assign them to the corresponding processors. Finally, assign all remaining tasks to the processor  $P_m$ .

### 6.3.3 Genetic Operators

In HAR, there are three genetic operators: (1) reproduction, (2) crossover, and (3) mutation as described in the following.

#### 6.3.3.1 Reproduction

The reproduction process forms a new population by selecting individuals in the old population based on their fitness values. There are several different methods for selecting individuals, but a “roulette wheel” method is commonly used and becomes a choice of HAR. In this method, each individual is assigned an interval, whose length is proportional to its fitness. Then a random number between 1 and  $\sum_{i=1}^n fitness(T_i)$  is

generated and used as an index into the roulette wheel to locate the corresponding individual. Individuals with higher fitness value are more likely to be selected and passed to the next generation.

### **6.3.3.2 Crossover**

The crossover positions are randomly chosen to cut the tasks into two halves, and if two conditions hold, the new individuals are always feasible.

1. The heights of the tasks next to the crossover positions are different.
2. The heights of all the tasks immediately in front of the crossover positions are the same.

In other words, tasks are partitioned into two sets  $V_1$  and  $V_2$ , and the left parts of the individuals contain only tasks in  $V_1$  and the right parts contain only tasks in  $V_2$ . As a result, there is no dependency between a task in  $V_1$  and a task in  $V_2$ . During the crossover operation, the left parts of each individual remain the same, while the right parts of the individuals are exchanged.

### **6.3.3.3 Mutation**

First, a task is randomly chosen. Then, another task with the same height is chosen at random. Finally, the positions of these two tasks are exchanged, resulting in a new schedule.

#### 6.3.3.4 Shortfalls

Corrêa *et al.* [Corrêa *et al.* 99] investigated HAR and made observations to point out the shortfalls.

##### **Observation 1 (Initial Population)**

A processor  $P_i$  has on average more tasks than  $P_{i+1}$ ,  $1 \leq i < m$ . This happens because the task distribution over the processors is not uniform.

##### **Observation 2 (Crossover)**

Since HAR assigns tasks to the processors according to their height, there is a chance to miss an optimal solution. Suppose that a task  $T_i$  has predecessors that all have been already scheduled. Thus, a task  $T_i$  itself is ready to be scheduled. However, if the height of  $T_i$  is larger than the height of the tasks currently being processed,  $T_i$  has to wait until all tasks whose height is smaller than the height of  $T_i$  have been assigned.

##### **Observation 3 (Absence of knowledge)**

HAR uses knowledge about the problem only in terms of a structural nature, through the verification of feasibility of the solutions. Thus, it lacks knowledge about the quality of individuals.



## **6.4 The Combined-Genetic List (CGL) Algorithm**

Corrêa *et al.* [Corrêa *et al.* 99] proved that the representation used in HAR cannot express the full range of possible schedules. To overcome this drawback, they proposed the full-search genetic algorithm (FSG), which is capable of spanning the entire space of possible schedules. Although FSG outperforms HAR, additional list-heuristics that leverage some knowledge about the scheduling problem were added to FSG for further improvement. This algorithm is called the combined-genetic list (CGL) algorithm.

### **6.4.1 Initial Population**

As in HAR, each individual of the initial population is generated at random. First, a task is randomly chosen among those for which all predecessors are already assigned. Then, a processor is also chosen at random. Notice that since this allocation method is not based on heights of tasks, it can possibly cover an entire search space, which is not possible with HAR.

### **6.4.2 Genetic Operators**

The genetic operators used in HAR are revised in order to take into account the new features in CGL.

### 6.4.2.1 Reproduction

The reproduction process uses a biased roulette wheel in the same way as HAR does.

### 6.4.2.2 Crossover

Instead of height of tasks as in HAR, CGL uses a set, which stems from the precedence relationships implied by the tasks assigned to the same processor. This set is defined as:

$$A(s) = A \cup \{(T_i, T_j) \mid (T_i, T_j) \notin A\},$$

where  $s$  is the schedule,  $T_i$  and  $T_j$  are in the same processor, and  $T_i$  is in the immediate front of  $T_j$ . In addition,  $A^+(s)$  is the transitive closure of  $A(s)$ . Then, the partition is performed as shown in Figure 6.5. Through this operation, we can observe that  $T_i$  and all of its predecessors are inserted in  $V_1$ , and equivalently,  $T_i$  and all of its successors are inserted in  $V_2$ .

To overcome the drawback of HAR, which is stated as absence of knowledge, CGL uses the list heuristic called *earliest date/most immediate successor first* (ED/MISF). This heuristic first computes the minimal introduction date of each free task and chooses the task with smallest introduction date. Then, a processor is chosen at random in a way so that the task can be assigned as soon as possible.

```

Choose task  $T_i \in T$  at random
Choose  $V_1$  or  $V_2$  at random

If  $V_1$ 
     $V_1 = V_1 \cup \{T_i\} \cup \{T_j \mid T_j \in T \text{ and}$ 
         $(T_i, T_j) \in (A^+(s_1) \cup A^+(s_2))\}$ 
Else
     $V_2 = V_2 \cup \{T_i\} \cup \{T_j \mid T_j \in T \text{ and}$ 
         $(T_i, T_j) \in (A^+(s_1) \cup A^+(s_2))\}$ 
End-If

Delete all tasks inserted into  $V_1$  or  $V_2$ 

```

**Figure 6.5 Crossover (CGL)**

### 6.4.2.3 Mutation

The new individual is formed by using a list heuristic under the same rules used in the crossover operation.

## 6.5 Other Genetic Algorithms Related to CGL

Zhong and Yang [Zhong & Yang 03] presented the Task Execution Order List (TEOL) algorithm. As in HAR and CGL, TEOL partitions tasks into sets  $V_1$  and  $V_2$  so that there is no dependency from a task in  $V_2$  to a task in  $V_1$ . The main characteristic of TEOL is to find the task's begin execution time in a schedule and create the list according to these execution times. The earlier a task begins to execute, the further up it is in the list.

Rinehart *et al.* [Rinehart *et al.* 03] pointed out the relative computational complexity of CGL and proposed the Bi-Chromosomal Genetic Algorithm (BCGA). BCGA decomposes a schedule into two independent structures: a task-to-processor assignment matrix that stores the assignment of tasks to processors, and a topological-sort vector representing the execution order of the tasks in the schedule. In addition, unlike HAR and CGL, BCGA uses a clustering method known as Dominant Sequence Clustering (DSC) when it partitions the tasks into sets.

## 6.6 Observations

Before presenting our algorithm, we first make several observations on the existing GAs to analyze their effectiveness and drawbacks.

### Observation 1: Clustering Techniques

All GAs introduced in this chapter use some clustering techniques, because it is necessary to partition tasks into sets when we perform crossover operation. For example, if it is one-point crossover, we need to partition tasks into two sets, and if it is two-point crossover, we need three sets, and so on. There are two types of clustering techniques in terms of their flexibility of organizing the sets as described below:

1. *Static Clustering*: Once tasks are clustered into the sets, they basically stay in the same sets. For example, in HAR, tasks are clustered according to their heights, and the order of height never changes. Also, in PGA, tasks are clustered using *blevel*, and the members in each set do not change.
2. *Dynamic Clustering*: When sets are formed, they are created dynamically and the members of each set may be different at each time. For example, in CGL,

two sets  $V_1$  and  $V_2$  are dynamically created by choosing the tasks at random. Also, in CFA, the structures of the clusters are constructed in a dynamic manner.

The main drawback of static clustering is that once clusters are formed, the members in each cluster do not change. Thus, it may not be able to search the entire solution space. Dynamic clustering does not suffer from this drawback. Therefore, dynamic clustering is more ideal than static clustering in terms of a possibility to find an optimal solution.

### **Observation 2: Homogeneous vs. Heterogeneous**

All algorithms introduced in this chapter assume that the system is homogeneous, meaning that all processors are identical. This assumption plays a very important role since the execution time of a task must be the same no matter which processor executes the task. In fact, some list heuristics - such as earliest date/most immediate successor first (ED/MISF) in CGL and task execution order list (TEOL) in TEOL - can predict the execution time in advance due to this assumption. In other words, if the system is heterogeneous, we cannot predict the exact starting and ending times of a task unless we actually make a schedule for the task. We assume that our system is heterogeneous, and therefore we cannot use the techniques devised only for homogeneous systems.

### **Observation 3: Task Allocation Methods**

There are two types of task allocation methods used for precedence constrained tasks.

1. *Height-based Allocation Method*: This method allocates the tasks according to their heights. An example of this method is HAR.

2. *Predecessor-based Allocation Method*: This method allocates a task as soon as its all predecessors are scheduled. The examples of this method are CGL, PGA, and TEOL.

As noted earlier in this chapter, the main drawback of HAR comes from this height-based allocation method. Predecessor-based allocation method is desirable to avoid this drawback.

## 6.7 Proposed Model

In this section, we present our Genetic Real-time Fault-tolerant Scheduling (GRFS) algorithm. GRFS is based on HAR and CGL, and it is designed to combine these two algorithms to obtain the maximum benefits out of them. The reasons we chose HAR and CGL for the foundation of GRFS are:

1. HAR is successful in simplifying the genetic operators by introducing height.
2. CGL compensates for the drawbacks of HAR.

Although GRFS adopts the ideas from HAR/CGL, we should note that there are some major differences between HAR/CGL and GRFS as listed below:

1. HAR/CGL are not specifically designed for real-time systems; therefore, they do not consider deadlines of tasks.
2. HAR/CGL are not designed for fault-tolerant systems, and as a result, they do not create backup copies for fault-tolerance.

Real-time fault-tolerant scheduling systems are under more constraints than normal multiprocessor scheduling systems. It is necessary to modify some parts of HAR/CGL so that GRFS can incorporate real-time and fault-tolerant features.

### **6.7.1 Requirements for Real-Time Fault-Tolerant Systems**

In this section, we discuss the additional features that may not be part of HAR/CGL but are necessary for GRFS to make it suitable to real-time fault-tolerant systems.

#### **Earliest-Deadline-First (EDF) Algorithm**

One of the most important characteristics of real-time systems is the timing constraints. In hard real-time systems, the consequences of not executing a task before its deadline may be catastrophic. We need to make every effort to make sure that all tasks meet their deadlines if possible. From this point of view, we use the *deadline-driven scheduling algorithm*, called the Earliest-Deadline-First (EDF) algorithm. The EDF algorithm, introduced by Liu and Layland [Liu & Layland 73], proved it is optimum in the sense that if a set of tasks can be scheduled by any algorithm, it can be scheduled by the deadline-driven scheduling.

In the EDF algorithm, a task with the earliest deadline will receive the highest priority. Then, these tasks are placed in a ready queue so that a task with highest priority is always taken out of the queue first. After the task is picked up, a scheduler assigns that task to a processor. Since our problem is NP-complete, it is not guaranteed for the EDF algorithm

to find an optimal solution. Nonetheless, we assume that the EDF algorithm still can produce a near-optimal solution.

### **Precedence Constraints**

In a precedence constrained environment, if there is a precedence relationship between two tasks such as  $T_i < T_j$ , then we must guarantee that task  $T_i$  is always executed before task  $T_j$  starts running. In HAR, the concept of height is used to sustain the precedence constraints, and we also use this idea in GRFS. However, as mentioned earlier, using heights of tasks incurs a negative impact on the search procedure. To overcome this problem, we introduce a hybrid allocation method in the following section.

### **Fault-Tolerance**

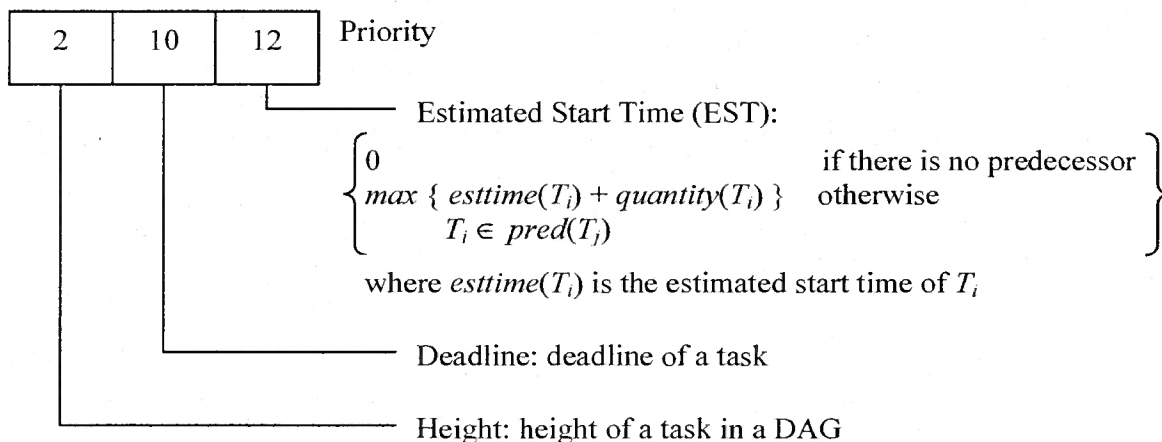
As mentioned in Chapter 4, there are two major types of fault-tolerant techniques: backup de-allocation and backup overloading. GRFS is capable of creating backup de-allocating copies and making a schedule using both techniques.

#### **6.7.2 Priority Assignment**

GRFS uses priorities to integrate precedence constraints and the EDF algorithm. A priority consists of three parts: (1) height; (2) deadline; and (3) estimated start time (Figure 6.6). The height indicates the level of a task in a DAG and is necessary to sustain the precedence relationships among tasks. The deadline is a task's deadline. Finally, the estimated start time denotes the possible start time for a task. After all tasks are assigned



priorities, they are placed into a queue and sorted in ascending order. Thus, it is guaranteed to: (1) maintain the precedence constraints; (2) a task with earliest deadline is scheduled first; and (3) a task may be scheduled as soon as its all predecessors are scheduled.



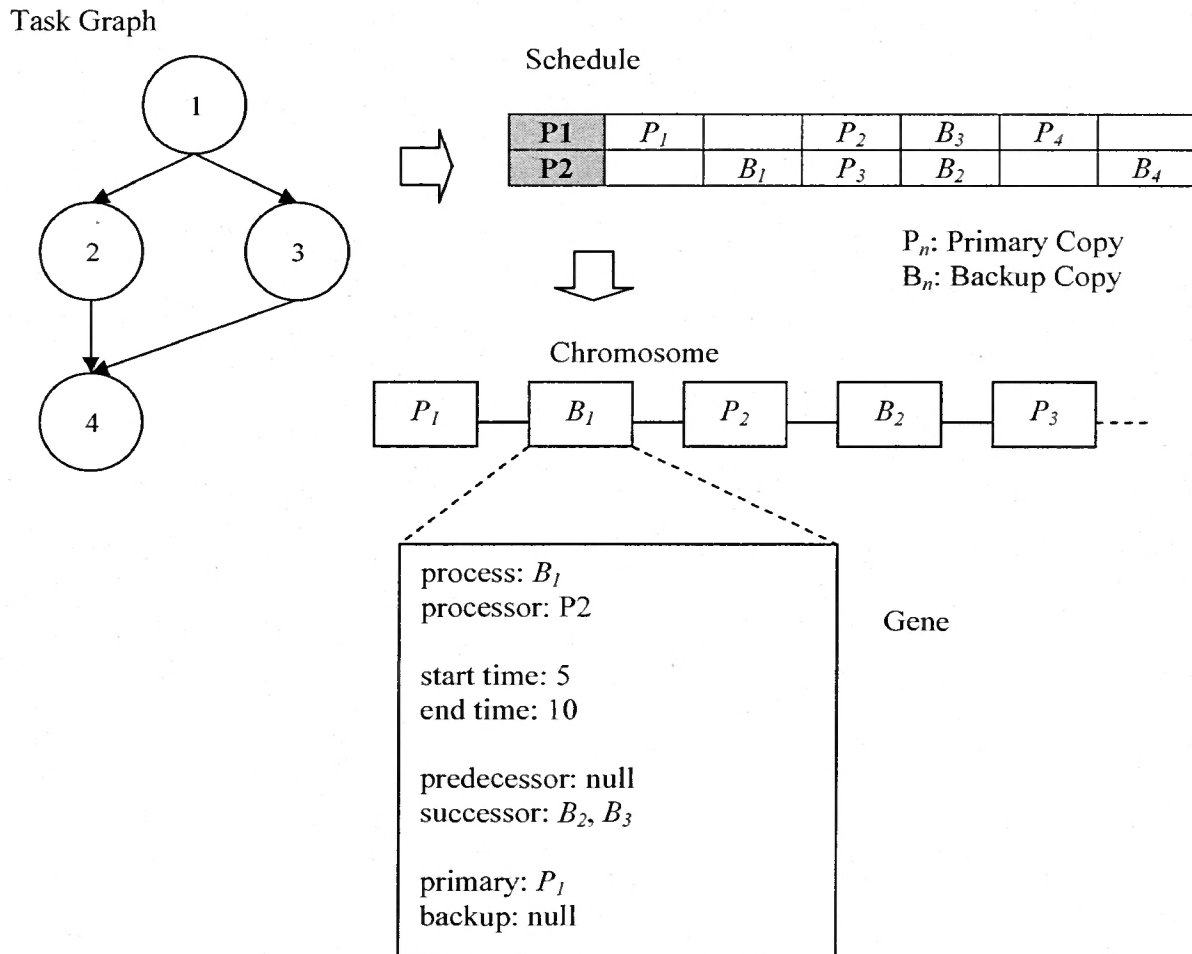
Name	Minimum Value	Maximum Value
Height	The first task to be executed has the minimum value of 0.	If all tasks are executed one after another, the maximum value is $N - 1$ , where $N$ is the total number of tasks.
Deadline	The minimum value is 1.	The maximum value equals the length of the schedule.
EST	The minimum value is 0.	$\sum_{i=1}^N quantity(T_i) - quantity(T_j)$ , where $T_j$ is the final task.

Figure 6.6 Priority in GRFS

### 6.7.3 Chromosome Encoding

One of the prominent differences among existing GAs is how to represent a solution. For example, HAR, CGL, PGA and TEOL all select string representation. Some other

algorithms use matrix representation as in [Wang & Korfhage 95]. In addition, it is still possible to combine more than one representation like BCGA, which has both string and matrix representations. Since GRFS is based on HAR and CGL, we also use string representation as depicted in Figure 6.7.



**Figure 6.7 String Representation**

In GRFS, a chromosome is represented by a linked-list, whose single element corresponds to a gene (Figure 6.7). A gene is categorized into either a primary copy or a

backup copy. Further, each gene has information about (1) its allocation and (2) its precedence relationships as shown in Table 6.2. GRFS can construct a schedule by using only information contained in each gene.

Allocation Information		Precedence Information	
Name	Type	Name	Type
Allocated Processor	String	List of Predecessors	Pointer
Starting Time	Integer	List of Successors	Pointer
Ending Time	Integer	Primary Copy	Pointer
Backup Overloaded	True/False	Backup Copy	Pointer

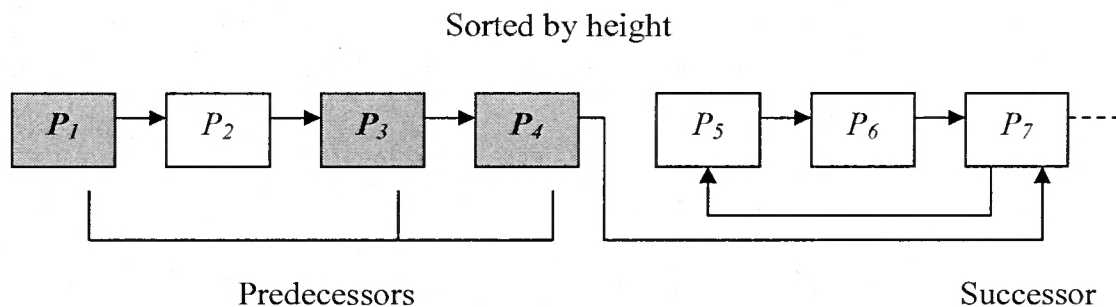
**Table 6.2 Information in genes**

We should mention something else about a gene. Each gene also has a priority which is mentioned earlier in this chapter, and the order of genes in a chromosome is organized according to their priorities. The order of genes in a chromosome represents the sequence of execution of each task.

#### **6.7.4 Hybrid Task Allocation Method (HTAM)**

GRFS uses heights of tasks to determine the order of execution of each task. However, this height-based allocation method has a relatively high risk to miss an optimal solution due to its inability to explore the entire solution space. To overcome this problem, we introduce the Hybrid Task Allocation Method (HTAM). HTAM is a combination of the height-based allocation method used in HAR and the predecessor-based allocation method used in CGL.

Since the order of genes in a chromosome is determined by the height of each task, normal allocation follows the heights of tasks in a DAG. However, once we discover that all predecessors have already been scheduled, we choose to either continue following the order of height or jump to their successor with the probability of 0.5 for each. In Figure 6.8, we assume that task  $P_7$  has predecessors  $P_1$ ,  $P_3$ , and  $P_4$ . Normal execution follows the height-based allocation method, thus task  $P_1$  through task  $P_4$  are executed according to their heights. After executing task  $P_4$ , all predecessors of task  $P_7$  have already been executed. Therefore, there are two choices: (1) execute task  $P_5$  as in the height-based allocation method or (2) execute task  $P_7$  as in the predecessor-based allocation method. In the figure, we choose to execute task  $P_7$ , and after that, the execution goes back to the original order. The same process is repeated for the rest of the list. In this way, we can eliminate the drawback of HAR and possibly search the entire solution space.



**Figure 6.8 Hybrid Task Allocation Method (HTAM)**

### 6.7.5 Processor Search

When creating a schedule for primary copies, any processor can be used for task allocation as long as there exist enough empty time slots in the processor. Nonetheless, this is not true when it comes to selecting a processor to allocate the backup copies. We define some notations as follows: (1)  $proc(T_i)$  is the processor to which task  $T_i$  is assigned; (2)  $T_i^P$  is the primary copy of task  $T_i$ ; and (3)  $T_i^B$  is the backup copy of task  $T_i$ .

#### Proposition 6.1

*A task  $T_i$  is guaranteed to execute in the presence of one permanent fault if and only if  $proc(T_i^P) \neq proc(T_i^B)$ .*

This is true because if both primary and backup copies of task  $T_i$  are scheduled on the same processor and that processor fails, then  $T_i$  is completely lost. A primary copy and its corresponding backup copy must be assigned to the different processors.

#### Proposition 6.2

*To allow a single fault to be tolerated,  $T_i^B$  and  $T_j^B$  can be overloaded if and only if  $proc(T_i^P) \neq proc(T_j^P)$ .*

This is true because if both primary copies are scheduled on the same processor and that processor fails, their corresponding backup copies must both be executed. Backup overloading is possible only when their primary copies are assigned to the different processors.

### 6.7.6 Earliest Empty Slot Search

In this section, we introduce two concepts: Earliest Available Time (EAT) and Earliest Start Time (EST).

#### Earliest Available Time (EAT)

Qin *et al.* [Qin *et al.* 02] used the concepts of *EAT* and *EST*. To define these two, we need more notations: (1)  $p(i)$  is the processor whose id is  $i$ ; (2)  $start(T_i)$  is the starting time of  $T_i$ ; (3)  $end(T_i)$  is the ending time of  $T_i$ ; (4)  $msg_{(ij)}(T_i, T_j)$  is the amount of time to take for  $T_i$ , which is assigned to  $p(i)$ , to send a message to  $T_j$ , which is assigned to  $p(j)$ ; and (5)  $EAT_i(T_j)$  is the earliest available time for  $T_j$  in  $p(i)$ . Before defining *EAT*, we should note that *EAT* for primary copies and backup copies are different because of the proposition below.

#### Proposition 6.3

*To allow  $T_i$  to execute in the presence of a fault,  $start(T_j^B)$  should be larger than or equal to  $end(T_j^P) + \delta$ , where  $\delta$  is the time necessary for  $T_j^B$  to detect the fault.*

This is true because  $T_j^B$  executes only when its primary copy fails. If  $T_j^B$  is scheduled before the completion of its primary copy, there is no way for  $T_j^B$  to know whether its primary copy failed or not.

We first define  $EAT$  for primary copies as:

$$EAT_i(T_j^P) = \max \{ \text{end}(T_k) + \text{msg}(\text{proc}(T_k), i)(T_k, T_j) \}.$$

$$T_k \in \text{pred}(T_j^P)$$

Next,  $EAT$  for backup copies is defined as:

$$EAT_i(T_j^B) = \max \{ \text{end}(T_j^P) + \delta, \text{end}(T_k) + \text{msg}(\text{proc}(T_k), i)(T_k, T_j) \}.$$

$$T_k \in \text{pred}(T_j^B)$$

Note that for primary copies, we consider only precedence relationships among primary copies. However, for backup copies, we consider not only precedence relationships among backup copies but also the finishing times of their primary copies.

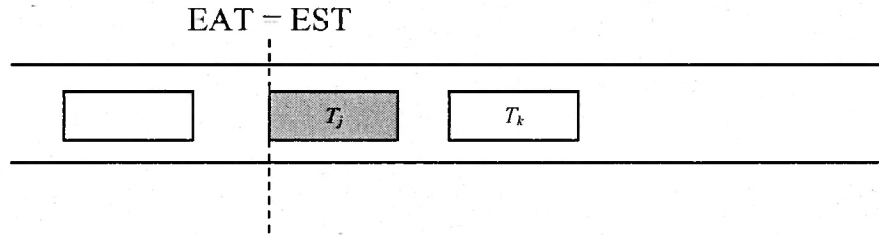
Although  $EAT$  is the earliest possible time for a task to start running, it is not always the case that a task actually starts executing exactly at  $EAT$ . For example, if another task is already scheduled at  $EAT$ , it is impossible for a task to start executing exactly at that point. Thus, we need to define  $EST$ , which indicates the actual earliest time for a task to start running.

### **Earliest Start Time (EST)**

We define  $EST_i(T_j)$  as the earliest time for task  $T_j$  to start executing in  $p(i)$  and consider two cases. We assume that task  $T_j$  is the task to be assigned and task  $T_k$  is the first task which has already been assigned to  $p(i)$  and satisfies the condition  $\text{end}(T_k) > EAT_i(T_j)$ .

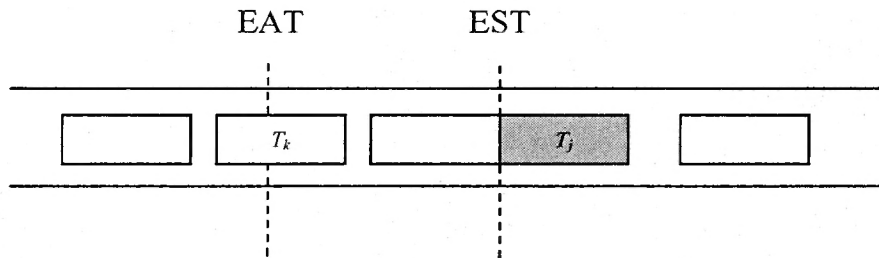
Case 1: Task  $T_k$  does not exist or  $start(T_k) - EAT_i(T_j) \geq end(T_j) - start(T_j)$ .

$$EST_i(T_j) = EAT_i(T_j)$$

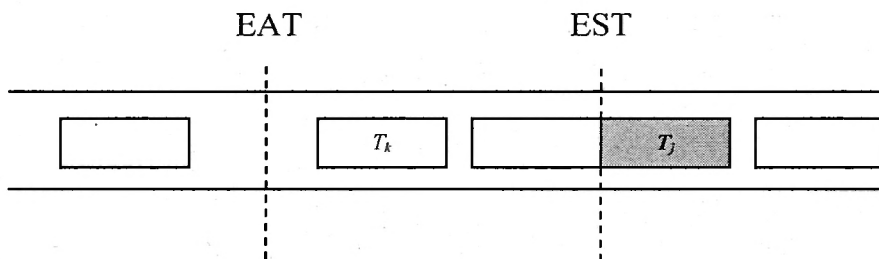


**Figure 6.9 EST (Case 1)**

Case 2: Task  $T_k$  is assigned at  $EAT_i(T_j)$  (Figure 6.10) or  $start(T_k) - EAT_i(T_j) < end(T_j) - start(T_j)$  (Figure 6.11).



**Figure 6.10 EST (Case 2-1)**



**Figure 6.11 EST (Case 2-2)**



In this case,  $EST_i(T_j)$  is calculated as shown in Figure 6.12. In this figure, we assume that task  $T_j$ ,  $T_k$  and  $T_l$  are all assigned to  $p(i)$ .

```

Task  $T_k$  is the first task satisfies  $end(T_k) > EAT(T_j)$ 
Task  $T_l$  is the task assigned immediately after  $T_k$ 

For until  $EST(T_j)$  is found
  If  $start(T_l) - end(T_k) \geq end(T_j) - start(T_j)$ 
     $EST(T_j) = end(T_k)$ 
  Else
     $T_k = T_l$ 
     $T_l =$  the task assigned immediately after  $T_k$ 
  End-If
End-For

```

**Figure 6.12 The algorithm to find EST**

Note that  $EST_i(T_j)$  is the earliest start time in  $p(i)$ . Since we want to find the earliest start time in the entire system, it is necessary to further define  $EST(T_j)$ . When GRFS tries to assign task  $T_j$  to a processor, it uses  $EST(T_j)$  to find the time slot for  $T_j$ .

$$EST(T_j) = \min \{ EST_i(T_j) \}$$

$$i = p(i) \in P$$

### 6.7.7 Initial Population

In general, the initial population in GAs is created at random. However, if the individuals in the initial population are far from optimal solutions, we may need to create many more generations than we wish, which requires both computational time and memory space. In fact, an extensive iteration is a drawback of GAs. To avoid this problem and speed up the process to converge, GRFS uses three heuristics to create the initial population.

1. *Earliest Start Time*: This approach finds the processor which can start executing the task earliest among other processors.
2. *Least Utility*: This approach finds the processor which is least utilized among other processors. In other words, the total length of the tasks assigned to the processor is the shortest among others.
3. *Random*: This approach selects the processor at random.

These heuristics make it possible to create the individuals with relatively high quality, and they guarantee diversity in the initial population. The process to create the initial population is illustrated in Figure 6.13.

```
k = initial population size
Repeat k times
  For each primary and backup copy
    Case first time
      Find processor with Earliest Start Time
    Case second time
      Find processor with Least utility
    Case other
      Find processor at random

    /* search time slot */
    Search EST
    Assign the task to the processor
  End-For

  Put the schedule into a queue
End-Repeat
```

**Figure 6.13 Initial population**

### 6.7.8 Fitness Function

A fitness function is a predefined problem-specific measure of fitness used to evaluate each individual in the current population. The total time required by a schedule to completely execute is called the schedule's *makespan*. In HAR, CGL, PGA, and BCGA, the makespan is used for their fitness functions. This approach also works for GRFS because (1) GRFS is designed for real-time systems, thus it is desirable to assign all tasks as early as possible and (2) if we can minimize the schedule length, the system may be able to accommodate more tasks. For these reasons, GRSF also uses the makespan for its fitness function, and it is defined as:

$$\max_{P_j \in P} \{ finish(P_j) \}$$

where  $finish(P_j)$  is the ending time for the final task in processor  $P_j$ .

### 6.7.9 Genetic Operators

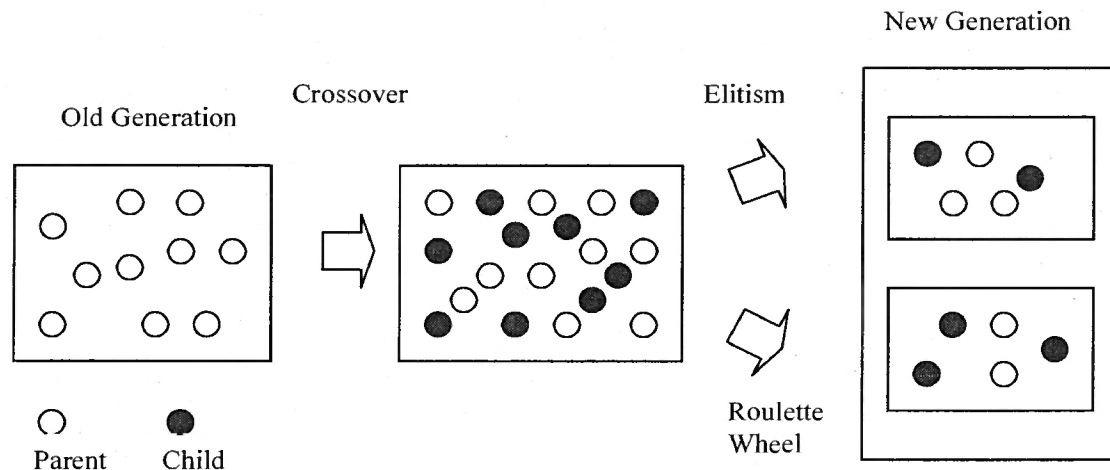
In this section, we describe three genetic operators used in GRFS: (1) reproduction; (2) crossover; and (3) mutation.

#### 6.7.9.1 Reproduction

“Reproduction” is the process of forming a new population of individuals by selecting individuals in the old population based on their fitness values. GRFS employs a technique which combines two selection methods known as *elitism* and *roulette wheel*. In elitism

selection, the worst (least fit) individual in the new population is replaced by the best (most fit) individual from the previous population. As a result, the best individual is always kept intact and transferred to the following generations. In roulette wheel selection, each individual is assigned an interval, whose length is proportional to its fitness. Then a random number is generated and used to select the individual. Unlike elitism selection, each individual has a chance to be selected even if its fitness is very low. However, the better the fitness of an individual is, the higher its probability to be selected.

In GRFS, during the crossover operation, the new individuals are produced and placed into the same population as their parents. Hence, after the crossover and mutation operations, all individuals, both parents and their children, are in the same population. Then, GRFS uses the elitism selection for selecting the half population and the roulette wheel selection for selecting the others in the new generation. This process is depicted in Figure 6.14 and Figure 6.15.



**Figure 6.14 Selection**

```

k = population size / 2

/* elitism selection */
Repeat k times
    Select and remove best individual from old population
    Insert selected individual into new population
End-Repeat

/* Construct a roulette wheel */
Probability for  $T_i = \frac{F(T_i)}{\sum_{j=1}^n F(T_j)}$ , where  $F(T_j)$  is the fitness

/* roulette wheel selection */
Repeat k times
    Generate a random number [0, 1]
    Search and remove individual using the random number
    Insert selected individual into new population
End-Repeat

```

**Figure 6.15 Reproduction**

### 6.7.9.2 Crossover

“Crossover” is a term used for the process to produce offspring from parent individuals. First, two individuals in the current population are selected as a parent. This selection may be at random; otherwise the fitness value for each individual is used. We infer that if the fitness values of two individuals are very close, they may look similar. Thus, if we breed these two individuals, the resulting offspring may also be similar to its parent. From this reason, GRFS chooses two individuals at random so that we may preserve the diversity in the population. After selecting parent individuals, a crossover position is chosen. Then two parent individuals are exchanged at the crossover position and result in

two offspring. A crossover position may be single or multiple. In GRFS, both one-point and two-point crossover operations are used with the probability of 0.5 for each. Each task in the individual has the allocated processor, and the major purpose of crossover is to change these processors. An example of two-point crossover is illustrated in Figure 6.16.

Although parent individuals are chosen at random, it is still possible for offspring to be identical to its parent or other existing individuals. Thus, after crossover, GRFS checks whether offspring is a duplicate of others, and if this is the case, the offspring is deleted so that all individuals in the population are unique. Finally, each task in the new individual is assigned to the processors using the HTAM. This process is shown in Figure 6.17.

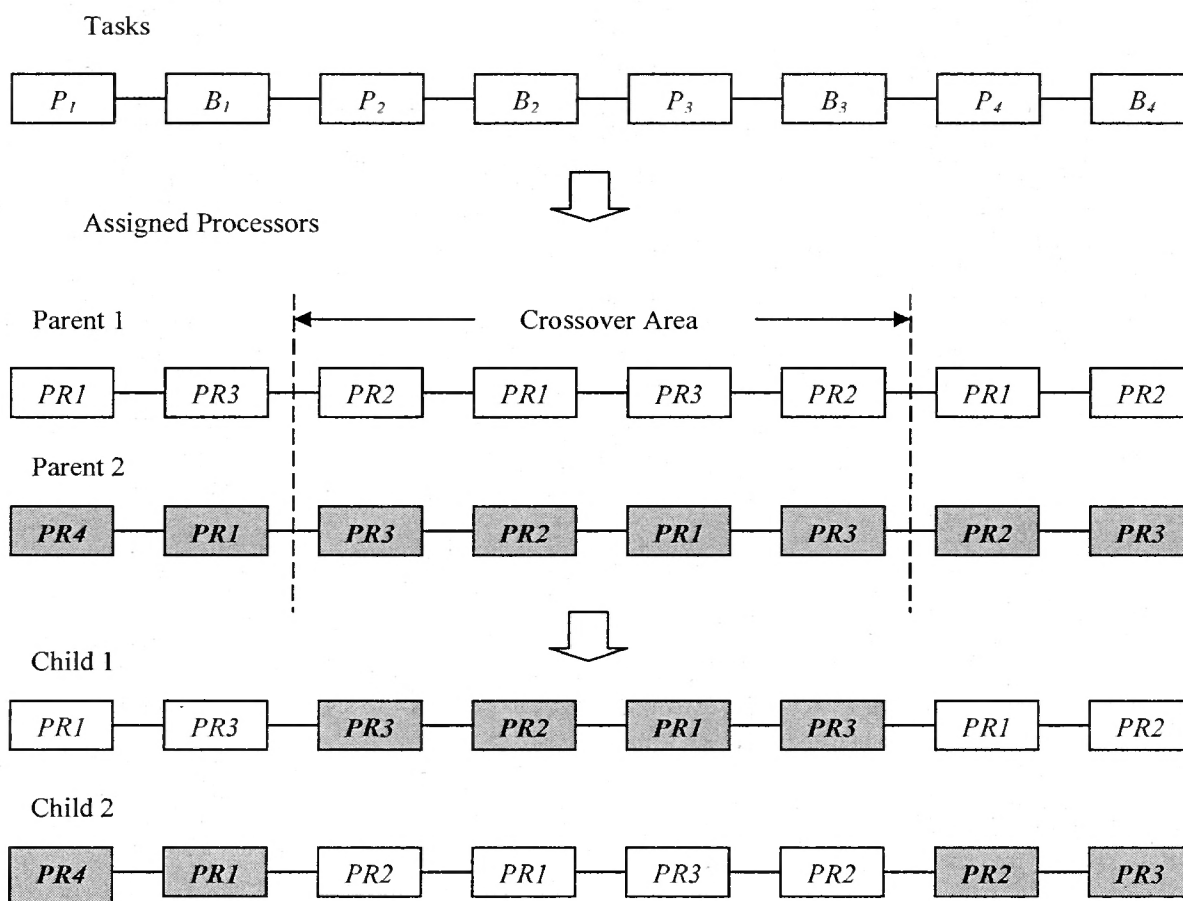


Figure 6.16 Two-point crossover

```
k = population size

Repeat k times
  Generate a random number [0, 1]

  /* check the probability */
  If the number < crossover_probability
    Select parent at random

    Generate a random number (1 or 2)

    If the number is 1
      Perform one-point crossover
    Else
      Perform two-point crossover
    End-IF

    /* check duplication */
    If offspring is a duplicate
      Delete offspring
      Go back to the repeat statement
    End-If

    For each primary and backup copy
      /* Hybrid Task Allocation Method */
      Select the task using HTAM

      /* search time slot */
      Search EST
      Assign the task to the processor
    End-For

    Put the schedule into a queue

  End-IF
End-Repeat
```

**Figure 6.17 Crossover**



### 6.7.9.3 Mutation

“Mutation” is the process to alter some segments in the individual. Mutation enables GAs to maintain diversity in the population, and as a result, it also prevents it from converging prematurely. There are several methods used for mutation. For example, two segments in the individual are selected at random and swapped so that the new individual is slightly different from the original one. In GRFS, the method called *rotation* is used. During the operation, two rotational positions are selected at random. Then, everything between these two positions in the individual is rotated either upwardly or downwardly. An example of upward rotation is illustrated in Figure 6.18.

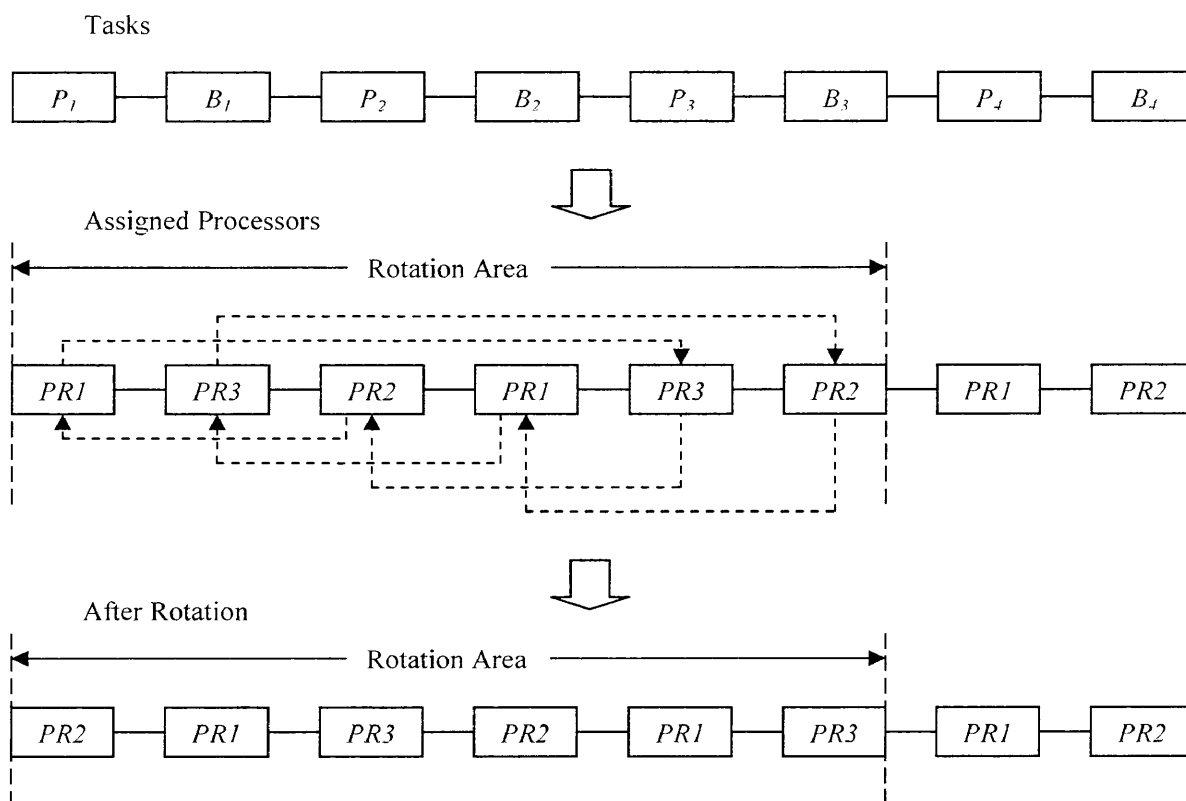


Figure 6.18 Upward rotation

The purpose of mutation in GRFS is the same as the purpose of crossover operation. After crossover operation, each task in the new individual has the assigned processor inherited from its parent. Then, each new individual has the probability, which is very low in general, to be mutated to further change the information of task allocation. Mutation enables GRFS to explore the wide range of solution space. This process is shown in Figure 6.19.

```

/* crossover operation */
Crossover

Generate a random number [0, 1]

/* check the probability */
If the number < mutation_probability
  Select two positions at random

  For each primary and backup copy in the range
    /* rotation */
    Select primary copy  $T_j^P$  which is right next to  $T_i^P$ 
    Select backup copy  $T_j^B$  which is right next to  $T_i^B$ 

    processor( $T_i^P$ ) = processor( $T_j^P$ )
    processor( $T_i^B$ ) = processor( $T_j^B$ )
  End-For

  For each primary and backup copy
    /* Hybrid Task Allocation Method */
    Select the task using HTAM

    /* search time slot */
    Search EST
    Assign the task to the processor
  End-For

  Put the schedule into a queue

End-If

```

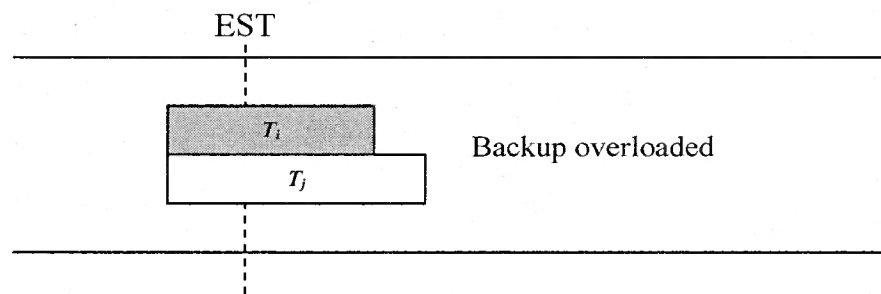
**Figure 6.19 Mutation**

### 6.7.10 Backup Overloading

In this section, we describe two different modes for backup overloading used in GRFS.

#### Natural Backup Overloading

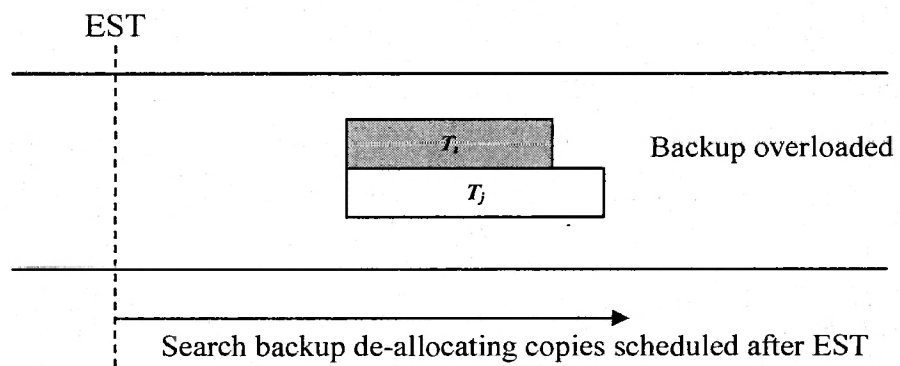
In this mode, the EST for a task is searched in the same manner as for the backup de-allocation technique. If the task can be assigned starting at its EST without overloading, then no overloading occurs. In other words, overloading occurs only when there is a backup de-allocating copy where the task is to be assigned.



**Figure 6.20 Natural backup overloading**

#### Compulsive Backup Overloading

Unlike natural backup overloading, in this mode, GRFS tries to overload as many tasks as possible. Each backup de-allocating copy scheduled after the EST is examined to check whether it is possible to overload the task, and if it is possible, backup overloading occurs. The primary concern in this mode is to increase the number of overloaded tasks rather than assign each task as early as possible.



**Figure 6.21 Compulsive backup overloading**

### 6.7.11 GRFS in Pseudocode

In this section, we present the pseudocode of the entire GRFS.

```

i = number of iterations
g = number of generations
k = initial population size
p = population size

Repeat i times
  /* graph and task creation */
  Generate a graph

  Generate primary and backup copies from the graph

  /* best depth first algorithm */
  Generate a schedule using BDF algorithm

  /* initial population */
  Repeat k times
    Create initial population using heuristics
  End-Repeat

  /* create generations */
  Repeat g times
    /* reproduction */
    Repeat p times
      Select an individual to form new generation
    End-Repeat

    /* crossover */
    Repeat p/2 times
      If random number < crossover_probability
        Perform crossover

        /* mutation */
        If random number < mutation_probability
          Perform mutation
        End-If
      End-If
    End-Repeat
  End-Repeat
End-Repeat

```

**Figure 6.22** The GRFS algorithm

## Chapter 7

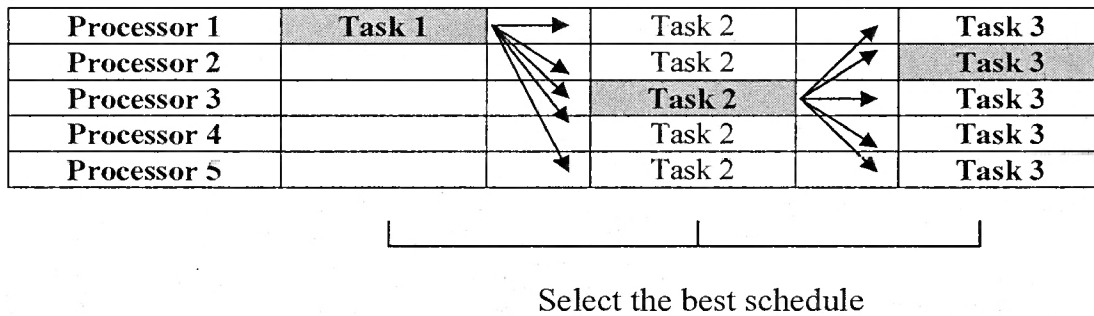
### Simulation Results and Analysis

In this chapter, we compare the simulation results from the heuristic algorithm called Best Depth First (BDF) and GRFS. The simulations were performed with a variety set of parameters to analyze the effectiveness of GRFS in the different settings.

#### 7.1 The Best Depth First (BDF) Algorithm

To analyze the effectiveness of GRFS, we need a means to measure its performance. For this reason, we modified the existing heuristic algorithm and provide the Best Depth First (BDF) algorithm. As its name implies, BDF employs a search method known as *depth-first* search. Depth-first search always expands one of the nodes at the deepest level of the tree. Only when the search hits a dead end does the search go back and expand nodes at more shallow levels.

Suppose that there are total of five processors available in a system. When BDF assigns the first task to a processor, it creates five different temporary schedules, where each schedule corresponds to each processor. Then, BDF evaluates each schedule and assigns the point according to its evaluation. The best schedule is always selected among others, and it is further expanded until the final schedule is obtained. This process is depicted in Figure 7.1.



**Figure 7.1 Schedule in the BDF algorithm**

## 7.2 Input DAG

At the beginning of each simulation, an input DAG is automatically created using two user-defined parameters, the number of tasks and the maximum degree in the graph. The *degree* of a vertex is the number of edge ends at that vertex. Hence, it determines the density of the graph. For instance, if the degree is three, the graph is sparse, and if the degree is six, the graph is dense. Let  $deg(v_i)$  be the degree of vertex  $v_i$ . Then, the degree for each vertex in the graph is chosen at random,  $1 \leq deg(v_i) \leq \text{maximum degree}$ . Other attributes of each task such as the computational amount of time and the deadline are set at random. Finally, an input DAG may be one connected component or have several disconnected components.

## 7.3 BDF vs. GRFS

In this section, we compare the simulation results from BDF and GRFS to analyze how GRFS improves the performance made by BDF. Depending on the maximum degree, we

use four different types of graphs. Table 7.1 shows the parameter list used during the simulations.

<b>Parameter</b>		<b>Value</b>
Graph Type	Maximum Degree	2 - 5
Population Size	Initial Population Size	10
	Maximum Population Size	15
Task/Processor	Number of Tasks	10 - 50
	Number of Processors	3 - 10
Numbers	Number of Iterations	5
	Number of Generations	50
Probability	Crossover Probability	1.0
	Mutation Probability	0.02

**Table 7.1 Parameter list (BDF vs. GRFS)**

Table 7.2 shows the overall result of the simulations. From this table, we can observe that on average GRFS outperforms BDF in all cases. Figure 7.2 through Figure 7.5 reflect the simulation results with different maximum degrees. Each graph in these figures has total of five cases with a varying number of tasks. Finally, if the improvement is positive, it implies that GRFS outperforms BDF, and if it is negative, BDF achieves a better performance than GRFS does.

<b>Maximum Degree</b>	<b>Finishing Time</b>		<b>Improvement (Average)</b>
	<b>BDF</b>	<b>GRFS</b>	
2	63.8	60.3	5.4 %
3	70.6	65.0	8.0 %
4	110.3	107.9	2.1 %
5	155.4	154.1	0.8 %

**Table 7.2 Overall result (BDF vs. GRFS)**



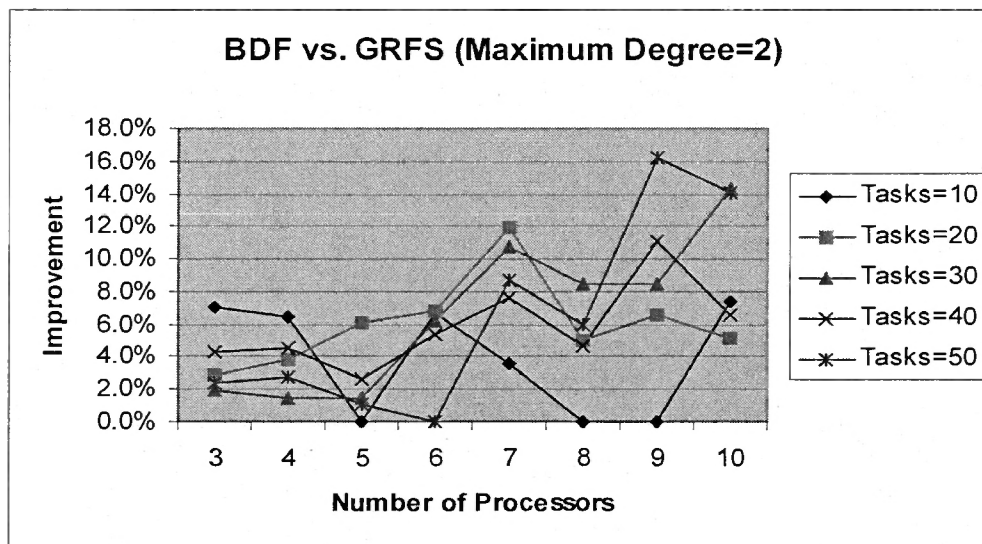


Figure 7.2 BDF vs. GRFS (maximum degree=2)

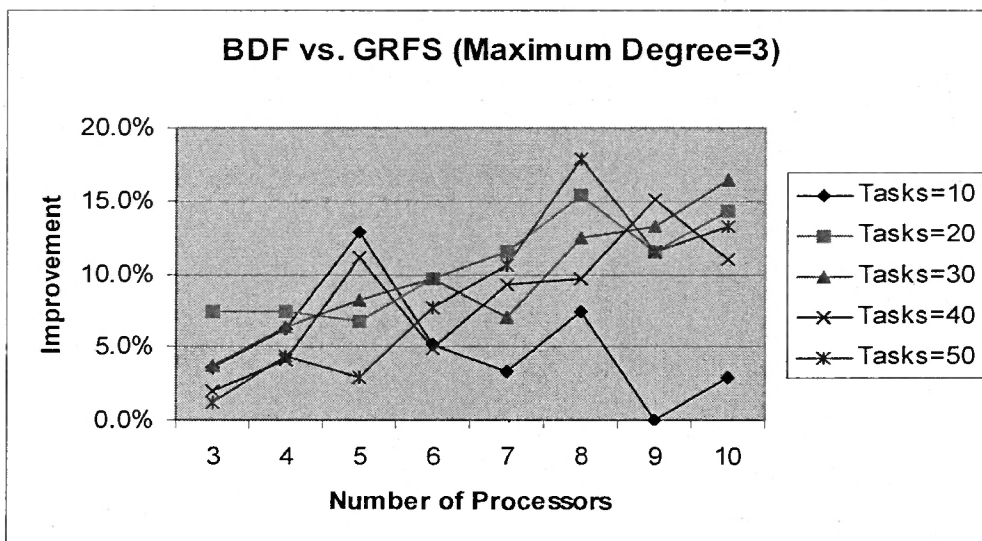


Figure 7.3 BDF vs. GRFS (maximum degree=3)

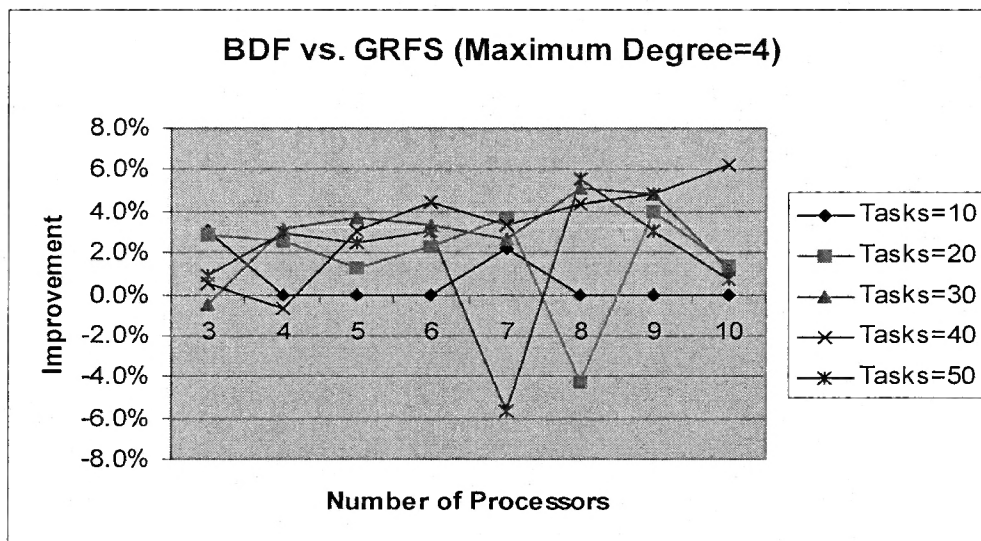


Figure 7.4 BDF vs. GRFS (maximum degree=4)

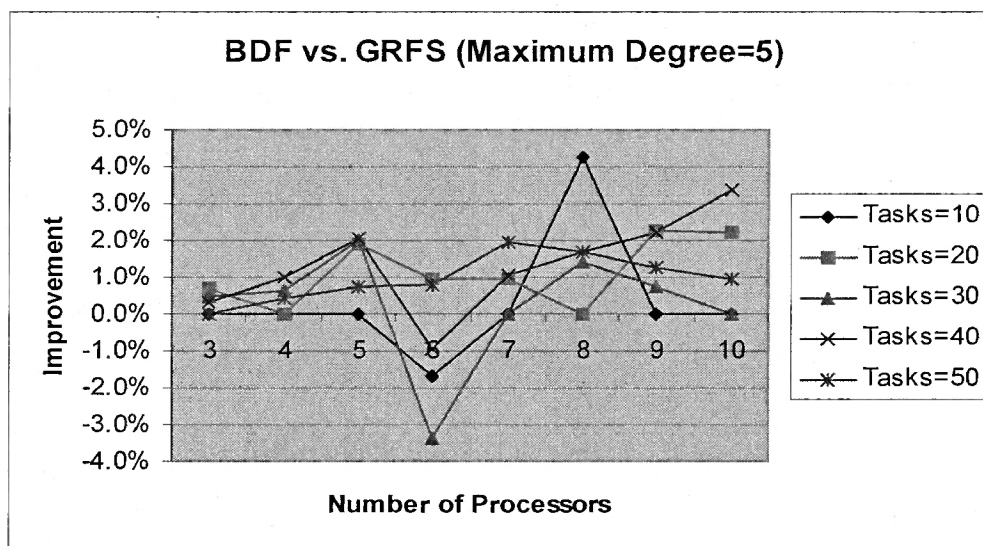
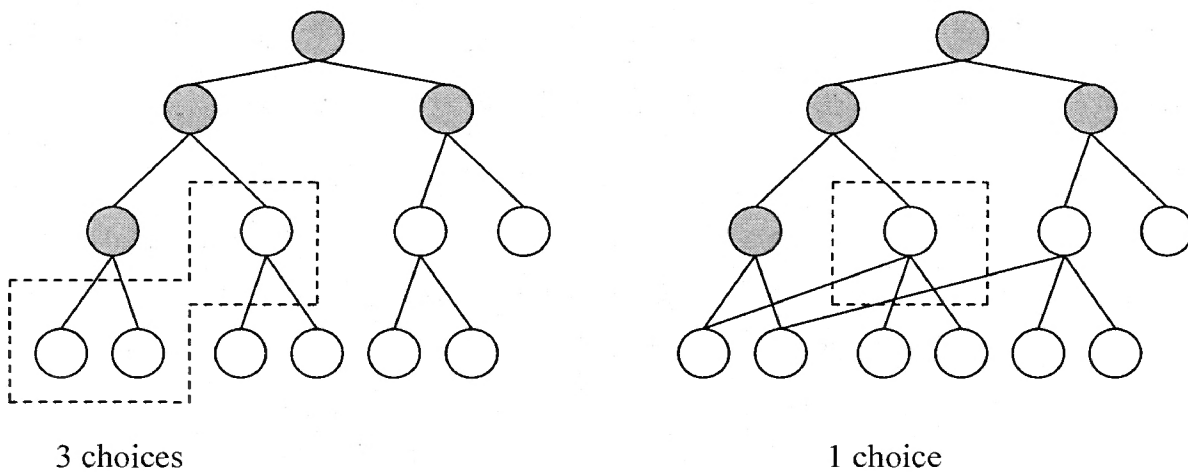


Figure 7.5 BDF vs. GRFS (maximum degree=5)

From these simulation results, we can say that although GRFS outperforms BDF on average, the improvements heavily depend on the maximum degree. This fact is very closely related to the precedence constraints. We explain the relationship between precedence constraints and performance using an example. In Figure 7.6, there is a sparse graph on the left side. We assume that the shaded tasks have already been assigned to the processors, and task allocation was done from left to right at each height in the graph. After assigning the fourth task in the graph, we have three choices: either its sibling or one of its children. However, as in the graph on the right side, if we add two edges to the original graph, we have only one choice because one of the predecessors of the two children has not been assigned yet. From this example, we can see that as the maximum degree increases, the number of choices to select a task decreases, and consequently, it gets harder to discover new solutions.



**Figure 7.6** Precedence constraints and performance

#### 7.4 Maximum Degree

In the previous section, we show that GRFS outperforms BDF. Thus, in the following sections, we focus on the performance of GRFS with different parameters. As the first parameter, we vary the maximum degree. Table 7.3 shows the parameter list, and Table 7.4 shows the overall result.

Parameter		Value
Graph Type	<b>Maximum Degree</b>	<b>2 - 5</b>
Population Size	Initial Population Size	10
	Maximum Population Size	15
Task/Processor	Number of Tasks	10 - 50
	Number of Processors	3 - 10
Numbers	Number of Iterations	5
	Number of Generations	50
Probability	Crossover Probability	1.0
	Mutation Probability	0.02

**Table 7.3 Parameter list (maximum degree)**

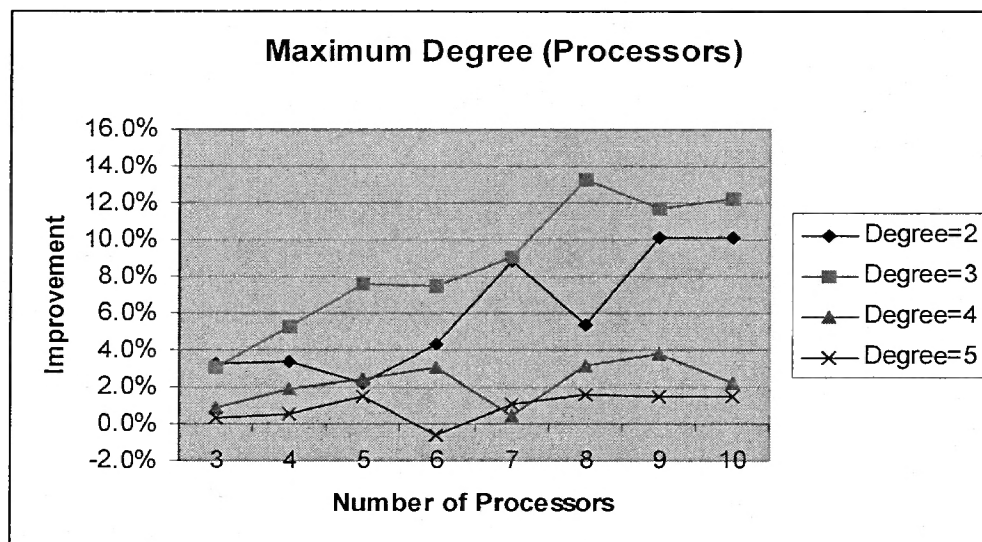
Maximum Degree	Improvement		
	Maximum	Minimum	Average
2	16.2 %	0 %	5.4 %
3	17.9 %	0 %	8.0 %
4	6.3 %	- 5.7 %	2.1 %
5	4.3 %	- 3.4 %	0.8 %

**Table 7.4 Overall result (maximum degree)**

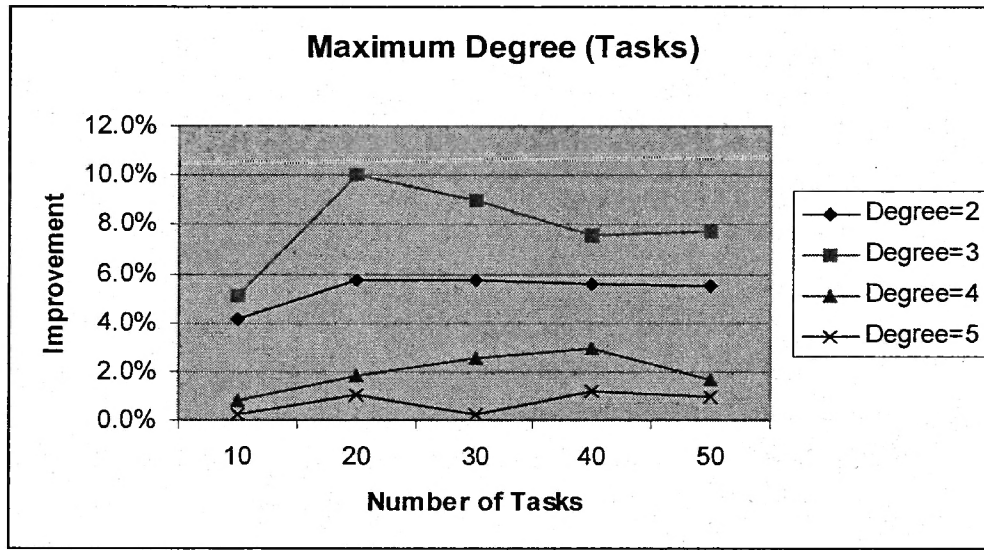
GRFS achieves the highest performance when the maximum degree is three, and as the degree increases, the performance decreases. Figure 7.7 and Figure 7.8 show the same simulation results from a different perspective. In Figure 7.7, we can observe that as the

number of processors increases, the performance also increases because GRFS can take advantage of the flexibility introduced by the number of choices to select a processor for a task allocation. Nonetheless, as the precedence constraints tighten, the effect of this advantage may not be so significant.

Figure 7.8 indicates another fact that as the number of tasks increases, the performance may or may not increase. For example, when the maximum degree is two, the performance reaches a peak with 20 tasks. When the maximum degree is four, the performance reaches a peak with 40 tasks. In general, as the number of tasks increases, the number of candidate solutions grows exponentially, and thus it becomes hard to improve the performance. However, if a graph is dense, increasing the number of tasks relaxes the precedence constraints to a certain degree, which results in improved performance.



**Figure 7.7 Maximum degree (processors)**



**Figure 7.8 Maximum degree (tasks)**

### 7.5 Number of Generations

In this section, we vary the number of generations to analyze the relationship between the number of generations and performance. Table 7.5 shows the parameter list, and Table 7.6 shows the overall result.

Parameter		Value
Graph Type	Maximum Degree	2 - 5
Population Size	Initial Population Size	10
	Maximum Population Size	15
Task/Processor	Number of Tasks	30
	Number of Processors	5
Numbers	Number of Iterations	5
	<b>Number of Generations</b>	<b>0 - 200</b>
Probability	Crossover Probability	1.0
	Mutation Probability	0.02

**Table 7.5 Parameter list (generations)**

Maximum Degree	Improvement		
	Maximum	Generation	Minimum
2	9.7 %	130	- 20.9 %
3	13.7 %	190	- 25.3 %
4	7.0 %	130	- 21.2 %
5	2.9 %	60	- 25.7 %

**Table 7.6 Overall result (generations)**

When the maximum degree is three, GRFS achieves the highest performance at 190 generations. However, simulation results show that if the maximum degree is between two and four, the performance reaches a near peak at generation 130. Also, we should note that when the maximum degree is five, the precedence constraints are too strict so

that even if we create new generations, the performance may not increase at all. Figure 7.9 depicts how the improvements move with the number of generations. GRFS has a relatively quick convergence, and this may be attributed to the fact that GRFS uses three distinct heuristics to create the initial population. In other words, the qualities of the initial individuals are relatively high, and this makes it possible to reach the convergence at early generations.

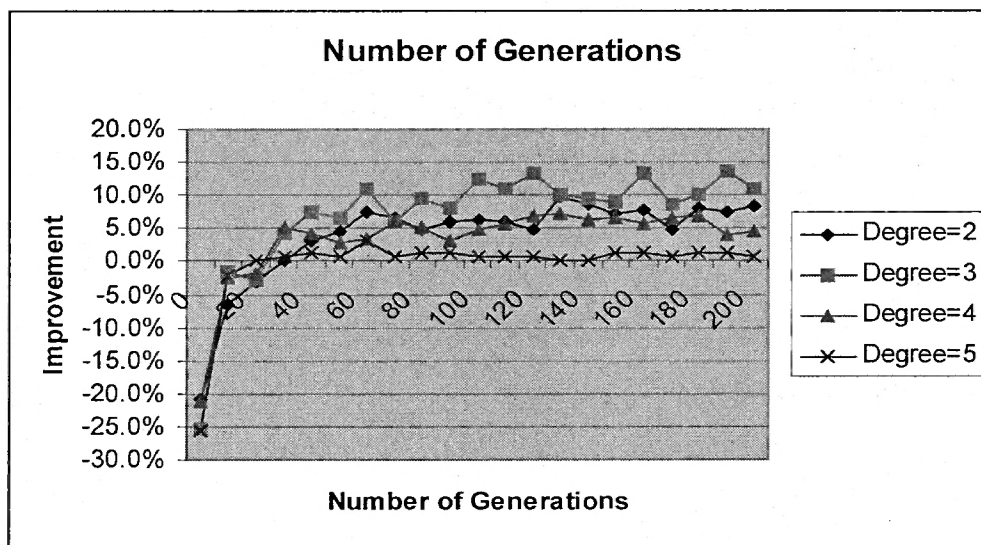


Figure 7.9 Number of generations



## 7.6 Initial Population Size

In this section, we vary the size of the initial population. Table 7.7 shows the parameter list, and Table 7.8 shows the overall result.

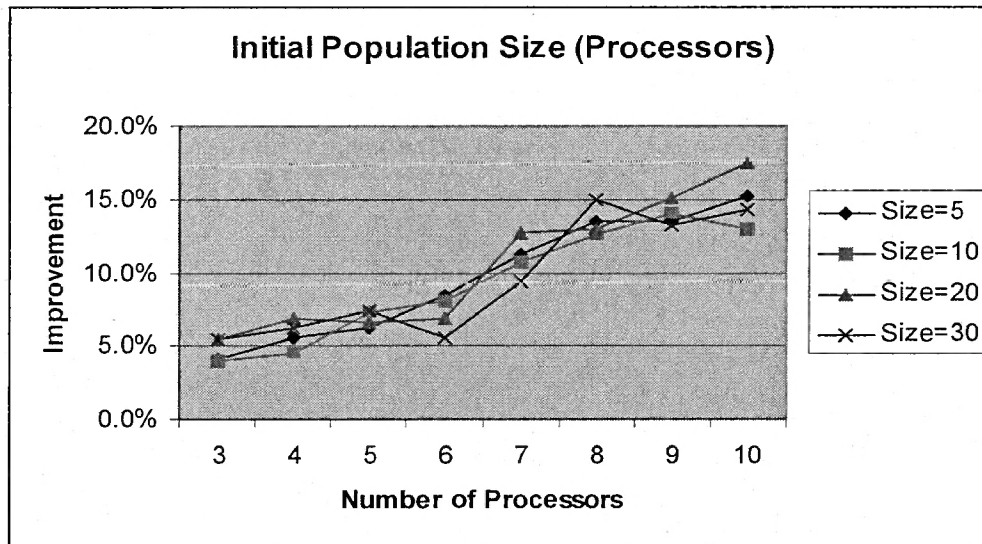
Parameter		Value
Graph Type	Maximum Degree	3
Population Size	<b>Initial Population Size</b>	<b>5,10,20,30</b>
	Maximum Population Size	15
Task/Processor	Number of Tasks	10 - 50
	Number of Processors	3 -10
Numbers	Number of Iterations	5
	Number of Generations	50
Probability	Crossover Probability	1.0
	Mutation Probability	0.02

**Table 7.7 Parameter list (initial population size)**

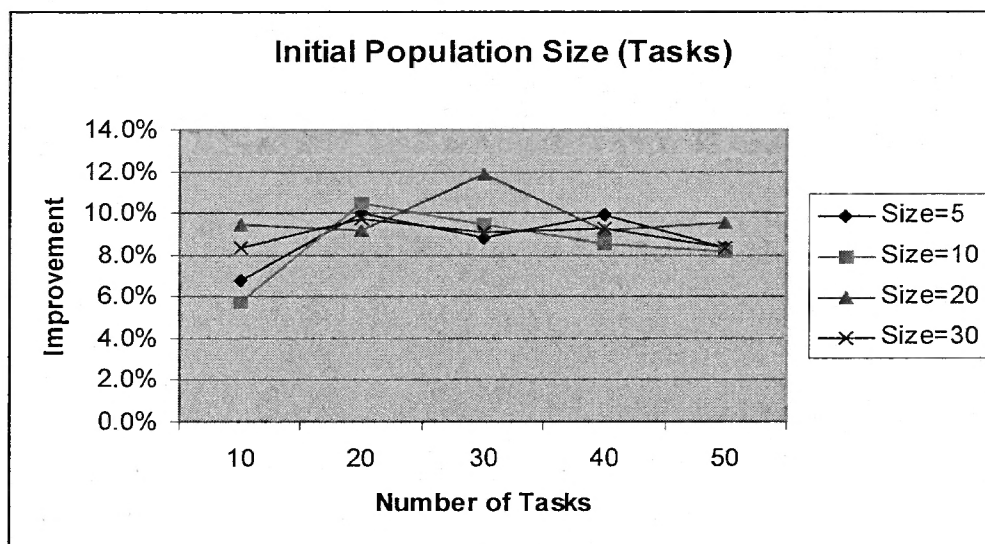
Initial Population Size	Improvement		
	Maximum	Minimum	Average
5	21.6 %	2.3 %	8.9 %
10	19.0 %	2.7 %	8.6 %
20	21.2 %	3.0 %	9.9 %
30	16.7 %	2.9 %	8.9 %

**Table 7.8 Overall result (initial population size)**

GRFS has the highest performance when the size of the initial population is 20. In general, if the size is too small, there is a risk of missing the good individuals in the initial population. In contrast, if the size is too large, there is also a risk of degrading the qualities of individuals in the population as the generations proceed. Since GRFS uses three distinct heuristics to create the initial population, even if its size is small, it does not affect the performance significantly. Figure 7.10 and Figure 7.11 show the results.



**Figure 7.10 Initial population size (processors)**



**Figure 7.11 Initial population size (tasks)**

## 7.7 Maximum Population Size

In this section, we vary the size of the maximum population. Table 7.9 shows the parameter list, and Table 7.10 shows the overall result.

Parameter		Value
Graph Type	Maximum Degree	3
Population Size	Initial Population Size	10
	<b>Maximum Population Size</b>	<b>5 - 60</b>
Task/Processor	Number of Tasks	10 - 50
	Number of Processors	3 -10
Numbers	Number of Iterations	5
	Number of Generations	50
Probability	Crossover Probability	1.0
	Mutation Probability	0.02

**Table 7.9 Parameter list (maximum population size)**

Maximum Population Size	Improvement		
	Maximum	Minimum	Average
5	13.0 %	- 22.9 %	- 0.2 %
10	18.0 %	0 %	7.6 %
20	19.7 %	2.3 %	10.1 %
30	25.0 %	4.1 %	10.9 %
40	21.7 %	3.5 %	11.2 %
50	22.6 %	3.2 %	12.2 %
60	25.6 %	2.4 %	11.7 %

**Table 7.10 Overall result (maximum population size)**

When the size of the maximum population is 5, GRFS has a negative improvement on average. This is because the probability of discovering the good solutions heavily depends on the number and quality of the individuals in the population. If the size of the population is too small, it is simply impossible to explore the wide range of solution

space. In contrast, if the size of the population is too large, it is also possible to keep the individuals with low fitness values in the population, which results in the degradation of the entire quality. We should note that GRFS has the stable improvements when the size of the maximum population increases beyond 20. This effect is brought by the fact that GRFS uses both elitism and roulette wheel schemes to select the individuals to form the new generations. No matter how many individuals exist in the population, it is always guaranteed to pass the best individuals to the following generations. Figure 7.12 and Figure 7.13 show the results.

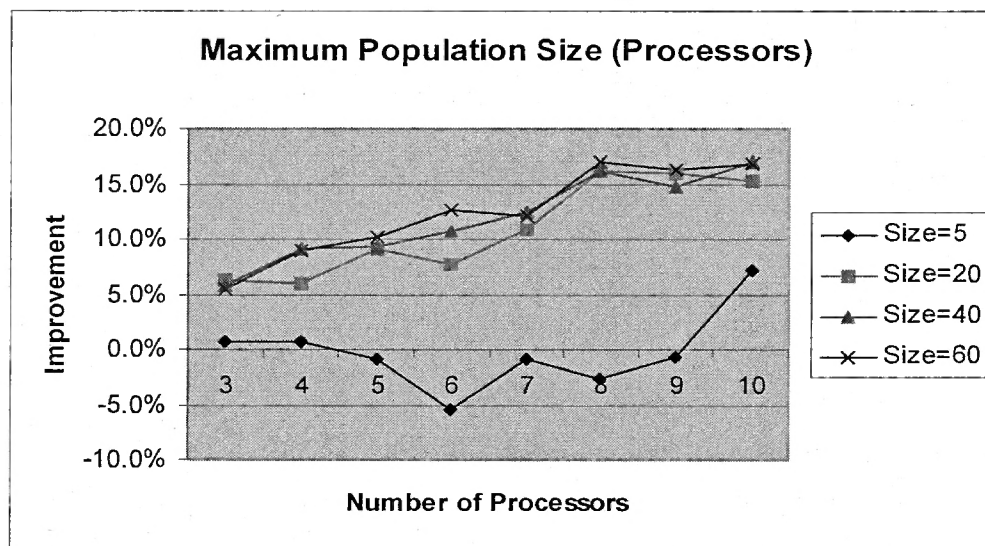


Figure 7.12 Maximum population size (processors)

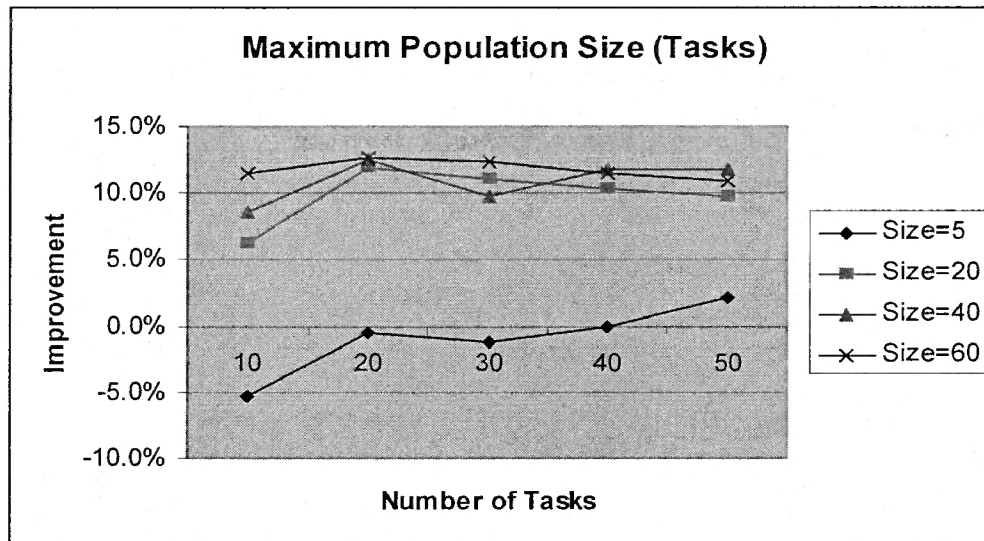


Figure 7.13 Maximum population size (tasks)

## 7.8 Backup Overloading

In this section, we analyze backup overloading technique in terms of finishing time and processor usage. Table 7.11 shows the parameter list.

Parameter		Value
Graph Type	Maximum Degree	3
Population Size	Initial Population Size	10
	Maximum Population Size	15
Task/Processor	Number of Tasks	10 - 50
	Number of Processors	3 - 10
Numbers	Number of Iterations	5
	Number of Generations	50
Probability	Crossover Probability	1.0
	Mutation Probability	0.02

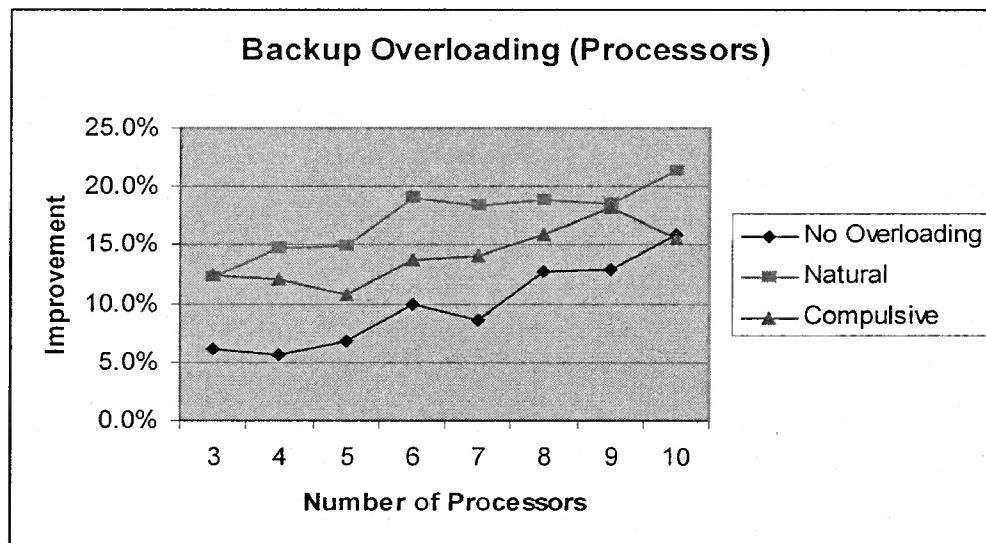
Table 7.11 Parameter list (backup overloading)

## Finishing Time

Overloading Mode	Improvement		
	Maximum	Minimum	Average
No overloading	19.8 %	2.6 %	9.2 %
Natural overloading	26.9 %	5.0 %	16.7 %
Compulsive overloading	24.2 %	5.3 %	13.8 %

**Table 7.12 Overall result1 (backup overloading)**

As we can observe from Table 7.12, natural overloading has the highest improvements on average. This is because in natural overloading mode, GRFS tries to assign each task to a processor as early as possible. In contrast, the major goal of the compulsive overloading mode is to reduce the workload of each processor by overlapping as many tasks as possible. Figure 7.14 and Figure 7.15 show the simulation results.



**Figure 7.14 Backup overloading (processors)**

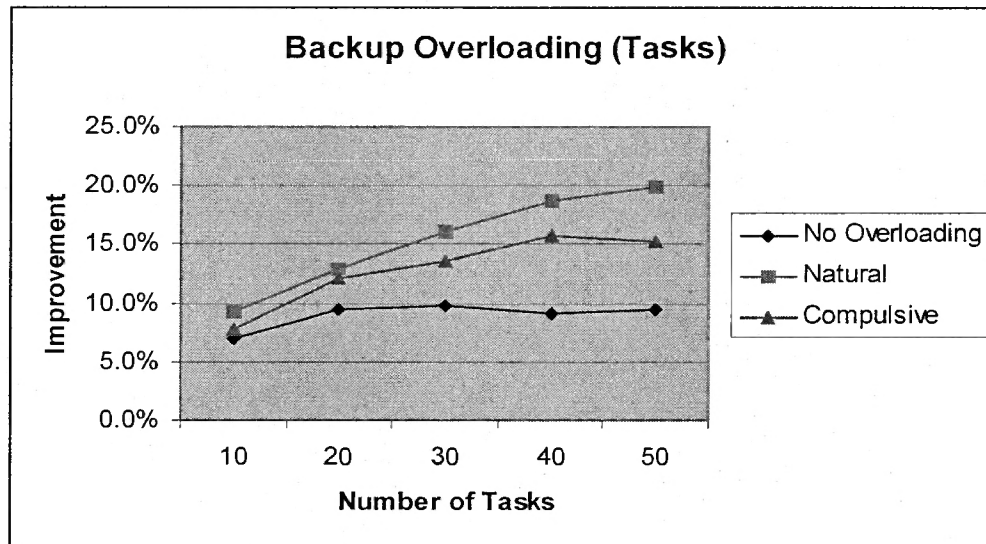


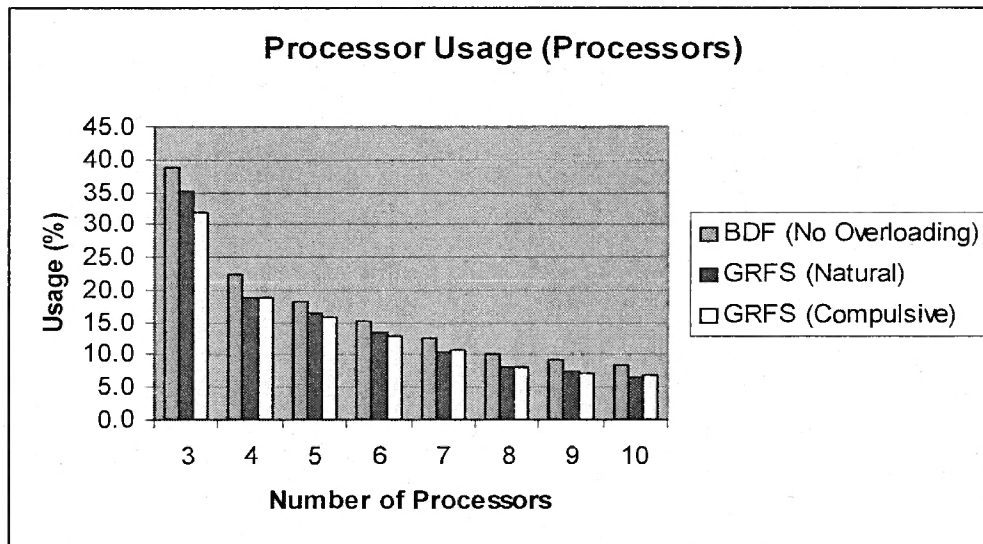
Figure 7.15 Backup overloading (tasks)

### Processor Usage

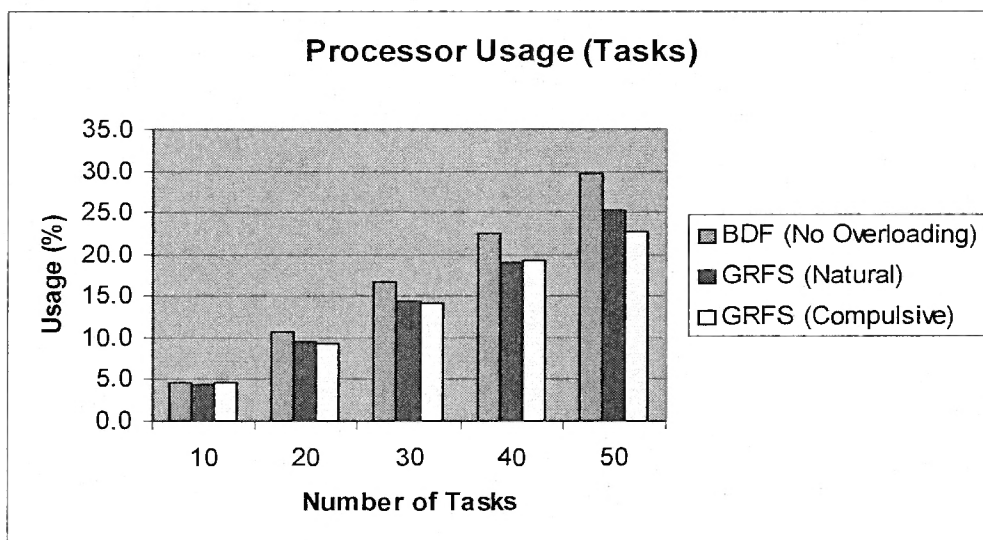
Overloading Mode	Processor Usage		
	Maximum	Minimum	Average
BDF with no overloading	69.3 %	2.3 %	16.8 %
GRFS with natural overloading	63.6 %	1.9 %	14.5 %
GRFS with compulsive overloading	49.3 %	1.8 %	14.0 %

Table 7.13 Overall result2 (backup overloading)

GRFS in compulsive overloading mode has the lowest processor usage, especially when the number of processors is small or the number of tasks is large (Figure 7.16 and Figure 7.17). This is because as the number of processors increases, the number of backup de-allocating copies scheduled in each processor decreases. As a result, a chance for backup overloading also decreases. This is also true if the number of total tasks is small.



**Figure 7.16 Processor usage (processors)**



**Figure 7.17 Processor usage (tasks)**



## Chapter 8

### Conclusion and Future Work

#### 8.1 Conclusion

A scheduling problem for both uniprocessor and multiprocessor systems is known to be NP-complete in its general form. Therefore, many researchers have proposed various heuristic algorithms to produce near-optimal solutions. Among these heuristics, three classical and also well-known algorithms are: the Rate-Monotonic (RM) algorithm, the Deadline-Monotonic (DM) algorithm, and the Earliest-Deadline-First (EDF) algorithm. Although these algorithms were originally proposed for uniprocessor systems, later on they were adapted for multiprocessor systems and also fault-tolerant systems.

As an alternative to traditional heuristic approaches, optimization methods such as simulated annealing, tabu search, and genetic algorithms have been adapted to solve various NP-complete problems. In fact, several researchers have tackled the scheduling problems on multiprocessor systems using these optimization methods, and proven their effectiveness. Nonetheless, almost none of these methods have been used for fault-tolerant systems.

In this thesis, we adapted a genetic algorithm and presented our Genetic Real-time Fault-tolerant Scheduling (GRFS) algorithm to take a new approach to address real-time fault-tolerant scheduling. We also modified the existing heuristic algorithm and provided the Best-Depth-First (BDF) algorithm to compare the simulation results with those from

GRFS. In addition, we used different types of DAG to test these algorithms in different environments.

The simulation results showed that GRFS outperformed BDF in almost all cases. GRFS has, overall, 8.0% higher performance than BDF, with the highest improvement 17.9% for a graph whose maximum degree is three. We also tested the performance of GRFS with different parameters and the result for each case is summarized below.

1. Maximum Degree

When the maximum degree is three, GRFS achieved the highest performance (8.0% on average and 17.9% as the highest).

2. Number of Generations

When the maximum degree is three, GRFS achieved the highest performance at 190 generations (13.7% as the highest).

3. Initial Population Size

When the initial population size is 20, GRFS achieved the highest performance (9.9% on average and 21.2% as the highest).

4. Maximum Population Size

When the maximum population size is 50 and 60, GRFS achieved the highest performance (12.2% on average when the size is 50 and 25.6% as the highest when the size is 60).

5. Backup Overloading

With natural backup overloading, GRFS achieved the highest performance (16.7% on average and 26.9% as the highest). As for processor usage, compulsive overloading resulted in the lowest (14.0% on average and 49.3% as the lowest).

The improvements achieved by GRFS are attributed to the fact that GRFS can take advantage of the flexibility when the precedence constraints in a graph are not so strict. This strategy also works well when the number of processors increases. We can conclude that when the restrictions in the system are not so strict and there exists some flexibility

in the system, the genetic algorithm has a good chance to maximize that flexibility and yield excellent outcomes.

As for backup overloading technique, we presented two different approaches. Since each approach has its own advantages and disadvantages, the choice should be made depending on the requirements and demands of a system. If minimizing the schedule length is the primary concern, natural backup overloading may be the choice. If reducing the processor utility is the major goal, compulsive backup overloading may work well.

## **8.2 Future Work**

In general, a scheduling system uses some graphical representations such as a Gantt chart to display the simulation results. Thus, it is necessary to provide a user-friendly interface. From this point of view, developing the simulator using a Graphical User Interface (GUI) is more desirable.

Since GAs require many iterations to produce the final solution, a fitness function and any other computations in GAs should be compact and fast. Although GRFS is not so complicated, it still has some computational overheads such as checking predecessors each time a task is assigned as in Hybrid Task Allocation Method (HTAM). It may be necessary to reduce some overheads to speed up the entire process.

Finally, more sophisticated way to integrate height-based allocation and predecessor-based allocation while emphasizing the importance of maintaining every single deadline may be necessary to realize more effective scheduling system.

## References

- [Amphlett & Bull 96] R. W. Amphlett and D. R. Bull, "Genetic Algorithm Based DSP Multiprocessor Scheduling," *IEEE International Symposium on Circuits and Systems*, Vol. 2, pages 253-256, May 1996.
- [Audsley *et al.* 91] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings, "Hard Real-Time Scheduling: The Deadline-Monotonic Approach," *Proceedings of the 8<sup>th</sup> IEEE Workshop on Real-Time Operating Systems and Software*, 1991.
- [Baker 85] J. E. Baker, "Adaptive Selection Methods for Genetic Algorithms," *Proceedings of the First International Conference on Genetic Algorithms and their Applications*, pages 101-111, 1985.
- [Baruah & Goossens 03] S. K. Baruah and J. Goossens, "Rate-Monotonic Scheduling on Uniform Multiprocessors," *IEEE Transactions on Computers*, 52(7), pages 966-970, July 2003.
- [Baruah *et al.* 90] S. K. Baruah, L. E. Rosier, and R. R. Howell, "Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic, Real-Time Tasks on One Processor," *Real-Time Systems*, 2(4), pages 301-324, 1990.
- [Beasley *et al.* 93] D. Beasley, D. R. Bull, and R. R. Martin, "An Overview of Genetic Algorithms: Part 1, Fundamentals," Technical Report, University of Purdue, 1993.
- [Bertossi *et al.* 99] A. A. Bertossi, L. V. Mancini, and F. Rossini, "Fault-Tolerant Rate-Monotonic First-Fit Scheduling in Hard-Real-Time Systems," *IEEE Transactions on Parallel and Distributed Systems*, 10(9), pages 934-945, September 1999.
- [Brindle 81] A. Brindle, "Genetic Algorithms for Function Optimization," Ph.D. Thesis University of Alberta, Canada, 1981.
- [Burchard *et al.* 95] A. Burchard, J. Liebeherr, Y. Oh, and S. H. Son, "New Strategies for Assigning Real-Time Tasks to Multiprocessor Systems," *IEEE Transactions on Computers*, 44(12), pages 1429-1442, December 1995.
- [Chung & Dietz 95] T. M. Chung and H. G. Dietz, "Adaptive Genetic Algorithms: Scheduling Hard Real-Time Control Programs with Arbitrary Timing Constraints," Technical Report, School of Electrical Engineering, Purdue University, May 1995.
- [Cormen *et al.* 90] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to Algorithms," MIT Press, Cambridge, MA, 1990.

- [Corrêa *et al.* 99] R. C. Corrêa, A. Ferreira, and P. Rebreyend, "Scheduling Multiprocessor Tasks with Genetic Algorithms," *IEEE Transactions on Parallel and Distributed Systems*, 10(8), pages 825-837, August 1999.
- [Cramer 85] N. L. Cramer, "A Representation for the Adaptive Generation of Simple Sequential Programs," *Proceedings of the International Conference on Genetic Algorithms and their Applications*, pages 183-187, July 1985.
- [Davis 91] L. Davis, "Handbook of Genetic Algorithms", Van Nostrand Reinhold, New York 1991.
- [De Jong 75] K. A. De Jong, "An Analysis of the Behavior of a Class of Genetic Adaptive Systems," Ph.D. Thesis, University of Michigan, 1975.
- [Dhall & Liu 78] S. Dhall and C. Liu, "On a Real Time Scheduling Problem," *Operations Research*, 26(1), pages 127-141, January-February 1978.
- [DiNatale & Stankovic 95] M. DiNatale and J. A. Stankovic, "Applicability of Simulated Annealing Methods to Real-Time Scheduling and Jitter Control," *Proceedings of the 16<sup>th</sup> IEEE Real-Time Systems Symposium*, pages 190-199, December 1995.
- [Dorn 95] J. Dorn, "Iterative Improvement Methods for Knowledge-based Scheduling," *AI Communications*, 8(1), pages 20-34, March 1995.
- [Fang 94] H. L. Fang, "Genetic Algorithms in Timetabling and Scheduling," Ph.D. Thesis, Department of Artificial Intelligence. University of Edinburgh, Scotland, 1994.
- [Garey & Johnson 79] M. R. Garey and D. S. Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness," W. H. Freeman and Company, San Francisco, 1979.
- [Ghosh *et al.* 97] S. Ghosh, R. Melhem, and D. Mossé, "Fault-Tolerance through Scheduling of Aperiodic Tasks in Hard Real-Time Multiprocessor Systems," *IEEE Transactions on Parallel and Distributed Systems*, 8(3), pages 272-284, 1997.
- [Ghosh *et al.* 98] S. Ghosh, R. Melhem, D. Mossé, and J. S. Sarma, "Fault-Tolerant Rate-Monotonic Scheduling," *Real-Time Systems*, 15(2), pages 149-181, 1998.
- [Glover 89] F. Glover, "Tabu Search-Part I," *ORSA Journal on Computing*, 1(3), pages 190-206, 1989.
- [Goldberg 89] D. E. Goldberg, "Genetic Algorithms in Search, Optimization, and Machine Learning," Addison-Wesley, MA, 1989.

[Goldberg 90] D. E. Goldberg, "A note on Boltzmann Tournament Selection for Genetic Algorithms and Population-Oriented Simulated Annealing," *Complex Systems*, pages 445-460, 1990.

[Holland 75] J. H. Holland, "Adaptation in Natural and Artificial Systems," The University of Michigan Press, Ann Arbor, 1975.

[Hou & Shin 94] C. J. Hou and K. G. Shin, "Replication and Allocation of Task Modules in Distributed Real-Time Systems," *Twenty-Fourth International Symposium on Fault-Tolerant Computing*, pages 26-35, June 1994.

[Hou *et al.* 94] E. S. H. Hou, N. Ansari, and H. Ren, "A Genetic Algorithm for Multiprocessor Scheduling," *IEEE Transactions on Parallel and Distributed System*, 5(2), pages 113-120, February 1994.

[Isović & Fohler 00] D. Isović and G. Fohler, "Efficient Scheduling of Sporadic, Aperiodic, and Periodic Tasks with Complex Constraints," *Proceedings of the Real-Time Systems Symposium*, The 21<sup>st</sup> IEEE, pages 207-216, November 2000.

[Jalote 98] P. Jalote, "Fault Tolerance in Distributed Systems," Prentice-Hall, Inc, NJ, 1998.

[Jeffay & Stone 93] K. Jeffay and D. L. Stone, "Accounting for Interrupt Handling Costs in Dynamic Priority Task Systems," *Proceedings of IEEE Real-Time Systems Symposium*, pages 212-221, December 1993.

[Joseph & Pandya 86] M. Joseph and P. Pandya, "Finding Response Times in a Real-Time Systems," *The Computer*, Vol. 29, pages 390-395, October 1986.

[Kianzad and Bhattacharyya 01] V. Kianzad and S. S. Bhattacharyya, "Multiprocessor Clustering for Embedded Systems," *Proceedings of the European Conference on Parallel Computing*, pages 697-701, August 2001.

[Kirkpatrick *et al.* 83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, 220, pages 671-680, 1983.

[Koza 92] J. R. Koza, "Genetic Programming: On the Programming of Computers by Means of Natural Selection," MIT Press, MA, 1992.

[Lawler & Wood 66] E. L. Lawler and D. E. Wood, "Branch-and-bound methods: A survey," *Operations Research*, pages 699-719, July 1966.

[Lee and Chen 03] Y-H. Lee and C. Chen, "A Modified Genetic Algorithm for Task Scheduling in Multiprocessor Systems," National Chiao Tung University, Taiwan R.O.C, 2003.

[Lehoczky *et al.* 89] J. Lehoczky, L. Sha, and Y. Ding, "The Rate Monotonic Algorithm: Exact Characterization and Average Case Behavior," *Proceedings of the Real-Time Systems Symposium*, 1989.

[Leung & Whitehead 82] J. Y. T. Leung and J. Whitehead, "On the Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks," *Performance Evaluation*, 2(4), pages 237-250, December 1982.

[Liu *et al.* 94] J. W. S. Liu, W. K. Shin, K. J. Lin, R. Bettani, and J. Y. Chung, "Imprecise Computations," *Proceedings of the IEEE*, 82(1), pages 83-94, January 1994.

[Liu & Layland 73] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM*, 20(1), pages 46-61, January 1973.

[Manimaran & Murthy 98] G. Manimaran and C. S. R. Murthy, "A Fault-Tolerant Dynamic Scheduling Algorithm for Multiprocessor Real-Time Systems and Its Analysis," *IEEE Transactions on Parallel and Distributed Systems*, 9(11), pages 1137-1152, November 1998.

[Oh & Son 93] Y. Oh and S. H. Son, "Tight Performance Bounds of Heuristics for a Real-Time Scheduling Problem," Technical Report CS-93-24, University of Virginia, May 1993.

[Peng *et al.* 97] D. Peng, K. G. Shin, and T. F. Abdelzaher, "Assignment and Scheduling Communicating Periodic Tasks in Distributed Real-Time Systems," *IEEE Transactions on Software Engineering*, 23(12), pages 745-758, December 1997.

[Qin *et al.* 02] X. Qin, H. Jiang, and D. R. Swanson, "An Efficient Fault-tolerant Scheduling Algorithm for Real-time Tasks with Precedence Constraints in Heterogeneous Systems," Technical Report TR-UNL-CSE 2002-0501, University of Nebraska-Lincoln, February 2002.

[Ramamritham & Stankovic 94] K. Ramamritham and J. Stankovic, "Scheduling Algorithms and Operating Systems Support for Real-Time Systems," *Proceedings of the IEEE*, 82(1), pages 55-67, January 1994.

[Ramanathan 97] P. Ramanathan, "Graceful degradation in real-time control applications using (m,k)-firm guarantee," *Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing*, pages 132-141, June 1997.

[Rinehart *et al.* 03] M. Rinehart, V. Kianzad, and S. S. Bhattacharyya, "A Modular Genetic Algorithm for Scheduling Tasks Graphs," Technical Report, UMIACS-TR-2003-66, University of Maryland, June 2003.

[Sabine & Sha 94] N. M. Sabine and E. H. M. Sha, "Including Selective Fault-Tolerance in Real-Time Multiprocessor Schedules," Research Report CSE-TR-94-23, September 1994.

[Sarkar 89] V. Sarkar, "Partitioning and Scheduling Parallel Programs for Multiprocessors," MIT Press, 1989.

[Stankovic *et al.* 94] J. A. Stankovic, M. Spuri, M. D. Natale, and G. Buttazzo, "Implications of Classical Scheduling Results for Real-Time Systems," *IEEE Transactions on Computer*, 28:16 – 25, June 1994.

[Syswerda 89] G. Syswerda, "Uniform Crossover in Genetic Algorithms," *Proceedings of the Third International Conference on Genetic Algorithms and their Applications*, pages 2-9, 1989.

[Thesen 98] A. Thesen, "Design and Evaluation of Tabu Search Algorithms for Multiprocessor Scheduling," *Journal of Heuristics*, 4(2), pages 141-160, 1998.

[Tindell *et al.* 92] K. Tindell, A. Burns, and A. Wellings, "Allocating Hard Real Time Tasks: An NP-Hard Problem Made Easy," *Real-Time Systems*, 4(2), pages 145-165, May 1992.

[Ullman 75] J. Ullman, "NP-Complete Scheduling Problems," *Journal of Computing System Science*, Vol. 10, pages 384-393, 1975.

[Wang & Korfhage 95] P. Wang and W. Korfhage, "Process Scheduling Using Genetic Algorithms," *IEEE Symposium on Parallel and Distributed Processing*, pages 638-641, 1995.

[Whitley 89] D. Whitley, "The GENITOR Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is Best," *Proceedings of the Third International Conference on Genetic Algorithms*, pages 116-121, 1989.

[Xu and Parnas 90] J. Xu and D. L. Parnas, "Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations," *IEEE Transactions on Software Engineering*, 16(3), pages 360-369, March 1990.

[Xu 93] J. Xu, "Multiprocessor Scheduling of Processes with Release Times, Deadlines, Precedence, and Exclusion Relations," *IEEE Transactions on Software Engineering*, 19(2), pages 139-154, February 1993.



[Yang & Gerasoulis 94] T. Yang and A. Gerasoulis, "DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors," *IEEE Transactions on Parallel and Distributed Systems*, 5(9), pages 951-967, September 1994.

[Zhong & Yang 03] Y-W. Zhong and J-G. Yang, "A Genetic Algorithm for Tasks Scheduling in Parallel Multiprocessor Systems," *Proceedings of the Second International Conference on Machine Learning and Cybernetics*, pages 1785-1790, November 2003.