INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

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

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

**A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600**

Order Number 9516954

**On-line hard real-time scheduling of parallel tasks on
partitionable multiprocessors**

Babbar, Davender, Ph.D.

The Ohio State University, 1994

U·M·I

300 N. Zeeb Rd.
Ann Arbor, MI 48106

ON-LINE HARD REAL-TIME SCHEDULING OF PARALLEL TASKS ON PARTITIONABLE MULTIPROCESSORS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the Graduate
School of The Ohio State University

By

Davender Babbar, B.Tech., M.S.

* * * * *

The Ohio State University

1994

Dissertation Committee:

Phillip Krueger

Mukesh Singhal

P. Sadayappan

Approved by

Adviser

Department of Computer and
Information Science

© Copyright by

Davender Babbar

1994

DEDICATED TO
my parents and my wife

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude and sincere thanks to my advisor Prof. Phillip Krueger for his guidance, encouragement and support. His high standards of research and scientific approach taught me what constitutes quality research. Not only he is a great teacher but also a continuous source of inspiration and moral support. His encouragement and critical feedback helped me at every step of the research discussed in this dissertation. I would like to thank the other members of my committee, Prof. Mukesh Singhal and Prof. P. Sadayppan for their advice and critical reviews.

I am indebted to Prof. Mervin Muller for his support and advice. He has been a constant source of wisdom and encouragement. As a teacher and a guide he introduced me to the art of experimental performance analysis.

I would like to thank all my friends who helped me stay sane and enjoy life. These friends made my long stay OSU full of memorable moments. My sincere gratitude to all my friends, including: Sanjay, Sona, Kaushik, Anup, Rohini, Sunil, Jyothi, Kumar, Tarik, Ashwani, Sapna, Amit, Manish, Rohit, Jyotsna, Usha, Ratna, Kalyan, and Feroze. My sincere thanks to Shivnandan Kaushik for proof-reading parts of my thesis. I would like to thank Manas Mandal and Pete Ware for their help with typesetting this dissertation.

This dissertation would not have been possible without the love, affection and moral support of my wife Savita and my parents. This dissertation is dedicated to my family members who stood by me during my difficult times.

VITA

September 1, 1966	Born - Faridabad, Haryana, India
1988	B.Tech. Electronics Engineering, Banaras Hindu University, India
1990	M.S. Computer Science, The Ohio State University
1988-present	Dept. of Computer and Information Science, The Ohio State University.

Publications

Research Publications

D. Babbar and P. Krueger, "On-line Scheduling of Hard Real-Time Parallel Tasks on Partitionable Multiprocessors", *Proc. International Conference on Parallel Processing*, vol. II, pages 29–38, Aug. 1994.

D. Babbar and P. Krueger, "A Performance Comparison of Processor Allocation and Job Scheduling for Mesh-Connected Multiprocessors", accepted for the Sixth Symposium on Parallel and Distributed Processing, Oct. 1994.

Fields of Study

Major Field: Computer and Information Science

Studies in:

Operating Systems	Prof. Phillip Krueger
Parallel and Distributed Systems	Prof. Mukesh Singhal
Software Engineering	Prof. Gary Perlman

TABLE OF CONTENTS

DEDICATION	ii
ACKNOWLEDGEMENTS	iii
VITA	v
LIST OF TABLES	x
LIST OF FIGURES	xi
CHAPTER	PAGE
I Introduction	1
1.1 Real-Time Systems	1
1.1.1 Aspects and Constraints	2
1.2 Real-Time Scheduling	5
1.3 Problem Statement	8
1.3.1 Motivation	9
1.4 Thesis Organization and Overview	10
II Related Work	13
2.1 Introduction	13
2.2 Hard Real-Time Scheduling Algorithms for Uniprocessor Systems	17
2.2.1 Static Scheduling Algorithms	17
2.2.2 Dynamic Scheduling Algorithms	20
2.3 Multiprocessor Systems	23
2.4 Hard Real-Time Scheduling Algorithms for Multiprocessor Systems . .	24
2.4.1 Static Scheduling Algorithms	24
2.4.2 Dynamic Scheduling Algorithms	27

2.5	Scheduling of Parallel Tasks	29
2.5.1	Processor Allocation	29
2.5.2	Real-Time Scheduling of Parallel Tasks	30
2.5.3	Similarity to Bin-Packing Results	32
2.6	Imprecise Computations	33
III	Framework	34
3.1	System Model	34
3.2	Process Model	35
3.3	Workload Model	37
3.4	Performance	38
IV	Problem Decomposition and Design of Heuristics	40
4.1	Introduction	40
4.2	Problem Decomposition and Solution Framework	42
4.3	Subcube Identification	45
4.3.1	The Buddy Strategy	47
4.3.2	Gray Code Strategy	54
4.3.3	Multiple Gray Code Strategy	56
4.3.4	Optimal Strategy	57
4.4	Job Ordering	58
4.4.1	First-Come-First-Served	58
4.4.2	Largest-Dimension-First	59
4.4.3	Earliest-Deadline-First	59
4.4.4	Minimum-Laxity-First	60
4.4.5	Complexity of Job Ordering	60
4.5	Subcube Selection	61
4.5.1	First-Suitable-Subcube	61
4.5.2	Earliest-Start-Time	64
4.5.3	Stacking	66
4.6	Overall Complexity	69
4.7	Performance Comparison	71
4.7.1	Introduction	71
4.7.2	Results	73
4.8	Appropriate Level of Preemption	88
4.8.1	Levels of Preemption	88
4.8.2	Performance Comparison	91

4.9	Summary and Conclusions	94
V	Complex Workloads	98
5.1	Introduction	98
5.2	Resource Reclaiming When Jobs Complete Early	100
5.2.1	The Need for Flexibility	102
5.2.2	Data-Structures	104
5.2.3	The LST Subcube Selection Heuristic	106
5.2.4	Making Use of the “Holes”	107
5.2.5	Performance Comparison of the Subcube Selection Policies	115
5.3	Priority Constraints	124
5.3.1	Strict Priority Ordering	126
5.3.2	Exploring Intermediate Orderings	128
5.3.3	Results	130
5.4	Summary and Conclusions	136
VI	Mesh-Connected Systems	138
6.1	Introduction	138
6.2	Processor Allocation Algorithms	140
6.2.1	Overview of the Algorithms	141
6.2.2	Performance Evaluation	154
6.2.3	Conclusions	163
6.3	Submesh Identification and Submesh Selection	165
6.3.1	Search Strategy	166
6.3.2	Submesh Selection	167
6.3.3	Submesh Identification	170
6.4	Job Ordering	172
6.5	Performance Comparison	172
6.6	Summary and Conclusions	178
VII	Summary and Future Research	181
7.1	Summary	181
7.2	Future Research	191

APPENDICES

A	Scheduling of Independent Parallel Tasks with a Single Deadline	196
	A.1 Problem Definition	196
	A.2 Proof	197
	BIBLIOGRAPHY	199

LIST OF TABLES

TABLE		PAGE
1	Scheduling of Real-Time Tasks on Multiprocessors	32
2	EDF is Not Optimal for Multiprocessor Systems	60
3	A Sample Job Set	61
4	Summary of the Processor Allocation Strategies	152
5	Time-List	169
6	Job Start Times	170

LIST OF FIGURES

FIGURE	PAGE
1 Classification of Real-Time Scheduling Algorithms	14
2 Data Structure for Maintaining the Schedule	49
3 Updates Required	51
4 Efficient Updates	52
5 Two Updates at One Level	53
6 Subcube Recognition by Buddy and Gray Code Strategies	55
7 Schedule Prepared By the FSS Algorithm	63
8 Schedule Prepared By the EST Algorithm	65
9 Identifying h-Subcube Boundaries	67
10 Schedule Prepared By the Stacking Algorithm	68
11 Work Miss Ratio% vs. Maximum Job Dimension	73
12 Work Miss Ratio% vs. Offered System Load	75
13 Work Miss Ratio% vs. Mean Laxity%	76
14 Work Miss Ratio% vs. System Size	77

15	Work Miss Ratio% vs. Offered System Load	79
16	Work Miss Ratio% vs. Mean Laxity%	80
17	Work Miss Ratio% vs. Preemption-Migration Cost	81
18	Work Miss Ratio% vs. Subcube Selection Policy	82
19	Work Miss Ratio% vs. Offered System Load	84
20	Work Miss Ratio% vs. Mean Laxity%	85
21	Work Miss Ratio% vs. Preemption-Migration Cost	86
22	Work Miss Ratio% vs. System Size	87
23	Work Miss Ratio% vs. Offered System Load	93
24	Work Miss Ratio% vs. Preemption-Migration Cost	95
25	Job Start-Time Advancing is not Always Possible	109
26	Resource Usage Methods vs. Subcube Selection Policies	114
27	Work Miss Ratio% vs. Offered System Load	118
28	Work Miss Ratio% vs. Early-Completion Ratio	119
29	Work Miss Ratio% vs. Mean Job Laxity	121
30	Work Miss Ratio% vs. Preemption-Migration Cost	122
31	Work Miss Ratio% vs. System Size	124
32	Work Miss Ratio% vs. Offered System Load	133
33	Work Miss Ratio% vs. Perc. of HP Jobs	135

34	Work Miss Ratio% vs. Mean Laxity%	136
35	A 2D Mesh of Size 8×5	141
36	Mean Job Response Time/Mean Allocation Overhead vs. Offered System Load	156
37	Mean Job Response Time vs. Offered System Load	157
38	Mean Job Response Time vs. Size of the Mesh System	159
39	Mean Submesh Allocation Time vs. Offered System Load/System Size	160
40	Mean Job Response Time vs. Cost Per Operation/Offered System Load	161
41	Mean Job Response Time vs. Offered System Load	163
42	Mean Job Response Time vs. Offered System Load	164
43	A Sample Schedule	169
44	Mean Job Response Time vs. Offered System Load	175
45	Mean Job Response Time vs. Mean Laxity%	176
46	Mean Job Response Time vs. Early Completion Ratio	177
47	Mean Job Response Time vs. Offered System Load	178
48	Mean Job Response Time vs. Mean Laxity%	179
49	Mean Job Response Time vs. Early Completion Ratio	180

CHAPTER I

Introduction

The use of computer systems to monitor and control real-time processes in industrial, medical, scientific, environmental, military and other applications vital to our modern day life-style is experiencing rapid growth. Multiprocessors have emerged as an important means of computing for such applications, due to their capability for high throughput and reliability through component multiplicity. This thesis addresses the issue of processor management in real-time multiprocessor systems.

1.1 Real-Time Systems

A real-time system is one in which the correctness of a computation depends not only on the logical correctness of the result but also on the time at which the result is produced.

For example, in an automated factory, a robotic arm may be used to pick up objects from a conveyor belt. The belt is moving and the robot has a small window of time in which to pick up an object. If the robot is late, the object will not be there anymore. In this case, the system fails to do the job correctly even though it went to the right place. This implies that, unlike many systems where there is a clear separation between correctness and performance, the two are tightly interrelated in real-time systems. Time is an explicit parameter in ensuring system integrity in real-time applications.

Typically, a real-time system consists of a *controlling system*, a *controlled system* and an *environment*. For example, in an automated factory the controlled system is the factory floor with its robots, the assembling stations, and the assembled parts. The controlling system is the computer and human interfaces that manage and coordinate the activities of the robots and the assembling stations. The environment consists of all other objects and activities on the factory floor which are beyond the control of controlling system. The environment may affect and possibly be affected by the activities of the controlling and controlled systems.

Advances in hardware and software technologies during the past few decades have resulted in a rapid growth in the extent and variety of real-time systems, and this trend is expected to continue in the future. The complexity of real-time systems spans a wide spectrum from simple control of laboratory experiments, to complex applications such as the Space Shuttle control system. Very often, real-time systems are associated with critical applications in which human lives or expensive machinery are at stake. Real-time systems' missions are often long-lived and uninterruptible, making maintenance or reconfiguration difficult. Examples of current real-time computing systems are the control systems in chemical and nuclear plants, control of engines in automobiles, flight control systems, air-traffic control systems, multiple autonomous and cooperating robots [30], distributed real-time simulations [37], real-time communications in large scale distributed real-time systems, and multi-site theater battle management systems [80].

1.1.1 Aspects and Constraints

The most important characteristic of a real-time system is that the system can never return to a prior state. The effects of a real-time system's actions or inactions on the controlled

system, and in turn on the real-world, are irreversible. Although many real-time systems have been constructed, the fundamental differences between real-time systems and other computer systems have not always been clearly understood. In real-time systems, the completion of a process *has a value to the system that varies with time*. Although the time to complete a process is of some importance in all computer systems, in a real-time system the completion time is viewed as a crucial part of the *correctness of the application software*; a computation that is late is frequently no better, or perhaps even worse, than one producing an incorrect result. This time dependent value is usually described in terms of deadlines by which computations must be completed, and the deadlines are generally traceable to the physical environment with which the system must interact. For example, if the computer controlling a robot does not command it to stop or turn in time, the robot might collide with another object on the factory floor. Needless to say, such a mishap can result in catastrophe.

Variation in the value of a task depends on the nature of the application. In applications where a late result is no better than an incorrect result (or no result), the value of the task to the system drops to zero after the deadline. Such tasks are referred to as *hard real-time tasks*. In the example of a robotic arm picking up objects from a moving conveyor belt, if the robotic arm is late in getting to the right place on the belt, it won't find the object there because the object has already moved ahead. Clearly, it is useless to get to the right place after the deadline has passed. This is an example of a hard real-time task. On the other hand, there are some applications where late results might still be useful to the system. For such applications, the value to the system for completing a task diminishes after the deadline but remains more than zero. An example of such an application is a video-conferencing system.

Late sound is better than no sound at all, so the value for completing the task diminishes after the deadline but remains greater than zero. Such tasks are referred to as *soft real-time tasks*.

It is not hard to find computing environments where both hard real-time and soft real-time tasks are present, along with tasks that are not time-critical. Tasks that are not time-critical are referred to as *non-real-time tasks*. Ideally, the computer should execute the time-critical tasks so that each task meets its deadline, and it should execute non-real-time tasks so as to minimize their response time.

One common denominator in all real-time systems is the need for *predictability*. Predictability means that it should be possible to demonstrate or prove that all requirements are met subject to the assumptions made, for example, concerning failures, workload and the environment [114]. To understand the meaning of predictability, let us consider the following example: System S guarantees a worst-case interrupt response time of 50 microseconds. This system is predictable in the sense that service to any new interrupt is *started* within 50 microseconds of interrupt's arrival. However, this does not guarantee that every interrupt's service will finish before a certain deadline. There might be overlapping interrupts which could affect the service time of each other. Consequently, in System S, the claim to predictability is extremely tenuous and is offered at a microscopic level. A system level predictability will include claims such as: all tasks meet their deadlines with 100% guarantee. However, it should be emphasized that often requiring 100% guarantees for all tasks is not necessary and may lead to overly conservative designs. For a complex real-time system that has both hard and soft real-time tasks, with some of the tasks being critical for systems integrity, a typical system-wide predictability requirement may be that all critical

tasks always meet their deadlines (i.e. 100% guarantee), and that 97% of non-critical hard real-time tasks and 95% of soft real-time tasks must meet their deadlines. Alternatively, for non-critical hard and soft real-time tasks, the requirement might be to maximize the total value the system receives for completing the tasks. The value awarded to the system for completing a task might be proportional to the importance of the task.

1.2 Real-Time Scheduling

Scheduling involves allocating resources to activities so that the system can meet certain performance requirements. In real-time systems, tasks have timing constraints associated with them. The system's goal is not to improve the response time or throughput, but to ensure that the maximum number of tasks complete within their timing constraints. This problem may be complicated by the fact that some of the tasks are more critical (more important to the system) than others and their successful completion within their timing constraints needs to be ensured even at the cost of some non-critical tasks. Other constraints, such as maintaining precedence relations among executions of a set of tasks or allocating processors to parallel tasks so as to minimize processor fragmentation, may also restrict the possible orders in which the system may execute the tasks. All these factors contribute to make the scheduling problem, in a more general sense the resource allocation problem, one of the most formidable problems in the design of real-time systems.

Timing constraints for tasks may be arbitrarily complicated, but the most common timing constraints are either *periodic* or *aperiodic*. Aperiodic tasks, also referred to as *sporadic* tasks, are characterized by random arrival times and distinct deadlines. A sporadic task has

a deadline on either the start time, the finish time or both. Due to the unpredictable nature of sporadic task arrivals, it is difficult to design a real-time system that maximizes the number of tasks that meet their deadlines. Sporadic tasks are most often created by dynamic events or abnormal conditions, such as the subject rotating his/her head in a virtual reality system or a human operator pushing a button on a console in a process control system. More complex types of timing constraints might occur for tasks such as spray painting a car on a moving conveyor belt. The task must be started after time t_1 and completed before time t_2 .

Periodic tasks are tasks that are executed repeatedly in a fixed time interval. The fixed interval is referred to as the *period* of the task. A period might mean *once per period T* or *exactly T time units apart*. Most scheduling algorithms for periodic tasks assume the end of the period to be the deadline for the periodic task, but quite often that's not the case. Periodic tasks may have deadlines different from the end of the period, but it is rarely after the end of the period. Most low-level application tasks that process information from sensors or control loops and those that activate actuators, are periodic in nature. For example, a radar that tracks flights produces data in a periodic fashion, at a fixed rate. Some periodic processes may exist from the time of system initialization, while others may come into existence dynamically e.g. a periodic task that monitors a particular flight, begins when the aircraft enters an air traffic control region and ends when the aircraft leaves the region. It is not difficult to find real-time applications where periodic and sporadic tasks are present together with non real-time tasks.

Scheduling algorithms can be broadly classified into two classes: *static* scheduling and *dynamic* scheduling. Static scheduling algorithms rely on complete a priori knowledge

about the task set and its attributes, such as deadlines, computation times, precedence constraints, and future release times. A static scheduling algorithm operates on a given set of tasks and produces a schedule which is fixed for all time. In contrast, a dynamic scheduling algorithm may have complete knowledge of currently active tasks (possibly including active tasks with future release times), but has no knowledge about arrival times or resource requirements of future tasks.

Because it is common in real-time systems, off-line scheduling is often equated to static scheduling, but this need not be true. In building any real-time system, off-line scheduling (analysis) should always be done. In many cases, the designers identify the maximum set of tasks with their worst-case resource requirements and apply a static scheduling algorithm to produce a static schedule. This schedule is then fixed and used on-line with well understood properties such as, given that all the assumptions about the tasks' worst-case resource requirements hold, all tasks will meet their deadlines. In other cases, for example under the Rate Monotonic approach [75], off-line analysis produces a static set of priorities to be used at run time. The schedule itself is not fixed, but the priorities that drive the scheduling process are fixed.

When a real-time system operates in a dynamic environment, it is not feasible to meet the assumptions of static scheduling (i.e. everything is known a priori). In such cases, an algorithm is chosen and analyzed off-line using the expected dynamic environmental conditions. The same dynamic algorithm is then used on-line. Usually, less precise statements can be made about the overall performance of the on-line algorithm.

Generally, the same scheduling algorithm with possible modifications can be applied to

static or dynamic scheduling, and be used either off-line or on-line. The important difference is what is known about the performance of the algorithm in each of these cases.

A scheduling approach based on the notion of dynamic guarantees provides both the flexibility of the dynamic approach and some of the predictability of the static approach. After a task arrives but before its execution begins, an attempt is made to create a schedule that contains previously guaranteed, but as yet unfinished tasks, along with the newly-arrived task. If the attempt fails and if the attempt is made sufficiently ahead of the deadline, time is available to take alternative actions. This approach provides for predictability with respect to individual arrivals. In contrast, if a system uses a purely priority-driven preemptive approach – say, using task deadlines as priorities, a task could be preempted any time during its execution. In such a case, it is not known whether a task will meet its deadline until the deadline arrives or until the task finishes, whichever comes first.

1.3 Problem Statement

The research presented in this thesis addresses the problem of on-line scheduling of sporadic parallel tasks in hard real-time systems based on partitionable multiprocessor systems. Parallel tasks are tasks that execute on one or more processors. A parallel computer is said to be *partitionable* if it can be partitioned into subsystems, each of which can be allocated to a different job. Distributed memory multiprocessors (also referred to as multicomputers), based on hypercube, mesh or tree topologies are highly partitionable.

A typical instance of the problem is as follows: A task arrives at the system at an unpredictable time and tells the system that it requires k processors for e units of time before time

d. The scheduler is responsible for determining if it is possible to allocate required resources to the task within its timing constraints. If it is possible to do so, the task is *guaranteed* to finish before its deadline; otherwise it is rejected.

To understand the complexity of the above problem, let us consider two simplifications. First let us assume that the problem involves only sequential real-time tasks, i.e. each task executes on one processor only. The problem remains NP-hard even after this simplification. Mok [84] has shown that dynamic scheduling of sequential real-time tasks on multiprocessors is NP-hard. For an alternative simplification, let us assume that the problem involves non-real-time parallel tasks. Dutt [31] has shown that optimal dynamic scheduling of parallel tasks on multiprocessors is a NP-hard problem.

The problem addressed in this thesis involves not only on-line scheduling of real-time tasks in a multiprocessor environment, but also dynamic processor allocation for parallel tasks. Since both of these problems have been shown to be NP-hard, a polynomial time solution for the combined problem is infeasible. Consequently, we rely on heuristic approaches.

1.3.1 Motivation

The growing need for efficient algorithms for dynamic scheduling of real-time tasks on parallel computers was the driving force behind the research presented in this thesis. Parallel computers are emerging as an important computing means for real-time applications, due to their capability for high throughput and reliability through component multiplicity. However, increasing numbers of multicomputers are being used in partitioned mode, because few jobs can effectively use the large number of processors available. Such multicomputer

environments, where multiple jobs can execute simultaneously, are also referred to as *space sharing environments*.

With continuing advances in technology, the sophistication and complexity of real-time applications is increasing rapidly. With increasingly complex applications, it is impossible to enumerate the set of tasks that might occur simultaneously, which precludes the use of static strategies for task scheduling. On-line algorithms that can provide guarantees are needed for such complex dynamic environments. Because the time required to make scheduling decisions strongly affects the ability of the system to meet jobs' deadlines, decisions must be made quickly.

1.4 Thesis Organization and Overview

This dissertation begins with an overview of related work in Chapter II. This chapter places the work presented in this thesis in perspective with the work done in the area of real-time scheduling. Chapter III describes the system model and process model assumed in this research. Assumptions regarding the workload are also discussed in this chapter along with a discussion of the performance metrics used in the evaluation and comparison of various heuristics.

Chapter IV describes the approach used for designing heuristic-based algorithms for hard real-time scheduling of parallel tasks on partitionable multiprocessors. The scheduling problem is divided into three subcomponents, and heuristics are designed for each of the components. Simulation studies are conducted to evaluate and compare the performance of these heuristics. These studies also determine whether the performance of a heuristic is de-

pendent on the heuristics used for other components.

In Chapter IV, we also address the issue of determining an appropriate level of preemption for the scheduling problem addressed in this thesis. Preemption improves resource utilization and in turn, helps maximize the number of tasks that are able to meet their deadlines. However, preemption incurs considerable overheads, because of the required context switches and possible job migrations, since a job might be moved to a different set of processors after preemption.

In real-time systems that provide guarantees, jobs are scheduled with respect to their worst-case execution times. However, jobs typically execute for much less than their worst-case execution times. For algorithms that maintain a work-plan (i.e. an execution schedule for all the tasks present in the system), this brings up the problem of efficient use of resources left unused due to early job completions. Chapter V describes the issues related to early job completions and extensions of the basic approach to deal with these problems. This chapter also addresses the handling of priority constraints in the workload. Such constraints might arise when some tasks are more important to the system than others. These important tasks need to be guaranteed to meet their deadlines even at the cost of non-critical tasks.

Chapter VI describes the application of the suggested heuristic-based scheduling algorithms to mesh-connected systems. A typical workload in mesh-connected systems consists of requests for submeshes of arbitrary sizes. Several submesh allocation algorithms have been suggested for general-purpose systems. This chapter describes their extension to real-time systems and compares their performance under a hard real-time environment.

The results and contributions of the research presented in this thesis are summarized in

Chapter VII. This chapter also discusses possible future extensions to the work presented in this thesis.

CHAPTER II

Related Work

2.1 Introduction

Although little research has addressed the problem of scheduling real-time parallel tasks on partitionable parallel architectures, a large amount of work has been done in the area of real-time scheduling. This chapter places the research presented in this thesis in perspective with the myriad results available in the domain of real-time scheduling. Comprehensive surveys of results in the area of real-time scheduling are available in [18, 38, 39, 52, 67, 115].

Real-time scheduling has many dimensions and there is no accepted taxonomy to classify the work done in this area. Possible schemes for classification include uniprocessor vs. multiprocessor systems, static vs. dynamic scheduling, preemptive vs. non-preemptive scheduling, periodic vs. aperiodic tasks, and distributed vs. centralized systems. Classifications based on other features such as the presence of precedence constraints, multiple resource requirements, or overloaded conditions, are also possible but not very common. The classification followed in the remainder of this thesis is shown in Figure 1. As indicated in Figure 1, we focus on the scheduling algorithms and results available for centralized hard real-time systems.

Figure 1: Classification of Real-Time Scheduling Algorithms

Even though the focus of this thesis is on centralized systems, the results discussed are relevant to distributed systems as well. One of the biggest concerns in distributed scheduling is how to enforce globally important performance objectives in distributed systems. Towards this end, most distributed scheduling algorithms have two common components: (1) a *global* task scheduling strategy between nodes, and (2) a *local* scheduling policy for individual nodes. The local scheduling policy is very often similar to the scheduling strategies used in centralized systems. Some distributed scheduling algorithms are compared in [13], based on the methods used for sharing scheduling information.

Scheduling problems can be classified into two categories: *static* scheduling and *dynamic* scheduling. As mentioned in Chapter I, static scheduling algorithms have complete knowledge about the set of tasks to be scheduled. The characteristics of all tasks and their timing constraints are known to the scheduling algorithm. On the other hand, dynamic sche-

duling algorithms work with much less information. Dynamic algorithms may have complete knowledge about the characteristics and constraints of tasks currently present in the system, but they have no knowledge of the tasks that will arrive in future. Dynamic algorithms are harder to design, and usually less precise statements can be made about their performance. The majority of scheduling algorithms in the literature are designed to schedule task-sets for which all characteristics are known. Very few results are available for real-time dynamic scheduling algorithms. In this chapter, we look at preemptive as well as non-preemptive algorithms. Preemption makes the scheduling problem easier in most of the cases, however non-preemptive algorithms are easier to implement and have a smaller scheduling overhead. Unless stated otherwise, the performance metric used for the algorithms discussed in this chapter is the *guarantee ratio*, the proportion of tasks that successfully meet their timing constraints.

Before we review the related work, we define a few terms.

Latest Start Time (LST): The LST of a job is the the latest time by which the job must start executing in order to meet its timing constraints.

Laxity: The laxity of a job is defined as the time difference between the current time and the Latest Start Time (LST) of the job. Essentially, laxity is a measure of the flexibility available for scheduling a task.

Optimal Scheduling Algorithms: A *static* scheduling algorithm is optimal if, for any set of tasks, it always produces a schedule that satisfies the constraints of tasks whenever any other algorithm can do so. A *dynamic* scheduling algorithm is said to be *optimal* if it always produces a feasible schedule whenever a static algorithm with complete prior knowledge of all

possible tasks can do so.

NP-Completeness and NP-Hardness: NP is the set of all problems that can be solved in polynomial time by a non-deterministic computer. A problem R is said to be NP-complete if $R \in NP$ and all other problems in NP are polynomial transformable to R. A problem R is NP-hard if all problems in NP are polynomial transformable to R but it is not shown that $R \in NP$.

Task: A task is a granule of computation treated by the scheduler as a unit of work to be allocated processor time or scheduled. A task t is characterized by its arrival time a , ready time r (also referred to as the *release time*), worst-case execution time e and deadline d . Arrival time of a task is the time at which the task arrives at the system, whereas the ready time or release time of a task refers to the earliest time at which the task can start executing. In many studies, tasks are assumed to be ready to execute as soon as they arrive, so, their arrival time a and ready time r are identical. A task's characteristics are made known to the system at the time of its arrival.

Preemptable vs. non-preemptable tasks: a task is said to be preemptable if its execution can be interrupted by other tasks at any time and resumed at a later point in time. On the other hand, a task is non-preemptable if it must run to completion once it starts. Whether a task is preemptable or not is mainly determined by the nature of the application environment.

2.2 Hard Real-Time Scheduling Algorithms for Uniprocessor Systems

2.2.1 Static Scheduling Algorithms

Non-Preemptive Algorithms

The general form of the scheduling problem where tasks have arbitrary arrival times, arbitrary execution times and individual deadlines, has been shown to be NP-hard [35]. However, if all tasks are restricted to have identical execution times or identical ready times, the problem can be solved in polynomial time even under precedence constraints [65, 85]. Kise [54] developed an $O(n^2)$ (n is the number of tasks to be scheduled) algorithm for the case where a task t_1 has earlier ready time than task t_2 if and only if task t_1 's deadline is earlier than task t_2 's deadline. McMahon and Florian [81] proposed a heuristic algorithm for sequencing jobs on a single processor in order to minimize maximum lateness subject to tasks' release times and deadlines.

Jeffay et al. [51] derived a set of necessary and sufficient conditions for a set of periodic or sporadic tasks with integral release times to be *schedulable*. A task set τ is said to be schedulable if it is possible to schedule the executions of all tasks of τ such that all tasks meet their deadlines. Schedulability conditions can be verified in pseudo-polynomial time. It was also shown that the non-preemptive *Earliest Deadline First* (EDF) algorithm is able to schedule any set of periodic or sporadic tasks with integral release times, if the set is schedulable by any algorithm that does not use inserted idle time. Under the EDF scheduling policy, tasks with earlier deadlines and earlier ready times are chosen to run before tasks with later deadlines and later ready times.

Many researchers have attempted to solve the general case of the scheduling problem by

developing efficient enumeration algorithms with strong bounding conditions. For uniprocessor systems, Bratley, Florian and Robillard [11] developed an implicit enumeration algorithm to determine schedule for tasks with arbitrary ready times and deadlines. Baker and Su [6] used a similar approach to minimize the maximum lateness of tasks. Erschler et al. [32] developed a necessary condition for scheduling tasks with arbitrary ready times and deadlines. Their theories can be used to limit the search space of an enumeration algorithm.

Preemptive Algorithms

If preemption is allowed, the problem of scheduling a set of tasks with arbitrary release times, arbitrary execution times and individual deadlines can be solved in polynomial time, even under precedence constraints [65]. In the absence of precedence constraints, EDF is optimal for static as well as dynamic scheduling when preemption is allowed [24]. Horn [48] developed an $O(n^2)$ algorithm to schedule tasks with arbitrary ready times and deadlines. This approach is based on the Earliest-Deadline-First (EDF) policy.

Preemptive scheduling of periodic tasks on a uniprocessor system is one of the most extensively studied problems in real-time systems. Based on the assignment of either a fixed or a dynamic priority value to each task, two alternative approaches have been proposed to solve this problem. In the fixed priority approach, task priorities are computed once and maintained unaltered during the entire task life time. In the dynamic priority approach, task priorities are dynamically computed and may be altered at run-time. Several preemptive scheduling policies have been developed for fixed and dynamic priority systems, including the Rate Monotonic (RM), the Earliest-Deadline-First (EDF) and the Minimum-Laxity-First (MLF) approaches. The idea behind the Rate Monotonic algorithm developed by Liu

and Leyland [75] is to assign different fixed priorities to tasks with different execution rates, with the highest priority being assigned to the task with the highest frequency (i.e. smallest period), and the lowest priority to the task with the lowest frequency (i.e. longest period). At any time, the low-level scheduler simply chooses to execute the highest priority task. The EDF and MLF policies implement dynamic priorities. With EDF, the earlier the task deadline, the higher its priority, while with MLF, a task's priority is inversely proportional to the amount of laxity it has.

To determine the schedulability of a set of periodic tasks, we only have to consider the instances of the periodic tasks within a time interval between zero and the *least-common-multiple* (also referred to as the hyper-period) of the tasks' periods. Then, the Rate Monotonic, EDF or MLF scheme is applied to determine whether all the instances of the periodic tasks in the interval can be scheduled to meet their deadlines. If they can be scheduled, then all the other instances of the periodic tasks can also be scheduled.

Liu and Leyland [75] showed that the Rate Monotonic and Earliest-Deadline-First scheduling algorithms are optimal static (fixed) priority and dynamic priority algorithms, respectively in a uniprocessor preemptive scheduling environment. However, the Rate Monotonic algorithm assumes that the deadline of a periodic task is same as the period of the task. Teixeira [120] removed this restriction and presented a fixed-priority assignment scheme that allows the relative deadlines of periodic tasks to be different from their periods. *Deadline Monotonic* priority assignment, proposed by Leung and Whitehead [70], also removes the constraint of deadline being equal to the period, within a static priority scheduling scheme. Sha et al. [69] presented a technique to modify the periods of tasks in such a way that while

the timing constraints of tasks continue to be met, better processor utilization is achieved. This modification involves breaking up one periodic task into two, each with half the computation time and half the period as the original task. Mok [83] proved that the problem of deciding whether it is possible to schedule a set of periodic processes that use semaphores only to enforce mutual exclusion is NP-hard.

Several techniques have been used for scheduling sporadic tasks along with periodic tasks in a uniprocessor system [68]. The first policy consists of scheduling sporadic tasks as background tasks, i.e. sporadic tasks are allowed to execute only when no periodic tasks are active. Another policy, termed *polling*, creates a periodic process with a fixed priority to execute sporadic tasks. The main problem with this policy is the incompatibility between the cyclic nature of this policy and the bursty nature of the aperiodic tasks. Some policies, including *Priority Exchange* [68], *Deferrable Server* [119], *Sporadic Server* [112] and *Slack Stealing* [96], create a high priority periodic server to improve the responsiveness of sporadic tasks. These policies differ in the way they manage the high priority of their periodic servers and their implementation complexity.

2.2.2 Dynamic Scheduling Algorithms

Theoretically, any static scheduling algorithm for centralized systems can be used as a dynamic algorithm. However, with a few exceptions, most of the algorithms that are optimal for static scheduling are not optimal for dynamic scheduling. Furthermore, because run-time cost is an important factor for dynamic scheduling, most sophisticated static algorithms, including many multi-stage polynomial algorithms, are not appropriate for dynamic scheduling.

Non-Preemptive Algorithms

Since non-preemptive hard real-time scheduling of tasks with arbitrary arrival times, arbitrary execution times and arbitrary deadlines is NP-hard under static scheduling [35], there is no possibility of a polynomial time dynamic scheduling algorithm for this problem. In general non-preemptive scheduling is less efficient in terms of resource utilization than preemptive scheduling. However, in many cases non-preemptive scheduling algorithms are easier to implement than preemptive algorithms. For example, in hard real-time environments where tasks may need multiple resources, it is easier to implement non-preemptive scheduling algorithms than preemptive algorithms.

Zhao and Ramamritham [127] compared a set of heuristics for scheduling a set of n tasks in a system having r resources. Tasks have arbitrary execution times, arbitrary deadlines, and may request any number of resources. This problem is known to be NP-complete, therefore, heuristics-based approaches were attempted. It was found that simple heuristics like Shortest-Job-First, Longest-Job-First, Earliest-Deadline-First and Minimum-Laxity-First did not perform very well. However, an algorithm that uses combinations of these heuristics works nearly as well as the optimal algorithm.

Baker and Su [6] compared four simple heuristic algorithms that schedule tasks according to an order determined by ready times, by deadlines, by the average of the ready times and deadlines, and by both the ready time and the deadline, respectively. With a limited number of tests, they showed that the last two heuristics perform better than the first two.

Preemptive Algorithms

The Earliest-Deadline-First (EDF) scheduling algorithm, which is optimal for static scheduling of preemptive tasks, can be used for dynamic scheduling as well. Dertouzos [24] showed that the EDF algorithm is optimal for scheduling preemptable tasks with arbitrary arrival times. Since the task with the earliest deadline gets the highest priority, the resulting priorities are naturally dynamic. Dertouzos's algorithm does not address the cost of invoking the algorithm to guarantee tasks at run-time. Ramamritham and Stankovic [94] described a guarantee scheme that is based on the EDF policy and takes into account the run-time cost.

A variant of EDF scheduling is Minimum-Laxity-First (MLF) scheduling [25], where the task with the minimum laxity is executed first. The main difference between EDF and MLF is that unlike EDF, MLF takes the execution times of the tasks into consideration.

While EDF and MLF allow for 100% processor utilization, the problem with them is that their performance degrades rapidly during overloaded intervals [77]. There is no way to guarantee which tasks will fail in such situations. An overloaded interval is defined as the interval of time $[t_x, t_y]$ such that the sums of execution times of all tasks having arrival times and deadlines within this interval is greater than $(t_x - t_y)$. A number of heuristic algorithms have been proposed to deal with overloads [44, 121, 130]. Stewart [116] proposed a variant of EDF scheduling algorithm, called the *Maximum-Urgency-First* (MUF) algorithm, where each task has two fixed priorities and a dynamic priority, which is inversely proportional to the task's laxity. The first fixed priority is proportional to the task's criticality and has precedence over the task's dynamic priority. The other fixed priority, called the user priority,

has lower precedence than the task's dynamic priority. The idea is to use the task's criticality to help the on-line algorithm distinguish more important tasks from less important tasks. Baruah et al. [9] proved that for a dual processor system, no on-line scheduling algorithm can guarantee a cumulative value greater than one-half the value obtainable by a *clairvoyant* (one that knows about future events) scheduler.

Schwan and Zhou [101] proposed a dynamic uniprocessor scheduling algorithm that performs feasibility analysis (decides whether a new task can be scheduled to meet its timing and precedence constraints) and accepts or rejects tasks at the time of their arrival. The algorithm is based on the EDF strategy and has worst-case time complexity of $O(n \log n)$.

Craig and Woodside [22] developed an analytical model to compare task scheduling disciplines for systems where tasks have random arrivals, deadlines and low system utilization (up to 20%). The scheduling disciplines that were compared included Shortest-Job-First (SJF), Earliest-Deadline-First (EDF), Minimum-Laxity-First (MLF) and First-Come-First-Served (FCFS). EDF was found to have the best performance.

2.3 Multiprocessor Systems

To enhance processing power, computer systems may be equipped with more than one processors. Such systems are called *multiprocessor* systems. Multiprocessor systems with thousands of processors are already available, and we expect to see systems with tens of thousands of processors in the near future. Multiprocessors can be broadly classified into two categories: shared memory multiprocessors and distributed memory multiprocessors. In shared memory multiprocessor systems, all processors have access to a globally shared

memory. Apart from the shared memory, processors may have private memory as well. On the other hand, in distributed memory multiprocessors, there is no memory shared among processors. Each processor has private memory accessible only to that processor. Because of the lack of shared memory the only means of communication among processors in distributed memory multiprocessors is through message passing.

Shared memory multiprocessors are easier to program than distributed memory multiprocessors, but lack the potential for scalability. As the number of processors in the system grows, the shared memory becomes a bottleneck and, also, the complexity and cost of the interconnection network connecting the processors rises steeply. On the other hand, distributed memory multiprocessors, by virtue of having no resources shared among all processors, do not have any inherent impediments to scalability. Potential for scalability is one of the main reasons distributed memory multiprocessors have enjoyed increased attention, especially in the last two decades. The research work presented in this thesis is targeted for distributed memory multiprocessors.

2.4 Hard Real-Time Scheduling Algorithms for Multiprocessor Systems

2.4.1 Static Scheduling Algorithms

Non-Preemptive Algorithms

Many non-preemptive scheduling problems for multiprocessor systems have been shown to be NP-hard. Non-preemptive scheduling in multiprocessor systems has been shown to be NP-hard even when all tasks are restricted to have identical release times and a common deadline [125]. However, polynomial time optimal algorithms are available for special

cases where all tasks are independent and have unit computation times [63, 109, 110, 111]. Garey and Johnson [35] proposed an $O(n^3)$ non-preemptive algorithm for scheduling a set of tasks with unit computation times, arbitrary start-times, arbitrary deadlines and precedence constraints on a dual processor system. The same problem, however, is NP-complete when execution times are allowed to be either one or two units, even if all tasks share the same release time and deadline [33]. Gunsfield [41] proposed a polynomial time heuristic for scheduling a set of tasks with release times and deadlines on multiple processors in order to minimize the maximum lateness of tasks.

Bratley, Florian, and Robillard [12] developed a heuristic-based multi-stage enumeration algorithm to schedule tasks with arbitrary release times and deadlines in order to minimize the elapsed time to complete all jobs. Since its worst-case complexity is exponential, this algorithm is designed to run off-line.

Preemptive Algorithms

The problem of preemptive scheduling of a set of tasks with arbitrary execution times, precedence constraints and an overall deadline, on a multiprocessor system has been shown to be NP-hard [124]. However, the problem can be solved in polynomial time if the number of processors is restricted to no more than two [86] or the tasks are assumed to be independent [48]. Horn [48] proposed an optimal $O(n^3)$ algorithm, which is based on the network flow method and considers only processors with identical processing speed. This approach was extended by Martel [79] to consider processors with different speeds. The extended scheduling problem is more difficult and the complexity of Martel's algorithm is $O(m^2n^4 + n^5)$, where m is the number of processors in the system and n is the number

of tasks to be scheduled. Both approaches may be extended to periodic tasks, in the same manner as suggested in Section 2.2.1, by considering all the task instances in a hyper-period and determining whether all the tasks can be scheduled to meet their timing constraints. However, if the periods of tasks are relatively prime, these approaches may not be practical because the number of instances of periodic tasks to be considered is large and the cost becomes high.

Bratley, Florian, and Robillard [11] proposed a preemptive algorithm to schedule tasks with arbitrary release times and deadlines in order to minimize the elapsed time to complete all jobs.

Many researchers have adopted a partitioning approach to solve the problem of scheduling periodic tasks. The idea behind these approaches is to partition the set of periodic tasks among a minimum number of processors such that each partition of the periodic tasks can be scheduled on one processor according to the EDF, MLF or Rate Monotonic priority scheme. Davari and Dhall [23] showed that if the EDF scheme is used, a *bin-packing* algorithm can be used to determine a suboptimal partition pattern of periodic tasks among multiple processors. Bannister and Trivedi [7] proposed a *best-fit* partition scheme. Their approach can be used in conjunction with both the EDF scheme and Rate Monotonic priority scheme. For the Rate Monotonic priority scheme, Dhall and Liu [26] developed a *next-fit* scheme and a *first-fit* partition scheme. Later, Davari and Dhall [23] improved these schemes and developed a more efficient *next-fit* partitioning scheme. The time complexity of this improved scheme is $O(n)$.

As described above, many of the scheduling algorithms designed for periodic tasks are

based on a fixed-priority assignment scheme. The advantage of the fixed-priority schemes is that they have small scheduling overheads. However, these schemes are inflexible, due to the large overheads involved in changing the priority assignment once it has been *fixed* on a system.

2.4.2 Dynamic Scheduling Algorithms

Dynamic scheduling of hard real-time tasks on multiprocessor systems is a difficult problem. Very few results are available in this area.

Non-Preemptive Algorithms

Almost all non-preemptive multiprocessor scheduling problems are NP-complete, even when ready times and deadlines of all tasks are the same [124, 125]. If tasks are allowed to have arbitrary deadlines, the problem is exacerbated.

Garey and Johnson [33] showed that the multiprocessor scheduling problem with 2 processors, independent tasks and arbitrary computation times is NP-complete.

Ramamritham et al. [95] proposed a heuristic based approach for dynamic scheduling of tasks with arbitrary deadlines and multiple resource requirements. They formulated the scheduling problem as a search problem and used heuristic functions to guide the search. Starting with an empty partial schedule, each step of the search extends the current partial schedule with one of the tasks yet to be scheduled. The heuristic functions help choose the task that extends the current partial schedule. Two algorithms were compared, one that considers *all* the tasks that are yet to be scheduled while selecting the task to augment the current partial schedule. The second, for the same purpose, considers a small subset of tasks with

shortest deadlines. The second algorithm has linear time complexity and is shown to be very effective when the maximum allowable scheduling overhead is fixed.

Preemptive Algorithms

Mok [83] showed that EDF scheduling is not optimal for multiprocessor systems. He also proved that for two or more processors, no deadline scheduling can be optimal without complete a priori knowledge of the (a) deadlines, (b) computation times, and (c) start times of the tasks.

This result implies that any of the classical scheduling theory algorithms that requires knowledge of start times can not be used on-line. This result also points out that we can not hope to develop an optimal on-line algorithm for the general case where tasks have arbitrary release times and deadlines.

Hong and Leung [47] showed that for multiprocessor systems no optimal scheduling algorithm can exist for task sets with two or more distinct deadlines. They also proposed an optimal on-line scheduling algorithm for task sets with one common deadline and any system with one or more processors. The proposed algorithm is based on McNaughton's algorithm [82] that finds the shortest preemptive schedule for a set of independent tasks on p processors. For the same model (i.e. task systems with one common deadline and $p \geq 1$), Sahni and Cho [100] proposed a so-called *nearly on-line* optimal scheduling algorithm. A nearly on-line scheduling algorithm is like an on-line algorithm, except that it has the additional information of when the next release time is.

2.5 Scheduling of Parallel Tasks

One of the assumptions made in classical scheduling theory is that a task is always executed by one processor at a time. With advances in parallel architectures and parallel algorithms, this assumption is no longer true. Tasks that execute on more than one processor are referred to as *parallel tasks*. Two models of parallel tasks have been considered. Under the first model, a task requires a fixed number of processors n , for its execution, where $n \geq 1$. Under the second model, a task may be executed by $1, 2, \dots, m$ processors, with the execution time depending on the number of processors assigned. At scheduling time, the task may be assigned any number processors between 1 and m . Once the number of processors assigned to a task is determined, it remains fixed throughout the execution of the task.

2.5.1 Processor Allocation

Scheduling of parallel tasks in distributed memory multiprocessors, such as hypercube systems and mesh-connected systems, brings up the problem of processor allocation. Dynamic allocation and deallocation of processors can fragment the system. A multiprocessor system is said to be fragmented if there are enough free processors to fulfill the needs of a requesting job, but the free processors do not form a *subunit* big enough to execute the requesting job. Subunit is defined as a part of the multiprocessor system that can be allocated to independent jobs. For example, subcube is a subunit of a hypercube systems and submesh is a subunit of a mesh-connected system. Minimizing system fragmentation and recognizing available subunits in a fragmented system are the two primary goals of processor allocation schemes. Several processor allocation strategies have been proposed for general-purpose (non-real-

time) environments on hypercube and mesh-connected systems. We review processor allocation strategies for hypercube systems in Chapter IV. Processor allocation strategies for mesh-connected systems are reviewed and compared in Chapter VI. We also discuss how these allocation schemes can be extended for hard real-time environments.

2.5.2 Real-Time Scheduling of Parallel Tasks

The problem of scheduling parallel tasks can be reduced to a multiprocessor scheduling problem by restricting each task to execute on one processor only. Therefore, all infeasibility results available for the classical multiprocessor real-time scheduling problem are applicable to the scheduling problem of parallel tasks as well. Since non-preemptive multiprocessor scheduling is NP-hard under both the dynamic and static cases, non-preemptive scheduling of parallel tasks is also NP-hard.

Blazewicz et al. [10] showed that finding an optimal non-preemptive schedule for a set of parallel tasks with equal execution times and empty precedence constraints is NP-hard for systems with arbitrary numbers of processors. However, the problem can be solved in polynomial time for systems with fixed numbers of processors. They also presented a polynomial time preemptive algorithm, based on a linear programming formulation, for finding a minimum length schedule for a task set consisting of independent tasks, with each task requiring either one processor or k processors, where k is fixed.

It can be shown that the problem of static scheduling of independent preemptive parallel tasks is NP-complete even when tasks have unit execution times and a single deadline but require arbitrary numbers of processors. A proof for this result is given in appendix A. For a restrictive case, for example scheduling on hypercube systems, where the total number

of processors and the number of processors required by each task is an exact power of 2, the problem of finding a minimum length preemptive schedule can be solved in polynomial time, as shown in [15, 129].

Under the second model of parallel tasks, where the number of processors allocated to a task is determined at the time of scheduling and the execution time of a task depends on the number of processors allocated to it, the problem of finding a minimum length preemptive schedule has been shown to be strongly NP-hard for systems with arbitrary numbers of processors and independent tasks [29]. Blazewicz et al. [10] showed that a polynomial time non-preemptive algorithm exists for the problem of minimizing schedule length when all tasks have unit processing times and require an arbitrary number of processors between 1 and k , where k is a fixed integer. The case for which k is not fixed is shown to be NP-complete. Table 1 summarizes the results available for scheduling of parallel tasks. In Table 1 the term “Sequential Tasks” refers to tasks that execute only on one processor, “Var. Parallel Tasks” refers to parallel tasks for which the number of processors is decided at the time of scheduling, and “Fix. Parallel Tasks” refers to parallel tasks for which number of processors required is fixed at compilation time. The column titled “Fix. Parallel Tasks on Hypercubes” lists results for the case where fixed parallel tasks are scheduled on a hypercube system and each task requires a perfect subcube. Because of the restrictions on the number of processors required by each task and the total number of processors available in the system, scheduling of fixed parallel tasks on hypercube systems is easier than scheduling of fixed parallel tasks requiring arbitrary numbers of processors on a fully-connected system with arbitrary numbers of processors.

Table 1: Scheduling of Real-Time Tasks on Multiprocessors

	Sequential Tasks	Var. Parallel Tasks	Fix. Parallel Tasks	Fix. Par. Tasks on Hypercubes
Static Non-Pre.	NP-Hard	NP-Hard	NP-Hard	NP-Hard
Static Preempt.	Polynomial Horn [48]	NP-Hard	NP-Hard Appendix A	Open <i>Polynomial for Single Deadline</i>
Dynamic Non-Pre.	NP-Hard	NP-Hard	NP-Hard	NP-Hard
Dynamic Preempt.	NP-Hard	NP-Hard	NP-Hard	NP-Hard

2.5.3 Similarity to Bin-Packing Results

Bin packing algorithms have been an active area of research for many decades. In the bin-packing problems, each bin has a maximum capacity and boxes to be placed in the bins require some percentage of the capacity. The goal is either, given a fixed number of bins, to pack them with boxes so as to minimize the maximum length of any bin or to fill the bins to maximum capacity and minimize the number of bins required. The bins may be thought of as computer systems with multiple processors that provide a computing capacity up to the deadline of the set of jobs. Jobs require some amount of processing time on a certain number of processors.

This analogy between bin-packing and job-scheduling allows the results and heuristics developed for the bin-packing problem to be applied to the problems of static scheduling in multiprocessors. For example, the heuristics used by Chen et al. [15] and Zhu et al. [129] for preemptive scheduling of parallel tasks on hypercube systems, were adopted from the heuristics developed for bin-packing problems.

2.6 Imprecise Computations

Recent work in real-time scheduling has begun to consider *imprecise computations*. Imprecise computations are computations that can return results at almost any time during their execution. Iterative algorithms are a good example of imprecise computations. The longer such computations run, the more precise their results are. Ideally, a process executes until a result with a desired small tolerance has been obtained. However, when time is limited, the process can be terminated prematurely, producing a result that may be acceptable, but not as precise as desired.

Liu et al. [76] discuss a formulation of the problem and present algorithms that take into account the quality of the overall result. Shih et al. [107] present three algorithms for scheduling real-time preemptive, imprecise tasks on a processor such that the total error is minimized.

CHAPTER III

Framework

Before we discuss processor management techniques for real-time systems, it is important to define the context of the problem. This chapter describes the system model, process model, workload model, performance measurement and performance indices for the problem addressed in this thesis.

3.1 System Model

The research presented in this thesis is targeted for distributed memory multiprocessors. There is no shared memory in the system and message passing is the only means of communication among processes executing on different processors. In this thesis, we focus on hypercube systems and mesh-connected systems, though the concepts may be applied to other distributed memory architectures as well.

Hypercubes are widely used interconnection topologies for parallel computers. The hypercube (or binary n -cube) topology enjoys numerous advantageous architectural properties [99], such as logarithmic diameter, high bandwidth, scalability due to regular structure and fault-tolerance due to natural architectural redundancy. Many regular algorithms and machine topologies can be mapped onto hypercube with their adjacency properties pre-

served (or nearly preserved) [45]. The nodes of a hypercube can be imagined to lie at the vertices of an n -dimensional cube, linked by communication channels that lie along the edges of the cube. Each node is linked to n -neighboring processors and the entire hypercube system contains $N = 2^n$ processors.

The mesh topology for distributed-memory multiprocessors has drawn considerable attention due to its simplicity, regularity and suitability for VLSI implementation [87, 98]. An $N_1 \times N_2 \dots \times N_d$ mesh consists of $\prod_{i=1}^{t=d} N_i$ processors connected in an n -dimensional array, where a node represents a processor and an edge denotes a communication link. Many scientific algorithms and several topologies can easily be mapped onto mesh-connected systems. Several mesh-connected systems have been built or are under construction including the ILLIAC IV [8], Touchstone DELTA [49] and Intel Paragon [50], Tera Computer System [1], and K2 Parallel Processor [2].

We assume there is a host processor connected to the multiprocessor system. This host processor is responsible for preparing the schedule, processor allocation/deallocation and task dispatching.

3.2 Process Model

In this thesis, the terms *task* and *job* are used interchangeably. In Section 2.1, we defined a task and identified its attributes. We assume in this thesis that the number of processors required for a task T_i has been determined before it is submitted to the system. Research addressing this step includes [16] and [99]. In this thesis, we deal with sporadic tasks; tasks' arrival times are assumed to be arbitrary. Job arrivals are assumed to be stochastic; i.e. no

knowledge about future arrivals is assumed. All jobs are assumed to be independent. Precedence constraints are handled by assuming that a job's ready time occurs after its precedence constraints have been satisfied. This model encompasses both periodic and sporadic tasks by considering any single execution of a periodic task as a sporadic task. Overlapping executions can either be avoided by properly selecting the deadlines of the instances of periodic tasks or by allowing multiple instances of a process to be simultaneously schedulable.

A *task set* τ consists of k independent tasks to be scheduled. Task set τ is said to be *feasible* on an n -dimensional hypercube if there is a preemptive schedule for the tasks in τ such that each task T_i is executed within its execution window $I_i = [A_i, D_i]$. Such a schedule is called a *feasible schedule*. A scheduler is said to be optimal for n -dimensional hypercube systems if it constructs a feasible schedule for every feasible task set.

Precedence and consistency constraints (e.g. for tasks involved in mutual exclusion) are addressed through the definition of the arrival time, but are not otherwise explicitly addressed in this model. Thus, if two tasks are related through a precedence relationship, the latter task will not be ready for scheduling until the first one completes. Future research should extend this work to consider precedence relations and synchronization constraints in the scheduling process.

We assume that all processes are fully preemptable and restartable once they start executing. Preemptability, as defined in Section 2.1, means that a process can be suspended at any time during its execution and other processes can be executed, normally followed later by the restart of the original process from the point at which it suspended processing. The cost of preemption is accounted for by adding an overhead for each preemption and resume

to task's execution time.

Unless stated otherwise, all tasks are assumed to have the same priority, i.e. all tasks are equally important to the system. In Chapter V, we deal with the problems encountered when some of the tasks have higher priorities because they are more valuable to the system e.g. tasks that handle emergency situations.

3.3 Workload Model

In this study, a Poisson job arrival process is assumed. The Poisson process has been found to model natural physical and organic processes realistically [55], and is commonly used to model random, independent arrivals of jobs to computer systems from an external population. We refer to the load imposed on the system by these job arrivals as the *offered system load*. If λ is the job arrival rate, X is the mean execution time, P is the mean number of processors in a subunit request, and N is the number of processors in the multiprocessor system, then the offered system load is $\frac{\lambda X P}{N}$.

Similar to Shivaratri and Singhal [108], to model a real-time workload, we assume that subunit hold times are normally distributed with a standard deviation equal to the mean, and with the distribution truncated at 0 and at one standard deviation greater than the mean. For convenience and with no loss of generality, we assume a mean subunit hold time of 1 time unit. Laxities are also assumed to be normally distributed, again truncated at 0 and one standard deviation above the mean. Subunit hold times are assumed to be independent of subunit sizes.

3.4 Performance

Due to their complexity, the systems studied in this research do not lend themselves to analytical performance modeling. Instead, a process-oriented discrete event simulator based on a custom simulation package developed by Krueger [60], was prepared to provide a hard real-time environment. Krueger's simulation package provides support for event handling, process management and random number generation. This package is similar to the CSIM package in functionality and has been extensively used for simulating scheduling algorithms for both loosely and tightly-coupled systems.

To allow steady state performance measurements, the lengths of the simulations were chosen to be sufficiently long (typically 100,000 time units, where the mean subunit hold time is 1 time unit), so that the effects of start-up conditions are negligible. All performance measures reported throughout this study have confidence intervals of 5% or less at the 90% confidence level. The method of independent replication [64] was used to achieve this level of accuracy.

Performance Indices

For hard real-time systems, two useful measures of performance are the Job Miss Ratio, which is the ratio of jobs refused to the total number of jobs that arrive at the system, and the Work Miss Ratio, which is the ratio of refused work to the offered workload. The load offered by a job is computed as its subunit hold time multiplied by the number of processors in its subunit. To better understand these performance metrics, consider the following: Let us say that the total number of jobs that arrived at the system in a certain time interval is J_a , and these J_a jobs offered a load of ρ_a to the system. The system was able to guarantee

only J_g of these jobs, which offered a workload of ρ_g . Over this time interval, the Job Miss Ratio is $\frac{J_a - J_g}{J_a}$ and Work Miss Ratio is $\frac{\rho_a - \rho_g}{\rho_a}$. In all the cases we present, we have found the plots of Job Miss Ratio to exhibit the same trends as that of Work Miss Ratio. Therefore, we display the Work Miss Ratio plots only. To improve the understandability of the plots, we plot *Work Miss Ratio%*, which is the Work Miss Ratio multiplied by 100.

CHAPTER IV

Problem Decomposition and Design of Heuristics

4.1 Introduction

Scheduling of sporadic tasks is one of the most difficult problems in the design of hard real-time systems. The problem of on-line scheduling in a hard real-time environment is to make a sequence of decisions dynamically by assigning system resources to real-time tasks. Each such scheduling decision must be made with no prior knowledge of the task requests to be made in future. The primary objective of a scheduling algorithm in such an environment is to maximize the *guarantee ratio* – the proportion of tasks that can be guaranteed, at the time they arrive to the system, to complete by their deadlines.

An on-line hard real-time scheduling algorithm works as follows: when a sporadic task arrives, its execution time, deadline and other attributes are made known to the system. The on-line scheduler is called upon to decide if the newly-arrived task, along with any tasks in the system that are as yet unfinished, can be scheduled so that all tasks will meet their deadlines. If it is possible to meet all deadlines, the system executes the tasks according to the schedule constructed by the on-line scheduler. Otherwise, the newly-arrived task is rejected. This process is repeated whenever a new task arrives.

Dynamic multiprocessor scheduling of hard real-time tasks is still in its infancy. Only a few on-line algorithms address hard real-time scheduling under practical assumptions. Since the time required to make scheduling decisions strongly affects the ability of the system to meet deadlines of tasks, decisions must be made quickly. As discussed in Chapter I, on-line scheduling of hard real-time parallel tasks is a NP-hard problem. Therefore, the goal of the research presented in this thesis was to explore *heuristic*-based algorithms that are able to make scheduling decisions quickly.

To be able to determine if a task can be guaranteed at the time of its arrival, the scheduler needs to maintain a work-plan, which we will refer to as the *schedule*. Using scheduling algorithms that maintain a work-plan, the system can provide an instantaneous snap shot of the current predictability of the newly-arrived job. At any point in time, the scheduler can identify the tasks that will meet their deadlines. This predictability provides the system with the flexibility to choose the jobs it would like to guarantee, in case all jobs can not meet their deadlines. The flexibility to choose critical jobs to be guaranteed provides many advantages, including graceful degradation, the ability to handle overloads and the ability to make intelligent decisions concerning the overall operation of the system. For example, in a distributed system, if the schedulers provide on-line guarantees, the decision to transfer tasks to other nodes can be made in time to meet the tasks' deadlines.

The algorithms presented in this chapter are applicable to a large class of partitionable architectures, such as hypercube systems and mesh-connected systems. In this chapter, we focus our discussion on their application to hypercube architectures. Chapter VI deals with the application of these algorithms and other specific issues for mesh-connected systems.

In this research, we assume that jobs request a complete subcube and the size of the subcube required for the job has already been determined by the time the job arrives to the scheduler. We do not address the problem of determining subcube sizes; this problem has been addressed by others, including Chen [16] and Saad [99]. Since we are interested in on-line algorithms that make scheduling decisions quickly, simplicity of the scheduling process is important. To reduce the complexity of search for a feasible schedule, the algorithms we study make only limited use of preemption: The jobs that are currently executing may be preempted in favor of a newly-arrived job that would otherwise not meet its deadline. We consider alternative approaches with different levels of preemption in Section 4.8.

The design of heuristics for a complex problem such as the one addressed in this research, is a formidable task. It can be simplified, however, by decomposing the problem into logical subproblems and designing heuristics for the subproblems. In the next section, a decomposition of the problem is presented. Heuristics for the subproblems are discussed in Sections 4.3, 4.4 and 4.5. Time complexities of the scheduling algorithms are compared in Section 4.6. Section 4.7 compares the performance of the heuristics and presents the results. Section 4.8 addresses the problem of determining an appropriate level of preemption in the scheduling process.

4.2 Problem Decomposition and Solution Framework

Hard real-time schedulers for parallel tasks in partitionable multicomputers not only have to deal with timing constraints, but also with space constraints, i.e. how should the processors be allocated to jobs so that processor fragmentation is minimized. Three major components

of a scheduler for such environments can be identified – *subunit identification*, *subunit selection* and *job ordering*. In this chapter, subunit identification and subunit selection are referred to as subcube identification and subcube selection, respectively, since in this chapter we deal with hypercube systems exclusively. The subcube identification component is responsible for identifying *suitable subcubes*. A suitable subcube is one that is large enough to execute the requesting job and has enough free time to finish the job before its deadline. The subcube selection component determines which subcube to allocate to the requesting job from the set of suitable subcubes identified by the subcube identification component. The subcube selection component is also responsible for deciding the time at which the job starts executing, if the selected suitable subcube has more available time than is required for the job. Job ordering determines the order in which jobs are considered for scheduling. All three components of the scheduler play important roles. If the subcube identification algorithm is not able to recognize certain available subcubes, some jobs may unnecessarily be rejected. The subcube selection policy can have a significant effect on processor fragmentation. A judicious selection of subcubes by the subcube selection policy can limit processor fragmentation and minimize resource waste due to external fragmentation. In addition, the order in which jobs are considered for scheduling can have a significant effect on the scheduler's ability to meet all timing constraints and, thus, find a feasible schedule.

Before we discuss the algorithms and heuristics for each of the three components of the problem, we look at the overall framework of the solution approach to understand the role played by each of the components.

Two-Phase Scheduling Approach

The delay in making scheduling decisions can be reduced by making scheduling a two-phase process. Instead of completely recreating the schedule on every new job arrival, scheduling a newly-arrived job can be made a two-phase process, where the second phase is optional. In the first phase, the scheduler simply attempts to add the newly-arrived job to the existing schedule without rescheduling any previously-scheduled jobs. To schedule the job, the subcube identification component looks at the current system state and identifies all suitable subcubes. If the set of suitable subcubes identified is non-empty, the subcube selection component selects a subcube from the set for allocation to the job. The subcube selection component also determines the start time of the job, if the selected suitable subcube has more available time than needed for the job. If no suitable subcube is found, the newly-arrived job can not be scheduled under the current schedule. In such cases, the scheduler has the option to go to the second phase and try to build a completely new schedule that includes the new job. The decision to continue to the second phase is based on the amount of time available before the job's deadline. If the available time is more than the sum of the worst-case execution time of the scheduler's second phase and the job's execution time, the second phase is initiated; otherwise the new job is rejected.

The second phase consists of trying to find a feasible schedule by rescheduling previously-scheduled jobs. The scheduler attempts to construct a completely new schedule in order to accommodate the newly-arrived job. The third component – job ordering comes into play at this point. All as yet unfinished jobs present in the system, along with the newly-arrived job are placed in an ordered list (called the *job-queue*) that is ordered according to the job order-

ing discipline. Starting with an empty schedule, the scheduler schedules the jobs one by one in the order in which they appear in the job-queue. The procedure followed to schedule a job is the same as that used in the first phase, the only exception being that if the job can not be scheduled, the scheduling process is aborted. This procedure is repeated for all the jobs in the job-queue, or until a job is encountered that can not be scheduled. If all jobs can be scheduled without any violation of the timing-constraints, the scheduler has found a feasible schedule for the job set that includes the newly-arrived job and all previously-guaranteed but as yet unfinished jobs. If a feasible schedule is found, all currently executing jobs are preempted and the new schedule is adopted. If the scheduler encounters a job that it is unable to schedule, on the other hand, the scheduling process is aborted and the scheduler declares that the new job can not be guaranteed, since a feasible schedule for the job set has not been found. While preparing a new schedule in the second phase of the scheduling process, the scheduler takes into account the overheads of preempting all currently executing jobs and migrating jobs that are allocated on a different subcube under the new schedule.

While this two-phase approach does not improve the scheduler's ability to find a feasible schedule, it has the potential to considerably improve efficiency by avoiding unnecessary reschedulings and preemptions.

4.3 Subcube Identification

The subcube identification problem is closely related to the classic processor allocation problem for hypercube systems. The processor allocation problem deals with finding a currently available subcube of a given size in a general-purpose (i.e. non-real-time) system. Since the

scheduler dynamically allocates subsets of processors to different tasks, *processor fragmentation* occurs. Processor fragmentation is said to occur when there are enough processors available for a particular period of time to satisfy the need of the requesting task, but these available processors do not form a big enough subcube. Processor allocation algorithms have two responsibilities: Minimize processor fragmentation and recognize available subcubes.

Subcube identification is the counterpart of the processor allocation problem for real-time systems. In real-time systems, where schedulers maintain a work-plan in order to provide on-line guarantees to new job arrivals, processors are allocated not only for immediate use, but also for future use. Every job present in the system is either executing currently on a set of processors or has been scheduled to start execution at a future time on a set of processors. In such an environment, scheduling a job involves searching for subcubes of the desired size that are either currently available or will have enough free time in the future to finish the requesting job before its deadline. The strategies proposed for the classic processor allocation problem can be extended in the time dimension to solve the subcube identification problem in hard real-time systems. In this section, we review some of the processor allocation strategies and propose extensions to convert them into subcube identification components for hard real-time environments.

Several strategies for processor allocation in hypercube systems have been proposed, including Buddy [90], Single and Multiple Gray Code [17], and MSS [31]. These strategies vary widely in complexity and *subcube recognition ability* – the ability to identify available subcubes in a hypercube system. Strategies that can identify a free subcube whenever one

is available are said to be *recognition complete*. The Multiple Gray Code and MSS strategies are recognition complete, but carry *much* greater overhead than Buddy. Also, better subcube recognition ability does not necessarily result in better system performance. For general-purpose systems, Krueger et al. [62] showed that despite its poor subcube recognition ability, the Buddy strategy performs almost as well as the Multiple Gray Code and MSS strategies, even when overheads of the algorithms are not taken into account. To study the relationship between subcube recognition ability and performance in hard real-time systems, we compare the performance of the Buddy strategy with that of the Single and Multiple Gray Code strategies in a hard real-time environment. In the comparison is included an *Optimal strategy* – a strategy that completely eliminates external fragmentation and has perfect subcube recognition ability. We describe these strategies as originally proposed for general-purpose environments and also present their extensions for hard real-time environments.

4.3.1 The Buddy Strategy

The Buddy Strategy, originally proposed for storage allocation [57], has since been applied to processor allocation for general-purpose hypercube systems. For job i , requesting a subcube of dimension k within a hypercube of dimension n , the Buddy strategy is as follows: Find the smallest integer j , $0 \leq j \leq 2^{n-k} - 1$, such that all processors in the subcube $\#[j2^k, (j+1)2^k - 1]$ (i.e. the processors numbered from $j2^k$ to $(j+1)2^k - 1$) are available, and allocate these processors to job i . If no such j exists, no subcube can currently be allocated to job i .

The Buddy strategy has been shown to be *statically optimal* [17] in the sense that only minimal subcubes are used by the strategy to accommodate each sequence of incoming requests when processor relinquishment is not considered (i.e. static allocation only).

Extension to Real-Time Systems

To extend the Buddy strategy to the real-time environment, allocation is extended into the time dimension. In addition to processors being allocated for immediate use, reservations are taken for future use. For each processor, the scheduler maintains the *Earliest Available Time* (EAT) – the time at which the processor will become available for use. The EAT of a processor is equal to the completion time of the last job scheduled on that processor or the current time, whichever is later. For job i , requesting a subcube of dimension k within a hypercube of dimension n , the real-time Buddy strategy is as follows: Find the smallest integer j , $0 \leq j \leq 2^{n-k} - 1$, such that all processors in the subcube $\# [j2^k, (j+1)2^k - 1]$ have earliest available times before the latest-start-time (LST – as defined in Section 2.2) of job i , and allocate these processors to job i . If no such j exists, no subcube can complete job i before its deadline. In such a case, job i is rejected.

Data-Structures

A simple implementation of the Buddy strategy for general-purpose systems uses 2^n allocation bits to represent the availability of processors. A bit having value 0 indicates that the corresponding processor is available, while 1 indicates a processor in use. For real-time systems, the array of allocation bits can be replaced with an array of available times, where each entry in the array contains the EAT of the corresponding processor. The efficiency of

such an algorithm can be improved by keeping track of the EATs of processors at the subcube level as well as at the processor level. In our proposed real-time Buddy algorithm, a logical map of the subcubes of each possible size within the hypercube is maintained. An efficient data-structure is used for maintaining this logical map, which contains vital information about each subcube that is recognizable under the Buddy strategy. For a hypercube of dimension 3, the data-structure is shown in Figure 2.

Figure 2: Data Structure for Maintaining the Schedule

The single node at level 3 represents the entire hypercube, which is of dimension 3. At level 2 there are two nodes, which represent the two subcubes of dimension 2 that are recognized by the Buddy Strategy. Similarly level 1 represents the subcubes of dimension 1 and level 0 represents the dimension-0 subcubes (individual processors). Let E_{ij} denote the i^{th} entry on level j . It should be noted that each entry at levels 1 and above is a collective representation (called the 'parent') of the two entries at the next lower level. For example, in this figure, E_{01} is the parent of E_{00} and E_{10} . E_{00} and E_{10} are called 'children of' E_{01} . For

notational simplicity, we consider the ‘children of’ and ‘parent of’ relations to be transitive, so, E_{02} is considered a parent of E_{01} and E_{11} as well as E_{00} , E_{10} , E_{20} and E_{30} . For entry E_{ij} , the set of children $C_{E_{ij}}$ and the set of parents $P_{E_{ij}}$ can be represented as follows:

$$C_{E_{ij}} = \{E_{(i \times 2^{(j-k)})_k} \text{ through } E_{((i+1) \times 2^{(j-k)} - 1)_k}, \forall k, 0 \leq k < j\} \quad (4.1)$$

$$P_{E_{ij}} = \{E_{\lfloor \frac{i}{2^{(k-j)}} \rfloor k}, \forall k, j < k \leq d\}, \text{ where } d \text{ is dimension of the hypercube} \quad (4.2)$$

Each entry in the logical map maintains information about the status of the subcube it represents, including the *Earliest Available Time* – the time at which that particular subcube will become available, and the *Last Job Dimension* – the dimension of the job last scheduled on this subcube.

To schedule a job j_k that requires a subcube of dimension d_k , the subcube identification component examines the entries in level k of the map from left to right, starting with the leftmost entry in that level. Subcubes corresponding to the entries that have early enough EATs to finish the job j_k before its deadline are identified as suitable subcubes. The subcube selection component selects one subcube from the set of suitable subcubes identified and schedules the job j_k on that subcube.

When a job is scheduled on a subcube, the map must be updated. If the job is scheduled on subcube S_k , all its ‘children’ along with its ‘parents’ need to be updated, as shown in Figure 3.

For a large hypercube, carrying out all these updates could be very time consuming. To reduce the time complexity of updates, the proposed scheduler makes use of the fact that entries at any level of the map are searched from left to right. Taking advantage of this fact, the scheduler simply updates the leftmost of all the entries that need to be updated on each

Figure 3: Updates Required

level and makes a note with that entry that its EAT also applies to the following entries on that level.

Figure 4 presents an example of this update method. In this example, a new job is scheduled on the first entry on level 2, i.e. E_{02} . Its parent, E_{03} , and all its children – E_{01} , E_{11} , E_{00} , E_{10} , E_{20} and E_{30} – need to be updated. Instead of updating all the children, we update only the leftmost child on each level. E_{01} , the leftmost child of E_{02} on level 1, and E_{00} , the leftmost child of E_{02} on level 0, are updated. With E_{01} a note is kept that the state information of E_{01} is also applicable to the next entry on that level, i.e. E_{11} . Similarly, with E_{00} a note is kept that the state information of E_{00} is also applicable to the next three entries on that level, i.e. E_{10} , E_{20} and E_{30} .

Using this technique, only one entry typically needs to be updated at each level of the map. However, there are some situations in which two entries need to be updated at some levels. These exceptions occur when the scheduler needs to update an entry E_{ij} whose current status is applicable to the next m entries on that level, while the new update is applicable

Figure 4: Efficient Updates

to only the next n entries, and $n < m$. In such a case, the current status of entry E_{ij} is copied to the first entry beyond the applicability range of the new update, i.e. to entry $E_{(i+n+1)j}$. The new update is then carried out on entry E_{ij} . Figure 5 depicts such a situation. The new job is scheduled on E_{01} , so the scheduler needs to update its children, E_{00} and E_{10} . Since the current status information of E_{00} is applicable to the next three entries, we copy the current status of E_{00} to E_{20} , reduce its applicability to only one following entry, and apply the new update to E_{00} , indicating that this update applies to E_{10} as well.

Complexity

Scheduling overhead is a critical factor in the choice of algorithms for real-time systems. Therefore, it is important to consider the time-complexity of the proposed algorithms. Since the Buddy strategy imposes a linear ordering on the nodes of the hypercube, all suitable subcubes can be identified in a single scan of all the processors in the hypercube system. Since the status of each subcube is maintained in the logical map, the scheduler needs to look

Figure 5: Two Updates at One Level

at the entries of level k of the logical map only, when searching for suitable subcube for a job of size k . The worst-case complexity for a job that requires a subcube of dimension k is $O(2^{n-k})$, where 2^n is the number of nodes in the hypercube and 2^{n-k} is the number of entries in level k of the logical map. For a job requesting a subcube of dimension 0, 2^n entries may need to be examined. Therefore, the worst-case complexity of subcube identification with the Buddy strategy is $O(N)$, where $N = 2^n$ is the number of nodes in the hypercube system.

4.3.2 Gray Code Strategy

The Gray Code (GC) strategy for processor allocation in hypercube systems was proposed by Chen and Shin [17]. In general-purpose systems, the same data-structure as used by the Buddy strategy (an array of 2^n allocation bits) can also be used for the Gray Code strategy. The Gray Code strategy improves its subcube recognition ability over the Buddy strategy by using novel mappings between the allocation bits and node processors. As shown in Figure 6, the Gray Code strategy can identify twice as many subcubes as the Buddy strategy.

To describe the Gray Code strategy, the following notation and terminology is borrowed from [17].

Let $\{g_1, g_2, \dots, g_n\}$ be a sequence of distinct integers. For $1 \leq i \leq n$ the *partial rank* r_i of g_i is defined as the rank of g_i in the set $\{g_1, g_2, \dots, g_i\}$, when the set is rearranged in the ascending order. Let A be a sequence of binary strings of length $n - 1$, $n > 1$. Then a sequence of binary strings of length n , denoted by $A^{b \setminus k}$, $b \in \{0, 1\}$, can be obtained by either inserting a bit b into the position immediately to the right of the k th bit of every string in A , if $1 \leq k \leq n - 1$; or prefixing a bit b to every string in A , if $k = n$. Also, let A^* denote the sequence of binary strings obtained from A by reversing the order of strings in A . For example, if $A = \{00, 01, 11, 10\}$, we have $A^{1 \setminus 2} = \{010, 011, 111, 110\}$, and $A^* = \{10, 11, 01, 00\}$. Using the above notation, Gray Codes are formally defined as follows.

Definition: Let G_n be the GC with parameters g_i , $1 \leq i \leq n$, where $\{g_1, g_2, \dots, g_n\}$ is a permutation of $Z_n = \{1, 2, \dots, n\}$. Then, G_n is defined recursively as follows.

$$G_1 = \{0, 1\}, \quad (4.3)$$

$$G_k = \{G_{k-1}^{0 \setminus r_k}, (G_{k-1}^*)^{1 \setminus r_k}\}, 2 \leq k \leq n \quad (4.4)$$

where r_k is the partial rank of g_k . Since the cardinality of Z_n is n and all its members are distinct, $n!$ distinct permutations of Z_n are possible. Therefore, there are $n!$ distinct Gray Codes for a dimension n hypercube.

For job i , requesting a subcube of dimension k within a hypercube of dimension n , the GC strategy is as follows: Determine the least integer j such

that all $(i \bmod 2^n)$ th allocation bits are 0's (indicating free processors), where $i \in \#[j2^{k-1}, (j+2)2^{k-1} - 1]$. If such a j is found, then allocate processors with addresses $G_n(i \bmod 2^n)$ (G_n is the Gray Code being used) to the requesting job and set the allocation bits to 1's. Since the nodes corresponding to the first and last allocation bits are adjacent to each other, a circular search is allowed in the GC strategy.

Figure 6: Subcube Recognition by Buddy and Gray Code Strategies

We extended the Gray Code processor allocation strategy proposed by Chen and Shin for use as the subcube identification component for scheduling hard real-time tasks. The real-time extension of the Gray Code strategy is similar to the one discussed for the Buddy strategy. Instead of maintaining the busy/free status of each processor, the scheduler keeps track of the EAT of each processor, and the subcube identification component looks for subcubes in which all the component processors have early enough EATs to finish the requesting job before its deadline.

The data-structure proposed for the Buddy strategy can not be used for the Gray Code strategy, because under the Gray Code strategy each processor can be part of two subcubes of a given size. As shown in Figure 6, under the Gray Code strategy, processor 2 is part of two subcubes of dimension 2 – the subcube consisting of processors 0 through 3 and subcube consisting of processors 2 through 5. Similarly, each processor is part of two subcubes of each size. On the other hand, under the Buddy strategy, each processor is part of only one subcube of each size. Therefore, for the Gray Code strategy, the data-structure must be modified. The modified data-structure for the Gray Code strategy has twice as many entries in each level of the map and each node has two immediate parents instead of one. Since the Gray Code strategy examines twice as many subcubes as the Buddy strategy, its complexity is twice the complexity of the Buddy strategy. However, the worst-case time complexity of the Gray Code strategy is still $O(N)$, where N is the number of processors in the hypercube system.

4.3.3 Multiple Gray Code Strategy

Since different GCs are associated with different sets of recognizable subcubes, subcube recognition ability improves as the number of GCs used by an allocation strategy increases. If more than one GC is used, the strategy is referred to as Multiple Gray Code (MGC) strategy. For processor allocation, MGC looks through the GCs sequentially for available subcubes until a free subcube is found or all the GCs have been examined. The procedure for using a GC is the same as described under the single Gray Code strategy. Chen and Shin [17] showed that every subcube in a hypercube is recognizable by at least one GC. Therefore, complete subcube recognition can be achieved if all the $n!$ GCs are used, where $n = \log_2 N$,

is the dimension of the hypercube system. Chen and Shin [17] further showed that since there are overlaps among the sets of subcubes recognized by different GCs, it is possible to achieve complete subcube recognition by using no more than $C_{\lfloor n/2 \rfloor}^n$ Gray Codes. The Multiple Gray Code (MGC) strategy determines the set of $C_{\lfloor n/2 \rfloor}^n$ Gray Codes required for complete subcube recognition.

The MGC strategy as proposed for general-purpose systems can be extended for real-time systems in the same way as discussed for the Buddy and GC strategies. The scheduler maintains the EATs of all processors. The subcube identification component looks for sets of processors with early enough EATs to finish the requesting job before its deadline. However, the data-structure proposed for the Buddy strategy can not be used for the MGC strategy, because under MGC, a given node of the hypercube belongs to a different set of subcubes of a given size, depending upon the Gray Code being used. Therefore, an array of EATs is used for the implementation of the MGC strategy. Since in the worst-case, the MGC strategy needs to examine subcubes recognizable under all $C_{\lfloor \log N/2 \rfloor}^{\log N}$ Gray Codes, the MGC strategy has a worst-case time complexity of $O(C_{\lfloor \log N/2 \rfloor}^{\log N} \times N)$, where N is the number of nodes in the hypercube system.

4.3.4 Optimal Strategy

Although the Optimal strategy is not a practical strategy, it is used here for comparative reasons to find an upper bound on the performance improvement due to the subcube identification strategy. It should be noted that this strategy is not optimal for the overall problem, it is optimal only for subcube identification and subcube selection. The Optimal strategy

completely eliminates processor fragmentation i.e. if sufficient number of processors are available, they form a large enough subcube for the requesting job.

An implementation of the Optimal strategy will involve frequent job preemptions and job reschedulings. During reschedulings jobs may be scheduled on subcubes other than the ones they were executing on under the old schedule. Therefore, several jobs might need to be migrated to different subcubes. Since job preemption and job migration are very time consuming activities, it is not practical to use the Optimal strategy in hard real-time environments.

4.4 Job Ordering

Job Ordering comes into play in the second phase of the two phase scheduling approach. When a new schedule is constructed for the job set consisting of all as yet unfinished jobs present in the system and the newly-arrived job, the job ordering discipline decides the order in which jobs are considered for scheduling.

4.4.1 First-Come-First-Served

Under the First-Come-First-Served (FCFS) heuristic, jobs are considered for scheduling in the order of their arrival. The job with the earliest arrival time is scheduled first. The FCFS heuristic, which is most commonly used in general-purpose systems, is not particularly suited for real-time systems, because neither does it pay any attention to the tasks' timing constraints, nor does it attempt to reduce processor fragmentation in the system. However, FCFS is included in the discussion to provide a baseline for comparison and highlight the significance of job ordering in the scheduling process.

4.4.2 Largest-Dimension-First

Largest-Dimension-First (LDF) is the most commonly used heuristic in bin-packing problems. Chen et al. [15] and Zhu et al. [129] used heuristics similar LDF for static scheduling in a hypercube system under the assumption that all jobs have a single deadline. LDF is effective in reducing external fragmentation in the system. However, since LDF does not consider timing constraints of jobs, it is not well suited for environments where jobs have distinct deadlines. LDF is included in the discussion to compare the relative importance of minimizing processor fragmentation and taking the timing constraints of tasks into account.

4.4.3 Earliest-Deadline-First

Earliest-Deadline-First (EDF) is one of the most commonly used heuristics for static, as well as dynamic, scheduling in real-time systems when jobs have distinct deadlines. Under EDF, jobs are scheduled in ascending order according to their deadlines. The job with the earliest deadline is scheduled first. Dertouzos [24] showed that EDF is optimal for uniprocessor systems, though it has been shown to be suboptimal for multiprocessor systems [84]. The suboptimality of EDF for multiprocessor systems can easily be shown through the following example: Let us assume that the job set shown in Table 2 needs to be scheduled on a two-processor system. When scheduled according to the EDF order, jobs j_1 and j_2 will be scheduled on processors one and two, respectively and j_3 will not be able to meet its deadline. Thus, EDF fails to find a feasible schedule for the given job set. However, it is quite obvious that all three jobs can be scheduled to meet their respective deadlines, if job j_3 is scheduled on one of the processors and jobs j_1 and j_2 are scheduled on the other processor,

Table 2: EDF is Not Optimal for Multiprocessor Systems

Job	Number of Processors	ExecutionTime	Deadline
j_1	1	2	3
j_2	1	2	5
j_3	1	5	6

with j_1 executing before j_2 .

4.4.4 Minimum-Laxity-First

Minimum-Laxity-First (MLF), like EDF, takes individual job deadlines into account, but unlike EDF, it also considers their execution times. Laxity, as defined in Chapter II, is a measure of how much flexibility the scheduler has in scheduling a job. Under MLF, jobs are scheduled in ascending according to their laxities. The job with the minimum laxity gets the highest priority for scheduling. MLF has also been referred to as Least-Laxity-First (LLF) in the literature. Like EDF, MLF is optimal for uniprocessor systems [25], but not for multiprocessor environments. Although MLF will prepare an optimal schedule for the job set shown in Table 2, other examples of multiprocessor scheduling can be presented where MLF is not optimal.

4.4.5 Complexity of Job Ordering

Under all the job ordering heuristics discussed in this section, the overhead of ordering is equal to the overhead of sorting the list of jobs. If m is the number of jobs currently present in the system, including the newly arrived job, the worst-case time complexity of job ordering is $O(m \log m)$.

Table 3: A Sample Job Set

Job	Subcube Dimension	ExecutionTime	Deadline
j_1	2	1.0	1.1
j_2	3	1.0	1.2
j_3	3	1.0	2.1
j_4	2	2.0	3.2
j_5	1	1.0	3.3
j_6	3	1.0	4.1
j_7	1	2.5	4.5
j_8	3	2.0	4.8

4.5 Subcube Selection

As discussed earlier, the subcube selection policy is responsible for selecting a subcube for allocation to the requesting job from the set of suitable subcubes identified by the subcube identification strategy. In this section, we discuss three heuristics for subcube selection.

4.5.1 First-Suitable-Subcube

First-Suitable-Subcube (FSS) is the simplest heuristic. Under FSS, the first suitable subcube identified by the subcube identification strategy is allocated to the requesting job.

FSS takes the first suitable subcube identified by the subcube identification component and allocates it to the requesting job. The job will start executing at the EAT of the subcube allocated to the job.

Since FSS simply takes the first suitable subcube identified by the subcube identification component, FSS's subcube selection process takes constant time.

To see how scheduling process works when FSS is used with the real-time Buddy strat-

egy, consider the job set presented in Table 3. To explain the example, we need to define a few terms: We define a *Homogeneous Subcube*, or *h-subcube*, as a subcube recognized under the Buddy strategy, with all its processors having the same EAT. The EAT of the h-subcube is the EAT of its processors. When two adjacent h-subcubes of dimension k are treated as a single h-subcube of dimension $k + 1$, we say that the two h-subcubes have been *coalesced*. The EAT of the coalesced h-subcube is equal to the greater of the EATs of the two constituent h-subcubes. It is worth pointing out that coalescing two adjacent h-subcubes gives rise to some idle-time on one of the subcubes, since the h-subcubes have different EATs. This idle-time in the schedule can be viewed as a bounded ‘hole’ in the packing. Conversely, when only a portion of an h-subcube is allocated to a job, we say that the h-subcube has been *split*. Unnecessary ‘splitting’ of h-subcubes leads to fragmentation and adversely affects the schedulability of jobs that need large subcubes.

Figure 7 shows how this sample job set would be scheduled under the FSS algorithm when used with the Buddy strategy. FSS begins its search at the lowest numbered processor and schedules the job on the first subcube that is big enough for the job and whose processors have early enough EATs to finish the job before its deadline. We assume for this example that jobs have already been ordered by the job ordering component. To schedule job j_1 , FSS begins its search at processor 0 and finds that processors 0 through 3, which form a dimension-2 h-subcube, are available early enough to finish job j_1 before its deadline. Job j_1 is scheduled on processors 0 through 3, thereby splitting the h-subcube formed by processors 0 through 15 into two parts. Next, job j_2 needs a dimension-3 subcube. Under the Buddy strategy there are two subcubes of dimension 3, one consisting of processors 0

Figure 7: Schedule Prepared By the FSS Algorithm

through 7 and the other consisting of processors 8 through 15. Since not all the processors of the first dimension-3 subcube are available before the LST of job j_2 , the job is scheduled on processors 8 through 15. For job j_3 , processors 0 through 7 are available before its LST, so job j_3 is scheduled on processors 0 through 7 starting at time 1.0. To form this dimension-3 subcube, two dimension-2 h-subcubes (processors 0 through 3, which become available at time 1.0 and processors 4 through 7, which become available at time 0.0) are effectively coalesced. This coalescing leaves some idle-time (0.0 to 1.0) at processors 4 through 7 (indicated by the shaded area in Figure 7). This idle-time is similar to a bounded hole in the packing problem, and is difficult to fill because of the difficulty of finding appropriate job(s) to fit the hole, and because of the overhead involved in keeping track of such holes. In fact, the proposed Buddy algorithm makes no attempt to fill such holes. The Single and Multiple Gray Code strategies also suffer from this problem, because due to the large

overheads involved, the implementations of these strategies do not keep track of slots of free time sandwiched between busy times in the schedule of a processor. The coalescing, which created this particular hole, was unnecessary, though. If job j_3 had instead been scheduled on the second dimension-3 subcube (processors 8 through 15), this coalescing could have been avoided.

Continuing, jobs j_4 , j_5 and j_6 are scheduled on the first large-enough subcube that can finish the job before its deadline. Using the same strategy, job j_7 is scheduled on processors 0 and 1. By scheduling job j_7 on these processors, a dimension-3 h-subcube (consisting of processors 0 through 7) is ‘split’ into smaller subcubes (one dimension-2 subcube and two dimension-1 subcubes). This split could have been avoided by instead scheduling job j_7 on processors 12 and 13, after they finish executing job j_5 . Due to this unwise selection, FSS is unable to schedule the last job of the job set, j_8 , so it is unable to find a feasible schedule for the job set. This example highlights the limitation of the FSS algorithm: Unintelligent selection of subcubes for processor allocation, which leads to unnecessary splits and coalesces of h-subcubes, thereby hurting the algorithm’s ability to find feasible schedules for job sets. As we will see in Section 4.7, these limitations have a significant effect on performance.

4.5.2 Earliest-Start-Time

The Earliest-Start-Time (EST) heuristic is similar to FSS except that instead of the first suitable subcube identified, the suitable subcube having the earliest available time is allocated to the requesting job. Since EST schedules jobs on the earliest available subcubes, no job will ever be scheduled on a subcube where it must wait to start execution while a currently

available suitable subcube is present in the system. Thus, EST potentially improves the resource utilization of the system.

Since EST examines the EATs of all suitable subcubes identified by the subcube identification strategy before making its selection, in the worst-case when each processor in the system is a suitable subcube for a dimension-0 job, EST will have to examine the EAT's of N subcubes, where N is the number of nodes in the system. Therefore, the worst-case time complexity of EST is $O(N)$.

Figure 8: Schedule Prepared By the EST Algorithm

For the job set presented in Table 3, the schedule prepared by EST (shown in Figure 8) is identical to the schedule prepared by FSS. Jobs j_1 and j_2 are scheduled on currently available subcubes of dimension 2 and 3 respectively. For scheduling job j_3 , two subcubes of dimension 3 are available and both are available after time 1.0. EST picks the first of the

two and schedules job j_3 on processors 0 through 7 to start execution at time 1.0. Jobs j_4 , j_5 and j_6 are scheduled on subcubes available at time 1.0, no other subcube can start executing these jobs before time 1.0. For job j_7 there are several options, but the earliest subcube is available only after time 2.0. Job j_7 is scheduled on processors 0 and 1 to start execution at time 2.0. Despite its ability to improve resource utilization, EST fails to find a feasible schedule for the task set presented in Table 3. Like FSS, EST also suffers from potential processor fragmentation problem.

4.5.3 Stacking

The objective of the Stacking algorithm is to avoid the weaknesses that arise in FSS and EST. The Stacking algorithm tries to minimize processor fragmentation by avoiding unnecessary coalesces and splits of h-subcubes. Coalesces lead to 'holes' in the schedule and splits hurt the ability to schedule future jobs that need larger subcubes. The approach used by the Stacking algorithm is to 'stack' equal-sized jobs in the time dimension. Instead of choosing the first available subcube that allows the job's deadline to be met, the Stacking algorithm chooses the earliest available subcube that allows the deadline to be met from among those that require the least number of splits or coalesces.

To accomplish its goal of avoiding unnecessary coalesces of h-subcubes, the Stacking algorithm needs an additional piece of information from the subcube identification component. For each suitable subcube identified, the subcube identification component specifies the dimension of the job last scheduled on that subcube. By examining the dimension of the last job scheduled on the subcube and the EAT of the subcube, the Stacking algorithm can determine whether the subcube under consideration is an h-subcube.

Figure 9: Identifying h-Subcube Boundaries

To select a subcube for a job that requires a subcube of dimension k , the Stacking algorithm examines all the suitable subcubes identified by the subcube identification component. If two adjacent subcubes are h-subcubes and have the same EAT, they are said to form an h-subcube of dimension $k + 1$. Similarly, four consecutive h-subcubes with the same available time are said to form an h-subcube of dimension $k + 2$. An example of identifying h-subcube boundaries is shown in Figure 9.

To avoid unnecessarily splitting large h-subcubes, the scheduler schedules the job on the first dimension- k subcube of the smallest h-subcube that has an early enough EAT to finish the job before its deadline. Among the h-subcubes of the same size, the one with the earliest EAT is selected. We will see that by choosing the earliest available h-subcube from among the smallest h-subcubes, the scheduler tends to 'stack' equal-sized jobs. As a special case, if the scheduler encounters an h-subcube of dimension k that is currently available, it searches no further and schedules the job on that subcube. In a single pass over all the

entries in level k of the logical map (Figure 2), the scheduler can identify the boundaries of h -subcubes and select the most suitable subcube for the job. Since for a given hypercube, the number of subcubes of a given dimension is fixed, it takes constant time to examine all the entries in a single level of the map. Therefore, the maximum time required to schedule a job can be predicted given the job size.

The Stacking algorithm needs to scan through all suitable subcubes only once before selecting a suitable subcube that involves the least number of splits and coalesces. Therefore, the worst-case time complexity of the Stacking algorithm is $O(N)$, the same as the worst-case complexity of EST.

Figure 10: Schedule Prepared By the Stacking Algorithm

A schedule prepared by the Stacking algorithm for the job set given in Table 3 is presented in Figure 10. FSS and EST failed to find a feasible schedule for this job set because

of their inability to avoid unnecessary coalesces and splits, leading to holes in the schedule. The Stacking algorithm avoids these holes by Stacking equal-sized jobs (job j_7 is stacked over j_5 , and j_3 is stacked over j_2). Consequently, by scheduling job j_8 over j_3 and j_2 , the Stacking algorithm succeeds in finding a feasible schedule for the job set, where FSS and EST failed.

4.6 Overall Complexity

Having looked at the complexities of the individual components of the scheduling process, in this section we consider the overall complexity of scheduling a job under the proposed two-phase approach. First we look at the joint complexity of subcube identification and subcube selection for different combinations of heuristics for each of the components. With the Buddy strategy the worst-case complexity of subcube identification is $O(N)$. For subcube selection, FSS takes constant time while EST and Stacking have the worst-case time complexity of $O(N)$. Therefore, the worst-case complexity of subcube identification and selection is $O(N)$ when the Buddy strategy is used for subcube identification.

Since the Gray Code strategy has the same worst-case time complexity as the Buddy strategy, the worst-case time complexity of subcube identification and selection remains $O(N)$ if the Gray Code strategy is used instead of Buddy for subcube identification.

For the Multiple Gray Code (MGC) strategy, the worst-case time complexity is $O(C_{\lfloor \log N/2 \rfloor}^{\log N} \times N)$. Since the MGC strategy has higher complexity than any of the subcube selection heuristics, its complexity will be the dominant factor in the joint complexity of subcube identification and selection if MGC is used for subcube identification. Therefore,

when MGC is used for subcube identification, the worst-case complexity of subcube identification and selection is $O(C_{\lceil \log N/2 \rceil}^{\log N} \times N)$.

In the first phase, scheduling a job entails three steps, subcube identification, subcube selection, and schedule update. Assuming that the Buddy strategy is used for subcube identification, the worst-case complexity of subcube identification and selection is $O(N)$. To update the schedule, no more than two entries need to be modified at each level of the map. Since updating an entry takes constant time and the number of levels is fixed for a given hypercube, the schedule update will require between T and $2T$ time, where T is a constant. Therefore, the overall complexity of the first phase is $O(N)$.

The second phase of the two-phase scheduling process involves sorting the jobs present in system, according to the job-scheduling discipline and scheduling them starting with an empty schedule. All heuristics considered for job ordering (FCFS, LDF, EDF and MLF) have the same worst-case time complexity, therefore, the choice of a specific heuristic does not affect the complexity. The overhead of sorting is $O(m \log m)$, where m is the number of jobs present in the system. The worst-case complexity of scheduling a job set consisting of m jobs is m times the worst-case complexity of scheduling a single job. Therefore, the worst-case complexity of scheduling the job set is $O(mN)$ when Buddy is used for subcube identification. The overall complexity of the second phase is $O(m \log m + mN)$ i.e. $O(mN)$.

In the worst-case both phases of the two-phase scheduling process are required to schedule a newly-arrived job. Therefore, when Buddy is used for subcube identification, the worst-case complexity of scheduling a job is $O(mN)$, where m is the number of jobs currently

present in the system and N is the number of nodes in the hypercube system.

4.7 Performance Comparison

4.7.1 Introduction

In this section, we present the results of a comparative performance study of heuristics for each of the three components of the scheduling process: Subcube identification, subcube selection and job ordering. For these comparisons, the assumptions made about the job arrival process, distributions of task execution times and laxities are the same as described in Section 3.4. The performance indices used are Work Miss Ratio% and Job Miss Ratio%, as described in Section 3.4.

Since subcube allocation, deallocation and schedule preparation is done by the host processor attached to the multiprocessor system, scheduling delays affect only the job being scheduled and any jobs waiting to be scheduled. Since these delays are similar under all algorithms, they are ignored. However, the overhead of preempting currently executing tasks may be significant and the number of task preemptions (after a successful second phase) may vary among different heuristics. Therefore, job preemption and migration overheads are taken into account and referred to as the preemption-migration cost in the remainder of this section. This cost is presented as a ratio of the job preemption and migration overheads to mean job execution time, which is assumed to be 1, as discussed in Section 3.4. For all the results presented in this section, the preemption-migration cost is assumed to be 0.001 of mean job execution time, unless stated otherwise. The overheads of job preemptions and migrations are taken into account by adding the preemption-migration cost to the execution

time of all tasks that are executing at the time, when a new schedule is adopted. This method of accounting for the overheads of job preemptions and migrations is used for all the results presented in this thesis.

We consider three distributions for the dimensions of the requested subcubes, the modified geometric distribution (mean = 2), the discrete uniform distribution, and a ‘reverse’ geometric distribution, which is simply a mirror image of the geometric distribution. All these distributions are truncated at 0 and $n - 1$, where n is the dimension of the hypercube (the reason for truncating the distributions at $n - 1$ instead of n is discussed later in this section). These distributions represent a wide range of possible workloads. While all subcube sizes are equally likely under the discrete uniform distribution, the geometric distribution has a high proportion of small jobs and the reverse geometric distribution has a high proportion of large jobs. The probability mass function for the truncated modified geometric distribution is $p(k) = \frac{p(1-p)^k}{\sum_{k=0}^{n-1} p(1-p)^k}$, where $p = \frac{1}{Mean+1}$. Alternatively, for the discrete uniform distribution, $p(k) = \frac{1}{n}$ and for reverse geometric distribution, $p(k) = \frac{p(1-p)^{n-1-k}}{\sum_{k=0}^{n-1} p(1-p)^k}$. Note that, since the reverse geometric distribution is simply a mirror image of the geometric distribution, $p(k)$ under the reverse geometric distribution is equal to $p(n - 1 - k)$ under the geometric distribution. Although the workload generated by any of these distributions is not expected to precisely match the workload of any real-life system, it is expected that the workload of any specific system will fall within the range of these distributions. Therefore, the performance of any specific system can be expected to fall within the range of the results presented.

4.7.2 Results

Before examining the scheduling issues, we begin with an important workload issue. Scheduling a job that requires a subcube of dimension n , where n is the dimension of the hypercube, can be expected to be difficult because such jobs need the entire hypercube to be available. If the scheduler waits until there are no smaller jobs in the system, it is likely to starve the size n jobs. Alternatively, scheduling size n jobs ahead of smaller jobs is likely to cause smaller jobs to miss their deadlines. In Figure 11, we examine the impact that al-

Figure 11: Work Miss Ratio% vs. Maximum Job Dimension

lowing large jobs in the workload has on performance. Figure 11(a) plots the overall Work Miss Ratio% against the maximum dimension of the jobs included in the workload, while Figure 11(b) plots the Work Miss Ratio% for jobs requesting subcubes of dimension 0, 2 and 4 under FSS (results for EST and the Stacking algorithms are similar). For all the results presented in Figure 11, the Buddy strategy was used for subcube identification and MLF was used for job ordering. Both plots assume a hypercube of dimension 8, an offered

system load of 0.4, mean laxity of 150% of the mean execution time, and geometrically distributed subcube sizes. Figure 11(a) shows that if size n jobs are included in the workload, performance is hurt badly under all algorithms. We have found this result to hold over a wide range of hypercube sizes, offered loads, laxities and subcube size distributions. Figure 11(b) shows that this negative effect is not limited to the size n jobs. The presence of size n jobs makes it harder for the scheduler to schedule smaller jobs as well. Because of their serious negative effect on performance, we assume that size n jobs are precluded from the workload, so the largest job size is $n - 1$.

Subcube Identification

In this section, we compare the performance of subcube identification strategies. Our results are presented in Figures 12 through 14. These figures assume the MLF policy for job ordering and EST for subcube selection. In each of Figures 12 through 14, subfigures (a), (b) and (c) assume a modified geometric distribution, a reverse geometric distribution and a uniform distribution, respectively, for the distribution of subcube sizes.

Figure 12 compares the performance of subcube identification policies against offered system load. For this figure, a hypercube system of dimension 8, and mean laxity of 150% of the mean execution time are assumed. The Buddy, Gray Code and Multiple Gray Code strategies have almost identical performance under all three distributions. The optimal strategy performs significantly better than all the other strategies. Although similar trends are observed in all three subfigures, the largest difference between the performance of the optimal strategy and the other three strategies is observed under the modified geometric distribution.

Figure 12: Work Miss Ratio% vs. Offered System Load

In Figure 13, performance of the subcube identification strategies is compared against mean job laxity. A hypercube system of dimension 8 and an offered system load of 0.4 is assumed for this figure. Again, the Buddy, Gray Code and Multiple Gray Code strategies have almost identical performance in all three graphs. The optimal strategy is slightly better than the rest of the strategies, but the differences are small. The largest difference between the performance of the optimal strategy and other three strategies again exists under the modified geometric distribution. Under the reverse geometric distribution all strategies

Figure 13: Work Miss Ratio% vs. Mean Laxity%

have nearly identical performance.

To study the effects of system size on the performance of subcube identification strategies, Figure 14 plots Work Miss Ratio% against system size. An offered system load of 0.4 and mean laxity of 150% of the mean execution time are assumed for this figure. The results presented in this figure are similar to those observed in Figure 12 and 13. The Buddy, and Gray Code and Multiple Gray Code strategies, again, have nearly identical performance, but the optimal strategy performs significantly better than the other strategies. Difference

Figure 14: Work Miss Ratio% vs. System Size

in the performance of the optimal strategy and other three strategies is again largest under the geometric distribution and smallest under the reverse geometric distribution.

The performance of the optimal strategy represents an upper bound on the performance improvements achievable through processor management. Considering the overheads involved in the implementation of the optimal strategy, the performance of the other three strategies is quite satisfactory. Since the scheduling overheads of the Buddy strategy are much smaller than the overheads of the Gray Code and the Multiple Gray Code strategies,

and all three strategies have nearly identical performance, the Buddy strategy is the strategy of choice for subcube identification.

Job Ordering

In this section, we present the results of a performance comparison of job ordering heuristics. Figures 15 through 18 present the results under different workload conditions. These figures assume a hypercube system of dimension 8 and use the Buddy strategy for subcube identification. The Stacking strategy is used for subcube selection in Figures 15 through 17. Results are similar for FSS and EST. In each of these figures, subfigures (a), (b) and (c) assume a modified geometric distribution, a reverse geometric distribution and a uniform distribution, respectively, for the distribution of subcube sizes.

Figure 15 compares the performance of job ordering policies against offered system load. Mean laxity of 150% of the mean execution time is assumed for all the plots in this figure. MLF and EDF have almost identical performance and, clearly, perform significantly better than FCFS and LDF. The relative performance of these policies is little affected by the offered system load.

Figure 16 compares the performance of job ordering policies against mean job laxity. All plots in this figure assume an offered system load of 0.4. Again, MLF and EDF have almost identical performance and perform significantly better than FCFS and LDF. Under the geometric distribution we find the largest difference between the performance of EDF(MLF) and the other two strategies. When mean job laxity is small, the Work Miss Ratio% is quite high under all job ordering policies, and when mean job laxity is large Work Miss Ratio% is quite small under all policies. This behavior is not surprising, because when laxities are

Figure 15: Work Miss Ratio% vs. Offered System Load

small, all algorithms can be expected to miss a large proportion of jobs, while all algorithms can guarantee most of the jobs when jobs have large laxities. The largest differences among the performance of job ordering policies are observed when mean job laxity is neither small nor large e.g. between 100% and 400% of mean execution time. Surprisingly, even when mean job laxity is as high as 500% of the mean execution time, EDF and MLF perform significantly better than FCFS and LDF.

The job ordering policy may have a significant effect on the number of jobs that need to

Figure 16: Work Miss Ratio% vs. Mean Laxity%

be preempted and migrated. Therefore, the relative performance of the job ordering heuristics depends on the preemption-migration cost. Figure 17 compares the performance of job ordering heuristics under variation in the preemption-migration cost. A mean laxity of 150% and an offered system load of 0.4 is assumed for all the plots in this figure. Once again, MLF and EDF have nearly identical performance when compared with increasing preemption-migration cost. The performance of all policies is severely affected when preemption-migration cost increases beyond 0.01 of mean job execution time. Drops in per-

Figure 17: Work Miss Ratio% vs. Preemption-Migration Cost

formance beyond the preemption-migration cost 0.01 of mean execution time are sharper for MLF and EDF than FCFS and LDF. Trends observed under all three subcube size distributions are similar except that performance under the reverse geometric and the uniform distributions is worse than under geometric distribution, for all job ordering heuristics.

To determine whether the performance of job ordering heuristics is dependent on the heuristics used for subcube selection, Figure 18 studies the performance of job ordering heuristics coupled with different subcube selection heuristics. A mean laxity of 150% of

Figure 18: Work Miss Ratio% vs. Subcube Selection Policy

the mean execution time and an offered system load of 0.4 are assumed for this figure. The results presented in this figure indicate that the performance of the job ordering heuristics depends on the strategy being used for subcube selection. For example, LDF is significantly better than FCFS when used with FSS, but only marginally better when used with EST and Stacking. MLF performs significantly better than EDF when used with FSS, but the two have almost identical performance when used with EST and Stacking.

These comparisons establish that the job ordering policy has a significant effect on the overall performance. A naive policy like FCFS is clearly a bad choice for this environment. The LDF policy, which does not take into account individual job deadlines, but does a good

job of packing the hypercube, performs better than FCFS but does not perform well compared to EDF and MLF. EDF and MLF have almost identical performance and perform significantly better than the performance of FCFS and LDF.

Subcube Selection

In this section, we compare the performance of subcube selection strategies. Figures 19 through 22 present the results of performance comparison of FSS, EST and the Stacking algorithm. Figures 19, 20 and 21 assume a hypercube system of dimension 8, whereas system size is varied in Figure 22. All results presented in this section assume that the Buddy strategy is used for subcube identification and MLF is used for job ordering. MLF is selected for job ordering because MLF performs marginally better than EDF under some workload conditions and significantly better than FCFS and LDF under all workload conditions considered.

Figure 19 plots the Work Miss Ratio% for FSS, EST and the Stacking algorithm against the offered system load, assuming a mean laxity of 150% of the mean execution. For all three algorithms, the Work Miss Ratio% increases monotonically as the system load increases. However, EST and the Stacking algorithm have a much lower Work Miss Ratio% than FSS and, thus, much superior performance. The Stacking algorithm performs better than EST under the geometric and uniform distributions but the two have almost identical performance under the reverse geometric distribution.

Figure 20 plots the performance of the three algorithms against mean laxity (as a percentage of the mean execution time). As mentioned earlier, the laxity of a job is defined as the time difference between current time and the LST of the job. An offered system load

Figure 19: Work Miss Ratio% vs. Offered System Load

of 0.4 is assumed for the plots of this figure. We expect to see little performance difference among the algorithms, when laxities are very small because all algorithms can be expected to miss a large proportion of jobs under such conditions. On the other hand, almost all jobs can be expected to be guaranteed by all three algorithms when laxities are very large. Consequently, we expect the greatest difference in performance at intermediate laxities. Figure 20 shows that the algorithms behave as expected. At low laxities all algorithms have a high Work Miss Ratio% and the difference in their performance is small. However, all

Figure 20: Work Miss Ratio% vs. Mean Laxity%

three algorithms show significant improvement in performance as the mean laxity increases. The Stacking algorithm performs increasingly better than FSS with increasing laxity, up to a mean laxity of about 150%. Beyond that, the difference in their performance decreases. When laxities are small, EST's performance is closer to FSS, however, with increasing laxities its performance approaches that of the Stacking algorithm.

Figure 21 explores the change in performance of the subcube selection algorithms with changes in preemption-migration cost. An offered load of 0.4 and mean laxity of 150% of

Figure 21: Work Miss Ratio% vs. Preemption-Migration Cost

the mean execution time are assumed for this figure. FSS is affected most by the increases in preemption-migration cost, because the number of jobs preempted and migrated under FSS is the largest. The performance of the Stacking algorithm is least affected by increases in preemption-migration cost. By limiting processor fragmentation, the Stacking algorithm is able to reduce the number of job preemptions and migrations required. This reduced number of preemptions and migrations results in increasingly better performance as preemption-migration cost increases.

Figure 22: Work Miss Ratio% vs. System Size

Figure 22 studies the effects of variation in system size on the performance of subcube selection algorithms. An offered load of 0.4 and a mean laxity of 150% of the mean execution time are assumed for this figure. FSS, clearly, has the worst performance under all three distributions. The Stacking algorithm performs better than EST under the modified geometric distribution, but the two have almost identical performance under other two distributions. Performance of all the algorithms, under all three distributions, gets worse as the system size increases.

4.8 Appropriate Level of Preemption

Preemption is a powerful tool for dealing with scheduling problems. In many cases, preemption helps to improve resource utilization; in other cases, it makes scheduling problems computationally tractable. There are many scheduling problems that are NP-hard under non-preemptive scheduling, though polynomial time solutions exist for them if task preemption is allowed. For example, dynamic scheduling of hard real-time tasks on a uniprocessor system has a polynomial time solution when task preemption is allowed, but is NP-hard when tasks are non-preemptable. For a hypercube system, static scheduling of parallel tasks with a common deadline is NP-hard when preemption is not allowed but has a polynomial time preemptive solution [15, 129]. Under the proposed two-phase scheduling approach for on-line scheduling of hard real-time parallel tasks, tasks are preempted if it is not possible to meet the timing constraints of a newly-arrived task under the current schedule. Preemption helps in improving the guarantee ratio, but there is significant overhead associated with task preemption and schedule rearrangement.

In this section, we study the significance of task preemption and schedule rearrangement in our proposed scheduling approach and determine the appropriate level of preemption for best system performance.

4.8.1 Levels of Preemption

Under the proposed two-phase scheduling approach, initiation of the second phase implies that the newly-arrived job can not be guaranteed to meet its timing constraints under the current schedule. During the second phase, the goal of the scheduler is to determine whether it

can guarantee the newly-arrived job by preempting all currently executing jobs and rescheduling them according to a different schedule. If such a schedule is found, all currently executing jobs are preempted and rescheduled according to the new schedule.

There are two kinds of costs associated with preempting currently executing tasks and adopting a new schedule – *job preemption* cost and *job migration* cost. Job preemption cost refers to the cost of preempting an executing process and resuming it later, while job migration cost refers to the cost of transferring a job from one set of processors to another after it has started executing. All currently executing tasks that are preempted in order to adopt the new schedule incur the preemption cost. Under the new schedule, some or all of the jobs may be scheduled on processors different than the ones on which they were executing under the prior schedule. All such jobs that had started executing under the old schedule and are now scheduled on a different set of processors will incur the cost of job migration. All executing tasks can be preempted in parallel and job migrations can take place in parallel as well. Still the cost of task preemption and migration can be significant.

To determine an appropriate level of preemption for good system performance, we consider two alternative approaches. The first approach reduces the amount of preemption by eliminating preemption from the scheduling process i.e. by having a non-preemptive second phase. The second approach uses higher amounts of preemption in the scheduling process by preempting all currently executing jobs and rescheduling them on every new job arrival. By constructing a new schedule on every new job arrival, the scheduler can minimize processor fragmentation and improve the guarantee ratio by sequencing the executions of jobs according to the job ordering discipline being used. We compare the performance of

these alternative approaches with the preemptive two-phase scheduling approach proposed in Section 4.2. In this comparison, we also include a fourth alternative in which there is no second phase at all, i.e. if the newly-arrived job can not be scheduled under the current schedule, it is rejected. So, the four approaches compared in this study are: (a) no second phase, (b) non-preemptive second phase, (c) preemptive second phase, and (d) new schedule on every new job arrival.

No Second Phase

Under this approach rescheduling of already scheduled jobs is completely eliminated. If the newly-arrived job can not be accommodated within the current schedule, it is rejected. This approach does not incur any overheads by preparing new schedules or preempting or migrating jobs. Low scheduling overheads help system performance, but the lack of preemptions is likely to hurt resource utilization and guarantee ratio.

Non-Preemptive Second Phase

Under a non-preemptive second phase, jobs that are currently executing are allowed to continue their execution; only jobs that have been scheduled but have not started their execution yet are allowed to be rescheduled. Thus, job preemption and migration is completely eliminated from the second phase. Reduction in scheduling overheads is expected to help system performance, but the lack of preemption might hurt the system's resource utilization and guarantee ratio.

Preemptive Second Phase

This is the approach described in Section 4.2 and used in all previous sections of this chapter.

A New Schedule on Every Job Arrival

Under the proposed two-phase scheduling approach, jobs are generally scheduled by adding them to the existing schedule. A completely new schedule is constructed only if the new job can not be guaranteed within the current schedule. The schedule that evolves by adding jobs to the existing schedule incrementally as jobs arrive may not be the best schedule in terms of resource utilization and ability to guarantee future arrivals. Better performance might be achieved by preparing a new schedule whenever a new job must be added to the schedule. Through constructing a new schedule on every new arrival, processor fragmentation in the system can be minimized and executions of jobs can be ordered according to the job ordering discipline used. Both reducing processor fragmentation and proper ordering of job executions help improve system's resource utilization and guarantee ratio. Constructing a new schedule on every new arrival also implies many more preemptions and job migrations than the preemptive second phase approach. We wish to see whether the cost of these preemptions and job migrations offsets the benefits achieved by constructing a new schedule.

4.8.2 Performance Comparison

In this section, we compare the performance of our proposed scheduling scheme under four cases: (1) no second phase at all (2) non-preemptive second phase (3) preemptive second phase, and (4) constructing a new schedule on every new job arrival. Assumptions about distributions of job execution times and job laxities are the same as described in Section 3.4. Subcube size distributions are the same as described in Section 4.7. Job Miss Ratio% and Work Miss Ratio% are the indices used for the performance comparison, with only Work

Miss Ratio% plots being shown here, since the trends observed for the two performance indices are identical. In order to look at the inherent performance of the algorithms, all scheduling overheads, including job preemption and migration costs, are ignored in Figure 23. Figure 24 studies the effects on preemption-migration cost on the four approaches being compared. The fourth approach, which constructs a new schedule on every new job arrival has the largest overheads, because it involves the largest number of job preemptions and migrations. This approach benefits the most when all overheads are ignored. The preemptive two-phase scheduling approach also gets some benefits from ignoring all overheads. The other two approaches, no second phase and non-preemptive second phase, are low overhead approaches, which do not stand to gain by ignoring all scheduling overheads.

Figure 23 presents the results of a performance comparison of the EST, Stacking, and FSS algorithms under the four approaches mentioned above. Beginning with EST, we find that its performance is worst when there is no second phase i.e. if a newly-arrived job can not be accommodated within the current schedule, it is rejected. Performance improves slightly if a non-preemptive second phase is allowed. On the other hand, allowing preemption of currently executing tasks in the second phase improves the performance dramatically. However, only a small additional performance gain can be achieved by constructing a new schedule on every new arrival. Even though constructing a new schedule on every new job arrival incurs a great deal of overhead, performance gains are not significant even when all overheads are ignored. Preemptive second phase is the appropriate level of preemption for FSS and Stacking because for both the algorithms, preemptive second phase is not only better than the approaches with no second phase and non-preemptive second phase,

Figure 23: Work Miss Ratio% vs. Offered System Load

but also better than the approach of constructing a new schedule on every new arrival. The reason for this unintuitive behavior is as follows: Whenever a new schedule is constructed, jobs are scheduled in the order determined by the job ordering policy, which in this case is MLF. Due to proper ordering, jobs can be stacked higher in the time dimension. Both FSS and Stacking have the tendency to stack up jobs in the time dimension, even if there are available resources in the system to execute the jobs immediately. This hurts the system's resource utilization and guarantee ratio. As a result we see performance degrade when a

new schedule is constructed on every new job arrival, even when all costs are being ignored. EST does not have this problem because it schedules jobs on the earliest available subcubes, which avoids the stacking of jobs whenever possible and keeps resource utilization high.

The four levels of preemption are compared against preemption-migration cost in Figure 24. Since the first two approaches, with no second phase and a non-preemptive second phase, do not require any job preemptions and migrations, their performance is not affected by variation in preemption-migration cost. The third approach involving preemptive second phase, shows slight degradation in performance with increases in preemption-migration cost. As expected, increases in the preemption-migration cost have a big impact on the performance of the fourth approach, which constructs a new schedule on every new job arrival. Similar trends are observed for all three subcube selection algorithms i.e. FSS, EST and Stacking.

From the results presented in Figures 23 and 24, it can be concluded that the proposed two-phase scheduling approach with preemptive second phase has an appropriate level of preemption. Less preemption hurts system performance badly and more preemption does not improve system performance significantly even if all costs are ignored.

4.9 Summary and Conclusions

Because of their random arrival times and hard deadlines, sporadic tasks present a formidable scheduling problem in hard real-time environments. No previous work has addressed this problem for parallel tasks on partitionable multiprocessors. The motivation behind the work presented in this chapter was to design scheduling algorithms that are able to make schedul-

Figure 24: Work Miss Ratio% vs. Preemption-Migration Cost

ing decisions quickly, since in dynamic real-time environments jobs may miss their deadlines if the scheduling algorithm takes too long in making scheduling decisions.

This chapter discussed a logical decomposition of the problem into three components: *subunit identification*, *subunit selection*, and *job ordering*. To improve the efficiency of scheduling, a two-phase scheduling approach was adopted for on-line scheduling of tasks. In this chapter, we also discussed the role of each of the components in the proposed two-phase scheduling process. Several heuristic-based approaches were evaluated for each com-

ponent. Several algorithms have been proposed for processor allocation in hypercube systems, including the *Buddy*, *Single Gray Code* and *Multiple Gray Code* strategies. These strategies were proposed for general-purpose (i.e. non-real-time) systems. We have proposed extensions to these strategies for hard real-time environments. Efficient data-structures were designed to maintain the information needed by the scheduler so that the scheduling decisions can be made quickly. A performance comparison of these strategies under hard real-time workloads shows that the Buddy strategy, despite its simplicity and low scheduling overheads, performs as well as the Single and Multiple Gray Code strategies, even if the overheads of the algorithms are ignored. Based on the results of this study, the Buddy strategy was adopted for subcube selection.

For job ordering we compared four heuristics: First-Come-First-Served (FCFS), Largest-Dimension-First (LDF), Earliest-Dimension-First (EDF), and Minimum-Laxity-First (MLF). FCFS was found to perform poorly, which is not surprising since FCFS ignores tasks' timing constraints. LDF improves upon FCFS, but was found to be significantly worse than EDF and MLF. EDF and MLF had almost identical performance, but MLF was found to be marginally better than EDF under some workload conditions. We also found that the relative performance of the job ordering heuristics is not independent of the heuristics adopted for subcube selection. However, EDF and MLF were always better than FCFS and LDF. Based on these results, MLF was selected as the job ordering discipline for our continuing work.

Three heuristics were proposed for subcube selection: First-Suitable-Subcube (FSS), Earliest-Start-Time (EST) and the Stacking strategy. FSS simply selects the first subcube

identified that can complete the requesting job in time to meet its deadline. FSS has the tendency of clustering jobs according to their sizes in different parts of the hypercube systems. EST examines all subcubes that can complete the requesting job before its deadline and selects the one with earliest available time. EST improves resource utilization by eliminating such cases where a job has to wait for busy processors while it could have been serviced by another set of currently available processors. Both FSS and EST suffer from processor fragmentation because they take no specific measure to control processor fragmentation in the system. The Stacking algorithm corrects this shortcoming of FSS and EST. It reduces processor fragmentation by judiciously selecting subcubes for allocation to jobs and stacking equal-sized jobs in the time dimension. We compared the performance of FSS, EST and Stacking and found the Stacking algorithm to perform significantly better than FSS and EST when subcube sizes are geometrically distributed and jobs have small laxities or preemption-migration cost is high. Under most other workload conditions considered, Stacking and EST have similar performance, though both perform considerably better than FSS.

In this chapter, we also addressed the problem of determining an appropriate level of preemption in the scheduling process. We compared four approaches involving different levels of preemption. The results of the study established that the proposed two-phase scheduling approach with a preemptive second phase has the right level of preemption to achieve good system performance.

CHAPTER V

Complex Workloads

5.1 Introduction

In this chapter, the proposed scheduling framework for on-line scheduling of hard real-time parallel tasks is extended to handle more complex workloads. We relax some of the assumptions of our workload model and extend our framework to accommodate the resulting workloads.

In our workload model, the execution time of a job is made known to the scheduler at the time the job arrives at the system. However, it is almost impossible to estimate the exact execution time of a job, because of its dependence on run-time factors such as the size of the input, actual input values, system state etc. To ensure that tasks meet their deadlines in the worst case, most hard real-time schedulers schedule tasks with respect to their worst-case execution times [34, 47, 95, 127]. Research addressing the determination of worst-case execution times of jobs includes [43, 92, 118]. Since the worst-case execution time is an upper bound of a job's execution time, most jobs execute much less than their worst-case execution times. Since resources are allocated to jobs for the duration of their worst-case execution times, some of the resources are left unused when jobs complete earlier than their

scheduled completion times. Effective utilization of these unused resources is essential for good system performance.

Utilizing resources left unused due to early job completions is not straight-forward, because all jobs present in the system have already been scheduled for the resources they require. To schedule jobs for the unused resources, the scheduler has to rearrange the execution schedule of the jobs. With the objective of minimizing the overheads, the scheduler tries to utilize the unused resources with minimal changes to the schedule. Making effective use of the resources left unused due to early job completions is the focus of Section 5.2. In this section, methods for utilizing resources left unused due to early job completions are proposed and evaluated. In addition to the three subcube selection heuristics proposed in Chapter IV, we propose another heuristic for subcube selection and compare all proposed subcube selection heuristics under an environment where jobs may complete earlier than their scheduled completion times.

The second type of workload complexity addressed in this chapter is related to the priority constraints of the workload. When scheduling sporadic tasks on-line, it is impossible to guarantee that all tasks will be able to meet their deadlines. Due to abnormal conditions such as overloads, unexpected combinations of events, or multiple failures, the scheduler may not be able to guarantee all tasks. To ensure that critical tasks meet their deadlines, tasks should be prioritized on the basis of their criticalness. The need to give preference to some tasks over the others on the basis of their importance to system, is referred to as the *priority constraints* of the workload. To accommodate priority constraints, we need to relax the assumption of our workload model that all tasks are equally important to the sys-

tem. Section 5.3 describes how our scheduling framework can be extended to handle priority constraints of the workload. The objective of the proposed priority handling scheme is to provide services to high-priority jobs as if no low-priority jobs were present in the system, and at the same time to minimize the negative effects on low-priority jobs.

5.2 Resource Reclaiming When Jobs Complete Early

Early job completions are a rule, not an exception in hard real-time systems. It is extremely difficult to estimate the exact execution time of a job, because the exact execution time of a job depends on many run-time factors like size of the input, actual input values, system state etc. Due to the lack of information about jobs' exact execution times, jobs must be scheduled with respect to their worst-case execution times, so that they can meet their deadlines even in the worst-case. On the average, however, jobs execute much less than their worst-case execution times, so most jobs complete earlier than their scheduled completion times. When a job completes early, the resources that were allocated to the job for the duration of its worst-case execution time are left unused for the remaining duration. There are some other situations as well where jobs are removed from the schedule and leave behind unused resources. For example, a task may be removed from the schedule during an operation mode change¹ [104] or when one of the copies of a replicated task completes successfully in a fault-tolerant system [19]. The effectiveness of the scheduler can potentially be improved by making use of resources left unused by jobs that complete early or jobs that are removed from the schedule for some other reason. In this section, we explore how much performance

¹Operation mode change refers to progress of a system from one phase of execution to another as its mission advances. A mode change is characterized by the deletion of some tasks, addition of new tasks, or changes in the parameters of some existing tasks.

gain can be achieved through making use of the resources left unused and what are good methods of utilizing these resources.

Two conceptually simple approaches for dealing with unused resources are as follows:

No Change in the Schedule: The scheduler may just ignore the early job completions and execute the tasks according to the existing schedule. This approach incurs no additional overheads but considerable resources are wasted. System performance may suffer significantly under this approach.

New Schedule on Every Early Job Completion: Another possible approach is to prepare a new schedule on every early job completion, in order to make use of the resources left unused by the completing job. This approach is good at utilizing resources but incurs considerable overhead. The overheads of preparing a new schedule on almost every job completion are comparable to the overheads of preparing a new schedule on every new job arrival. As shown in Section 4.8, the overheads of preparing a new schedule on every new job arrival can degrade system performance considerably. Similarly, the overheads of preparing a new schedule on every early job completion will offset any advantages of utilizing the unused resources and may have a negative effect on system performance.

Both the approaches suggested above can easily be ruled out. We want to consider methods that can utilize the unused resources without incurring excessive overheads. The main problem with making use of the unused resources is that the scheduler maintains a schedule for use of the system resources by currently present tasks in the system. This schedule is required for providing on-line guarantees, but to schedule jobs for the unused resources, this schedule needs to be rearranged. Our approach is to find methods that can utilize the un-

used resources with minimal changes to the schedule. To maintain the schedule, we design data-structures that incur low overheads for schedule alterations.

In this chapter, again we focus on hypercube systems only and assume that the Buddy strategy is used for subcube identification. In Section 5.2.1, we discuss the flexibility needed from the data-structures in order to minimize the overheads of frequent schedule modifications. Section 5.2.2 describes the data-structures used by the scheduler in order to effectively handle early job completions. We propose a new subcube selection heuristic in Section 5.2.3. Three methods for utilizing unused resources are proposed and compared in Section 5.2.4. Section 5.2.5 compares the subcube selection heuristics in the presence of early job completions.

5.2.1 The Need for Flexibility

In previous chapters, we assumed that every job executes for the exact duration of its declared execution time. Under this model, the scheduler has complete knowledge of the job's characteristics once the job arrives at the system. When early job completions are considered, the scheduler no longer has complete knowledge of jobs' characteristics. It knows the upper bound on a job's execution time but does not know the job's exact execution time. The decisions made by the scheduler, on the basis of this incomplete knowledge, may need to be altered once a job's exact execution time is known, i.e. when the job actually completes. In order to provide on-line guarantees, the scheduling algorithms considered in this thesis maintain a work-plan, i.e. an execution schedule for all the tasks present in the system. To make use of the resources as they become available due to early completions of jobs, the work-plan may need to be modified frequently. With additional resources being

available due to early job completions, the scheduler may be able to guarantee some of the jobs that were not guaranteed at the time of their arrival, provided it is not too late for the jobs to meet their deadlines. These jobs need to be added to the schedule.

Due to the need for frequent alterations to the the schedule, flexibility is required from the data-structures that are used to maintain the schedule. The data-structures need to be efficient not only for adding jobs to the schedule but also for removing jobs from the schedule. In order to make use of the resources that become available due to early job completions, some of the jobs have to be scheduled on these resources, and this may require rescheduling some previously-scheduled jobs on different sets of processors or changes in the *start-times* of some of the jobs. The start-time of a job is the time at which the job is scheduled to start its execution on a pre-specified set of processors. To accommodate frequent changes to the schedule, the data-structures should be flexible for adding jobs to the schedule as well as for removing jobs from the schedule.

Another requirement imposed on the data-structures is that they should not hide any slots of free times sandwiched between durations of busy times in the schedules of processors. The reason for this requirement is as follows: When a job completes earlier than its scheduled completion time, the processors that were executing the job, will have some free time, since no other job has been scheduled on those processors for that duration. The scheduler attempts to make use of this free time by rescheduling some other job(s) on those processors without recreating the schedule completely, since preparing a new schedule after every early job completion is inefficient. Even if the scheduler may not be able utilize the available time on these processors immediately, it should have access to these free time slots, in

order to use them for future job arrivals.

The data-structure described in Chapter IV for maintaining the schedule, though efficient for adding jobs to the schedule, does not meet the additional requirements imposed by early job completions. It does not provide access to the slots of available time on processors and is not efficient for removing jobs from the schedule. New data-structures were designed to provide the flexibility required under the early job completion model without excessive overheads.

5.2.2 Data-Structures

The scheduler maintains two data-structures that meet the requirements imposed by early job completions. Both the data-structures keep track of jobs that are currently present in the system. The first data-structure, referred to as the *Start-Time-Job-List* is a doubly-linked list of all jobs present in the system, maintained in increasing order according to job start-times.

The second data-structure, the *Root-Job-List*, maintains the list of all currently present jobs in the system, ordered according to *roots* of the jobs. The root of a job is the lowest numbered processor among the set of processors, on which the job is scheduled to execute. Jobs with the same root are maintained in increasing order according to their start-times. Both data-structures maintain a list of the jobs present in the system, though in different orders.

To implement these data-structures, only one entry is maintained for each job present in the system. This entry is linked into both the data-structures and maintains information about the start-time of the job and the set of processors on which this job is scheduled to execute. For adding a job to the schedule or for removing a job from the schedule, only

the entry associated with the job needs to be linked to or unlinked from the data-structures. Information about the state of the system is distributed among the job entries linked in the two data-structures. By scanning through the entries for all the jobs present in the system, the scheduler can get all the information about the state of the system. Available times of processors can be derived by scanning through the job entries. No separate data-structure is maintained to keep track of processor available times. Next, we look at how the proposed data-structures are updated when a job is added to or removed from the schedule. We also point out how these data-structures meet the requirements of the early job completion model.

Adding a Job to the Schedule

To identify all suitable subcubes, the subcube identification component searches through the Root-Job-List, starting with the top of the list. Since the Buddy strategy is used for subcube identification, only subcubes recognizable under the Buddy strategy are considered. The Root-Job-List maintains a list of jobs in increasing order of their roots. Therefore, for every candidate subcube, the subcube identification component can find all the jobs that are scheduled on the processors of that subcube. By examining all the jobs scheduled on the processors of a subcube, the scheduler can determine the time durations for which the subcube is available. Once all durations of available time on all subcubes are known, the subcube selection component can decide which subcube to schedule the job on and for what time duration. Since the scheduler must look through the list of all currently present jobs in order to schedule a job, the time complexity of scheduling under the current schedule is $O(m)$, where m is the number of jobs present in the system.

The second data-structure, Start-Time-Job-List, is used to find the earliest time at which any job is scheduled on a given set of processors. This information is required while scheduling jobs in slots of free time on processors that are created due to early job completions.

Removing a Job from the Schedule

Removing a job from the schedule at the time of a job's completion or during a schedule rearrangement involves unlinking the entry corresponding to the job from both the data-structures. Since a job has only one entry, only one search and two unlink operations are required. A pointer to the job entry can be saved with the job at the time of its scheduling. This pointer is passed back to the scheduler at the time of job's completion, to eliminate the need for search. Using this technique, jobs can be removed from the schedule in constant time.

5.2.3 The LST Subcube Selection Heuristic

Three subcube selection heuristics, FSS, EST and the Stacking algorithm were proposed and evaluated in Chapter IV. In this section, we propose another algorithm for subcube selection. The ability of the data-structures proposed in Section 5.2.2 to provide access to slots of free time in the schedules of processors, makes another subcube selection algorithm feasible – Latest-Start-Time (LST). Since the data-structure used in Chapter IV did not provide this access, this heuristic was not used there. The need for LST to have access to slots of available times in the schedules of processors is explained in the following description.

Latest-Start-Time: The Latest-Start-Time (LST) heuristic is the opposite of the EST heuristic. The goal of LST is to schedule jobs to start execution as late as possible, but just

in time to complete before their deadlines. Once a job has been guaranteed, it must be serviced, but not necessarily immediately. Immediate time is more precious than distant time, because immediate time can be used to service jobs that need urgent service. Therefore, postponing the execution of jobs, whenever possible, can improve the system's guarantee ratio by allowing it to guarantee jobs having very short laxities that arrive later. The LST algorithm examines all subcubes of the requested size and schedules the job on the subcube where it can start its execution as late as possible. The latest time at which job i can start its execution and still meet its deadline is $D_i - E_i$, where D_i is the deadline and E_i is the worst-case execution time of job i .

When a job is scheduled on a subcube using the LST heuristic, a slot of free time is created on the subcube between the completion time of the previous job scheduled on the subcube and the start-time of the job being scheduled. These slots of free time in the schedules of processors are used to schedule jobs that arrive later and have tight laxities. Therefore, it is mandatory that the scheduler should have access to these slots of free time, so that they can be used for scheduling future job arrivals.

5.2.4 Making Use of the “Holes”

As discussed earlier, most jobs in hard real-time systems complete earlier than their scheduled completion time and leave behind some unused resources because the resources are allocated to jobs for the entire duration of their worst-case execution time. Resources left unused due to early job completions are similar to “holes” in a packed bin that appear after the bin has been packed. Considerable system resources may be wasted due to these holes in the packing if appropriate measures are not adopted to utilize these holes. Making

efficient use of the resources left unused due to early job completions is crucial for good system performance. In this section, we propose and compare three methods for making use of the unused resources. The proposed methods represent three *levels* of schedule rearrangements performed. The first level performs only minimal changes; the modifications increase progressively with increase in the level. We refer to these three levels as level one, two and three. Level two does everything that level one does plus some additional changes; similarly level three does everything that level two does plus some more. The overheads incurred by these methods are proportional to the amount of schedule alterations they perform.

The proposed methods are described next, followed by the results of a comparative performance study of these methods.

Level 1: Advancing Job Start-Times

A simple method for making use of the resources left unused due to early job completions is to advance the start-times of already scheduled jobs. When a job completes earlier than its scheduled completion time, we can advance the start-times of jobs that are scheduled next on that subcube and start executing them early. Advancing job start-times makes use of the unused resources and improves the probability of future jobs being guaranteed. The simple schedule modification performed by level one helps improve resource utilization and, in turn, improve the guarantee ratio of the system. The only overhead involved in this method is in searching the Root-Job-List for the jobs scheduled next on a set of processors. The worst-case complexity of this search is $O(m)$, where m is the number of jobs present in the system. However, it is not always possible to make use of the unused resources by advanc-

ing the start-times of jobs. Figure 25 depicts one such situation, where job j_3 executing on processors 6 and 7 completes early, but it is not possible to advance the start-time of job j_4 , scheduled next on those processors, because job j_2 is still executing.

The advantage of this method is that it is simple and incurs only a small overhead. On the negative side, this method is not useful in situations such as that depicted in Figure 25.

Figure 25: Job Start-Time Advancing is not Always Possible

Level 2: Scheduling Unguaranteed Jobs and Advancing Start Times

Since it is not always possible to advance the start-times of already scheduled jobs, the proposed start-time advancing method is not adequate to make use of all the resources left unused due to early job completions. The additional resources made available through early job completions may alternatively be used for *unguaranteed* jobs. The unguaranteed jobs are jobs that could not be guaranteed to complete before their deadlines at the times of their

arrival. Retaining unguaranteed jobs in the system with a hope to guarantee them later, when some jobs consume less than their allocated share of resources and complete earlier than their scheduled completion time, can help improve the system's guarantee ratio.

These two methods – advancing start-times and retaining unguaranteed jobs so they can be guaranteed later, can be combined to improve system performance beyond that achievable through either of the methods alone. In the level 2 approach, when a job completes early, the scheduler tries to schedule one or more unguaranteed jobs on the processors left unused by the early completing job, in such a way that the start-times of jobs scheduled to start later on the processors of that subcube are not affected. If no such unguaranteed job can be scheduled in that slot, the start-times of jobs scheduled next on that subcube are advanced to the current time, if possible, and their execution starts immediately. Two kinds of overheads are incurred by this method. The first overhead involves finding the duration of available time on the set of processors with early job completion. This can be done by finding the start-time of the job scheduled next on those processors. The other overhead involves examining the unguaranteed jobs present in the system to see if they can be scheduled to complete within that available time. The worst-case time-complexity of this method is $O(w + g)$, where w is the number of waiting jobs present in the system and g is the number of unguaranteed jobs present in the system.

Level 3: Scheduling Unguaranteed Jobs and Advancing Start-Times Coupled with Job Transfers

The concept of advancing start-times of already scheduled jobs can be further generalized to include jobs scheduled on subcubes other than the one that has available unused resources.

In the level 3 approach, the scheduler first attempts to schedule any unguaranteed jobs on the processors left idle by an early job completion, without affecting the start-times of jobs scheduled next on those processors. If no unguaranteed jobs can be scheduled or all of the processors can not be used by unguaranteed jobs, the scheduler selects the largest *waiting*² job (largest in terms of number of processors required) that can be scheduled on those processors, without affecting the start times of other already scheduled jobs. The selected job is either a job already scheduled on those processors or a job scheduled on a different set of processors. If we assume that the executable image of a job is pre-loaded on the processors on which the job is scheduled to execute, this method involves potential *job transfers*. Job transfer refers to moving a job from one set of processors to another before the job has started its execution. Overheads involved in job transfers are usually much smaller than the overheads incurred by *job migrations*. As explained in Section 4.8, job migration refers to moving a job to a different set of processors, after it has started executing on a set of processors. In the worst-case, the scheduler has to examine all the waiting jobs present in the system twice, first to find the duration of the available time on the processors, and second to determine which job(s) could be transferred to the set of processors left unused due to early job completion. Therefore, the worst-case time-complexity of this method is $O(w + g)$, where w is the number of waiting jobs present in the system and g is the number of unguaranteed jobs present in the system.

²A waiting job is a job that is scheduled to execute on a set of processors, but has not started executing yet.

Performance Comparison

In this section, we compare the performance of the three methods proposed for utilizing the resources left unused due to early job completions. Since the performance of these methods may depend on the heuristic used for subcube selection, we compare their performance under the three subcube selection heuristics proposed in Chapter IV and LST described in Section 5.2.3. For these comparisons, the assumptions made regarding the job arrival process, the distributions of execution times and laxities are the same as described in Chapter III. Work Miss Ratio% and Job Miss Ratio% (as described in Chapter III) are used as performance metrics. As described in Section 4.7, three distributions were considered for subcube sizes – a uniform distribution, a modified geometric distribution and a reverse geometric distribution. Preemption-migration cost is assumed to be 0.001 for all the results presented in this section. As described in Section 4.7, the overheads of job preemptions and migrations are taken into account by adding the preemption-migration cost to the execution time of all tasks that are executing at the time, when a new schedule is adopted.

Subcube hold time or task execution time, as described in Section 3.4, is considered to be the worst-case execution time in this chapter. The actual execution time of a task is derived from its worst-case execution time by multiplying it by a factor f . Factor f is chosen from a truncated exponential distribution with mean m_{asd} and truncated at the mean worst-case execution time 1, where m_{asd} is the desired mean actual execution time. The ratio of m_{asd} to the mean worst-case execution time is referred to as *Early Completion Ratio* (EC Ratio). Since it is not possible for an exponential distribution, which is truncated at 1, to have a mean greater than 0.5, the exponential distribution is replaced with the *reverse exponential*

distribution if the desired m_{asd} is greater than 0.5. The reverse exponential distribution, is a mirror image of the exponential distribution. The relationship between exponential and reverse exponential distributions is similar to the relationship between geometric and reverse geometric distributions, as described in Section 4.7,

Calculations of the offered system load for all the results presented in this chapter are based on actual job execution times, rather than worst-case job execution times.

Results

Figure 26 presents the results of a performance comparison study of the proposed methods for utilizing unused resources. For the results presented in this figure, we assume a hypercube system of dimension 8, maximum job size of dimension 7 (reasons are explained in Section 4.7), an offered system load of 0.4 and a mean job laxity of 150% of the mean worst-case execution time. The EC Ratio is assumed to be 0.1, because, in general, worst-case execution times of tasks are much higher than their mean execution times. Job transfer overheads were ignored for these results, because these overheads depend on whether a waiting job has been pre-loaded or not, at the time of transfer.

The results presented in Figure 26 show that under the level 1 method of hole usage, i.e. advancing start-times, all subcube selection strategies perform quite poorly. LST performs considerably worse than all other heuristics. The EST and Stacking strategies have nearly identical performance and perform significantly better than FSS. Performance improves considerably when the level 2 method is adopted, i.e. when the scheduler uses the holes created by early job completions to schedule unguaranteed jobs as well as to advance start-times. The Stacking algorithm performs significantly better than all other heuristics.

Figure 26: Resource Usage Methods vs. Subcube Selection Policies

LST shows the largest improvement in performance when the level 2 method is used instead of the level 1 method. LST shows further improvement going from the level 2 to the level 3 method. The level 3 method allows for waiting jobs (jobs that have been scheduled but have not started executing yet) from other subcubes to be scheduled in the newly created holes. This gives the scheduler more flexibility in selecting jobs to be scheduled in the holes. FSS and EST show small improvements in performance going from the level 2 to the level 3 methods, whereas the Stacking algorithm does not benefit from the additional scheduling flexibility provided by the level 3 method.

We see that LST goes from the worst under the level 1 method to the best under the level 3 method. The reason for the big improvement in LST's performance is as follows: Apart from the holes created in the schedule due to early job completions, LST introduces additional holes in the schedule by the virtue of its *modus-operandi*. Under LST, jobs are intentionally scheduled to start executing as late as possible, allowing for holes in the schedule, so that the future job arrivals with tighter laxities can be scheduled in those holes without any modifications to the current schedule. Therefore, the schedule prepared with the LST heuristic has inherently many more and bigger holes than schedules prepared with the other three heuristics. Considerable resources are wasted and performance suffers if these holes are not utilized properly. Moving from the level 1 method to the level 3 method we improve the ability to use holes. Since LST results in many more holes than the other heuristics, it achieves the biggest improvements in its performance. The Stacking algorithm attempts to minimize the number of holes, therefore, we do not see much change in its performance going from the level 2 to the level 3 method.

5.2.5 Performance Comparison of the Subcube Selection Policies

The objective of subcube selection algorithms is to minimize processor fragmentation in the system. Different algorithms succeed to a different extent in their objective. Remaining processor fragmentation shows up in the schedule as holes. Since under the proposed extended framework we have the ability to make use of these holes, this may significantly change the relative performance of subcube selection heuristics as compared to that observed in Section 4.7. In this section, we compare subcube selection heuristics under the extended

framework, where resources wasted due to holes in the schedule can be utilized to a certain extent.

In Chapter IV, three subcube selection heuristics, First-Suitable-Subcube (FSS), Earliest-Start-Time (EST), and the Stacking algorithm, were proposed and compared. Another subcube selection heuristic, LST was proposed in Section 5.2.3. We compare the performance of these four heuristics under an environment where jobs may complete earlier than their scheduled completion time.

For all the results presented in this section, our assumptions about the job arrival process, and distributions of subcube hold times and laxities are the same as described in Section 3.4. The procedure for determining the actual execution times of jobs is the same as described in Section 5.2.4. A hypercube system of dimension 8 and maximum job size of dimension 7 are assumed. Unless stated otherwise, an EC Ratio of 0.1 and a preemption-migration cost of 0.001 of mean worst-case execution time are assumed. Since it results in the best performance, we assume that the level 3 method of utilizing unused resources, which involves scheduling unguaranteed jobs and advancing start-times coupled with job transfers, is employed for making use of the resources left unused due to early job completions.

In each of Figures 27 through 31, subfigures (a), (b) and (c) assume a modified geometric distribution, a reverse geometric distribution and a uniform distribution, respectively, for the distribution of subcube sizes.

Results

We begin by comparing the performance of the subcube selection policies against offered system load. Figure 27 plots Work Miss Ratio% for each subcube selection policy against

offered system load. The mean laxity is assumed to be 150% of the mean worst-case execution time. Under a modified geometric distribution for subcube sizes, there are only marginal differences in performance between the subcube selection policies. The LST, EST and the Stacking algorithm have nearly identical performance when offered system load is less than 0.5. Beyond that, EST performs worse than LST and Stacking. FSS has the worst performance when offered system load is below 0.6, however, at higher loads EST is worse than FSS. Differences among the performances of subcube selection policies are even smaller under the uniform and reverse geometric distributions. LST performs slightly better than the other heuristics, particularly at low loads.

It is not surprising to find only a small difference between the performance of subcube selection heuristics. The objective of the subcube selection component is to select subcubes in such a way that processor fragmentation and thus, holes, in the schedule are minimized. However, under the new framework, the scheduling algorithms have the ability to make use of the holes in the packing, which reduces the differences in the performance of subcube selection heuristics.

Next, we study the effects of changes in the EC Ratio on the the performance of subcube selection heuristics. Results of this study are presented in Figure 28. An offered system load of 0.4 and a mean laxity of 150% of the mean worst-case execution time are assumed for these results. The performance of all the algorithms suffers when the EC Ratio increases. This result is a bit counter-intuitive: the more accurate knowledge we have of jobs' execution times, the worse we do. The reason for this behavior is as follows: When the EC Ratio is small, the holes created in the schedule by early job completions are much bigger in the

Figure 27: Work Miss Ratio% vs. Offered System Load

time dimension, whereas a high EC Ratio results in smaller holes. Small holes can not be used effectively, so considerable resources are wasted. On the other hand, when the holes are large, jobs can easily be scheduled in those holes and resource waste is minimized. Performance of FSS is significantly worse than all other algorithms under all three distributions. LST is significantly better than the others under the reverse geometric distribution and for low EC Ratios under the Uniform distribution. Under the modified geometric distribution, LST, EST, and Stacking have nearly identical performance for small EC Ratios, but with

increases in EC Ratio EST gets a little better than Stacking and LST gets a little worse than Stacking.

Figure 28: Work Miss Ratio% vs. Early-Completion Ratio

In Figure 29, we study the variation in the performance of the subcube selection algorithms with changes in the mean laxity of the jobs constituting the workload. An offered system load of 0.4 is assumed for this figure. Note that all plots in Figure 29 use a logarithmic scale for the X and Y axes. Laxities are represented as a ratio of mean worst-case

execution time. Under the modified geometric distribution, EST and Stacking have almost identical performance. LST performs slightly better than EST and Stacking and considerably better than FSS, especially when mean laxity is high. FSS is significantly worse than all others when mean laxity is more than 50% of mean worst-case execution time. Under the reverse geometric distribution FSS, EST, and Stacking have nearly identical performance; LST performs slightly better than the others. Results are similar under the Uniform distribution. For small laxities all algorithms have equally bad performance. From moderate to high laxities FSS and Stacking are nearly identical and EST performs slightly better than them. LST performs a little better than all other heuristics.

These results show that for tight laxities all algorithms perform quite poorly and for large laxities all algorithms perform well. It is not surprising to see only small differences between the algorithms on the two extremes. When laxities are small, all algorithms miss many jobs, and when laxities are large, all algorithms are able to schedule most of the jobs. Intermediate laxities show more difference in the performance of the algorithms.

The subcube selection strategy can potentially have a significant effect on the amount of job preemptions and migrations required in the scheduling process. Therefore, variation in the cost of job preemption and migration can significantly change the relative performance of the subcube selection policies. Figure 30 presents the performance of subcube selection policies against preemption-migration cost. An offered system load of 0.4 and a mean laxity of 150% of mean execution time are assumed. Again, a logarithmic scale is assumed for both the axes. Under all three distributions, the performance of all algorithms compared degrades sharply when preemption-migration cost increases beyond 0.01 of mean execution

Figure 29: Work Miss Ratio% vs. Mean Job Laxity

time. Under the modified geometric distribution FSS is significantly worse than the other algorithms when preemption-migration cost is low, but the difference in the performance of FSS and other heuristics decreases with increasing cost. Performance of EST degrades most quickly with increases in preemption-migration cost under all three distributions, which indicates that EST involves the largest number of job preemptions and migrations. Stacking is nearly identical to EST under the geometric distribution and worse under the other two distributions, when preemption-migration cost is low, but significantly better than EST for high

preemption-migration cost under all three distributions. Under the uniform and reverse geometric distributions, LST is a significantly better than all other algorithms, but differences are relatively small under the modified geometric distribution.

Figure 30: Work Miss Ratio% vs. Preemption-Migration Cost

Next, we compare the performance of the subcube selection policies against the size of the hypercube system. Results of the comparison are presented in Figure 31. For all the results presented in this figure, we assume an offered system load of 0.4 and a mean laxity

of 150% of mean worst-case execution time. Under all three distributions considered, performance of the four subcube selections algorithms degrades when system size increases. Under the modified geometric distribution, the rate of performance degradation is considerably higher for FSS than the other algorithms, therefore, differences in the performance of FSS and other heuristics widen with increases in the system size. Among the other heuristics, LST performs marginally better than EST and Stacking. There are no significant differences in the performance of EST and Stacking. The rate of drop in performance with increases in the system size is nearly identical for LST, EST and Stacking. The trends observed for the reverse geometric and uniform distributions are significantly different from trends observed for the modified geometric distribution. Differences in the rates of performance degradation for EST, LST and Stacking are larger than the differences observed under the modified geometric distribution. Under the reverse geometric distribution as well as the uniform distribution, LST is significantly better than the other three heuristics. EST has the second best performance in both the cases. The difference between the performance of LST and EST is most significant under the reverse geometric distribution.

The results presented in this section show that LST performs significantly better than the other heuristics, particularly under the uniform and reverse geometric distributions. In particular, the performance of LST is significantly better than the other heuristics when the system size is large, preemption-migration cost is high, or the EC Ratio is small. EST and Stacking have nearly identical performance under most of the workload conditions considered. FSS is considerably worse than the other heuristics almost under all workload conditions considered.

Figure 31: Work Miss Ratio% vs. System Size

5.3 Priority Constraints

On-line scheduling algorithms dealing with sporadic tasks can not ensure that all tasks will meet their deadlines. Due to overloaded intervals or unexpected failures, some tasks are not able to meet their deadlines. To make sure that all critical tasks meet their deadlines, tasks should be prioritized on the basis of their importance to system. Critical tasks should be guaranteed to meet their deadlines, even if some non-critical tasks have to have their

guarantees revoked because of it. In order to accommodate these priority constraints of the workload, we need to change our workload model, which assumes that all tasks are equally important to the system. In this section, we discuss the extension of our proposed framework, to honor priority constraints of the workload and show how priority constraints can be handled effectively under the extended framework.

We propose two policies: one that maintains strict priority order and the other that attempts to minimize the negative effects of priorities on low-priority jobs without hurting high-priority jobs, by considering intermediate schedules between the extremes of strict priority schedules and schedules prepared without considering task priorities. The goal of the proposed priority handling policy is to provide service to high priority jobs, as if no low-priority jobs were present in the system and, at the same time minimize the effects of priority constraints on low-priority jobs, as much as possible. Although, in this section we assume, there are only two priority levels – high-priority and low-priority, the priority handling techniques discussed here can easily be extended to multi-level priority systems.

The notion of priorities used in this section is strictly based on the criticalness of tasks. These priorities should not be confused with the priorities used by many real-time scheduling algorithms, which are based on task characteristics such as timing constraints, execution times or number of processor required.

Provisional Guarantees

When all tasks are equally important to the system, i.e. they all have same priority, tasks are guaranteed to complete before their deadlines in spite of future arrivals. However, when tasks have different priorities, cases such as the following may arise: Suppose a low-priority

task T_l has been guaranteed and a new high-priority task T_h arrives. Also suppose that T_h can not be guaranteed unless T_l is removed from the schedule. In such a case, previously-guaranteed low-priority task T_l is removed from the schedule and it is said that the guarantee of T_l has been *revoked*. Therefore, the initial guarantees provided to low-priority tasks are *provisional*. A low-priority task is guaranteed to complete before its deadline if no high-priority task arrives that has a conflict with the low-priority task. That is the semantics of guarantees used for low-priority jobs throughout this section.

Guarantee revocation is like breaking a promise; a task that was guaranteed to meet its deadline is no longer guaranteed after the guarantee revocation. For a task, guarantee revocation might be worse than being rejected at the time of arrival, because at arrival-time some alternative measures could have been adopted, but now it might be too late for that. Therefore, it is important for the scheduler to minimize the number of guarantee revocations as much as possible.

5.3.1 Strict Priority Ordering

Our approach to priority handling is centered around the job ordering component, described in Section 4.4. The other two components of the scheduling process, subcube identification and subcube selection are not affected by the priority constraints of the workload.

In Chapter IV, four heuristics – FCFS, LDF, EDF and MLF, were proposed and compared for job ordering. In this section, we propose another heuristic, Priority Ordering (PO), to handle workloads with priority constraints.

Priority Ordering (PO): Under the Priority Ordering heuristic, jobs are scheduled in increasing order according to their priorities, which are based on the importance of the task

to the system. Jobs with the same priority may be ordered using any other job ordering discipline such as EDF, MLF, LDF, or FCFS. In this chapter, we assume that MLF is used for ordering jobs that have the same priority. We refer to the schedule prepared by scheduling jobs in the order determined by PO, as the *Priority Schedule*.

Schedules prepared by scheduling jobs in the order determined by EDF, MLF, LDF or FCFS, without paying any attention to task priorities, are referred to as *No-Priority* schedules. In this chapter, No-Priority Ordering is assumed to be MLF.

Handling Priorities

Both high-priority and low-priority jobs are handled identically under both phases of the scheduling process. Job priorities are not taken into account, unless the scheduler fails to schedule the newly-arrived job under the regular scheduling process. In such a case, the scheduler examines the priority of the new job. If it is a low-priority job, it is rejected; otherwise the scheduler constructs a Priority Schedule.

If any high-priority job fails to meet its timing constraints under the Priority Schedule, the new job is rejected; otherwise the Priority Schedule is adopted. If in adopting the schedule, some or all of the low priority jobs can not be scheduled, they are removed from the schedule and it is said that the guarantees of these jobs have been *revoked*. These low-priority jobs with revoked guarantees are kept with the unguaranteed jobs present in the system. In the future, whenever additional resources become available due to early job completions, the scheduler first attempts to guarantee any high-priority unguaranteed jobs present in the system, using these resources. After considering high-priority unguaranteed jobs, if some resources are still available, the scheduler will try to guarantee low priority jobs that

had their guarantees revoked. If some resources are still available, other low-priority unguaranteed jobs will be considered for scheduling.

5.3.2 Exploring Intermediate Orderings

The priority handling mechanism described in Section 5.3.1, considers schedules prepared by scheduling jobs in the order determined by either Priority Ordering or No-Priority Ordering. Schedules prepared by these two orderings represent the two extremes. No-Priority ordering does not take priorities into account, therefore, high-priority jobs may be hurt by the low-priority jobs present in the system. On the other extreme, Priority Ordering strongly favors high-priority jobs and may unnecessarily hurt low-priority jobs. Possibly, there are *Intermediate Orderings* that give high-priority jobs enough preference so that all high-priority jobs meet their deadlines, but minimize the number of low-priority jobs that have to be removed from schedule. Our goal is to search for these Intermediate Orderings in order to help the low-priority jobs without hurting high-priority jobs. The schedule prepared by scheduling jobs in the order determined by an Intermediate Ordering is referred to as *Intermediate Schedule*. The search for Intermediate Orderings is conducted only if all high-priority jobs are able to meet their deadline under the Priority Schedule; otherwise the new job is rejected and no changes are made to the current schedule.

We adopt a *binary search* approach to search for Intermediate Orderings between the two extremes of Priority Ordering and No-Priority Ordering. The search is guided by the *quality* of the Intermediate Orderings. The quality of an ordering is measured in terms of the number of low-priority jobs that will have their guarantees revoked under the schedule prepared by scheduling jobs in that order. An ordering with a lower number of guarantee

revocations under its schedule is said to be *better* than orderings with higher numbers of guarantee revocations under their schedules. If the number of guarantee revocations is zero under an ordering, the ordering is said to be the *best*. There might be multiple best orderings. If a high-priority job fails to meet its deadline under an ordering, the ordering is declared to be invalid. An invalid ordering is considered to be worse than all valid orderings. The scheduler maintains two lists, hp-list and lp-list. Both the lists are ordered according to the No-Priority Ordering, which is assumed to be MLF in this chapter. To construct a schedule, the scheduler first schedules all the jobs from hp-list and then the jobs from the lp-list. Within each list, jobs are scheduled in the order in which they appear in the list.

Priority Ordering is assumed to be the starting point of the search. All high-priority jobs are in the hp-list and all low priority jobs are in the lp-list. In the first step of the search, the scheduler takes half the low-priority jobs from the lp-list and puts them in the hp-list and constructs a schedule. If the new ordering is better than the previous ordering (in the first step Priority Ordering is the previous ordering), half of the remaining jobs from the lp-list are moved to the hp-list and yet another schedule is constructed. On the other hand, if the ordering prepared in the first step is worse than the previous ordering, half of the low-priority jobs are transferred from hp-list back to the lp-list and another schedule is constructed. In each step of the search, the number of jobs transferred between lists is half the number of jobs transferred in the previous step. The direction of the search, i.e. jobs are added to the hp-list or removed from the hp-list, is determined by the quality of new ordering compared with the previous ordering. The maximum number of steps required is $\log_2 LP$, where LP is the number of actual low-priority jobs in the system. If at any point during the search,

a schedule is found where all jobs meet their deadlines, the search stops and that schedule is adopted. At the end of the search, the schedule with the minimum number of guarantee revocations encountered by the scheduler during the search is adopted.

It should be pointed out here that due to the heuristic nature of the algorithm used to find a schedule, the quality of the latest schedule may not monotonically increase with increasing number of steps in the search. The overhead of the search depends on the number of Intermediate Orderings examined, i.e. the number of steps in the search. To ensure that the job that is being scheduled does not miss its deadline because of the overhead of the search process, before each step the scheduler checks if the job being scheduled has more laxity than the worst-case time requirement of preparing a schedule. If the job being scheduled does not have enough laxity, the search stops; otherwise the next step of the search is carried out.

5.3.3 Results

In this section, we compare schedules prepared by three types of job orderings – No-Priority Ordering that completely ignores task priorities, Priority Ordering that strictly adheres to task priorities, and Intermediate Ordering that is determined by a binary search between Priority Ordering and No-Priority Ordering.

For all the results presented in this section, we assume that the Buddy strategy is used for subcube identification and EST for subcube selection. A hypercube system of dimension 8, maximum job size of dimension 7 (reasons are explained in Section 4.7), and preemption-migration cost of 0.001 of mean worst-case execution time are assumed. Assumptions about distributions of worst-case job execution times and laxities are as described in Section 3.4.

Actual execution times are derived as described in Section 5.2.4. An Early Completion ratio (EC Ratio) of 0.1 is assumed for all the results presented. Unless stated otherwise, the ratio of high-priority jobs in the workload is assumed to be 10%. This is a reasonable assumption, because the proportion of critical tasks in a system's workload is not expected to be high. We also explore the effects of modifying this assumption; results of this study are presented in Figure 33. For subcube sizes, again three distributions were considered – a modified geometric distribution, a reverse geometric distribution and a uniform distribution, as discussed in Section 4.7. In order to find an upper bound on the performance gains achievable through Intermediate Ordering, overheads of searching for Intermediate Ordering are assumed to be negligible.

Keeping the goals of the priority policies in mind, three performance indices are used to compare the performance of the three orderings. The performance indices used are the LP Work Revoked Ratio, HP Work Miss Ratio at Arrival and LP Work Miss Ratio. The LP Work Revoked Ratio is the ratio of work corresponding to low-priority jobs that had their guarantees revoked to the work corresponding to all low-priority jobs that arrived at the system. HP Work Miss Ratio at Arrival is the ratio of work corresponding to the high-priority jobs that could not be guaranteed at the time of their arrival to the work corresponding to all high-priority jobs that arrived at the system. LP Work Miss Ratio is the ratio of work corresponding to low-priority jobs that missed their deadlines to the work corresponding to all low-priority jobs that arrived at the system. To improve the understandability of the plots, we plot ratio%, which is the ratio multiplied by 100. Trends observed under all three distributions are found to be nearly identical. Therefore, we present the results for the modified

geometric distribution only.

We begin by comparing the performance of No-Priority Ordering, Priority Ordering and Intermediate Ordering against the offered system load. Results of this comparison are presented in Figure 32. For these results, the mean laxity of jobs is assumed to be 150% of mean worst-case execution time. Figure 32(a) plots the LP Work Revoked Ratio% against the offered system load. Since under No-Priority Ordering, no job ever has its guarantee revoked, its LP Work Revoked Ratio% is zero. Intermediate Ordering is able to reduce the LP Work Revoked% by a small margin, over Priority Ordering. Figure 32(b) compares the HP Work Refused at Arrival% for the three orderings. The results show that the Priority and Intermediate Orderings have almost identical performance but No-Priority Ordering performs considerably worse than these two. The LP Work Miss Ratio% is plotted in Figure 32(c). Performance of all three Orderings is nearly identical. Priority and Intermediate Orderings have almost identical performance as No-Priority Ordering with respect to low-priority jobs. Differences in the performance of Priority and Intermediate Orderings are not comparable to differences observed in their performance for the LP Work Revoked%. The reason for this behavior is that many low-priority jobs that have their guarantees revoked are guaranteed again when additional resources become available due to early job completions. Even though many tasks whose guarantees are revoked are eventually able to meet their deadlines, still it is important to minimize the number of guarantee revocations. Guarantee revocation creates uncertainty for the task, as it becomes unknown whether the task will be able to meet its deadline, and by the time the task finds out that it will not be able to meet its deadline, it might be too late to take alternative steps.

These results show that the Intermediate Ordering policy reduces the number of low-priority jobs that have their guarantees revoked by a small percentage, but has no negative effects on the performance of high-priority jobs.

Figure 32: Work Miss Ratio% vs. Offered System Load

Next, we look at the performance of the three priority handling policies against variation in the percentage of high-priority jobs in the workload. Figure 33 plots these results assuming an offered system load of 0.4 and a mean laxity of 150% of mean worst-case execution

time. Since No-Priority Ordering completely ignores priorities, the performance of low-priority as well as high-priority jobs is independent of the ratio of high-priority jobs in the workload. Intermediate Ordering is able to reduce the LP Revoked Ratio% over Priority Ordering (Figure 33(a)) while having the same HP Work Refused at Arrival% (Figure 33(b)). Figure 33(b) shows that the HP Work Refused at Arrival% increases for Priority Ordering as well as Intermediate Ordering and approaches the performance of No-Priority Ordering as the percentage of high-priority jobs in the workload increases from 10% to 90%. As shown in Figure 33(c), the LP Work Miss Ratio% is significantly lower for Intermediate Ordering than for Priority Ordering, particularly when proportion of high-priority jobs in the workload is high. Priority Ordering and Intermediate Ordering perform considerably worse than No-Priority Ordering, as far as LP Work Miss Ratio% is concerned.

Figure 34 presents the results of a comparison of the three priority handling policies against the mean laxity. For the results presented in this figure, it was assumed that the offered system load is 0.4. The results of this study are similar to the ones presented in Figures 32 and 33. Intermediate Ordering is able to improve the performance of low-priority jobs marginally without affecting the performance of high-priority jobs. Priority Ordering and Intermediate Ordering both significantly improve the performance of high-priority jobs. As shown in Figure 34(b), HP Work Refused at Arrival% is significantly lower under Priority and Intermediate Orderings as compared to No-Priority Ordering. The results presented in Figure 34(c) show that all three orderings have almost identical performance with respect to the LP Work Miss Ratio%.

From the results presented in this section, it can be concluded that the proposed priority

Figure 33: Work Miss Ratio% vs. Perc. of HP Jobs

handling policies are able to shield high-priority jobs from the effects of low-priority jobs present in the system. Performance of high-priority jobs is extremely good even at very high offered system loads and moderate job laxities. The Intermediate Ordering policy is able to reduce the number of low-priority jobs that have their guarantees revoked, by a small proportion, without any negative effects on the performance of high-priority jobs. Future research should investigate better methods of searching for intermediate Orderings and better heuristics to guide the search.

Figure 34: Work Miss Ratio% vs. Mean Laxity%

5.4 Summary and Conclusions

In this chapter, we addressed handling two types of workload complexities. The first complexity is related to the reuse of resources left unused due to early job completions. To ensure that tasks meet their deadlines in the worst-case, in hard real-time systems, tasks are scheduled with respect to their worst-case execution times. Since the worst-case execution time is an upper bound on a job's execution time, most jobs complete earlier than their

scheduled completion times. Early completing jobs leave behind unused resources, because resources are allocated to the jobs for the duration of their worst-case execution times. Effective use of resources left unused is crucial for good system performance, because most of the jobs in real-time systems execute for much less than their worst-case execution times. We proposed and evaluated three methods for making use of the resources left unused by early job completions. It was found that retaining unguaranteed jobs in the system and advancing the start-times of previously-scheduled jobs considerably improve system performance by utilizing the resources left unused due to early job completions. Subcube selection heuristics proposed in Chapter IV along with another subcube selection heuristic, proposed in this chapter LST, were compared under the assumption that jobs may complete earlier than their scheduled completion times. LST was found to perform significantly better than EST and Stacking, both of which had nearly identical performance. In general, performance was found to be less sensitive to the subcube selection heuristic because the new framework made it possible to reuse the holes effectively.

Priority constraints are the second type of workload complexity addressed in this chapter. Our approach to priority handling is confined to the job ordering component; subcube identification and subcube selection are not affected. We showed that priority constraints can be gracefully handled with the proposed Priority Ordering policy. We also explored another approach that searches for Intermediate Ordering between the extremes of strict Priority Ordering and No-Priority Ordering, in order to help mitigate the negative effect of priorities on low-priority jobs. We found Intermediate Ordering to improve the performance of low-priority jobs marginally without affecting the performance of high-priority jobs.

CHAPTER VI

Mesh-Connected Systems

6.1 Introduction

Due to its simplicity, regularity and suitability for VLSI implementation [87, 98], the mesh topology for distributed-memory multiprocessors has drawn considerable attention in recent years. In this chapter, we apply the proposed approaches for on-line scheduling of hard real-time parallel tasks to mesh-connected architectures and focus on issues specific to mesh-connected systems.

In previous chapters, while dealing with hypercube systems, we assumed that tasks always request complete subcubes, and therefore focused on processor allocation strategies that allocate complete subcubes. However, in most mesh-connected systems, typical workloads consist of requests for submeshes of arbitrary size. Therefore, in this chapter, we consider processor allocation strategies that can allocate submeshes of arbitrary size. We follow the same two-phase scheduling approach and problem decomposition strategy described in Chapter IV. The problem of on-line scheduling of hard real-time parallel tasks on mesh-connected architectures is decomposed into three components: *submesh identification*, *submesh selection* and *job ordering*. The roles of the three components are identical to those described for subcube identification, subcube selection and job ordering, respec-

tively, in Chapter IV. The submesh identification component is responsible for identifying all *suitable* submeshes of the requested size. A suitable submesh is defined as a submesh that is big enough in each dimension to execute the requesting job and has enough free time to complete the job before its deadline. The submesh selection component is responsible for selecting a submesh to execute the job from the set of suitable submeshes identified by the submesh identification component. The job ordering component is used in the second phase of the two-phase scheduling approach to determine the order in which jobs should be considered for scheduling.

Like the subcube identification problem, the submesh identification problem is closely related to the processor allocation problem. Therefore, we start with a discussion of processor allocation strategies for mesh-connected systems. Several processor allocation strategies have been proposed in recent years for mesh-connected systems, but no thorough study has been conducted to compare the performance of these algorithms. We begin by comparing these strategies for non-real-time environments, then discuss their extension in the time dimension, so they can be used for submesh identification in hard real-time environments. In Section 6.2 we describe and compare the processor allocation strategies proposed for mesh-connected systems in recent years. Section 6.3 discusses the extension of these strategies in the time dimension to use them for submesh identification in hard real-time environments. A discussion of job ordering heuristics appears in Section 6.4. The results of a performance comparison study of the heuristics proposed for submesh identification, submesh selection and job ordering are presented in Section 6.5. Finally the chapter concludes with a summary and conclusions drawn from the research presented in this chapter.

6.2 Processor Allocation Algorithms

In general-purpose systems, processor allocation strategies are responsible for finding a currently available submesh of a given size. Since dynamic allocation and deallocation of submeshes can lead to processor fragmentation, minimizing processor fragmentation is also a goal of the processor allocation algorithms. Several processor allocation algorithms have been proposed for mesh-connected systems, but these algorithms have not been compared under a single set of assumptions. In this section, we present an overview of these algorithms and compare their performance for general-purpose systems.

In recent years several processor allocation strategies have been proposed for mesh-connected systems. These strategies vary greatly in their complexity and ability to recognize available submeshes. In this chapter, we compare the performance of eight processor allocation strategies including the 2-Dimensional Buddy System [72], Buddy [61], Frame Sliding [20], Adaptive Scan [27], First-Fit and Best-Fit [128], and the Adjacency strategy [105]. In addition we propose a modified Adjacency strategy having reduced overhead. All these strategies were originally proposed with the assumption of First-Come-First-Served (FCFS) job scheduling. In this chapter, we also propose alternative versions of these strategies that make use of Scan scheduling [62]. Scan is found to significantly improve performance for all the processor allocation strategies.

The algorithms are compared for a wide range of workloads and system sizes under the umbrella of a single set of performance metrics. Because of the large differences in their complexities, this study also compares the processor allocation/deallocation overheads of these algorithms, and examines their effects on performance.

6.2.1 Overview of the Algorithms

Although many of the algorithms considered in this chapter can be extended to higher dimension meshes, we focus our attention on two-dimensional meshes. All the algorithms assume there is a host processor connected to the mesh system. This host processor is responsible for submesh allocation, submesh deallocation and task dispatching.

A two-dimensional mesh $M(a, b)$ consists of $a \times b$ nodes arranged in a $a \times b$ rectangular grid. A node in a mesh system refers to a processor. Each node can be identified by its coordinates $\langle x, y \rangle$, where x refers to its row position and y denotes its column position. Figure 35 shows an $M(8,5)$ mesh and the addresses of its nodes. A submesh in $M(a, b)$, denoted as $S(w, h)$, is a subgrid of $M(a, b)$ such that $1 \leq w \leq a$ and $1 \leq h \leq b$.

Figure 35: A 2D Mesh of Size 8×5

Since complete descriptions of the processor allocation strategies compared in this study is beyond the scope of this chapter, we briefly overview the strategies, highlighting their important characteristics and inherent limitations.

Two Dimensional Buddy Strategy

The Two-Dimensional Buddy System, a generalization of the one-dimensional binary buddy system (1DBS), was proposed by Li and Cheng [72, 73] for processor allocation in two-dimensional mesh-connected multiprocessors. 2DBS is usable only in square mesh systems of size $(p \times p)$, where $p = 2^n$ for integer n . It can allocate only square submeshes with side lengths restricted to exact powers of two. The sizes of requested submeshes are 'rounded up' (or 'dilated') to the nearest power of 2. For a request of arbitrarily size $w \times h$, the 2DBS algorithm would allocate a submesh of size $2^k \times 2^k$, where $k = \lceil \log_2(\max(w, h)) \rceil$. For example, a submesh $S(8, 8)$ is allocated to a request that actually needs a submesh $S(1, 5)$. This leads to a large number of processors (about 92% in this example) being wasted due to internal fragmentation. Expected internal fragmentation for the 2DBS is 42% [72], when the side-lengths of requests are drawn from a uniform distribution.

Under 2DBS each processor has a unique address (x, y) , where $0 \leq x, y \leq 2^n - 1$. Submeshes are specified as $S(x, y, k)$, where (x, y) is the address of the lowest leftmost processor in the submesh (base), and 2^k is its size, $0 \leq k \leq n$. Li and Cheng formally define an allocatable submesh (or *block*) as follows: (1) $S(0, 0, R)$ is a block; and (2) if $S(x, y, k)$ is a block and $k > 0$ then $S(x, y, k - 1)$, $S(x + 2^{k-1}, y, k - 1)$, $S(x, y + 2^{k-1}, k - 1)$, and $S(x + 2^{k-1}, y + 2^{k-1}, k - 1)$ are also blocks, and are *buddies* of each other.

For a mesh system of size $W \times H$, where $W = H = 2^n$, 2DBS maintains a set of $n + 1$ *free block lists* (FBLs). Let FBL_i be a doubly linked list of all available blocks of size 2^i , $0 \leq i \leq n$. In addition, a status array S of length $l = \sum_{i=0}^n 4^i$ is used to keep the status of each possible block. The status of a block consists of two fields, a bit b and a pointer ptr .

The bit b indicates whether the corresponding block is available. The pointer ptr points to the corresponding block in the free list if the block is available, otherwise it is *nil*. With the help of these two data structures, determining the existence of available buddies, and inserting/removing blocks from FBLs can be done in constant time. To allocate a submesh $S(2^k, 2^k)$, list FBL_k is examined. If it is non-empty, the first entry from the list is removed and the corresponding submesh is allocated to the requesting task. If list FBL_k is empty, but some list $FBL_j, k < j \leq n$, is non-empty, the first entry from the list FBL_j is removed, the corresponding submesh is split into four submeshes of size $2^k \times 2^k$ and three submeshes of each size $2^{k+i} \times 2^{k+i}, 1 \leq i \leq j - k$. One of the $2^k \times 2^k$ submeshes is allocated to the requesting tasks, while an entry for each of the remaining submeshes is inserted in the appropriate list and the status array is updated. Since at the most n such splits are required, the worst case time complexity of allocation is $O(n)$ or $O(\log N)$, where $N = W \times H = 2^{2n}$ is the number of nodes in the mesh system.

The deallocation procedure coalesces a released submesh with its buddies, if they are all currently available, to form a bigger submesh. The entries for the coalesced submeshes are removed from the FBLs and a new entry is made for the newly formed submesh in the appropriate FBL. This procedure is repeated for the newly formed submesh until no more coalescing is possible. The status array is properly adjusted at each step. In the worst case, n coalescing operations are necessary, so the deallocation procedure also has worst case time complexity $O(\log N)$.

Buddy Strategy

Krueger [61] proposed that by imposing a suitable linear ordering on the processors of a mesh-connected system, the Buddy strategy can be used for processor allocation in mesh systems. Under this scheme an *allocatable* submesh of size k within a mesh having 2^n linearly ordered processors is defined as the set of 2^k processors in the range $\#[j2^k, (j+1)2^k - 1]$, for integer j , $0 \leq j \leq 2^{n-k} - 1$. Like 2DBS, requested submesh sizes are rounded up to the size of the smallest allocatable submesh that can contain the requested submesh, resulting in internal fragmentation.

Since the numbering scheme determines the shape of submeshes that are allocatable, it affects the level of internal fragmentation. The numbering scheme proposed by Krueger allows allocation not only of square submeshes, but also rectangular submeshes where both side lengths are power of 2 and the ratio of the two side lengths is two. Under the proposed numbering scheme, a 2-dimensional mesh containing 2^n processors (referred to as M_n), is numbered using the following matrix operations. For $n = 1, 3, 5, \dots$ matrix M_n is defined recursively:

$$M_n = \begin{bmatrix} M_{n-1} & B_{n-1} + M_{n-1} \end{bmatrix} \quad (6.1)$$

Similarly, for $n = 2, 4, 6, \dots$

$$M_n = \begin{bmatrix} M_{n-1} \\ B_{n-1} + M_{n-1} \end{bmatrix} \quad (6.2)$$

$M_0 = \begin{bmatrix} 0 \end{bmatrix}$. In each case, B_n has the same dimension as M_n , and:

$$B_n = [b_{ij}], \text{ where } b_{ij} = 2^n, \forall i, j \quad (6.3)$$

With each increment in n , a mesh alternates the direction of its growth between its x and y axes. For example, since $B_0 = \begin{bmatrix} 0 \end{bmatrix}$, $M_1 = \begin{bmatrix} 0 & 1 \end{bmatrix}$. Continuing, $B_1 = \begin{bmatrix} 2 & 2 \end{bmatrix}$, so:

$$M_2 = \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix} \quad (6.4)$$

Continuing further:

$$M_3 = \begin{bmatrix} 0 & 1 & 4 & 5 \\ 2 & 3 & 6 & 7 \end{bmatrix} \quad (6.5)$$

$$M_4 = \begin{bmatrix} 0 & 1 & 4 & 5 \\ 2 & 3 & 6 & 7 \\ 8 & 9 & 12 & 13 \\ 10 & 11 & 14 & 15 \end{bmatrix} \quad (6.6)$$

It should be noted that the submeshes of mesh M_n , ranging in size from 0 to $n - 1$, are either square or rectangular (with width vs. height ratio = 2) in shape.

The Buddy allocation strategy is more flexible in the shape of submeshes it can allocate than 2DBS. 2DBS rounds up the sizes of the requested submeshes so that the side lengths are equal and an exact power of two. Therefore, the total number of processors to be allocated is a power of 4. The Buddy strategy, on the other hand, can allocate rectangular submeshes, so it requires the total number of processors to be a power of 2. Consequently, the Buddy strategy substantially reduces internal fragmentation over 2DBS. Expected internal fragmentation for the Buddy Strategy is only 25% when the side-lengths of requests are drawn from a uniform distribution [72]. Implementation of the Buddy strategy [62] is similar to that of 2DBS, and results in the same worst case time complexity $O(\log N)$, where N is the number of nodes in the mesh system, for processor allocation as well as deallocation.

Frame Sliding Strategy

The Frame Sliding (FS) strategy, proposed by Chuang and Tzeng [20], is applicable to meshes with arbitrary sizes and shapes. FS allocates a submesh of the requested size precisely, which eliminates internal fragmentation. For an incoming request $S(w, h)$, FS considers a *frame* of size $(w \times h)$. A frame is simply a submesh of the requested size and is identified by its lower left corner, or *base*. A frame is said to be *available* if all the processors in the frame are free, otherwise it is *unavailable*. A base is called *free* (*busy*) if its frame is *free* (*busy*).

To fulfill an incoming request $S(w, h)$, the FS strategy searches for an available frame of the requested size. The search starts from the lowest leftmost available node. If processors in the currently examined frame are not all available, the frame is slid over the “plane” of the mesh system to the next candidate frame, which is either w nodes away in the x -dimension or h nodes away in the y -dimension, depending on the current position of the frame. The frame first slides along the x -dimension in strides of w from left to right, until the frame exceeds the boundary of the mesh system. At that point a vertical slide with a stride of h takes place. The right side of the frame is aligned with the right boundary of the mesh system and further sliding takes place, again, in the horizontal direction, but from right to left. In general, the search process is carried out by examining a sequence of frames which altogether cover the entire mesh “plane”. During the search, if an available frame is found, it is assigned to the requesting task and search stops. Otherwise, the search continues until all the candidate frames have been examined.

To check if all the processors in a frame are free, the FS strategy simply checks for an

overlap between the address of the frame and the address of any already allocated submesh. The addresses of all the allocated (*busy*) submeshes are stored in a list. This check with respect to already allocated submeshes can be done in time linear to the number of allocated submeshes. The allocation step takes $O((N \times N_P)/(w \times h))$ time, where N is the number of nodes in the mesh system and N_P is the number of allocated submeshes, which is same as the number of currently running tasks in the system. In the worst case, N_P is $O(N)$. The deallocation steps involves removing the deallocated submesh from the *busy set*, the set of currently allocated submeshes. If a pointer to the corresponding entry in the busy set is passed to the allocated submesh at the time of allocation, it can pass the pointer back to the host processor at the time of deallocation and deallocation can be done in constant time. It should be noted that although the FS strategy is better than Buddy and 2DBS at finding a submesh for an incoming request, it is not *recognition complete*. Because of the fixed vertical and horizontal strides, FS may not be able to find a free submesh even though one is available [27].

Adaptive Scan Strategy

The Adaptive Scan (AS) strategy was proposed by Ding and Bhuyan [27] as an improvement over the Frame Sliding strategy. The AS strategy is also applicable to mesh systems of all sizes and shapes and it eliminates internal fragmentation by allocating submeshes of exactly the requested sizes. Like the FS strategy, AS searches for a frame having all its processors available. However, instead of using fixed strides of w and h , it uses a fixed vertical stride of 1 and an adaptive stride for the horizontal direction (x -dimension). The search starts at the lower left corner of the mesh system. If the current node can not serve as a base

node, it must belong to one or more busy or *coverage* submeshes. The *coverage* of a busy submesh with respect to an incoming task T is a submesh none of whose processors can serve as the base of any free submesh to accommodate the task T . Let $\{(x_i, y_i, x'_i, y'_i)\}$ be the addresses of all the busy and coverage submeshes to which the current node belongs, where $\langle x_i, y_i \rangle$ indicates the lower left corner of a submesh and $\langle x'_i, y'_i \rangle$ indicates the upper right corner. Let x_{max} be the maximal value of the x_i 's in these submeshes. $x_{max} + 1$ will be the next node to be considered as a candidate base. If the scan fails to find a base node in the current row, it moves to the left corner of the next row and the search continues in a left to right direction.

If no free frame is found to satisfy the current request, the orientation of the required frame is rotated by 90 degrees i.e. the side lengths are switched and the search procedure is repeated for the new frame. Since the algorithm can adjust the orientation of the frame and the stride during the search, it is called the Adaptive Scan strategy. The allocation procedure has worst case complexity $O(N \times N_P)$, where N is the number of nodes in the mesh system and N_P is the number of allocated submeshes. Submesh deallocation can be done in constant time using the same technique explained for the Frame Sliding strategy. Adaptive Scan is recognition complete, since it not only searches for the requested frame $S(w, h)$ but also for the rotated frame $S'(h, w)$, and always finds an available frame if one exists.

First-Fit and Best-Fit Strategies

The First-Fit (FF) and Best-Fit (BF) strategies proposed by Yahui Zhu [128] are applicable to mesh systems of arbitrary sizes and shapes. Like the Frame Sliding and Adaptive Scan

strategies, FF and BF are able to allocate submeshes of the requested sizes precisely, which eliminates internal fragmentation.

The idea behind FF and BF is to check for all possible frames of the requested size without checking all the processors in the mesh system. Two binary (0/1) arrays are used to speed the search process. The first array, called the *busy* array, is used to store the allocation state of the mesh, with a 1(0) value indicating that the corresponding processor is busy (idle). The second array, called the *coverage* array, is computed from the busy array with respect to an incoming request. The coverage array is simply the union of coverages of all the busy submeshes in the system. As defined earlier, the *coverage* of a busy submesh with respect to an incoming task T is a submesh none of whose processors can serve as the base of any free submesh to accommodate the task T . In the *coverage array*, each element represents the state of the base of a frame, and a value 1(0) indicates that the base is busy (free). Once the coverage array is constructed, the processor allocation problem reduces to finding a 0 entry, i.e. a free base in the coverage array. In order to reduce external fragmentation, the coverage array is searched in a certain order, say from left to right column and from top to bottom within each column, so that corner areas of free regions are picked for processor allocation. True to their names, FF stops its search as soon as it finds a free base (0 entry) in the coverage array, while BF searches the entire coverage array in order to find the “best fit” region for the current request.

The BF strategy tries to choose a *corner* from the smallest 0 (free) region, i.e. the best fit frame for the current request. A corner is defined as an element that serves as the ends of consecutive sequences of zero's in both the row and the column it is located in. The lengths

of the row and column zero sequences are called the *row length* and *column length*, respectively, of the corner. Multiplying the row length and the column length gives the *area* of the corner. The BF strategy determines the number of busy neighbors for each corner. A boundary element is considered as having a “busy neighbor” beyond the boundary. A corner with four busy neighbors is called a “perfect fit”. The frame corresponding to the first perfect fit corner encountered by BF is assigned to the requesting task. If no perfect fit corner is found, the frame corresponding to the corner having the highest number of busy neighbors and smallest area is allocated to the current request.

The FF and BF strategies are not recognition-complete because if they fail to honor the original request for submesh $S(w, h)$, they will not attempt to allocate a rotated submesh $S'(h, w)$. Both the FF and BF strategies have worst-case time complexity of $\Theta(N)$ for processor allocation as well as deallocation.

Adjacency Strategies

Sharma and Pradhan proposed a strategy [105] which we refer to as the Adjacency (ADJ) strategy. To allocate a submesh, ADJ considers only submeshes residing on the periphery of submeshes already allocated to other tasks, along with any free meshes at the four corners of the mesh system. Since all the free submeshes in the system are eventually incident either on an allocated submesh or on a corner of the mesh system, ADJ will always find a free submesh if it exists, despite the fact that it does not examine all possible submeshes in the system. ADJ allocates the free submesh having the highest *boundary value* to the requesting task. The boundary value of a free node $\langle x, y \rangle$ is the sum of the number of allocated neighbors of $\langle x, y \rangle$ and the number of mesh systems' boundary points on which it lies.

The boundary value of a free submesh is the sum of the boundary values of all the nodes on the periphery of the submesh.

The ADJ strategy further prunes the search space by not considering all of the submeshes on the periphery of allocated submeshes. It is sufficient to consider the candidate submeshes along the four corners of each allocated submesh, sliding the candidate submesh along allocated submesh's boundary in case the candidate submesh overlaps with some other allocated submesh. However, the submeshes at the corners of the submesh system *do not* slide in case of an overlap, they are rejected immediately. ADJ is recognition complete. The worst-case processor allocation complexity for ADJ is $O(N_p^3)$. Processor deallocation, as for FS and AS, takes constant time.

The ADJ strategy as proposed by Sharma and Pradhan is a *best-fit* strategy, since it looks for the submesh having the highest boundary value among the set of possible free submeshes. We also consider a *first-fit* variant of this strategy, which we refer to as ADJFF. Under ADJFF we first look at the candidate submeshes along the four corners of the mesh system and then consider the candidate submeshes along the periphery of already allocated submeshes. The search stops as soon as a free submesh is found. ADJFF is also recognition complete and has same worst-case time complexities as ADJ for processor allocation and deallocation.

Table 4 presents a concise summary of the salient characteristics of the algorithms discussed in this section. In Table 4, N refers to the number of processors (nodes) in the mesh system and N_p refers to the number of jobs present in the system.

Table 4: Summary of the Processor Allocation Strategies

Strategy	Allocation	Deallocation	Internal Fragmentation	Recognition Complete
2DBS	$O(\log N)$	$O(\log N)$	YES	NO
Buddy	$O(\log N)$	$O(\log N)$	YES	NO
FS	$O(N \times N_P)$	$\Theta(1)$	NO	NO
AS	$O(N \times N_P)$	$\Theta(1)$	NO	YES
FF	$\Theta(N)$	$\Theta(N)$	NO	NO
BF	$\Theta(N)$	$\Theta(N)$	NO	NO
ADJ	$O(N_P^3)$	$\Theta(1)$	NO	YES
ADJFF	$O(N_P^3)$	$\Theta(1)$	NO	YES

Job Scheduling

Job scheduling is an important component of processor management in multiprocessors. For hypercube systems, Krueger, Lai and Radiya [62] showed that the Scan scheduling algorithm results in considerably better performance than other disciplines, including First-Come-First-Served, Last-Come-First-Served, Shortest-Job-First and Smallest-Job-First. For meshes, Krueger [61] showed that for the Buddy and 2DBS allocation strategies, Scan job scheduling results in significantly better performance than disciplines such as First-Come-First-Served or Largest-Job-First. In this section, we show how Scan scheduling can be used with other mesh processor allocation strategies as well, with similar improvements in performance.

The order in which jobs are scheduled can have a considerable effect on external fragmentation. The Scan scheduling discipline exploits this potential, reducing external fragmentation by grouping together jobs requiring equal-sized submeshes. When used with Buddy and 2DBS, which restrict submesh sizes to exact powers of two, Scan maintains

$\log_2 N + 1$ separate queues, one for each possible size for submesh requests. On arrival, a new job joins the end of the queue corresponding to the size of the submesh it requires. All jobs of a given size (including those that arrive before the list empties) are allocated before the scheduler moves on to the next queue. Possible alternatives for Scan, based on the direction of scanning, include *ScanUp* and *ScanDown*. When queue i is empty, *ScanUp* continues with queue $i + 1$ modulo $\log_2 N + 1$, while *ScanDown* continues with queue $i - 1$ modulo $\log_2 N + 1$. For both versions of Scan, enqueueing a newly arrived job requires constant time. In the worst-case, it may be necessary to search all the queues to find the next job to be executed, so the time complexity of Scan is $O(\log N)$.

Because the other processor allocation strategies we consider can allocate submeshes of arbitrary size, the number of possible sizes is potentially very large. Consequently, it is not efficient to maintain a separate queue for each possible job size. Instead, Scan when used with these strategies, maintains only a single ordered queue and a moving pointer. The pointer always points to the job to be scheduled next, if any jobs are in the queue. When a job is scheduled, the pointer is moved to the next job in the queue, if there is one. Otherwise the pointer is set to point to the beginning of the queue. Newly arrived jobs are placed in the queue according to the queue ordering discipline, irrespective of the position of the pointer. For *ScanUp*, jobs are ordered from smallest to largest number of processors required, while for *ScanDown* jobs are ordered from largest to smallest. Finding a new job to be executed next takes constant time. However, the worst-case complexity of inserting a newly-arrived job in the queue is $O(p)$, where p is the number of jobs in the queue.

Because we have found *ScanUp* to perform slightly better than *ScanDown* in all cases,

we use it exclusively in this chapter, referring to it simply as Scan.

6.2.2 Performance Evaluation

We take a user-oriented perspective on performance, concentrating on mean job response time as a performance index. The response time of a job is defined as the time that elapses from the moment the job arrives to the system until it completes execution. Mean job response time is the sum of the mean job queuing delay and the mean time that a job holds a submesh during its execution. We do not consider performance indices such as system utilization and fragmentation, since a user is unlikely to be concerned about those factors beyond their effect on job response time.

For this study, our assumptions about the job arrival process are same as described in Chapter III. We also assume that submesh hold times are exponentially distributed (for convenience, and without loss of generality, we assume a mean of 1 time unit), and independent of submesh request sizes. It is assumed that jobs provide no deterministic a priori information regarding their execution times. We consider only square mesh-systems for this study, because some of the algorithms are not applicable to non-square mesh systems.

Little information is available regarding submesh request sizes in typical workloads of mesh-connected systems. For this study we consider four representative distributions of submesh sizes: uniform, geometric, uniform-increasing and uniform-decreasing. The two side lengths of the submesh are chosen independently from the specified distribution. The maximum possible size of a submesh is equal to the size of the mesh system. For the geometric distribution, we assume a mean equal to one-fourth the mesh size. The uniform-increasing and uniform-decreasing distributions are uniform distributions in interval. Within

the interval, side-lengths are uniformly distributed. If W is the side-length of the mesh system (note that we are considering only square mesh systems) and $P_{[a,b]}$ denotes the probability of a side-length being in the interval a to b , for the uniform-increasing distribution:

$$P_{[1, \frac{W}{8}]} = 0.2, P_{[\frac{W}{8}+1, \frac{W}{4}]} = 0.2, P_{[\frac{W}{4}+1, \frac{W}{2}]} = 0.2, \text{ and } P_{[\frac{W}{2}+1, W]} = 0.4,$$

and for the uniform-decreasing distribution:

$$P_{[1, \frac{W}{8}]} = 0.4, P_{[\frac{W}{8}+1, \frac{W}{4}]} = 0.2, P_{[\frac{W}{4}+1, \frac{W}{2}]} = 0.2, \text{ and } P_{[\frac{W}{2}+1, W]} = 0.2.$$

Most previous studies [20, 27, 61, 72, 128], have assumed processor allocation and deallocation overhead to be negligible. However, due to the large differences in the time complexities of the algorithms compared in this study, it is important that overheads be considered. We measured the processor allocation and deallocation times for each strategy on a Sun SPARCstation SLC running SunOS 4.0.3. We realize these results are implementation-dependent, but we believe the trends shown by these results indicate inherent characteristics of the algorithms. Since processor allocation/deallocation and task dispatching is generally done by a separate host processor attached to the mesh-connected system, the allocation/deallocation overheads affect only the job whose request is being serviced by the host processor and the jobs waiting to be scheduled, if there are any. The significance of these overheads depends on the ratio of overhead to the execution time of a job. To study the effects of overhead, we consider a wide range of ratios.

Due to the analytical intractability of this system in general case, we rely on discrete-event simulation for this study. The lengths of the simulation runs were chosen to be sufficiently long (typically 100,000 time units, where the mean submesh hold time is 1 time unit) to reduce the effect of start-up conditions to negligible levels, allowing steady-state

measurements. We report the system as unstable if no steady state results exist. A system is said to be *unstable* if it is unable to perform work at the rate it arrives, leading to job queues that grow without bound and an infinite mean response time. All results reported in this section have confidence interval of 5% or less at 90% confidence level. The method of independent replication [64] was used to achieve this level of precision.

Results

Comparison of Processor Allocation Strategies

Since these algorithms were originally presented under the assumption of First-Come-First-Served job scheduling, we assume the same in this section. We consider alternative job scheduling discipline in the following section.

Figure 36: Mean Job Response Time/Mean Allocation Overhead vs. Offered System Load

To reduce the size of the set of processor allocation strategies we consider, we begin with a pairwise comparison of similar algorithms. Figure 36(a) plots mean job response

time against the offered system load, assuming negligible overhead. To examine overhead, Figure 36(b) plots mean submesh allocation overhead against the offered system load. Both plots assume a uniform distribution for the side-lengths of submesh requests and an overall mesh system of size 64×64 . Note that Figure 36(b) uses a logarithmic scale for the Y-axis. Starting with ADJ and ADJFF, we note that in Figure 36(a) these algorithms have almost identical performance, but Figure 36(b) shows that ADJFF has about half the overhead of ADJ. Continuing with First-Fit and Best-Fit, the two strategies have almost identical mean job response time but Best-Fit has much higher overhead than First-Fit. In addition to the results presented, these trends have been found to hold for mesh systems having a wide range of sizes and for a wide range of distributions for the side-lengths of submesh requests. In light of these results, we will not consider the ADJ and Best-Fit algorithms in the remainder of our comparisons, focusing instead on ADJFF and First-Fit.

Figure 37: Mean Job Response Time vs. Offered System Load

To compare the remaining algorithms, we begin by comparing their performance under

increasing offered system load. Figure 37 plots mean job response time against offered system load, assuming uniformly distributed side-lengths for submesh requests (Figure 37(a)), and geometrically distributed side-lengths (Figure 37(b)). To illustrate the inherent limitations of each strategy, we again assume negligible overhead for these figures. The relative overheads of these strategies and their effects on performance will be examined shortly. Beginning with Figure 37(a) we see that among the algorithms, 2DBS performs most poorly, with the system becoming unstable before the offered system load reaches 0.25. At the other end of the spectrum, ADJFF has the best performance, with the system remaining stable up to an offered system load of 0.52. Among the other strategies, Adaptive Scan comes closest to ADJFF in performance. Surprisingly, the First-Fit strategy is not much better than the Frame Sliding strategy.

Continuing with Figure 37(b), we consider a workload having geometrically distributed request side-lengths. The picture presented by Figure 37(b) is significantly different from that of Figure 37(a). All the algorithms perform more poorly here than under a uniform distribution. Adaptive Scan is affected least by this change in workload and has performance almost identical to that of the ADJFF strategy. Clearly, this workload is more challenging than the uniformly distributed workload and reveals the deficiencies of the processor allocation strategies. Though they are not shown, the uniform-increasing distribution shows trends similar to those in Figure 37(a), and the uniform-decreasing distribution shows trends similar to those in Figure 37(b).

Another significant aspect of the performance of processor allocation strategies is sensitivity to the size of the mesh system. Figure 38 plots mean job response time against the size

Figure 38: Mean Job Response Time vs. Size of the Mesh System

of the mesh system, assuming an offered system load of 0.3 and again assuming negligible allocation overhead. Figure 38(a) assumes that the side-lengths of the requested submeshes follow a uniform distribution, while a geometric distribution is assumed for Figure 38(b). The performance of ADJFF and Adaptive Scan is little affected by changes in the size of the mesh system. Buddy and Frame Sliding, however, show significant drops in performance with increasingly large systems. Once again, the geometric distribution presents a more challenging workload and helps differentiate the performance of the algorithms. The performance of 2DBS is so poor that it is beyond the bounds of these plots.

Next, we compare the overhead of submesh allocation and deallocation under these strategies. Figure 39(a) plots the mean submesh allocation time against offered system load, assuming a mesh system of size 64×64 , while Figure 39(b) plots the mean submesh allocation time against the size of the mesh system, assuming an offered system of load of 0.3. Both the plots assume a uniform distribution for the side-lengths of requested submeshes, and both the plots use a logarithmic scale for the Y-axis.

Figure 39: Mean Submesh Allocation Time vs. Offered System Load/System Size

Clearly, the First-Fit strategy has *much* higher overhead than any of the other strategies under all levels of offered system load and for all sizes of mesh systems. Adaptive Scan has the second highest overhead, though it is much smaller than the First-Fit's overhead. However, under Adaptive Scan, overhead increases with increasing offered system load, as well as with increasing size of the mesh system. Buddy has the smallest overhead, and is little affected by changes in the offered load or the size of the mesh system. The ADJFF and Frame Sliding strategies have almost identical overhead for a mesh system of size 64×64 as shown by Figure 39(a), but for Frame Sliding (Figure 39(b)), the overhead increases rapidly with increases in the size of the mesh system, whereas ADJFF shows only a minor increase.

Despite the large differences in the worst-case time complexity of processor deallocation for the different strategies, the measured deallocation overheads did not show differences comparable to the differences in allocation overheads. These results are not included here, but appear in [5].

We next consider the effect of processor allocation/deallocation overhead on performance.

Figure 40: Mean Job Response Time vs. Cost Per Operation/Offered System Load

As discussed earlier, this effect depends on the ratio of overhead to the execution time of a job. To simulate processor allocation/deallocation overhead in a way that is independent of the machine and the execution environment under which the simulation is running, we take the following approach: We calculate the overhead based on the number of significant operations involved in a particular instance of allocation or deallocation, and multiply that count by the mean cost per operation, which is a parameter to the simulation. Significant operations include splitting a submesh into smaller pieces, inserting items in an ordered list, checking for overlap between two submeshes, or merging a submesh with its buddies. Trivial operations such as checking for a flag or testing a condition, are not included in the count. While differences exist between the actual overheads incurred by different significant operations, the overhead calculated in this fashion were found to be proportionate to the actual measured times, within reasonable bounds. By varying the mean cost per operation, we are

able to vary the allocation/deallocation overhead and study the performance of processor allocation strategies under varying ratios of overhead to mean job execution time.

Figure 40(a) plots mean job response times for each algorithm against the mean cost per operation, assuming a 64×64 mesh system and an offered system load of 0.3. The mean cost per operation is relative to the mean execution time of the jobs. Note that this figure has a logarithmic scale for the X-axis. Not surprisingly, the high overhead of First-Fit causes the system to become unstable even when the cost per operation is small. The low overhead of Buddy-Scan, on the other hand, allows the system to remain stable even when the mean cost per operation is very high. ADJFF also fares well. These results indicate that Buddy and ADJFF will perform significantly better than all other strategies if the overhead to execution time ratio is large.

Figure 40(b) plots mean job response time against offered system load, taking into account the overhead of processor allocation and deallocation. For Figure 40(b) the cost per operation is assumed to be 0.0001 time units, and the side-lengths of the requested sub-meshes are chosen from a uniform distribution. Comparing Figure 40(b) with Figure 37(a), the biggest loss in performance occurs for First-Fit, which now performs even more poorly than 2DBS.

Comparison of Job Scheduling Policies

In this section we compare the performance of the Scan and FCFS job scheduling disciplines, when used with each of the processor allocation strategies. Figures 41 and 42 plot mean job response time against offered system load, for various algorithm pairs. These graphs assume a mesh system of size 64×64 and ignore the processor allocation/deallocation

overhead. Figure 41 assumes a uniform distribution for the side-lengths of requested sub-meshes, while Figure 42 assumes a geometric distribution. For each allocation strategy, the FCFS version is depicted by a Solid line, while the Scan version is depicted by a dotted line. These figures show that replacing FCFS with Scan scheduling results in a significant performance improvements for *all* the processor allocation strategies. Under uniform workload (Figure 41), Buddy shows the most improvement among all the strategies, while ADJFF shows the most improvement under geometric workload (Figure 42).

Figure 41: Mean Job Response Time vs. Offered System Load

6.2.3 Conclusions

In this section, we have compared the performance of several processor allocation strategies proposed for general-purpose mesh-connected systems for a wide range of workloads and system sizes under the umbrella of a single set of performance metrics. This study in-

Figure 42: Mean Job Response Time vs. Offered System Load

cludes all processor allocation algorithms known to the author and takes processor allocation/deallocation overhead into account. We also proposed and evaluated a first-fit variant of the Adjacency strategy, as well as new versions of the strategies using an alternative job scheduling discipline – Scan.

Among processor allocation strategies, we found the Adjacency strategies to have better performance than all the other strategies under all the workload conditions considered. If jobs are short, Buddy which has a very small processor allocation/deallocation overhead, was found to perform very well. For job scheduling, we found that Scan scheduling improves the performance significantly over FCFS, irrespective of the processor allocation strategy being used.

In comparing their first-fit and best-fit variants, neither the Best-Fit strategy nor the standard Adjacency strategy show significant improvement in the performance over their first-fit counterparts, though they have almost twice the processor allocation overhead. While this

result does not suggest that better best-fit heuristics are not possible, it suggests that first-fit strategies should be given increased attention.

The performance improvements observed when each of the processor allocation strategy was used with the Scan job scheduling discipline suggest that job scheduling policies can have a significant effect on performance. Future research should investigate alternative job scheduling policies, including policies designed for use with specific processor allocation algorithms.

6.3 Submesh Identification and Submesh Selection

The role of the submesh identification component in the scheduling process is similar to that explained for the subcube identification component in Section 4.3. In hard real-time environments where the scheduler maintains a work-plan, the submesh identification component's responsibility is to identify submeshes of the desired size that are either currently available or will have enough free time in the future to finish the requesting job before its deadline. In this section, we describe extensions of the processor allocation strategies discussed in the previous section in the time dimension, so they can be used for submesh identification in hard real-time environments.

For mesh-connected systems, the submesh identification component is closely linked with the submesh selection component. Consequently, we also discuss the submesh selection component and heuristics suitable for it in this section. The relationship between the two components is made clear in the description of the procedure followed to search for a suitable submesh for a requesting job.

6.3.1 Search Strategy

The procedure followed to search for a suitable submesh in a given schedule is quite different than that explained for subcube identification in Section 4.3. Under the workload model assumed for hypercube systems, only perfect subcubes are requested by jobs. Assuming the Buddy strategy for subcube identification, we examine the schedule of each candidate subcube to determine if the processors of that subcube have enough available time to complete the job before its deadline.

However, in mesh-connected systems, since submeshes of arbitrary size may be requested, it is inefficient to consider submeshes of the desired size and for each candidate submesh determine if it can complete the requesting job before its deadline. Instead, the approach we adopt is to search for a time duration, equal to the worst-case execution time of the job to be scheduled, before the deadline of the job, in which *some* suitable submesh can execute the job. Each possible duration to be considered is referred to as a *candidate* time duration. The maximum number of candidate time durations is one more than the number of jobs currently present in the system. The submesh selection component is responsible for preparing the list of candidate time durations. This list is referred to as the *Time-List*. Each entry in the Time-List represents the start-time of a candidate time duration; the length of each candidate time duration is same as the worst-case execution time of the job being scheduled. The submesh identification component considers candidate time durations from the Time-List in the order in which they appear in the list, to determine if a suitable submesh can be found to execute the requesting job in a candidate time duration. The candidate time durations are considered in order until a suitable submesh is found or all the entries in the Time-List have

been considered.

6.3.2 Submesh Selection

The submesh selection component selects the time at which a job starts executing by deciding the order in which candidate time durations should be considered. Under the search strategy described above, two intuitive heuristics for submesh selection are EST and LST. As described in Chapter IV and V, EST starts executing the requesting job as early as possible, while LST elects to postpone the execution of a job as much as possible without violating the timing constraints of the job. Heuristics such as Stacking, as described for hypercube systems (Section 4.5), are not applicable for the submesh selection component, because under the proposed framework submesh selection does not select the submesh to execute the job; it just determines the candidate time durations and the order in which they should be considered for scheduling the requesting job. The concept similar to the Stacking algorithm is used by the Best-Fit heuristic for the submesh identification component. However, as explained later in this section, the Best-Fit heuristic does not perform well as compared to some other heuristics used for submesh identification. The procedure followed for preparing the list of candidate time durations is described next.

Preparing the list of Candidate Time Durations

To explain the procedure for preparing the Time-List – the list of candidate time durations we assume that EST is used for submesh selection. Modifications to this procedure for the LST heuristic are discussed later in this section.

To prepare the Time-List, the submesh identification component makes use of the fact that the earliest possible start-time for a job should be either the current time or a time when some previously-scheduled job is scheduled to complete. Considering this fact, the Time-List is nothing but the list of completion times of all jobs present in the system, ordered in increasing time order. If the completion times of two jobs are identical, there is only one entry for them in the Time-List. Current Time is added as a first entry to the Time-List and all those entries which are higher than the Latest Start Time (as described in Section 2.1) of the job being scheduled, are removed from the Time-List.

It should be noted that while preparing this list, no attention is paid to the size of the jobs or the set of processors on which jobs are scheduled; only the completion times of jobs are considered.

To better understand the process of preparing the Time-List, let us consider the following example: Suppose jobs J_1 to J_6 are currently scheduled with their start-times and completion times as shown in Figure 43. Job J_n , with worst-case execution time of 2 time units and deadline 10.0, is to be added to the current schedule. Assuming that the current time is 0.0 and EST is used for submesh selection, the Time-List is shown in Table 5. The Time-List is an ordered list of job completion times without the entries that are higher than the Latest Start Time of the J_n and with current time added as the first entry.

The procedure followed for preparing the Time-List, when the LST heuristic is used for submesh selection, is the opposite of that described for EST. If the execution of job J_n should be started as late as possible without violating its timing constraints, the deadline of job J_n , and the start-times of previously scheduled jobs represent possible completion times for J_n .

Figure 43: A Sample Schedule

Job	Completion Time	Time-List
J_1	4	0
J_2	9	3
J_3	3	4
J_4	10	7
J_5	12	
J_6	7	

Table 5: Time-List

The submesh selection component prepares a list of start-times of currently present jobs in the system, ordered in decreasing time order. All entries in the list that are higher than the deadline of J_n are removed from the list, because they can not be the possible completion times for J_n . The deadline of J_n is added to the beginning of the list. Now, each entry in the list represents the completion time of a candidate time duration. Since the Time-List is the list of start-times of candidate time durations, J_n 's worst-case execution time is subtracted from each entry in the list. The resulting list is the desired Time-List.

Job	Start Time	Time-List
J_5	8	8
J_4	6	6
J_2	5	4
J_6	2	3
J_3	1	0
J_1	0	

Table 6: Job Start Times

For the job set presented in Figure 43, the list of job start times and the Time-List, assuming the LST heuristic, are shown in Table 6.

The submesh selection component involves sorting the start-times or completion times of all jobs present in the system in order to prepare the Time-List. Therefore, the worst-case time complexity of submesh identification component is $O(m \log m)$, where m is the number of jobs present in the system.

6.3.3 Submesh Identification

To find a suitable submesh for the requesting job, the submesh identification component makes use of the Time-List prepared by the submesh selection component. Each entry in the Time-List is the start-time of a candidate time duration. Starting at the top of the list, candidate time durations are considered in order to determine if the requesting job can be scheduled in that time duration. Schedulability of the newly-arrived job in a candidate time duration is determined with the help of the processor allocation strategy.

Given a candidate time duration, the submesh identification component prepares a set,

(referred to as *Job-Set*), of all previously-scheduled jobs that are scheduled to execute during that time duration on any set of processors. Then, the processor allocation strategy is invoked to determine if the newly-arrived job can be scheduled on some submesh without any conflict with any of the jobs in the *Job-Set*. The processor allocation strategy determines if there be any available submesh to execute the new job after each job in the *Job-Set* has been allocated the set of processors on which it is scheduled. If such a submesh is found, it is allocated to the new job. The processor allocation strategy does not pay attention to the start-times and completion-times of jobs in the *Job-Set*, it only considers the sets of processors on which the jobs are scheduled.

If the processor allocation strategy fails to allocate a submesh to the requesting job, the next candidate time duration from the *Time-List* is considered. The search stops if a suitable submesh is found or there are no more entries in the *Time-List*.

The overhead of submesh identification depends on the processor allocation strategy used and the number of jobs present in the system. The number of jobs present in the system determines the maximum number of candidate time durations for scheduling the requesting job. For each candidate time duration considered, the processor allocation strategy is invoked once. Therefore, assuming there are m jobs present in the system, in worst case the processor allocation strategy will be invoked $m + 1$ times. If we assume the Adjacency strategy, which has worst-case time-complexity of $O(m^3)$, is used for processor allocation, the worst-case time complexity of submesh identification is $O(m^4)$. The Adjacency strategy is used to compute the time complexity of the submesh identification component, because it was found to have the best performance among the processor allocation strategies compared

in Section 6.2.2.

All proposed submesh identification heuristics use the same search procedure and, depending on the submesh selection heuristic, search for either the earliest or latest time duration for the execution of the requesting job. However, the submesh identification heuristics differ in the processor allocation strategy they use. We consider eight submesh identification heuristics based on the eight processor allocation strategies discussed in Section 6.2. We refer to the submesh identification heuristics by the same name as the processor allocation strategy they use. Therefore, the eight submesh identification heuristics considered are: 2DBS, Buddy, Frame Sliding, Adaptive Scan, First-Fit, Best-Fit, Adjacency, and ADJFF. The results of the performance comparison of these heuristics are discussed in Section 6.5.

6.4 Job Ordering

As described in Chapter IV, the job ordering component determines the order in which jobs are scheduled in the second phase of the two-phase scheduling approach. Job ordering heuristics considered for mesh-connected systems are the same as those considered for job ordering in hypercube systems – First-Come-First-Served (FCFS), Largest-Dimension-First (LDF), Earliest-Deadline-First (EDF), and Minimum-Laxity-First (MLF). These heuristics are described in detail in Section 4.4.

6.5 Performance Comparison

In this section, we present the results of a performance comparison of heuristics for the three components of the scheduling process discussed in the previous sections. Assumptions about the job arrival process, distributions of laxities, worst-case execution times and

actual execution times are as described in Sections 3.4 and 5.2.4. As described in Section 6.2.2, four representative distributions of submesh sizes: uniform, geometric, uniform-increasing and uniform-decreasing were considered. The two side lengths of a submesh are chosen independently from the specified distribution. The maximum possible size of a side-length of a requested submesh is half the side-length of the mesh system. This assumption was made in order to make the results comparable to the results for hypercube systems, which assumed that the maximum dimension of a requested subcube is one less than the dimension of the hypercube system. For the geometric distribution, we assume a mean equal to one-fourth the mesh size. We assume a mesh system of size 64×64 for all the results presented in this section. Job Miss Ratio% and Work Miss Ratio% (as described in Section 3.4) are used as performance indices. Since trends observed for the two indices are similar, we present Work Miss Ratio% plots only.

The results of the comparison of job ordering heuristics are nearly identical to those presented for hypercube systems in Section 4.7. Therefore, they are not repeated here. MLF and EDF were found to perform significantly better than FCFS and LDF under all the workload conditions considered. MLF was slightly better than EDF under a few workload conditions. For all the results presented in this section, MLF is used as the job ordering policy. Also, we display results only for a uniform distribution of side-lengths of submesh requests, because the trends are similar under other distributions.

Results

The 2DBS and Buddy strategies were dropped from the comparison because their performance was significantly worse than the other strategies. As was shown in Figure 39, First-

Fit (FF) and Best-Fit (BF) strategies have much larger scheduling overheads than other algorithms. If scheduling overheads are taken into account Figure 40 showed that, the performance of FF and BF is significantly worse than all other algorithms considered. Therefore, FF and BF are also dropped from the comparison. Submesh identification heuristics included in the comparison are the Adjacency strategy, ADJFF, Adaptive Scan and Frame Sliding.

Submesh Identification

We begin by discussing the results of a comparison of submesh identification heuristics against offered system load. These results are presented in Figure 44. A mean laxity of 150% of the mean worst-case execution time and an Early Completion Ratio of 0.1 are assumed for these results. For low offered loads, all the heuristics perform well. The Work Miss Ratio% is almost zero for all the algorithms, except Frame Sliding, up to an offered load of 0.5. Performance degrades sharply beyond loads of 0.7. Frame Sliding performs considerably worse than all the other algorithms being compared. ADJ performs slightly better than ADJFF and significantly better than Adaptive Scan.

Next, we compare the performance of the submesh identification heuristics against mean laxity. Results of the comparison are presented in Figure 45. An offered system load of 0.4 and an Early Completion Ratio of 0.1 are assumed for these results. It should be noted that a logarithmic scale is used for both the axes. Laxities are denoted as a ratio of the mean worst-case execution time. Not surprisingly, all the heuristics perform poorly when laxities are small and perform very well when laxities are sufficiently large. Frame Sliding, again, performs considerably worse than the other algorithms. The Adjacency strategy performs

Figure 44: Mean Job Response Time vs. Offered System Load

slightly better than ADJFF and significantly better than Adaptive Scan. For Adjacency, ADJFF and Adaptive Scan, the Work Miss ratio% is negligible when mean job laxity is more than 150% of the mean worst-case execution time.

Figure 46 compares the performance of the submesh identification heuristics against Early Completion Ratio. An offered system load of 0.4 and a mean laxity of 150% of the mean worst-case execution time are assumed for the results presented in this figure. The relative performance of the submesh identification heuristics is similar to that observed in Figures 44 and 45. Frame Sliding is substantially worse than the other three heuristics. The Adjacency strategy is marginally better than ADJFF but significantly better than Adaptive Scan. Performance of all the algorithms degrades with increasing Early Completion Ratio. These results are consistent with the results presented in Figure 28 for hypercube systems.

Submesh Selection

Two heuristics were compared for submesh selection – Earliest Start Time (EST) and

Figure 45: Mean Job Response Time vs. Mean Laxity%

Latest Start Time (LST). For the comparison of submesh selection heuristics, ADJFF was used for submesh identification and MLF was used for job ordering.

In Figure 47, we present the results of a comparison of the submesh selection heuristics against offered system load. A mean laxity of 150% of the mean worst-case execution time and an Early Completion Ratio of 0.1 are assumed for all the results presented in this figure. For offered system loads less than 0.5, both the heuristics have nearly identical performance. Beyond 0.5, EST performs significantly better than LST. The performance of both the algorithms degrades sharply beyond loads of 0.7.

Next, the two submesh selection heuristics are compared against the mean laxity. The results of this comparison are presented in Figure 48. An offered system load of 0.4 and an Early Completion Ratio of 0.1 are assumed for this figure. A logarithmic scale is used for the X-axis. Laxities are denoted as a ratio of the mean worst-case execution time. Both heuristics perform poorly when job laxities are small and perform well when laxities are

Figure 46: Mean Job Response Time vs. Early Completion Ratio

sufficiently large. When mean laxity is more than 150% of the mean worst-case execution time, the Work Miss Ratio% is negligible under both the heuristics. Overall, EST performs slightly better than LST, even when mean laxity is very small.

Figure 49 presents the results of a comparison of the submesh selection heuristics against Early Completion Ratio. For the results presented in this figure, we assume an offered system load of 0.4 and a mean laxity of 150% of the mean worst-case execution time. The performance of both the heuristics degrades with increases in the Early Completion Ratio. However, the performance degradation is sharper for LST than for EST. Therefore, the difference in the performance of the two heuristics increases with increases in the Early Completion Ratio. These trends are consistent with the trends observed for hypercube systems.

Based on the results discussed in this section, we conclude that the Adjacency strategy has the best performance among the heuristics considered for submesh identification, and EST performs better than LST for submesh selection. Contrary to the results observed for general-purpose systems, where the Adjacency strategy and ADJFF had nearly identical

Figure 47: Mean Job Response Time vs. Offered System Load

performance, the Adjacency strategy performs slightly better than its first-fit counterpart, ADJFF, in hard real-time environments.

6.6 Summary and Conclusions

In this chapter, we presented extensions of the proposed framework for on-line scheduling of hard real-time parallel tasks to mesh-connected systems. Since, typical workloads for mesh-connected systems consists of requests for submeshes of arbitrary size, we considered processor allocation strategies that can allocate submeshes of arbitrary size. A performance comparison study was conducted to compare the performance of these processor allocation strategies under a variety of workload conditions for general-purpose systems. The Adjacency strategy and its first-fit counterpart ADJFF were found to perform significantly better than the other strategies under all workload conditions considered.

We proposed extensions of these processor allocation strategies into the time dimension

Figure 48: Mean Job Response Time vs. Mean Laxity%

so they can work as submesh identification components in hard real-time environments. The submesh identification components based on the Adjacency strategy and its first-fit counterpart ADJFF were found to perform significantly better than the other heuristics under all workload conditions considered. The trends observed in the comparison of submesh identification components were similar to those found in the comparison of processor allocation strategies for general-purpose systems. The only exception being, the Adjacency strategy and ADJFF had nearly identical performance in general-purpose systems, but in hard real-time environment submesh identification heuristic based on the Adjacency strategy performed slightly better than the heuristic based on ADJFF.

EST and LST were the two heuristics considered for submesh selection. EST was found to perform better than LST under all workload conditions considered.

The relative performance of job ordering heuristics was nearly identical to that observed for hypercube systems. EDF and MLF were found to perform considerably better than LDF

Figure 49: Mean Job Response Time vs. Early Completion Ratio

and FCFS. MLF was slightly better than EDF under a few workload conditions.

In this chapter, we showed that the proposed framework for on-line scheduling of hard real-time parallel tasks, can be extended to mesh-connected systems with little modification. We believe that extension to other distributed memory architectures is also possible with little or no modification to the framework.

CHAPTER VII

Summary and Future Research

7.1 Summary

On-line scheduling of parallel hard real-time tasks on multiprocessor systems is a difficult problem that has received little attention from researchers. In the past, most real-time scheduling efforts have been devoted to static scheduling, which requires that the arrival times and other attributes of all tasks that may arrive at the system be known a priori. These algorithms have low run-time overheads, but fail to adapt to changes in the environment and are expensive to modify as a system evolves. Static algorithms are not particularly suitable for large systems due to the complexity involved in determining all possible sets of tasks that may be present in the system simultaneously. Dynamic scheduling algorithms overcome these limitations of static scheduling algorithms, though they have higher run-time overheads.

Since dynamic scheduling algorithms plan task schedules at run time, they can better respond to changes in the environment and better evolve with time. Modern-day real-time systems are becoming increasingly sophisticated and complex. They need more adaptive solutions than static algorithms can provide. Because of the need for adaptability and graceful handling of abnormal conditions in real-time applications such as multimedia applications, virtual reality systems, and space flight control missions, only dynamic scheduling al-

gorithms can meet the performance requirements of these systems. Therefore, we focused on dynamic scheduling algorithms in this thesis. The research presented in this thesis is the first work known to the author in the area of on-line scheduling of parallel tasks in hard real-time environments.

The research presented in this thesis, is targeted for distributed memory multiprocessors, including hypercube systems, mesh-connected systems and tree-based architectures. Distributed memory multiprocessors do not have any shared memory and the only means of communication among processors is through message passing. By virtue of having no resources shared among all processors, distributed memory multiprocessors, do not have any inherent impediments to scalability. The potential for scalability is one of the main reasons distributed memory multiprocessors have enjoyed increased attention, especially in the past two decades.

Multiprocessors are expected to have tens of thousands of processors in the future. Since few jobs can make use of such a large number of processors simultaneously, computing resources may be shared among independent jobs that can run concurrently. Such environments are called *space-sharing* environments and the multiprocessors which support space-sharing environments are called *partitionable* multiprocessors. Many multiprocessors are being used in space-sharing environments. The scheduling framework and heuristics presented in this thesis are targeted for space-sharing environments on distributed memory multiprocessors, with tasks that have hard timing constraints.

Under the workload model assumed for this work, the number of processors required for each task is determined before the task arrives at the system. The task arrival process is

stochastic, i.e. the system has no a priori knowledge of tasks' arrival times or attributes, before they arrive at the system. Tasks that have unpredictable arrival times are also referred to as *sporadic tasks*. It is assumed that there is a host processor connected to the multiprocessor system, which is responsible for processor allocation/deallocation, schedule preparation and task dispatching.

We presented a framework for solution to the problem of on-line scheduling of hard real-time parallel tasks on partitionable multiprocessors. Under the proposed framework on-line guarantees are provided, i.e. tasks are either guaranteed to complete before their deadlines or rejected at the time of their arrival. Since at the time of a task's arrival, its schedulability is known, there is ample opportunity to take alternative steps in case the task is not going to meet its deadline. For example, if an on-line guarantee algorithm running on a multiprocessor, which is part of a distributed system, finds that a newly-arrived task can not be guaranteed to complete before its deadline, it may be able to transfer the task, before it is too late, to some other node that can guarantee to complete the task before its deadline.

To improve the efficiency of scheduling, a two-phase scheduling approach was proposed, where the second phase is optional. In the first phase, the scheduler attempts to accommodate the newly-arrived job within the current schedule. If the scheduler is not able to do so and the task has more laxity than the worst-case execution time of the second phase, the second phase is initiated. The second phase consists of trying to find a feasible schedule by rescheduling previously-scheduled jobs. All as yet unfinished jobs present in the system, along with the newly-arrived job are placed in an ordered list and the scheduler, starting with an empty schedule, schedules the jobs one by one. If all jobs can be scheduled without

any violation of the timing-constraints, the scheduler has found a feasible schedule for the job set. If a feasible schedule is found, all currently executing jobs are preempted and the new schedule is adopted; otherwise the newly-arrived job is rejected.

To determine the right amount of preemption in the scheduling process, we compared the proposed preemptive two-phase scheduling approach with three alternative approaches (a) no second phase, (b) non-preemptive second phase, and (c) preparing a new schedule on every new job arrival. It was found that the system performance was significantly worse under the approaches with no second phase or a non-preemptive second phase as compared to the proposed preemptive two-phase approach. It was also shown that only marginal performance gains are achievable when a new schedule is constructed on every new job arrival, even if all scheduling overheads are ignored. When overheads of job preemptions and migrations are taken into account, performance is severely hurt if a new schedule is constructed on every new job arrival. From this study, we concluded that the proposed two-phase scheduling approach with a preemptive second phase has an appropriate level of preemption for good system performance.

The problem of scheduling a new job under a given schedule was decomposed into three components – *subunit identification*, *subunit selection* and *job ordering*. A subunit is defined as a part of the multiprocessor system that can be allocated to independent jobs. For example, subcube is a subunit of a hypercube systems and submesh is a subunit of a mesh-connected system.

The subunit identification component is responsible for identifying all *suitable* subunits present in the system. A suitable subunit is one that is large enough to execute the requesting

job and has enough free time to complete the job before its deadline. The problem of subunit identification is closely related to the problem of processor allocation in general-purpose systems. The processor allocation problem deals with finding a currently available subcube of a given size in a general-purpose (i.e. non-real-time) system, and doing so in a way that minimizes processor fragmentation.

Several processor allocation strategies have been proposed for hypercube and mesh-connected systems. Processor allocation strategies proposed for hypercube systems include, the Buddy strategy, the Single and Multiple Gray Code strategies and the MSS strategy. These strategies vary greatly in complexity and in their ability to recognize available subunits. The Multiple Gray Code and MSS strategies have perfect subcube recognition, i.e. if there is any available subcube in the system, they will find it. On the other hand, the Buddy strategy can not recognize all available subcubes, but incurs much smaller overheads than the Single and Multiple Gray Code and MSS strategies. Better subunit recognition ability does not necessarily result in better system performance. For general-purpose hypercube systems, Krueger et al. [62] showed that despite its poor subcube recognition ability, the Buddy strategy performs almost as well as the Multiple Gray Code and MSS strategies, even when overheads of the algorithms are not taken into account. To study the relationship between subcube recognition ability and performance in hard real-time systems, we compared the performance of subcube identification components based on the Buddy strategy, Single Gray Code strategy and Multiple Gray Code strategy in a hard real-time environment. An *Optimal strategy* that completely eliminates external fragmentation and has perfect subcube recognition ability was also included in the comparison. The Buddy strat-

egy, which has much smaller overheads compared to the Gray Code strategies, was found to perform as well as the Gray Code strategies even when scheduling overheads were not taken into account. Although, the optimal strategy performed considerably better than the other strategies, it is not a practical approach because of the large overheads involved in implementing this strategy.

For mesh-connected systems, several processor allocation strategies have been proposed in recent years, but no thorough study has been conducted to compare these algorithms under a single set of assumptions. We conducted a performance comparison study of processor allocation strategies, including 2DBS, strategy, the Buddy strategy, Frame-Sliding, Adaptive Scan, First-Fit & Best-Fit policies and the Adjacency strategy. We also proposed a first-fit variant of the Adjacency strategy and referred to it as ADJFF. The Adjacency strategy and ADJFF had nearly identical performance under all the workload conditions considered and ADJFF had half the overheads of the Adjacency strategy. The Adjacency strategy and ADJFF were found to perform considerably better than all other heuristics considered. We also considered an alternative job scheduling algorithm, Scan, for general-purpose mesh-connected systems. Scan was found to significantly improve system performance over FCFS under all processor allocation strategies.

The processor allocation strategies proposed for mesh-connected systems were extended into the time dimension, so they can be used as submesh identification components in hard real-time environments. A performance comparison of the submesh identification components showed that the submesh identification policy based on the Adjacency strategy performs slightly better than the submesh identification heuristic based on ADJFF, and signif-

icantly better than the submesh identification components based on the other processor allocation strategies.

The subunit selection component is responsible for selecting a subunit for allocation to the requesting job from the set of suitable subunits identified by the subunit identification component. For hypercube systems, we proposed and evaluated three subunit selection heuristics – *First-Suitable-Subcube* (FSS), the *Stacking algorithm*, and *Earliest-Start-Time* (EST). FSS simply selects the first suitable subcube identified by the subunit identification component, whereas EST selects the subcube that can start executing the requesting job earliest. The Stacking strategy attempts to minimize system fragmentation due to unnecessary subcube splits and coalesces by stacking equal-sized jobs in the time dimension. FSS requires constant time, while the worst-case time complexity of EST and Stacking is $O(N)$, where N is the number of processors in the hypercube system. FSS was found to have the worst performance among the three heuristics compared. EST and Stacking have nearly identical performance in several cases, while Stacking performs significantly better than EST when subcube sizes are geometrically distributed and jobs have small laxities or preemption-migration cost is high. For mesh-connected systems, the subunit selection component's responsibility is partly shared by the subunit identification component. Which subunit will be allocated to the requesting job is decided by the subunit identification component. However, the time at which the job starts executing is decided by the subunit selection component.

For the job ordering component, four heuristics were compared, First-Come-First-Served (FCFS), Largest-Dimension-First (LDF), Earliest-Deadline-First (EDF) and Minimum-Laxity-

First (MLF). FCFS orders jobs according to their time of arrival at the system, while under LDF, the job that requires the largest number of processors gets the highest priority. LDF, which is often used in bin-packing problems, is able to reduce processor fragmentation by reducing the number of holes in the schedule. However, it does not take into account the timing constraints of tasks. EDF and MLF are commonly-used heuristics for scheduling sequential real-time jobs on uniprocessor systems. EDF orders jobs in increasing order according to their deadlines, while MLF orders jobs in increasing order according to their laxities. EDF and MLF, both take timing constraints of tasks into account, however EDF considers only task deadlines, whereas MLF considers task execution times as well. The trends observed in the comparison of job ordering heuristics were nearly identical for hypercube systems and mesh-connected systems. Under both the cases, FCFS was found to perform considerably worse than the other three heuristics. LDF performed slightly better than FCFS but was significantly worse than EDF and MLF. EDF and MLF had nearly identical performance under almost all the workload conditions considered. Since MLF was found to be marginally better in certain cases, it was selected as the job ordering policy for our continuing work. This comparison of these heuristics shows that job ordering component has a significant effect on system's performance. The poor performance of LDF as compared to EDF and MLF shows that paying attention to timing constraints of jobs is more important than trying to reduce processor fragmentation by having a better packed schedule.

The proposed framework was extended to handle more complex workloads. Two types of workload complexities were considered. The first workload complexity considered is related to utilizing resources left unused by jobs that complete earlier than their scheduled

completion times. Since it is extremely difficult to determine the exact execution time of a job, jobs are scheduled with respect to their worst-case execution times in hard real-time systems, so they can be guaranteed to meet their deadlines even in the worst-case. The worst-case execution time of a job is an upper bound of a job's actual execution time, therefore most jobs complete earlier than their scheduled completion time. These early completing jobs leave behind some unused resources, because resources are allocated to jobs for the duration of their worst-case execution times. Resources left unused due to early job completions are like "holes" in a packed bin that appear after the bin has been packed. Considerable system resources may be wasted due to these holes in the packing if appropriate measures are not adopted to utilize these holes. We proposed and compared three methods for making use of the holes and found that retaining unguaranteed jobs in the system with a hope that the scheduler might be able to guarantee them later when additional resources become available due to early job completions, is very useful. The system's guarantee ratio can be improved significantly by scheduling unguaranteed jobs in the holes created by jobs completing earlier than their scheduled completion times. Also, transferring waiting jobs among the subunits to occupy the holes helps improve resource utilization and the system's guarantee ratio.

The second workload complexity considered was the presence of priority constraints among the jobs constituting the workload. We considered a two-level priority policy, where jobs are either high-priority jobs or low-priority jobs. The two-level policy can easily be extended to multi-level priority system. We showed that the proposed scheduling framework is suitable for handling priority constraints among tasks. For handling priority constraints,

the scheduler uses a Priority Ordering for scheduling the jobs. Priority Ordering schedules jobs in the order of their priorities. Within a priority level jobs are ordered using a secondary priority scheme, which was assumed to be MLF in our study. In order to accommodate newly-arrived high-priority jobs, it might be necessary to remove some previously-guaranteed low-priority jobs from the schedule. When a previously-guaranteed job is removed from the schedule, it is said that the guarantee of that job has been *revoked*. Guarantee revocation essentially breaks a promise and creates an environment of uncertainty for low-priority jobs. It is important that the Number of guarantee revocations be minimized. We proposed an approach that searches for Intermediate Orderings between the extremes of strict priority ordering and an ordering that completely ignores task priorities. It was found that the Intermediate Ordering can reduce the number of low-priority jobs that have their guarantees revoked without any negative effects on the performance of high-priority jobs.

The ability to utilize resources left idle due to holes in the schedule, makes another subcube selection algorithm feasible – Latest-Start-Time (LST). For hypercube systems, LST was compared with the other subcube selection heuristics, FSS, EST and Stacking in an environment where jobs may complete earlier than their scheduled completion times. LST was found to perform significantly better than the other heuristics, under most of the workload conditions considered. For mesh-connected systems, LST was compared with another submesh selection heuristic, EST. The results of a performance comparison of EST and LST were found to be in contrast with those for hypercube systems; EST performs significantly better than LST under most of the workload conditions.

In summary, we proposed a framework for on-line scheduling of hard real-time paral-

lel tasks on partitionable multiprocessors. A two-phase scheduling approach was presented to improve the efficiency of scheduling. The problem of scheduling a task under a given schedule was decomposed into three components and heuristics were proposed and compared for each of the components. Extensibility of the framework was shown by extending it to handle some complex workloads and by applying the proposed techniques to both hypercube systems and mesh-connected systems. Possible future extensions are suggested in the next section.

7.2 Future Research

Dynamic scheduling of real-time parallel tasks is a relatively new area of research. Few theoretical results or practical algorithms are available in this area. Many of the problems related to dynamic scheduling of real-time parallel tasks have been shown to be NP-hard. Therefore, we have to rely on heuristic-based approaches. In this thesis, we presented a framework and heuristics for on-line scheduling of hard real-time tasks.

The research presented in this thesis can be extended in several directions. The primary objective of these extensions should be to develop the current framework into a comprehensive scheduling approach that would (a) handle periodic and aperiodic tasks present in the same system, (b) handle preemptable and non-preemptable tasks present in the same system, (c) support multilevel priority policy to capture the criticalness of tasks, (d) handle precedence constraints of jobs, and (e) jointly schedule multiple resources in an environment where tasks share software and hardware resources. Each of these possible extensions are considered in detail in the remainder of this section. Most likely a suite of algorithms

will be required to provide this functionality, along with a policy to determine how these multiple algorithms should be used in order to adapt to the dynamic and evolving environments.

Under the proposed framework, periodic tasks are handled by treating each instance of a periodic task as a sporadic job. This scheme is simple and works, but it fails to take advantage of the predictable nature of periodic arrivals. Better techniques should be developed to make use of the additional information available about the arrival times and resource requirements of periodic tasks.

The proposed framework can be extended to handle *mixed* workloads, which consist of a mixture of soft real-time, hard real-time and non-real-time tasks. It is not hard to find systems where soft real-time tasks, hard real-time tasks and non-real-time tasks co-exist. The performance objectives for each class of tasks are different. The challenge is to meet the diverse performance objectives of each class of tasks present in the workload and, at the same time, meet overall system performance goals. One way to include hard real-time, soft real-time and non-real-time tasks in the same framework, is to treat each task's value to the system as a function of time. The value of a hard real-time task remains constant till its deadline but drops to zero immediately after the deadline, whereas the value of a soft real-time task is constant till its deadlines, but decreases gradually as the time progresses. The non-real-time tasks may need a fast response time, but their value does not change suddenly as in case of hard and soft real-time tasks. The objective of the scheduling algorithms will be to schedule tasks so as to maximize the value to the system, keeping in mind the time-dependent variation in the value of tasks.

Various preemptive and non-preemptive algorithms have been developed for the different forms of the real-time scheduling problem, though no work has been done for systems where preemptable and non-preemptable tasks may co-exist. Non-preemptability of certain tasks may result from various reasons, including, to ensure the consistency of the shared data-structures, to avoid deadlocks among tasks sharing resources, etc. Handling a mixture of preemptable and non-preemptable tasks present in the same system is difficult, but an even more complex problem is presented by *semi-preemptable* tasks that are preemptable during some parts of their execution but non-preemptable in other parts. In fact, this kind of task is quite common. Many algorithms and operating system tasks are not preemptable in their “critical section” but they might be preempted in other parts of their execution. The research presented in this thesis can potentially be extended to meet the requirements of systems where preemptive, non-preemptive and semi-preemptive tasks co-exist. In this thesis, we discussed both preemptive and non-preemptive scheduling approaches; the two can be combined. In fact, a mixed approach can also be devised that will take into account the preemptability of a task at a particular instant of time.

In order to make sure that critical tasks meet their deadlines, priority constraints are introduced in the workload, where task priorities are based on the importance of tasks to the system. A two level priority mechanism is discussed in this thesis to handle the priority constraints of tasks. A more comprehensive approach may be developed to handle multilevel priority constraints.

In our framework we handled precedence constraints by appropriately adjusting the ready times of tasks. This technique is not efficient for environments where tasks complete earlier

than their scheduled completion times. Handling of precedence constraints in such environments has another problem: Before a job's start-time can be advanced to make use of the resources left unused due to early job completions, the scheduler must make sure that no precedence constraints will be violated by this start-time advancement. Better techniques can be developed to handle precedence constraints of tasks. With each job J_i , we may maintain a list of jobs that directly follow J_i in the precedence order. Whenever there is a change in the completion time of J_i , due to early completion or change in the start-time, perform appropriate changes to all the jobs affected by this change.

The current work can be extended in yet another direction, to the joint scheduling of multiple resources in environments where tasks share hardware or software resources. Two types of solutions have been proposed for the problem of resource sharing among tasks in real-time environments: (a) run-time allocation of resources, and (b) pre-allocation of resources at the time of scheduling. Under the first category of solutions, tasks are scheduled for CPU(s) only. For access to other resources, the tasks must compete with each other at run-time. Several approaches have been proposed [93, 103] for bounding resource wait times for tasks, based on task priorities. However, the upper bounds on the wait times could be large under such schemes. This type of solution to the resource sharing problem can not be used in systems that provide on-line guarantees. For such environments, an alternative solution is employed. Tasks are *jointly* scheduled for all the resources they need at the time of their arrival. Some work has been done in the area of joint scheduling of system resources such as CPUs, I/O channels, data-structures, communication media and secondary storage etc. For example, Stankovic et al. [95] have developed a non-preemptive scheduling algo-

rithm for joint scheduling of multiple resources using a heuristic-based search technique, however, other approaches specially using preemptive scheduling should be investigated.

In this thesis, it was assumed that the number of processors required for executing parallel tasks is fixed at the time of task's compilation. As discussed in Chapter II, it might be possible to compile tasks in such a way that the number of processors required for executing a task is variable within some upper and lower bounds. The execution time of the task is a function of the number of processors allocated to the task. At the time of scheduling, the scheduler can decide how many processors can be allocated to each task in order to minimize processor fragmentation and to maximize the number of tasks that meet their deadlines. The problem of on-line scheduling of hard real-time parallel tasks remains NP-hard even under this model [29, 59], however, the flexibility in job sizes provided by this model is a powerful tool to combat the problem of processor fragmentation. The proposed framework should be extended to take advantage of the flexibility in job sizes in order to minimize processor fragmentation and maximize the number of tasks that are able to meet their deadlines.

APPENDIX A

Scheduling of Independent Parallel Tasks with a Single Deadline

In this section, we address the problem of static scheduling of independent parallel real-time tasks on uniform multiprocessors. We prove that the problem of finding an optimal schedule is NP-hard even under the simplifying assumption that all the tasks have an identical release time and a common deadline. This problem is a simplified special case of a more general real-time scheduling problem where tasks may have different deadlines.

A.1 Problem Definition

Instance: A set of tasks $\tau = (t_1, t_2, \dots, t_n)$, each task t_i is characterized by release time r_i , deadline d_i , computation time c_i and number of processor required p_i and number of processors in the multiprocessor machine m . $r_i = 0, \forall i$ and $d_i = D \forall i$.

Question: Does there exist a schedule in which all tasks will complete before the deadline D .

We refer to this problem as *Parallel Tasks with Single Due Date* (PTSDD) problem.

A.2 Proof

We prove that the PTSDD problem is NP-Hard by reducing the well-known Partition problem to this problem.

Partition Problem:

Instance: A set of integers $S = (s_1, s_2, \dots, s_r)$.

Question: Does there exist a subset S' such that:

$$\sum_{i \in S'} s_i = \sum_{i \in S - S'} s_i$$

Construction: To construct an instance of the PTSDD from an instance of Partition problem, do the following. Set the number of processors required for task t_i equal to integer s_i , i.e. $p_i = s_i$, set the computation time of all the tasks to be equal to 1, i.e. $c_i = 1, \forall i$, set the common deadline $D = 2$ and set the number of processors $m = \lfloor \frac{\sum_{i \in S} s_i}{2} \rfloor$.

Proof: It can be easily seen that constructing an instance of the PTSDD from an instance of the Partition problem can be done in polynomial time. Now, we show that a solution for the instance Partition problem exists iff a solution for the constructed instance of PTSDD exists. There are two possible cases:

(1). $\sum_{i \in S} s_i$ **is odd**. This is a trivial case. Since $m < \frac{\sum_{i \in S} s_i}{2}$. Obviously, there can not exist any schedule such that all tasks can finish before deadline D . Also, since $\sum_{i \in S} s_i$ is odd, there can not exist any partition.

(2). $\sum_{i \in S} s_i$ **is even**. Since $\sum_{i \in S} s_i$ is even, $m = \frac{\sum_{i \in S} s_i}{2}$ i.e. $2 * m = 1 * (\sum_{i \in S} s_i)$.

(a). *Existence of a schedule for the PTSDD problem implies existence of partition.*

Since the product of total available time and number of processors is equal to the sum of products of the number of processors required by each task and its execution time, in any feasible schedule all processors will be busy at all points of time between 0 and 2. Therefore, the sum of sizes of tasks executing simultaneously at any moment, is equal to m . Also, the sum of sizes of tasks not executing at that moment should also be equal to m , because $2m = \sum_{i \in S} s_i$. Therefore, if a solution for the PTSDD problem exists, the set of tasks executing simultaneously at any moment will constitute one partition and rest of the tasks will constitute the other partition.

(b). *Existence of a partition implies existence of a schedule for the PTSDD problem.*

If a partition exists, a schedule for the instance of PTSDD problem can be constructed easily.

Since a partition exists, there exists a subset S' such that:

$$\sum_{i \in S'} s_i = \sum_{i \in S - S'} s_i = m$$

Since all tasks have unit execution times, tasks corresponding to S' can be executed from time 0 to 1, and remaining tasks can be executed from time 1 to 2.

Therefore, a solution exists for the instance of the PTSDD problem iff there exists a partition. Thus, the partition problem is polynomial reducible to the PTSDD problem. Since, the Partition problem is known to be NP-complete, the PTSDD problem is also NP-complete.

BIBLIOGRAPHY

- [1] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera Computer System. In *Proc. Int. Conf. on Supercomputing*, pages 1–6, 1990.
- [2] M. Annaratone, M. Fillo, K. Nakabayashi, and M. Viredaz. The K2 Parallel Processor: Architecture and Hardware Implementation. In *Proc. Int. Symposium. on Computer Architecture*, pages 92–101, May 1990.
- [3] D. Babbar and P. Krueger. Online Hard Real-Time Scheduling for Hypercube Multiprocessors. Technical Report OSU-CISRC-9/93-TR30, The Ohio State University, Dept. of Computer and Information Science, Sept. 1993.
- [4] D. Babbar and P. Krueger. On-Line Hard Real-Time Scheduling of Parallel Tasks on Partitionable Multiprocessors. In *Proc. Int. Conf. on Parallel Processing*, volume II, pages 29–38, Aug. 1994.
- [5] D. Babbar and P. Krueger. Performance Comparison of Processor Allocation Algorithms for Mesh-Connected Systems. Technical Report OSU-CISRC-1/94-TR1, The Ohio State University, Dept. of Computer and Information Science, Jan. 1994.
- [6] K. R. Baker and Z. S. Su. Scheduling with Due Dates and Early Start Times to Minimize Maximum Tardiness. *Naval Research Logistics Quarterly*, 21, 1974.
- [7] J. A. Bannister and K. S. Trivedi. Task Allocation in Fault Tolerant Distributed Systems. In *Acta Informatica*. Springer-Verlag, New York, 1983.
- [8] G. Barnes, R. Brown, M. Kato, D. Kuck, D. Slotnick, and R. Stokes. Illiac IV Computer. *IEEE Transactions on Computers*, C-27:84–87, Jan. 1968.
- [9] S. Baruah, G. Koren, D. Mao, B. Mishra, A. Raghunathan, L. Rosier, D. Shasha, and F. Wang. On the Competitiveness of On-Line Real-Time Task Scheduling. In *Proc. Real-Time Systems Symposium*, Dec. 1991.
- [10] J. Blazewicz, M. Drabowski, and J. Weglarz. Scheduling Multiprocessor Tasks to Minimize Schedule Length. *IEEE Transactions on Computers*, 35(5):389–393, May 1986.

- [11] P. Brately, M. Florian, and P. Robillard. Scheduling with Earliest Start and Due Date Constraints. *Naval Research Logistics Quarterly*, 18(4), Dec. 1971.
- [12] P. Brately, M. Florian, and P. Robillard. Scheduling with Earliest Start and Due Date Constraints on Multiple Machines. *Naval Research Logistics Quarterly*, 22, 1975.
- [13] H. Y. Chang and M. Livny. Scheduling Under Deadline Constraints. In *Proc. Real-Time Systems Symposium*, Dec. 1986.
- [14] G. I. Chen and T. H. Lai. Preemptive Scheduling of Independent Jobs On A Hypercube. *Information Processing Letters*, 28:201–206, 1988.
- [15] G. I. Chen and T. H. Lai. Scheduling Independent Jobs On Partitionable Hypercubes. *Journal of Parallel and Distributed Computing*, 12:74–78, 1991.
- [16] M. S. Chen and K. G. Shin. Embedment of Interacting Task Modules into a Hypercube Multiprocessor. In *Proc. of the Second Hypercube Conference*, pages 121–129, Oct. 1986.
- [17] M. S. Chen and K. G. Shin. Processor Allocation in an N-Cube Multiprocessor Using Gray Codes. *IEEE Transactions on Computers*, pages 1396–1407, Dec. 1987.
- [18] S. C. Cheng, J. A. Stankovic, and K. Ramamritham. Scheduling Algorithms for Hard Real-Time Systems: A Brief Survey. In J. A. Stankovic and K. Ramamritham, editors, *Hard Real-Time Systems*. IEEE Computer Society Press, 1988.
- [19] H. Chetto and M. Chetto. Some Results of the Earliest Deadline Scheduling Algorithm. *IEEE Transactions on Software Engineering*, 15(10):1261–1269, Oct. 1989.
- [20] P. J. Chuang and N. F. Tzeng. An Efficient Submesh Allocation Strategy for Mesh Computer Systems. In *Proc. Int. Conf. on Distributed Computing Systems*, pages 156–263, 1991.
- [21] P. J. Chuang and N. F. Tzeng. A Fast Recognition-Complete Processor Allocation Strategy for Hypercube Computers. *IEEE Transactions on Computers*, 41(4):467–479, Apr. 1992.
- [22] D. Craig and C. Woodside. Rejection Rate for Tasks with Random Arrivals, Deadlines, and Preemptive Scheduling. *IEEE Transactions on Software Engineering*, 16(10):1198–1208, Oct. 1990.
- [23] S. Davari and S. K. Dhall. An On-line Algorithm for Real-Time Task Allocation. In *Proc. Real-Time Systems Symposium*, Dec. 1986.

- [24] M. Dertouzos. Control Robotics: The Procedural Control of Physical Processes. In *Proc. IFIP Congress*, 1974.
- [25] M. L. Dertouzos and A. K. Mok. Multiprocessor On-Line Scheduling of Hard Real-Time Tasks. *IEEE Transactions on Software Engineering*, 15(12):1497–1506, Dec. 1989.
- [26] S. K. Dhall and C. L. Liu. On a Real-Time Scheduling Problem. *Operations Research*, 26(1), 1978.
- [27] J. Ding and L. N. Bhuyan. An Adaptive Submesh Allocation Strategy for Two-Dimensional Mesh Connected Systems. In *Proc. Int. Conf. on Parallel Processing*, pages 193–200, 1993.
- [28] F. Dougliis and J. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software - Practice and Experience*, 21(8):757–785, Aug. 1991.
- [29] J. Du and J.Y-T. Leung. Complexity of Scheduling Parallel Task Systems. *SIAM Journal of Disc. Math.*, 2(4):473–487, Nov. 1989.
- [30] G. Dudek, M. Jenkins, and D. Wilkes. A Taxonomy for Swarm Robots. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 441–447, 1993.
- [31] S. Dutt and J. P. Hayes. Subcube Allocation in Hypercube Computers. *IEEE Transactions on Computers*, C-40(3):341–352, Mar. 1991.
- [32] J. Erschler, G. Fontan, C. Merce, and F. Roubellat. A New Dominance Concept in Scheduling N Jobs on a Single Machine with Ready Times and Due Dates. *Operations Research*, 31(1), 1983.
- [33] M. Garey and D. Johnson. Complexity Results for Multiprocessor Scheduling Under Resource Constraints. *SIAM Journal on Computing*, 4:397–411, 1975.
- [34] M. Garey and D. Johnson. Scheduling Tasks with Nonuniform Deadlines on Two-Processors. *Journal of the ACM*, 23(3), July 1976.
- [35] M. Garey and D. Johnson. Two-Processor Scheduling with Start-Times and Deadlines. *SIAM Journal on Computing*, 6:416–426, 1977.
- [36] M. R. Garey and D. S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, 1979.

- [37] K. Ghosh, R. M. Fujimoto, and K. Schwan. A Testbed for Optimistic Execution of Real-Time Simulations. In *Proc. IEEE Workshop on Parallel and Distributed Real-Time Systems*, May 1993.
- [38] K. Ghosh, B. Mukerjee, and K. Schwan. A Survey of Real-Time Operating Systems. Technical Report GIT-CC-93/18, Georgia Institute of Technology, College of Computing, Feb. 1994.
- [39] M. J. Gonzalez. Deterministic Process Scheduling. *Computing Surveys*, 9, Sept. 1977.
- [40] R. L. Graham. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal of Applied Mathematics*, 17(2):416–429, Mar. 1969.
- [41] D. Gunsfield. Bonds for Naive Multiple Machine Scheduling with Release Times and Deadlines. *Journal of Algorithms*, 5, 1984.
- [42] W. A. Halang. Real-Time Systems: Another Perspective. *Journal of Systems and Software*, pages 101–108, Apr. 1992.
- [43] M. G. Harmon, T. P. Baker, and D. B. Whalley. A Retargetable Technique for Predicting Execution Time. In *Proc. Real-Time Systems Symposium*, pages 68–77, Dec. 1992.
- [44] J. R. Hartisa, M. Livny, and M. J. Carey. Earliest Deadline Scheduling for Real-Time Database Systems. In *Proc. Real-Time Systems Symposium*, Dec. 1991.
- [45] J. P. Hayes, T. Mudge, and Q. F. Stout. A Microprocessor-based Hypercube Supercomputer. *IEEE Micro*, Oct. 1986.
- [46] J. Hong, X. Tan, and D. Towsley. A Performance Analysis of Minimum Laxity and Earliest Deadline Scheduling in a Real-Time System. *IEEE Transactions on Computers*, Dec. 1989.
- [47] K. S. Hong and J. Y-T. Leung. On-line Scheduling of Real-Time Tasks. *IEEE Transactions on Computers*, Oct. 1992.
- [48] W. A. Horn. Some Simple Scheduling Algorithms. *Naval Research Logistics Quarterly*, 21, 1974.
- [49] Intel Corporation. *A Touchstone DELTA System Description*, 1991.
- [50] Intel Corporation. *Paragon XP/S Product Overview*, 1991.

- [51] K. Jeffay, D. Stanat, and C. Martel. On Non-Preemptive Scheduling of Periodic and Sporadic Tasks. In *Proc. Real-Time Systems Symposium*, pages 129–139, Dec. 1991.
- [52] E. G. Coffman. Jr. *Computer and Jobshop Scheduling Theory*. Wiley-Interscience, New York, 1976.
- [53] J. Kim, C. R. Das, and W. Lin. A Top-Down Processor Allocation Scheme for Hypercube Computers. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):20–30, Jan. 1991.
- [54] H. Kise. A Solvable Case of One-Machine Scheduling Problem with Ready and Due Times. *Operations Research*, 26(1), 1978.
- [55] L. Kleinrock. *Queueing Systems: Volume 1, Theory*. John Wiley & Sons, 1975.
- [56] L. Kleinrock. *Queueing Systems: Volume 1, Computer Applications*. John Wiley & Sons, New York, 1976.
- [57] K. C. Knowlton. A Fast Storage Allocator. *Communications of the ACM*, 8(10):623–625, Oct. 1965.
- [58] R. Krishnamurti and E. Ma. An Approximation Algorithm for Scheduling Tasks on Varying Partition Sizes in Partitionable Multiprocessor Systems. In *Proc. Int. Conf. on Parallel Processing*, 1991.
- [59] R. Krishnamurti and B. Narahari. Preemptive Scheduling of Independent Jobs on Partitionable Parallel Architectures. In *Proc. Int. Conf. on Parallel Processing*, pages 268–275, 1992.
- [60] P. Krueger. *Distributed Scheduling for Changing Environments*. PhD thesis, Department of Computer Science, University of Wisconsin, Madison, Wisconsin, 1988.
- [61] P. Krueger. Processor Allocation and Job Scheduling for Mesh-Connected Multiprocessors. Technical Report OSU-CISRC-10/93-TR34, The Ohio State University, Dept. of Computer and Information Science, Oct. 1993.
- [62] P. Krueger, T.H. Lai, and V. Radiya. Job Scheduling is More Important than Processor Allocation for Hypercube Computers. *IEEE Transactions on Parallel and Distributed Systems*, 5(5):488–497, May 1994.
- [63] T. Lang and E. B. Fernandez. Scheduling of Unit-Length Independent Tasks with Execution Constraints. *Information Processing Letters*, 4(4), 1976.

- [64] S. S. Lavenberg. *Computer System Performance Modeling Handbook*. Academic Press, 1983.
- [65] E. L. Lawler. Optimal Scheduling of a Single Machine Subject to Precedence Constraints. *Management Science*, 19, 1973.
- [66] E. L. Lawler. Recent Results in the Theory of Machine Scheduling. In A. Bachem et al., editors, *Mathematical Programming: the State of the Art*. Springer-Verlag, New York, 1983.
- [67] E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Recent Developments in Sequencing and Scheduling: A Survey. In *Proc. NATO Advanced Study and Research Institute on Theoretical Approaches to Scheduling Problems*, July 1981.
- [68] J. Lehoczky, B. Sprunt, and L. Sha. Aperiodic Task Scheduling for Hard Real-Time Systems. In *Proc. Real-Time Systems Symposium*, Dec. 1988.
- [69] J. P. Lehoczky and L. Sha. Performance of Bus Scheduling Algorithms. In *Performance* 86, 1986.
- [70] J. Y-T. Leung and J. Whitehead. On Complexity of Fixed Priority Scheduling of Periodic Real-Time Tasks. *Performance Evaluation*, 2(4):237–250, 1982.
- [71] S. T. Leutenegger and M. K. Vernon. The Performance of Multiprogrammed Multiprocessor Scheduling Policies. In *Proc. of the 1990 ACM SIGMETRICS Conf. on Measurement & Modeling of Computer Systems*, pages 226–236, May 1990.
- [72] K. Li and K. H. Cheng. A Two-Dimensional Buddy System for Dynamic Resource Allocation in a Partitionable Mesh Connected System. *Journal of Parallel and Distributed Computing*, pages 79–83, 1991.
- [73] K. Li and K. H. Cheng. Job Scheduling in a Partitionable Mesh Using a Two-Dimensional Buddy System Partitioning Scheme. *IEEE Transactions on Parallel and Distributed Systems*, pages 413–422, 1991.
- [74] W. Lin and C. L. Wu. A Distributed Resource Management Mechanism for a Partitionable Multiprocessor System. *IEEE Transactions on Computers*, 37(2):201–210, Feb. 1988.
- [75] C. W. Liu and J. W. Leyland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

- [76] J. Liu, K. Lin, and S. Natarajan. Scheduling Real-Time, Periodic Jobs Using Imprecise Results. In *Proc. Real-Time Systems Symposium*, pages 252–260, Dec. 1987.
- [77] C. D. Locke. *Best Effort Decision Making for Real-Time Scheduling*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, 1986.
- [78] S. Majumdar, D. L. Eager, and R. B. Bunt. Scheduling in Multiprogrammed Parallel Systems. In *Proc. of ACM SIGMETRICS 1988*, pages 04–113, May 1988.
- [79] C. Martel. Preemptive Scheduling with Release Times, Deadlines, and Due Times. *Journal of the ACM*, 29(3), 1982.
- [80] J. McDonald and K. Schwan. Ada Dynamic Load Control Mechanism for Distributed Embedded Battle Management Systems. In *Proc. First Workshop on Real-Time Applications*, pages 156–160, 1993.
- [81] G. McMahon and M. Florian. On Scheduling with Ready Time and Due Dates to Minimize Maximum Lateness. *Operations Research*, 23, 1975.
- [82] R. McNaughton. Scheduling with Deadlines and Loss Functions. *Management Science*, 6:1–12, 1959.
- [83] A. K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment*. PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1983.
- [84] A. K. Mok and M. L. Dertouzos. Multiprocessor Scheduling in a Hard Real-Time Environment. In *Proc. Seventh Texas Conf. on Computer Systems*, Nov. 1978.
- [85] J. M. Moore. An N Job, One Machine Sequencing Algorithms for Minimizing the Number of Late Jobs. *Management Science*, 15(1), 1968.
- [86] R. R. Muntz and E. G. Coffman. Optimal Preemptive Scheduling on Two-Processor Systems. *IEEE Transactions on Computers*, 18:1014–1020, 1969.
- [87] P. Muzumdar. Evaluation of On-Chip Static Interconnection Networks. *IEEE Transactions on Computers*, C-36:365–369, 1987.
- [88] NCUBE Corporation. *NCUBE/ten: An Overview*, Nov. 1985.
- [89] F. Panzieri and R. Davoli. Real-Time Systems: A Tutorial. Technical Report UBLCS-93-22, Lab. for Computer Science, University of Bologna, Bologna, Italy, Oct. 1993.

- [90] J. L. Peterson and T. A. Norman. Buddy Systems. *Communications of the ACM*, 20(6):421–431, Jun. 1977.
- [91] J. L. Peterson and T. A. Norman. Buddy Systems. *Communications of the ACM*, pages 421–431, June 1977.
- [92] P. Puschner and C. H. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Internation Journal of Time Critical Computer Systems*, 1(2), Sep. 1989.
- [93] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, Norwell Massachusetts, 1991.
- [94] K. Ramamritham and J. A. Stankovic. Dynamic Task Scheduling in Hard Real-Time Distributed Systems. *IEEE Software*, 1(3):65–75, July 1984.
- [95] K. Ramamritham, J. A. Stankovic, and P.F. Shiah. Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):184–194, Apr. 1990.
- [96] Sandra Ramos-Thuel and J. P. Lehoczky. An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed Priority Preemptive Systems. In *Proc. Real-Time Systems Symposium*, pages 110–123, Dec. 1992.
- [97] Sandra Ramos-Thuel and J. P. Lehoczky. On-line Scheduling of Hard Deadline Aperiodic Tasks in Fixed Priority Systems. In *Proc. Real-Time Systems Symposium*, Dec. 1993.
- [98] G. Randade and S. L. Johnson. The Communication Efficiency of Meshes, Boolean Cubes and Cube Connected Cycles of Wafer Scale Integration. In *Proc. Int. Conf. on Parallel Processing*, pages 467–482, 1987.
- [99] Y. Saad and M. H. Schultz. Topological Properties of Hypercubes. *IEEE Transactions on Computers*, C-37:867–872, Jul. 1988.
- [100] S. Sahni and Y. Cho. Nearly On Line Scheduling of a Uniform Processor System with Release Times. *SIAM Journal on Computing*, 8:275–285, 1979.
- [101] K. Schwan and H. Zhou. Dynamic Scheduling of Hard Real-Time Tasks and Real-Time Threads. *IEEE Transactions on Software Engineering*, 18(8):736–747, Aug. 1992.
- [102] C. L. Seitz. The Cosmic Cube. *Communications of the ACM*, 28(1):22–23, Jan. 1985.

- [103] L. Sha, J. P. Lehoczky, and R. Rajkumar. Solutions for Practical Problems in Prioritized Preemptive Scheduling. In *Proc. Real-Time Systems Symposium*, Dec. 1986.
- [104] L. Sha, R. Rajkumar, J. Lehoczky, and K. Ramamritham. Mode Change Protocols for Priority Driven Preemptive Scheduling. *Int. Journal of Time Critical Computer Systems*, 1(3), Dec. 1989.
- [105] D. Sharma and D. Pradhan. A Fast and Efficient Strategy for Submesh Allocation in Mesh-Connected Parallel Computers. In *Proc. Symposium. on Parallel and Distributed Processing*, pages 682–689, 1993.
- [106] C. Shen, K. Ramamritham, and J. A. Stankovic. Resource Reclaiming in Real-Time. In *Proc. Real-Time Systems Symposium*, pages 41–50, Dec. 1990.
- [107] Wei-Kuan Shih and Jane W. S. Liu. On-Line Scheduling of Imprecise Computations to Minimize Error. In *Proc. Real-Time Systems Symposium*, Dec. 1992.
- [108] N. G. Shivaratri and M. Singhal. A Transfer Policy for Global Scheduling Algorithms to Schedule Tasks With Deadlines. In *Proc. Int. Conf. on Distributed Computing Systems*, pages 248–255, May 1991.
- [109] B. Simons. A Fast Algorithm for Multiprocessor Scheduling. In *Proc. 21st Annual Symposium on Foundations of Computer Science*, 1980.
- [110] B. Simons. Multiprocessor Scheduling of Unit-Time Jobs with Arbitrary Release Times and Deadlines. *SIAM Journal on Computing*, 12(2), 1983.
- [111] B. Simons and M. Siper. On Scheduling Unit-Length jobs with Multiple Release Time/Deadline Intervals. *Operations Research*, 32(1), 1984.
- [112] B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic Task Scheduling for Hard Real-Time Systems. *Journal of Real-Time Systems*, 1:26–60, 1989.
- [113] J. A. Stankovic and K. Ramamritham. The Design of the Spring Kernel. In *Proc. Real-Time Systems Symposium*, Dec. 1987.
- [114] J. A. Stankovic and K. Ramamritham. What is Predictability for Real-Time Systems. *Journal of Real-Time Systems*, 2(12):247–254, Dec. 1990.
- [115] J. A. Stankovic, M. Spuri, M. Di, and G. Buttazzo. Implications of Classical Scheduling Results for Real-Time Systems. Technical Report UM-CS-1993-023, Univ. of Massachusetts, Dept. of Computer and Information Science, Mar. 1993.

- [116] D. B. Stewart and P. K. Khosla. Real-Time Scheduling of Sensor-Based Control Systems. In *Proc. IEEE Workshop on Real-Time Operating Systems and Software*, May 1991.
- [117] H. S. Stone. Multiprocessor Scheduling with the Aid of Network Flow Algorithms. *IEEE Transactions on Software Engineering*, pages 85–93, Jan. 1977.
- [118] A. D. Stoyenko. *A Real-Time Language with A Schedulability Analyzer*. PhD thesis, University of Toronto, Aug. 1987.
- [119] J. K. Strosnider. *Highly Responsive Real-Time Token Rings*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, 1988.
- [120] T. Teixeira. Static Priority Interrupt Scheduling. In *Proc. Seventh Texas Conf. on Computing Systems*, Nov. 1978.
- [121] P. Thambidurai and K. S. Trivedi. Transient Overloads in Fault-Tolerant Real-Time Systems. In *Proc. Real-Time Systems Symposium*, Dec. 1989.
- [122] Thinking Machines Corporation. *The Connection Machine CM-5 Technical Summary*, Oct. 1991.
- [123] J. Y. Tien, C. T. Ho, and W. P. Yang. Broadcasting on Incomplete Hypercubes. *IEEE Transactions on Computers*, 42(11), Nov. 1993.
- [124] J. D. Ullman. NP-Complete Scheduling Problems. *Journal of Computing System Science*, 10:384–393, 1975.
- [125] J. D. Ullman. Complexity of Sequence Problems. In E. G. Coffman, editor, *Computer and Job-Shop Scheduling Theory*. J. Wiley, New York, 1976.
- [126] Q. Yang and H. Wang. A New Graph Approach to Minimizing Processor Fragmentation in Hypercube Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 4(10), Oct. 1993.
- [127] W. Zhao, K. Ramamritham, and J. A. Stanckovic. Preemptive Scheduling Under Time and Resource Constraints. *IEEE Transactions on Computers*, 36(8):949–960, Aug. 1987.
- [128] Y. Zhu. Efficient Processor Allocation Strategies for Mesh-Connected Parallel Computers. *Journal of Parallel and Distributed Computing*, pages 328–337, Dec. 1992.

- [129] Y. Zhu and M. Ahuja. Preemptive Job Scheduling on a Hypercube. In *Proc. Int. Conf. on Parallel Processing*, pages 301–304, Aug. 1990.
- [130] G. Zlokapa, J. A. Stankovic, and K. Ramamritham. *Well-Timed Scheduling: A Framework for Dynamic Real-Time Scheduling*. 1991.