THE DEADLINE-BASED SCHEDULING OF DIVISIBLE REAL-TIME WORKLOADS ON MULTIPROCESSOR PLATFORMS

SURIAYATI BT CHUPRAT

UNIVERSITI TEKNOLOGI MALAYSIA

UNIVERSITI TEKNOLOGI MALAYSIA

DECLARATION OF THESIS / UNDERGRADUATE PROJECT PAPER AND COPYRIGHT

Author's full name : **SURIAYATI BT CHUPRAT**

Date of birth : **12 FEBRUARY 1973**

Title : **THE DEADLINE-BASED SCHEDULING OF DIVISIBLE REAL-TIME WORKLOADS ON MULTIPROCESSOR PLATFORMS**

Academic Session: **2008/2009**

I declare that this thesis is classified as :

☐	**CONFIDENTIAL**	(Contains confidential information under the Official Secret Act 1972)*
☐	**RESTRICTED**	(Contains restricted information as specified by the organization where research was done)*
☑	**OPEN ACCESS**	I agree that my thesis to be published as online open access (full text)

I acknowledged that Universiti Teknologi Malaysia reserves the right as follows:

1. The thesis is the property of Universiti Teknologi Malaysia.
2. The Library of Universiti Teknologi Malaysia has the right to make copies for the purpose of research only.
3. The Library has the right to make copies of the thesis for academic exchange.

SIGNATURE

730212-12-5994

(NEW IC NO. /PASSPORT NO.)

Date : **17 JUNE 2009**

Certified by :

SIGNATURE OF SUPERVISOR

PROF. DR. SHAHARUDDIN SALLEH

NAME OF SUPERVISOR

Date : **17 JUNE 2009**

NOTES : * If the thesis is CONFIDENTAL or RESTRICTED, please attach with the letter from the organization with period and reasons for confidentiality or restriction.

"We hereby declare that we have read this thesis and in our opinion this thesis is sufficient in terms of scope and quality for the award of the degree of Doctor of Philosophy (Mathematics)"

Signature : .....

Name of Supervisor I : Professor Dr Shaharuddin Salleh

Date : 17 June 2009

Signature : Sanjoy Baruah .....

Name of Supervisor II: Professor Dr Sanjoy K. Baruah

Date : 17 June 2009

BAHAGIAN A – Pengesahan Kerjasama*

Adalah disahkan bahawa projek penyelidikan tesis ini telah dilaksanakan melalui kerjasama antara _____ dengan _____

Disahkan oleh:

Tandatangan : ..... Tarikh : .....

Nama : .....

Jawatan : .....
(Cop rasmi)

* *Jika penyediaan tesis/projek melibatkan kerjasama.*

BAHAGIAN B – Untuk Kegunaan Pejabat Sekolah Pengajian Siswazah

Tesis ini telah diperiksa dan diakui oleh:

Nama dan Alamat Pemeriksa Luar : **Prof. Madya Dr. Mohamed Othman**
Fakulti Sains Komputer & Teknologi Maklumat,
Universiti Putra Malaysia
43400 UPM, Serdang
Selangor

Nama dan Alamat Pemeriksa Dalam I : **Prof. Dr. Safaai bin Deris**
Pengarah,
Pusat Teknologi Maklumat dan Komunikasi (CICT),
UTM, Skudai

Pemeriksa Dalam II : **Prof. Madya Dr. Ali Abd. Rahman**
Fakulti Sains,
UTM, Skudai.

Nama Penyelia I : **Prof. Dr.Shaharuddin bin Salleh**

Nama Penyelia II : **Prof. Dr. Sanjoy K. Baruah**

Disahkan oleh Timbalan Pendaftar di Sekolah Pengajian Siswazah:

Tandatangan : ..... Tarikh : .....

Nama : **EN KHASSIM B ISMAIL**

THE DEADLINE-BASED SCHEDULING OF DIVISIBLE REAL-TIME WORKLOADS ON MULTIPROCESSOR PLATFORMS

SURIAYATI BT CHUPRAT

A thesis submitted in fulfilment of the
requirements for the award of the degree of
Doctor of Philosophy (Mathematics)

Faculty of Science
Universiti Teknologi Malaysia

JUNE 2009

I declare that this thesis entitled "*The Deadline-Based Scheduling of Divisible Real-Time Workloads on Multiprocessor Platforms*" is the result of my own research except as cited in references. The thesis has not been accepted for any degree and is not concurrently submitted in candidature of any degree.

Signature : .....

Name of Candidate : SURIAYATI BT CHUPRAT

Date : 17 JUNE 2009

To

my beloved husband, Abd Hadi,

my loving childrens, Aizat and Ainaa,

and

my loving and supportive parents,

Hj Chuprat and Hajjah Rabahya.

ACKNOWLEDGEMENT

First of all, I thank ALLAH (SWT), the Lord Almighty, for giving me the health, strength, ability to complete this work and for blessing me with supportive supervisors, family and friends.

I wish to express my deepest appreciation to my supervisor, Professor Dr Shaharuddin Salleh for his idea, support, enthusiasm, and patience. I have learnt an enormous amount from working with him. My special thanks to my co-supervisor Professor Dr Sanjoy K. Baruah for his guidance, support, respect, and kindness. The opportunity to collaborate with him during my five months visit at the University of North Carolina, Chapel Hill, USA has benefited my research tremendously.

I would also like to thank Professor Dr James H. Anderson for giving me the opportunity to attend his class on Real-time Systems at UNC. Thanks to the Real Time Systems Group of UNC (Nathan, Bjeorn, John, Aaron, Hennadiy) for sharing their dynamic discussions during the "Real-time Lunch" weekly meeting.

I am forever indebted to my employer Universiti Teknologi Malaysia (UTM) for granted me the study leave, funds and the facilities for my research. Thanks to all management staff of KST Kuala Lumpur, HRD Skudai and Canselori UTMKL.

Life would be harder without the support of many good friends. Thanks to Dr Zuraini and Dr Ruzana for being such a good mentor, Arbai'ah and Haslina for sharing the challenging PhD years, Dr Nazli, Dr Maslin and Dr Liza for their support and motivations. Thank you to all my friends in KST Kuala Lumpur and UTM Skudai.

Finally, I thank to all my family members for their love, patient and uncountable supports.

ABSTRACT

Current formal models of real-time workloads were designed within the context of uniprocessor real-time systems; hence, they are often not able to accurately represent salient features of multiprocessor real-time systems. Researchers have recently attempted to overcome this shortcoming by applying workload models from *Divisible Load Theory* (DLT) to real-time systems. The resulting theory, referred to as *Real-time Divisible Load Theory* (RT-DLT), holds great promise for modeling an emergent class of massively parallel real-time workloads. However, the theory needs strong formal foundations before it can be widely used for the design and analysis of real-time systems. The goal of this thesis is to obtain such formal foundations, by generalizing and extending recent results and concepts from multiprocessor real-time scheduling theory. To achieve this, recent results from traditional multiprocessor scheduling theory were used to provide satisfactory explanations to some apparently anomalous observations that were previously made upon applying DLT to real-time systems. Further generalization of the RT-DLT model was then considered: this generalization assumes that processors become available at different instants of time. Two important problems for this model were solved: determining the minimum number of processors needed to complete a job by its deadline; and determining the earliest completion time for a job upon a given cluster of such processors. For the first problem, an optimal algorithm called MINPROCS was developed to compute the minimum number of processors that ensure each job completes by its deadline. For the second problem, a Linear Programming (LP) based solution called MIN-ξ was formulated to compute the earliest completion time upon given number of processors. Through formal proofs and extensive simulations both algorithms have been shown to improve the non-optimal approximate algorithms previously used to solve these problems.

ABSTRAK

Model formal bagi beban kerja masa nyata asalnya direkabentuk dalam konteks sistem masa-nyata satu pemproses. Model ini kadangkala gagal mewakilkan secara tepat ciri-ciri sistem masa-nyata pemproses berbilang. Masalah ini cuba diatasi oleh para penyelidik dengan mengaplikasikan model beban kerja yang digunakan di dalam Teori Pembahagian Beban (DLT) kepada sistem masa nyata. Hasil aplikasi ini dikenali sebagai Teori Pembahagian Beban Masa Nyata (RT-DLT). Teori ini menunjukkan potensi yang meyakinkan bagi memodelkan beban kerja masa nyata selari dalam kelas besar. Walaubagaimanapun, sebelum teori ini boleh digunakan dalam merekabentuk dan analisis sistem masa nyata, ia memerlukan asas formal yang kukuh. Tujuan kajian tesis ini adalah untuk menghasilkan asas formal yang dimaksudkan dengan memperluaskan hasil kajian terkini dan menggunakan konsep dari teori sistem masa nyata pemproses berbilang. Untuk mencapai tujuan ini, hasil kajian terkini daripada teori penjadualan sistem masa nyata pemproses berbilang digunakan bagi menerangkan pemerhatian yang luar-biasa apabila Teori Pembahagian Beban diaplikasikan kepada sistem masa nyata. Tesis ini seterusnya mengkaji model Teori Pembahagian Beban Masa Nyata apabila berlaku keadaan di mana masa sedia pemproses-pemproses di dalam kluster adalah berbeza-beza. Dua masalah utama berjaya diselesaikan dalam kajian ini: menentukan bilangan minimum pemproses yang diperlukan untuk menyiapkan beban kerja sebelum sampai masa tamat; menentukan masa yang paling awal bagi menyiapkan sesuatu beban kerja. Bagi masalah pertama, satu algoritma optimal dinamakan MINPROCS telah dihasilkan. Dan untuk masalah kedua satu penyelesaian berasaskan Pengaturcaraan Lelurus yang dinamakan MIN-ξ telah direkabentuk. Melalui pembuktian formal dan beberapa siri simulasi, telah dibuktikan bahawa kedua-dua penyelesaian adalah optimal dan sekaligus algoritma yang sebelumnya digunakan untuk menyelesaikan masalah yang sama diperbaiki.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AN	All Nodes
ATLAS	AToroidal LHC ApparatuS
CMS	Compact Muon Solenoid
DAG	Directed Acyclic Graph
DLT	Divisible Load Theory
DM	Deadline Monotonic
EDF	Earliest Deadline First
EDZL	Earliest Deadline Zero Laxity
EPR	Equal Partitioning
EPU	Effective Processor Utilization
FIFO	First In First Out
IIT	Inserted Idle Time
LLF	Least Laxity First
LP	Linear Programming
MN	Minimum Nodes
MWF	Maximum Workload Derivative First
OPR	Optimal Partitioning
RM	Rate Monotonic
RT-DLT	Real-time Divisible Load Theory
SMP	Symmetric Shared Memory Multiprocessor
UMA	Uniform Memory Access
WCET	Worst Case Execution Time

LIST OF SYMBOLS

a_i	-	Arrival Time of i^{th} job
c_i	-	Execution Requirement of i^{th} job
C_m	-	Communication Cost
C_p	-	Computation Cost
c_i^j	-	Computation Time
C_i	-	Worst Case Requirement of i^{th} task
d_i	-	Deadline of i^{th} job
D_i	-	Deadline of i^{th} task
e_i	-	Worst Case Execution Time of i^{th} job
f_i	-	Completion Time of i^{th} job
J_i	-	i^{th} Job
I	-	Collection of Jobs
L_i	-	i^{th} Link
n	-	Number of Processors
$n^{\min}$	-	Minimum Number of Processors
p_i	-	Period or Inter-arrival between successive jobs
P_i	-	i^{th} Processor
r_i	-	Ready Time i^{th} job
s_i	-	Start Time i^{th} job
$S(t)$	-	Schedule as Integer Step Function
T_i	-	Minimum Inter-arrival separation of i^{th} task
t	-	Time

U_i	-	Utilization of i^{th} task
$U_{sum}(\tau)$	-	Total Utilization of a task system τ
$U_{\max}(\tau)$	-	Maximum Utilization of a task system τ
$V_{\max}(\tau)$	-	Maximum Utilization of a task system τ in non-preemptive system
$V_{sum}(\tau)$	-	Total Utilization of a task system τ in non-preemptive system
$e_{\max}(\tau)$	-	Maximum execution time of task τ

Greek Symbols

τ_i	-	i^{th} Task
σ_i	-	i^{th} Workload
α_i	-	i^{th} Fraction of Workload
β	-	Ratio of C_p and $(C_p + C_m)$
δ	-	Density of i^{th} task
$\delta_{\max}(\tau)$	-	Total Density of a task system τ
$\delta_{sum}(\tau)$	-	Largest Density of a task system τ
ξ_i	-	Execution Time of i^{th} workload
ϕ	-	Off set
$\chi(n)$	-	Cost of executing a job

LIST OF APPENDICES

CHAPTER 1

INTRODUCTION

1.1 Overview

Real-time computer application systems are systems in which the correctness of a computation depends upon both the *logical* and *temporal* properties of the result of the computation. Temporal constraints of real-time systems are commonly specified as deadlines within which activities should complete execution. For *hard-real-time systems*, meeting timing constraints is crucially important – failure to do so may cause critical failures and in some cases cause hazard to human life (Buttazzo, 2004). In *soft-real-time systems*, by contrast, the consequences of an occasional missed deadline are not as severe (Buttazzo et al., 2005). Given the central importance of meeting timing constraints in hard-real-time systems, such systems typically require guarantees prior to deployment – e.g., during system design time – that they will indeed always meet their timing constraints during run-time. This thesis is primarily concerned with hard-real-time systems.

Real-time computing will continue to play a crucial role in our society, as there are an increasing number of complex systems that needs computer control. Many next-generation computing applications such as automated manufacturing systems, defense systems (e.g. smart bombs, automotive, avionics and spacecraft control systems), high speed and multimedia communication systems, have significant real-time components (Liu, 2000; Buttazzo, 2004).

Such real-time application systems demand complex and significantly increased functionality and it is becoming unreasonable to expect to implement them upon *uniprocessor* platforms. Consequently, these systems are increasingly coming to be implemented upon *multiprocessor* platforms, with complex synchronization, data-sharing and parallelism requirements.

Formal models for representing *real-time workloads* have traditionally been designed for the modeling of processes that are expected to execute in uniprocessor environments. As real-time application systems increasingly come to be implemented upon multiprocessor environments, these same models have been used to model the multiprocessor task systems. However, these traditional models fail to capture some important characteristics of multiprocessor real-time systems; furthermore, they may impose additional restrictions ("additional" in the sense of being mandated by the limitations of the model rather than the inherent characteristics of the platform) upon system design and implementation.

One particular restriction that has been extended from uniprocessor models to multiprocessor ones is that each task may execute upon at most one processor at each instant in time. In other words, they do not allow task parallel execution. However, this is overly restrictive for many current multiprocessor platforms; to further exacerbate matters, this restriction is in fact one significant causal factor of much of the complexity of multiprocessor scheduling. Indeed, as Liu (1969) pointed out, *"the simple fact that a [job] can use only one processor even when several processors are free at the same time adds a surprising amount of difficulty to the scheduling of multiple processors."* Certainly, the next generation of embedded and real-time systems will demand parallel execution.

Recently, some researchers have studied extensions to the workload models traditionally used in real-time scheduling theory, to allow for the possibility that a single job may execute simultaneously on multiple processors. One of the more promising approaches in this respect has been the recent work of Lin et al. (2006a, 2006b, 2007a, 2007b, 2007c), that applies *Divisible Load Theory (DLT)* to multiprocessor real-time systems. The resulting theory is referred to as *Real-time Divisible Load Theory (RT-DLT).*

1.2 Research Problem and Motivations

Real-time Divisible Load Theory (RT-DLT) holds great promise for modeling an emergent class of massively parallel real-time workloads. ***However, the theory needs strong formal foundations before it can be widely used for the design and analysis of hard real-time safety-critical applications***. In this thesis, we address the general problem of obtaining such formal foundations, by generalizing and extending recent results and concepts from multiprocessor real-time scheduling theory. Within this general problem, here are some of the specific issues we address:

i. Prior research in RT-DLT has reported some apparently anomalous findings, in the sense that these findings are somewhat counter-intuitive when compared to results from "regular" (i.e., non-real-time) DLT. *What explains these (previously-identified) apparent anomalies in RT-DLT?*

ii. When the processors in a multiprocessor platform all become available at the same instant in time, the issue of scheduling a real-time divisible workload on such platforms is pretty well understood. However, the reality in many multiprocessor environments is that all the processors do not become available to a given workload at the same instant (perhaps because some of the processors are also being used for other purposes). *How does one extend RT-DLT to render it applicable to the scheduling of real-time workloads upon platforms in which all the processors are not made available simultaneously? Specifically we address two important problems:*

- Given a divisible job $\tau_i = (a_i, \sigma_i, d_i)$ and varying processor ready-times $r_1, r_2, r_3, \ldots$ what is the minimum number of processors needed to meet a job's deadline?

- Given a divisible job $\tau_i = (a_i, \sigma_i, d_i)$ and $\boldsymbol{n}$ (identical) processors with varying ready-times $r_1, r_2, \ldots, r_n$ upon which to execute it, what is the earliest time at which the job τ_i can complete execution?

1.3 Research Objectives

As stated above, the goal of this thesis is to develop strong formal foundations that enable the application of RT-DLT for the design and analysis of multiprocessor hard real-time systems. To achieve this goal, we must build theoretical foundations and accurate simulation environments for experimenting with, and explaining the behavior of, hard real-time DLT systems. Some of the specific objectives that we have identified as needing to be accomplished in order to achieve this goal are as follows:

i. To investigate the application of Divisible Load Theory (DLT) models to real-time workloads, in order to obtain a deep and detailed understanding of the behavior of such systems.

ii. To theoretically explain the apparent anomalies of Real-time Divisible Load Theory (RT-DLT).

iii. To extend RT-DLT so that they are able to handle cluster and workload models that are as general as possible. Specifically, we hope that these extensions will be applicable to platforms in which all processors do not become available simultaneously.

iv. To build efficient scheduling algorithms that will compute the exact minimum number of processors that must be assigned to a job in order to guarantee that it meets its deadline — on clusters in which all processors are not simultaneously available.

v. To develop efficient scheduling algorithms that minimize the completion time of a given divisible job upon a specified number of processors — on clusters in which all processors are not simultaneously available.

1.4 Scope of Research

In this thesis, we focus upon a particular formal model of real-time workloads that is very widely used in real-time and embedded systems design and implementation. In this model, it is assumed that there are certain basic units of work, known as *jobs* that need to be executed. Such jobs are generated by recurring processes known as periodic or sporadic tasks – each such task represents a piece of straight-line code embedded within a potentially infinite loop. This workload model is described in greater detail in Chapter 2.

There are several kinds of timing constraints considered in the real-time scheduling literature; in this thesis, we restrict our attention for the most part to just one of these kinds of constraints – meeting *deadlines* of jobs.

With respect to system resources, we will focus for the most part on *minimizing the number of processors* used. (Although other system resources, such as network bandwidth, energy, etc. are also important, optimization with respect to these resources does not lie within the scope of this thesis.)

Several different network topologies, such as stars, meshes, and trees, have been studied in DLT. We restrict our attention to the *single-level tree* topology, since this is one of the simpler models but nevertheless appears to contain most of the important issues that arise when DLT is extended to apply to real-time workloads.

1.5 Research Methodology

We conducted this research in six major phases, as shown in Figure 1.1. The six phases are: Literature Review, Analysis and Problem Formulations, Algorithms Design, Algorithms Implementation, Algorithms Evaluations and Documentation. Each of these phases will be described in greater detail in the following pages.

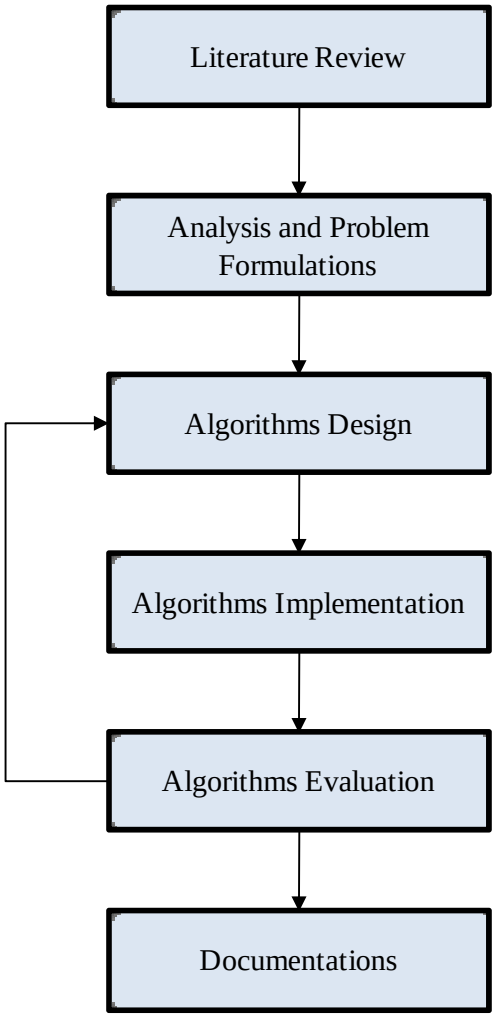

Figure 1.1 Conducted phases in this research

i. **Literature Review**

We performed literature review on various topics related to the research conducted in this thesis. The topic includes:

- State of the art of Real-time Systems
- State of the art of Real-time scheduling theory
- Current findings on Divisible Load Theory (DLT)
- Current findings on Real-time Divisible Load Theory (RT-DLT)

ii. **Analysis and Problem Formulations**

In this phase, we studied the applicability of DLT to multiprocessor scheduling of real-time systems. Specifically we analyzed series of work on RT-DLT (Lin et al., 2006a, 2006b, 2007a, 2007b, 2007c) and formulated three important problems arises upon these works. We explain these formulations in Chapter 3, 4 and 5 accordingly.

iii. **Algorithms Design**

As stated earlier, we formulated three significant problems detected from the work of Lin et al. (2006a, 2006b, 2007a, 2007b, and 2007c). For the first problem, we used existing scheduling theory to explain an anomalous observation of Lin et al. (2006a, 2006b, 2007a) when they first applied DLT to real-time multiprocessor scheduling. For the second problem, we designed an efficient algorithm to compute the minimum number of processors needed for a job to meet its deadline. To develop this algorithm, we used the first principle of RT-DLT found in Lin et al. (2006a, 2006b, and 2007a). And for the third problem, we formed a Linear Programming-based algorithm to compute the minimum completion time of a job execution. We present each detail design in Chapter 3, 4 and 5 respectively.

iv. **Algorithms Implementation**

In this phase, we developed series of simulations to compare the degree of improvement of our proposed algorithms to prior existing ones. For the second problem, we implemented the algorithm using C++ and for the third problem we developed the simulation programs using MATLAB.

v. **Algorithms Evaluation**

We evaluated our proposed algorithms by analyzing the results produced by our simulation programs. We compared the results produced by our algorithm with the ones produced by previous algorithms. In all comparisons, our algorithms showed significant improvement over pre-existing ones. We also provide lemmas and proofs to support our results and discussion in this thesis.

We conducted phase 3, 4 and 5 in three cycles for the three problems formulated.

vi. **Documentations**

Finally each contribution reported in this thesis was documented in technical publications. A list of papers published in the proceedings of conferences and journals are listed in Appendix A. The final and complete documentation is compiled in this thesis.

1.6 Thesis Organization

This thesis is organized into six chapters. Figure 1.2 shows the flow of the thesis organization; descriptions are given in the following pages.

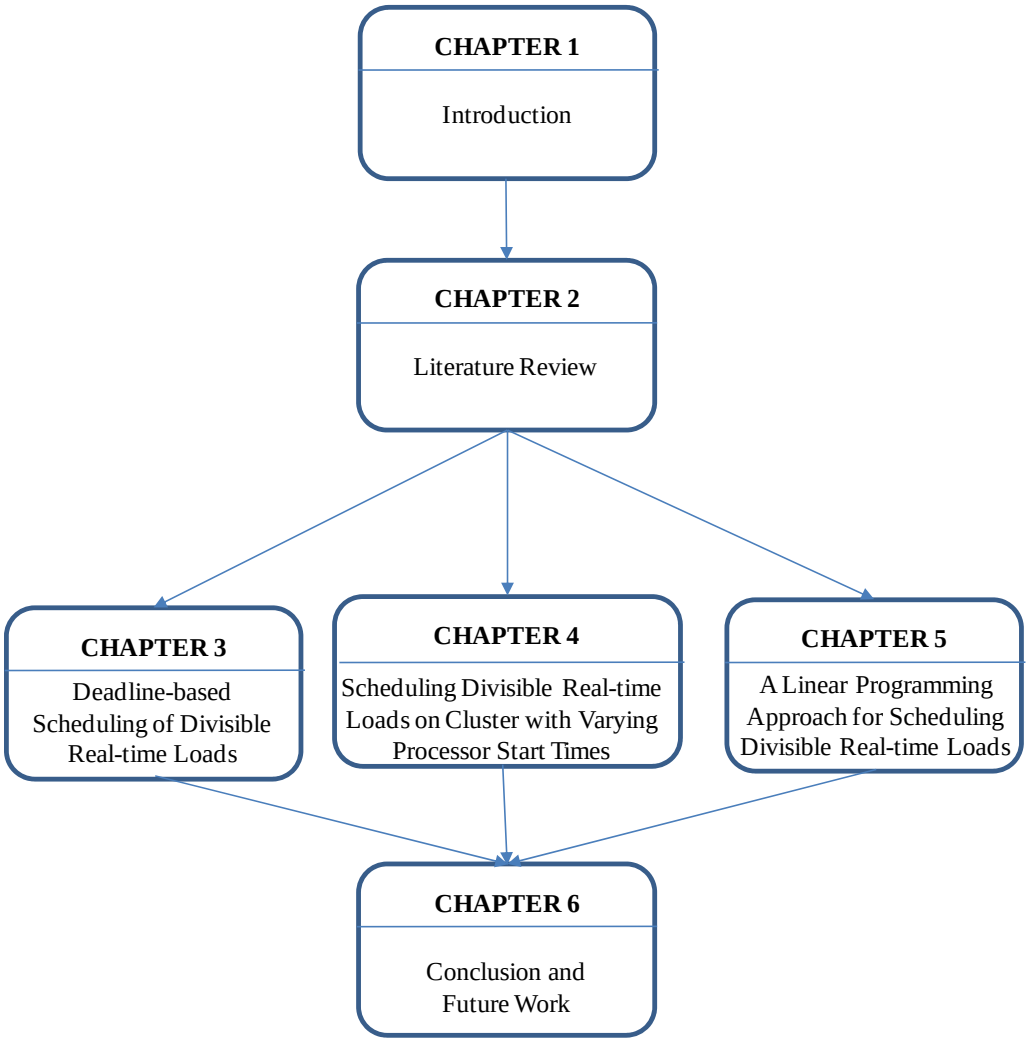

Figure 1.2 Thesis organization

This thesis explores two important research areas: Real-time Systems and Divisible Load Theory. In Chapter 2, we present some background information and review some of the prior results on real-time systems. The first part describes the basic concepts of real-time systems. We then briefly review some fundamental

results concerning real-time multiprocessor scheduling. The discussion mainly focuses on global multiprocessor scheduling with the Earliest Deadline First (EDF) scheduling algorithm. This chapter also discusses in greater detail the concept of Divisible Load Theory (DLT) and the application of this theory to multiprocessor scheduling of real-time systems, referred to as RT-DLT. We review some of the prior work done in RT-DLT, which we extend as part of this thesis.

In Chapter 3, we will report our first contribution presented in this thesis. We describe the initial work of Lin et al. (2006a, 2006b and 2007a) and their apparently anomalous findings with respect to a scheduling framework integrating DLT and EDF. We then present our results that provide a theoretical analysis to some of these anomalies.

In Chapter 4, we describe our study on scheduling problems in RT-DLT when applied to clusters in which different processors become available at different time-instants. We present an algorithm that efficiently determines the minimum number of processors that are required to meet a job's deadline. We then describe and discuss simulation results evaluating the proposed algorithm, and comparing it to previously-proposed heuristics for solving the same problem.

We have proposed a Linear Programming (LP) based approach to efficiently determine the earliest completion time for the job on a given processors which may become available at different times. This LP based approach is described in Chapter 5. We then present extensive experimental simulations to evaluate this LP based approach and consequently show how this approach significantly improves on the heuristic approximations that were the only techniques previously known for solving these problems.

Finally, we conclude our work and suggest directions for future research in Chapter 6.

CHAPTER 2

LITERATURE REVIEW

2.1 Introduction

In the previous chapter, we mentioned that the goal of our research is to further study the application of Divisible Load Theory (DLT) to the real-time systems workloads (henceforth, we referred as RT-DLT). In this chapter, we present some background information and review some of the prior results on real-time systems. The basic concepts of real-time systems will be presented in section 2.2.

In section 2.3, we will briefly review some fundamental results concerning real-time multiprocessor scheduling. The discussion will mainly focus on global multiprocessor scheduling with the Earliest Deadline First (EDF) scheduling algorithm. In section 2.4, we review recent works concerning parallel execution upon multiprocessor real-time systems. Specifically, we review the fundamental concepts of DLT and current findings on RT-DLT.

2.2 Real-time Systems

In designing a real-time system, there are three important components that must be specified: *Workload Models, Platform Models and Scheduling Algorithm.* This section will briefly explain the basic terms and concepts used to describe these components.

2.2.1 Real-time Workload

Real-time workloads are assumed to be comprised of basic units of execution known as *jobs*. Each job, $J_i = (a_i, c_i, d_i)$ is characterized by an arrival time $a_i \geq 0$, an execution requirement $c_i > 0$, and a relative deadline $d_i > 0$, and has the following interpretation: The job must execute for c_i time units over the time interval $[a_i, a_i + d_i)$. Other parameters associated with a job are the start time s_i and the completion time f_i. Figure 2.1 shows typical parameters of a real-time job.

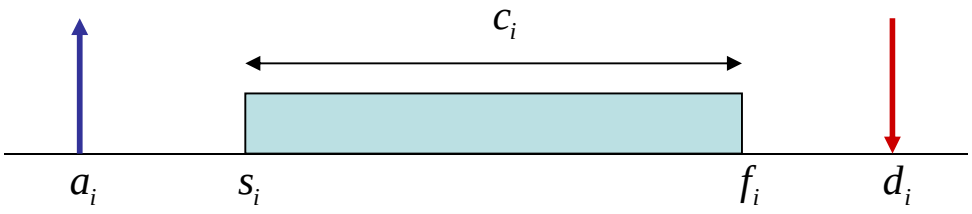

Figure 2.1: Typical parameters of a real-time job

Jobs are classified as being *preemptive* or *non-preemptive*. A preemptive job that is executing may have its execution interrupted at any instant in time and resumed later, at no cost or penalty; by contrast, the entire execution of a non-

preemptive job must occur in one contiguous time-interval of length c_i. In this thesis, we are primarily concerned with the scheduling of preemptive jobs.

In real-time systems, it is generally assumed that these jobs are generated by *recurring tasks*. Each such recurring task is assumed to model a piece of straight-line code embedded within a potentially infinite loop, with each iteration through the loop being modeled by a single job. Over time, a recurring task τ, initiates a real-time instance *I*, where *I* denotes a finite or infinite collection of jobs $I = \{J_1, J_2, ...\}$. *Task model* is a format and rules for specifying a task system. For every execution of the system, recurring task τ_i will generate a collection of real-time jobs. Several recurring tasks can be composed together into a recurring task system $\tau = \{\tau_1, \tau_2, ...\tau_m\}$.

Among widely used task models is the *periodic task model* (Liu and Layland, 1973). In this model, a periodic task τ_i is specified by a three tuple (ϕ_i, e_i, p_i), where ϕ_i is the offset of the first job generated by τ_i from start system time; e_i is the worst-case execution time (WCET) of any job generated by τ_i; and p_i is the *period* or inter-arrival time between successive jobs of τ_i.

The set of jobs generated by a periodic task τ_i with worst-case possible execution times is:

$$J^{P}_{WCET}(\tau_i) \overset{def}{=} \{(\phi_i, e_i, \phi_i + p_i), (\phi_i + p_i, e_i, \phi_i + 2p_i), (\phi_i + 2p_i, e_i, \phi_i + 3p_i), ...\}$$

Example 2.1: Consider a periodic task $\tau = \{\tau_1 = (0,4,8), \tau_2 = (10,5,15)\}$. The set of jobs generated by τ_1 and τ_2 with worst-case execution times are:

$$J^{P}_{WCET}(\tau_1) = \{(0,4,8), (8,4,16), (16,4,24), ...\}$$

$$J^{P}_{WCET}(\tau_2) = \{(10,5,15), (25,5,40), (40,5,55), ...\}$$

Figure 2.2 depicts the arrivals and executions of jobs generated by two periodic task described in Example 2.1.

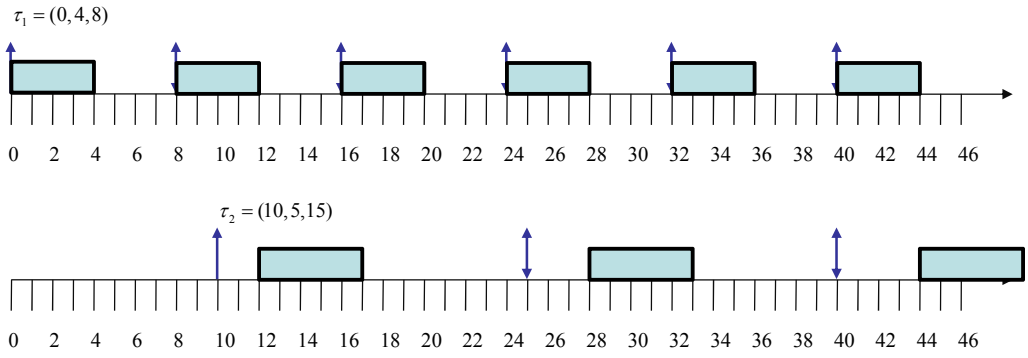

Figure 2.2: Example of arrivals and executions of jobs generated by periodic tasks $\tau_1 = (0,4,8)$ and $\tau_2 = (10,5,15)$.

The sporadic task model (Mok, 1983, Baruah et al., 1990) is another commonly used formal model for representing recurring real-time task systems. In the sporadic task model, a task $\tau_i = (C_i, D_i, T_i)$ is characterized by a worst-case execution requirement C_i, a (relative) deadline D_i, and a minimum inter-arrival separation T_i, also known as period. Such a sporadic task generates an infinite sequence of jobs, with successive job-arrivals separated by at least T_i time units.

Each job has a worst-case execution requirement equal to C_i and a deadline that occurs at D_i time units after its arrival time. A sporadic task system is comprised of several such sporadic tasks. Let τ denote a system of such sporadic tasks: $\tau = \{\tau_1, \tau_2, ..., \tau_m\}$ with $\tau_i = (C_i, D_i, T_i)$ for all $i, 1 \le i \le m$. Task system τ is said to be a constrained deadline sporadic task system if it is guaranteed that each task $\tau_i \in \tau$ has its relative deadline parameter no larger than its period: $D_i \le P_i$ for all $\tau_i \in \tau$.

Example 2.2: Consider a sporadic task $\tau = \{\tau_1 = (2,4,6), \tau_2 = (3,9,12)\}$. Since the sporadic task model specifies a minimum, rather than an exact, separation between the arrivals of successive jobs of each task, each sporadic task may generate infinitely many different sets of jobs. One such possible set of jobs generated by τ_1 and τ_2 with worst-case execution times are:

$$J^{S}_{WCET}(\tau_1) = \{(0,2,4),(6,2,10),(12,2,16),...\}$$
$$J^{S}_{WCET}(\tau_2) = \{(0,3,9),(12,3,21),(24,3,33),...\}$$

Figure 2.3 depicts the arrivals and executions of jobs generated by two sporadic task described in Example 2.2.

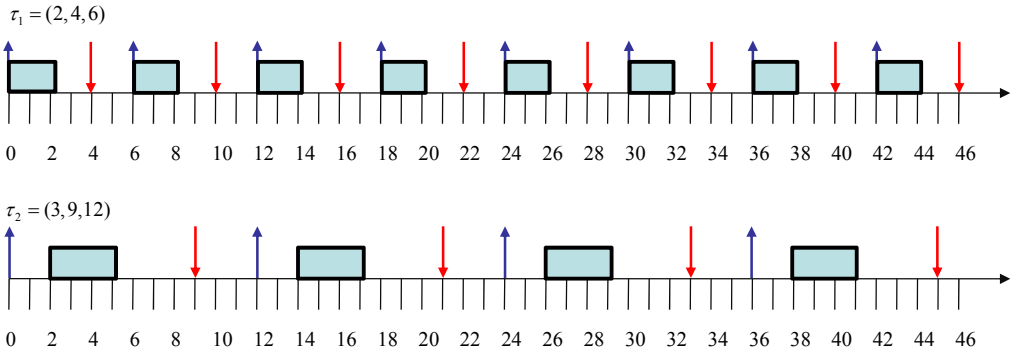

Figure 2.3: Example of arrivals and executions of jobs generated by sporadic tasks $\tau_1 = (2,4,6)$ and $\tau_2 = (3,9,12)$.

Additional task properties that we would like to define, as we will use in later chapters are the notions of *Utilization* and *Density*.

Utilization The utilization U_i of a task τ_i is the ratio of its execution requirement to its period:

$$U(\tau_i) = \frac{C_i}{T_i} \tag{2.1}$$

The total utilization $U_{sum}(\tau)$ and the largest utilization $U_{\max}(\tau)$ of a task system τ are defined as follows:

$$U_{sum}(\tau) \overset{def}{=} \sum_{\tau_i \in \tau} U(\tau_i) \tag{2.2}$$

$$U_{\max}(\tau) \overset{def}{=} \max_{\tau_i \in \tau} U(\tau_i) \tag{2.3}$$

Density The density δ of a task τ_i is the ratio of its execution requirement to the smallest of its relative deadline and its period:

$$\delta(\tau_i) = \frac{C_i}{\min(D_i, T_i)} \tag{2.4}$$

The total density $\delta_{sum}(\tau)$ and the largest density $\delta_{\max}(\tau)$ of a task system τ are defined as follows:

$$\delta_{sum}(\tau) \stackrel{def}{=} \sum_{\tau_i \in \tau} \delta(\tau_i) \tag{2.5}$$

$$\delta_{\max}(\tau) \stackrel{def}{=} \max_{\tau_i \in \tau} \delta(\tau_i) \tag{2.6}$$

One of the fundamental assumptions in these task models is that a job may be executing on at most one processor at each instant in time. In other words, parallel execution is not permitted. Keeping this in mind, now we will define the platform models in the following section.

2.2.2 Platform Model

Real-time scheduling theory has traditionally focused upon scheduling real-time workloads on *uniprocessor* platforms. In such platforms, there is only a single processor upon which the entire real-time workload is to execute. More recently, attention has been given to *multiprocessor* platforms, comprised of several processors. Typically, the letter *P* is used to denote processor(s). If a platform has n processors, the platform is denoted as $P_1, P_2, ..., P_n$.

An *identical* multiprocessor platform is a platform in which all the processors have the same capabilities and speed; more specifically, each processor is identical in terms of architecture, cache size and speed, I/O and resource access, and the access time to shared memory (Uniform Memory Access (UMA)).

Figure 2.4 shows a high-level illustration of a possible layout of an identical multiprocessor platform. All processors are connected at the bus level. This type of multiprocessor is commonly referred to as a *symmetric shared-memory multiprocessor (SMP)*, also known as *tightly-coupled* multiprocessor systems. On the other hand, loosely-coupled multiprocessor systems, also known as clusters, are comprised of multiple standalone computers interconnected via a high speed communication system such as Gigabit Ethernet. A Linux Beowulf cluster is an example of a *loosely-coupled* multiprocessor system.

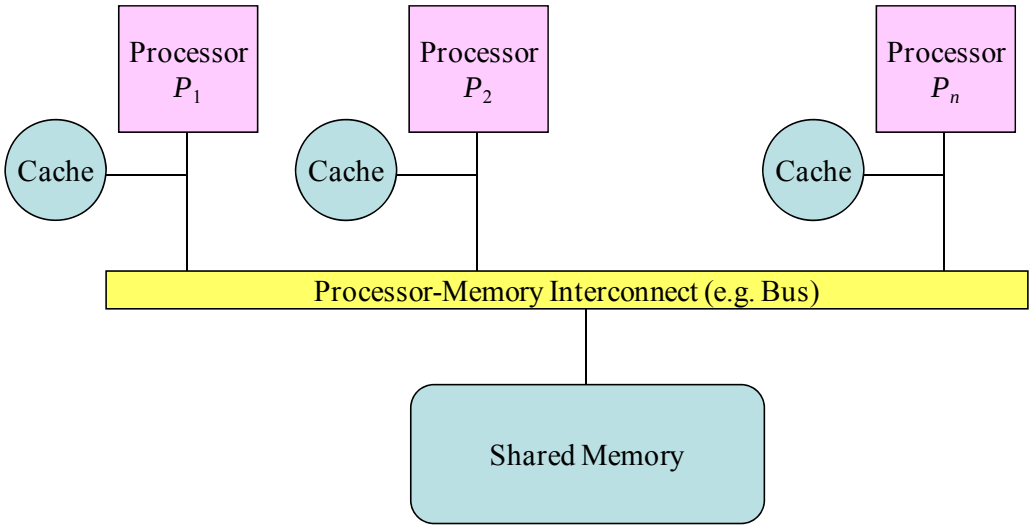

Figure 2.4: The layout of a symmetric shared-memory multiprocessor (SMP) platform.

A multiprocessor platform is called *heterogeneous* if each processor has different capabilities. Heterogeneous multiprocessors may be further classified into *uniform* and *unrelated* multiprocessors. The classification can be made in terms of execution speed of each processor.

Recall that, a task is a computation that is executed by the processor(s). When a single processor has to execute a set of tasks, or the number of tasks to be executed is greater than the number of processors in the processing platform, one need to

decide which task to be assigned to these processors. The set of rules for these assigning activities is known as *scheduling algorithm* (Cottet et al., 2002). We will describe this concept in greater detail, in the following section.

2.2.3 Scheduling Algorithms

Generally, a scheduling algorithm is a set of rules that allocates processor time to tasks. In real-time systems, processor-allocation strategies are driven by the need to meet timing constraints. A formal definition of a *uniprocessor schedule* for real-time task systems is given as follows:

Definition (from, (Buttazzo, 2004)): Given a set of *m* tasks, $\tau = \{\tau_1, \tau_2, ..., \tau_m\}$, a schedule is an assignment of tasks to processor, so that each task is executed until completion. More formally, a schedule can be defined as a function $S : \mathbb{R}^+ \rightarrow \mathbb{N}$ such that $\forall t \in \mathbb{R}^+, \exists t_1, t_2$ such that $t \in [t_1, t_2)$ and $\forall t' \in [t_1, t_2)\ S(t) = S(t')$. In other words, $S(t)$ is an integer step function and $S(t) = i$, with $i > 0$, means that task τ_i is executing at time *t, while* $S(t) = 0$ means that the processor is idle.

Before we further describe the scheduling approaches, we define some terms commonly used in describing properties of real-time scheduling algorithms.

- **Feasibility.** A schedule is said to be feasible if all tasks are completed according to set of specified constraints.

- **Schedulability**. A set of tasks is said to be schedulable if there exists at least one algorithm that can produce a feasible schedule.

- **Optimality**. A scheduling algorithm is said to be an optimal if the algorithm is able to produce a feasible schedule for any schedulable task set.

In *uniprocessor* scheduling, jobs will need to be organized and executed over a single processor, as illustrated in Figure 2.5. Arriving jobs will be stored in a job queue, and wait until the processor is ready for next execution. A scheduling algorithm, will select a job from the jobs queue, based on the scheduling policy used. For example, in the *Earliest Deadline First* (EDF) scheduling algorithm, the scheduler will choose a job with the nearest deadline. In other words, the job with the smallest deadline has the highest priority for the next execution. In some scheduling algorithm, preemption[1] is allowed, where the current job execution will be stop, allowing a job with higher priority be executed, and the preempted job will resume execution in other point of time. This kind of scheduling is illustrated in Figure 2.6.

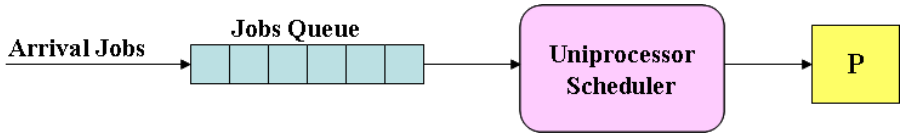

Figure 2.5: Uniprocessor scheduling

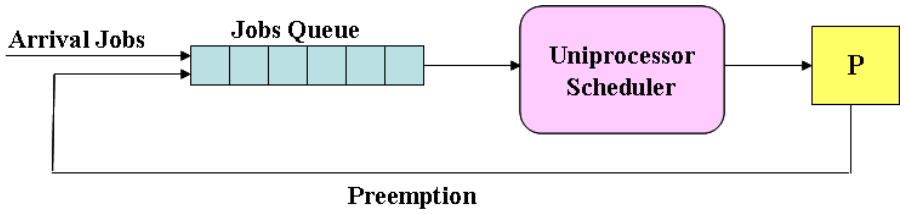

Figure 2.6: Uniprocessor scheduling with preemption

[1] Preemption is allowed in a system that permits interruption at any point of task execution.

On the other hand, in multiprocessor systems, the scheduler will organize and execute tasks over a set of processors. Two kinds of scheduling are defined for multiprocessor systems – one is known as *global scheduling* (Baker and Baruah, 2008) and the other is called *partitioned scheduling* (Baker and Baruah, 2008).

As illustrated in Figure 2.7, in global scheduling, arriving jobs will be placed in the global jobs queue. Then the global scheduler will organize which job to execute on each processor at each instant of time. In partitioned scheduling (see Figure 2.8) tasks will first be assigned to processors by a partitioning algorithm, and the generated jobs of each task are subsequently placed on the local job queue. Then, a uniprocessor scheduling is used to schedule jobs to each processor.

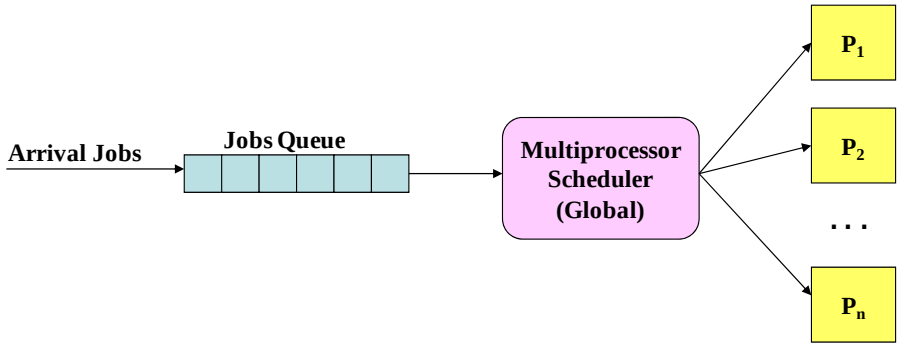

Figure 2.7: A multiprocessor global scheduling

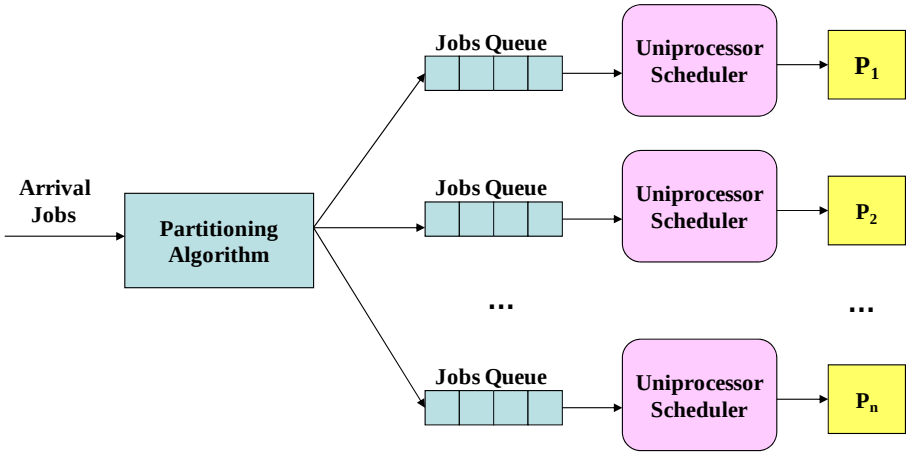

Figure 2.8: A multiprocessor partitioned scheduling

Scheduling in multiprocessor systems is more complex than in uniprocessor systems. Although it seems, having more processors provides the advantages in executing more jobs at each instant of time, but the organization and synchronization adds more complexity to the scheduling algorithms. These challenges have created a significant attention to multiprocessor scheduling research. We now review some of the important results concerning the real-time multiprocessor scheduling in the following section.

2.3 Real-time Multiprocessor Scheduling and EDF

Real-time scheduling is an important area of research in real-time computing as scheduling theory addresses the problem of meeting the specified timing requirements of the system (Stankovic and Ramamritham, 1985). Among the scheduling algorithms studied in real-time systems are: Earliest Deadline First (EDF) (Liu and Layland, 1973), Rate Monotonic (Liu and Layland, 1973; Baruah and Goossens, 2003), Deadline Monotonic (Leung and Whitehead, 1982), Least Laxity First (Dertouzous and Mok, 1989), Pfair-based algorithms (Baruah et al., 1996; Baruah et al., 1995) and Earliest Deadline with Zero Laxity (Cirinei and Baker, 2007; Baker et al., 2008).

Since our work is related to EDF, this section reviews the prior fundamental results obtained in EDF scheduling. EDF is a priority-driven scheduling algorithm that schedules the arriving tasks according to their deadlines. Specifically, tasks with the earlier deadlines have highest priority to be executed as soon as processors become available. EDF is known to be an excellent scheduling algorithm for uniprocessor platforms under a wide variety of conditions. For preemptive systems of independent jobs, it has been shown (Dertouzos, 1974) to be optimal in the sense that if any scheduling algorithm can schedule a given system to meet all deadlines, then EDF, too, will meet all deadlines for this system.

Figure 2.9 depicts an example of an EDF schedule on a uniprocessor platform, with a set of three tasks $\tau_1 = (0,1,3)$, $\tau_2 = (0,4,10)$ and $\tau_3 = (2,2,4)$. Both task τ_1 and task τ_2 arrived at t=0, and τ_1 is executed first, since it has an earlier deadline. At t=1, τ_2 begins its execution after τ_1 completes. When τ_3 arrives at t=3, τ_2 is preempted and resumes execution when τ_3 completes at t=4. In this example all tasks meet their respective deadlines.

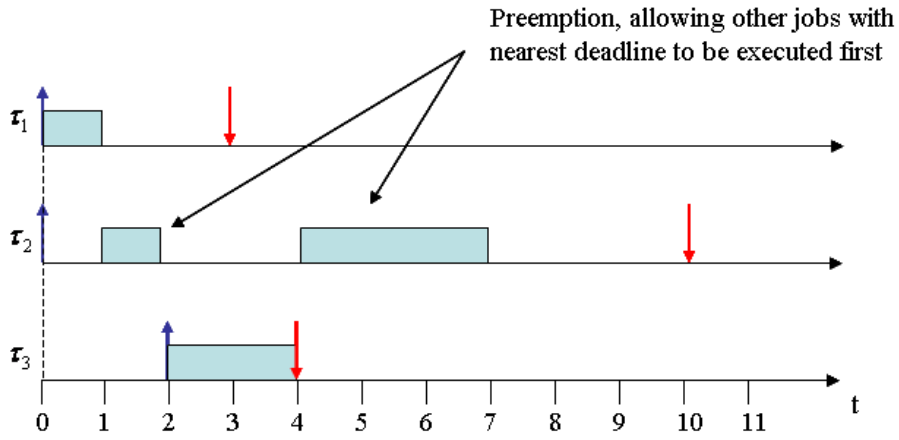

Figure 2.9: Example of an EDF schedule on uniprocessor platforms. Up arrows depicts job arrivals, and down arrows depicts job deadlines.

For non-preemptive systems, the EDF can be a non-optimal algorithm (Jeffay et al., 1991). Consider the same set of 3 tasks τ_1, τ_2, τ_3 and the same job arrivals as in Figure 2.7. Figure 2.10 shows that these jobs can be scheduled non-preemptively to meet all deadlines. Figure 2.11 exhibits that non-preemptive EDF fails to produce a feasible schedule.

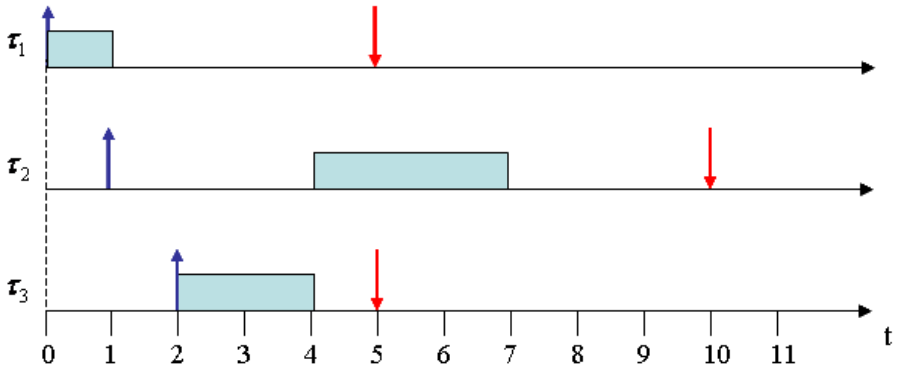

Figure 2.10: Feasible schedule exists for a non-preemptive system

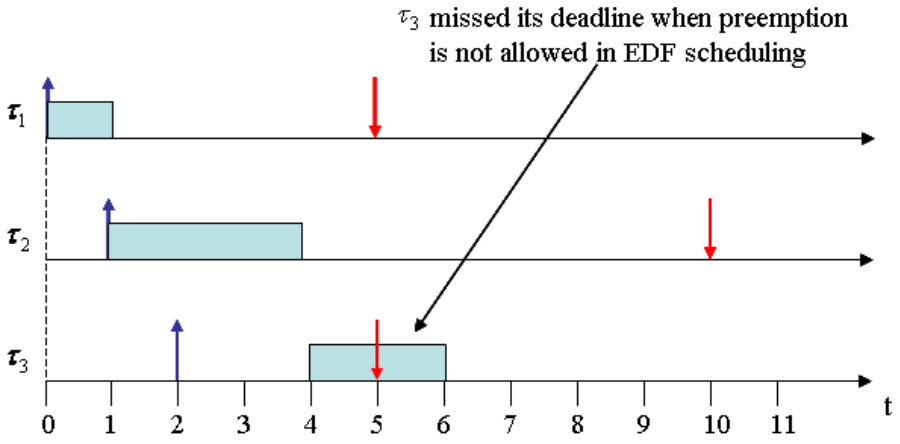

Figure 2.11: EDF schedule in a non-preemptive system. Task τ_2 gets executed first since $A_2 < A_3$, and τ_3 can only begin its execution when τ_2 completes at t=4, thus failing to meet its deadline.

For multiprocessor platforms, EDF is not quite as good an algorithm as it is on uniprocessors. Sufficient conditions and bounds have been obtained for using EDF for scheduling upon multiprocessors. One of the more important results concerning preemptive EDF multiprocessor scheduling was proved by Philips et al (1997).

Theorem 2.1 (from (Philips et al., 1997; Philips et al., 2002)) *If a real-time instance is feasible on m processors, then the same instance will be scheduled to meet all deadlines by EDF on m processors in which the individual processors are* $(2-\frac{1}{m})$ *times as fast as in the original system.*

Goossens et al. (2003) extended the above conditions for periodic task systems on identical multiprocessor platforms.

Theorem 2.2 (from (Goossens et al., 2003)) Periodic *task system* τ *is scheduled to meet all deadlines by EDF on an identical multiprocessor platform comprised of m unit-capacity processors, provided:*

$$U_{sum}(\tau) \le m-(m-1)U_{\max}(\tau) \tag{2.7}$$

For non-preemptive multiprocessor system, Baruah (2006) presented sufficient conditions for determining whether a given periodic task system will meet all deadlines if scheduled non-preemptively upon a multiprocessor platform using the EDF algorithm.

Theorem 2.4 (from (Baruah, 2006)) Any task system τ satisfying the following condition, will successfully scheduled by Non-preemptive EDF to meet all deadlines:

$$V_{sum}(\tau) \le m-(m-1)\times V_{\max}(\tau) \tag{2.8}$$

Where,

$$v(\tau_i,\tau) \overset{def}{=} \frac{e(\tau_i)}{\max(0,\, p(\tau_i)-e_{\max}(\tau))} \tag{2.9}$$

$$V_{sum}(\tau) \overset{def}{=} \sum_{\tau_i \in \tau} V(\tau_i, \tau) \tag{2.10}$$

$$V_{\max}(\tau) \overset{def}{=} \max_{\tau_i \in \tau} V(\tau_i, \tau) \tag{2.11}$$

Above, we have seen several sufficient conditions and schedulability bounds introduced in various investigation of EDF multiprocessor scheduling. However, it is also important to be aware that there exist task systems with very low utilization that are not EDF-schedulable on multiprocessor platforms. This phenomenon of task systems having low utilization but being not schedulable on a multiprocessor system is sometimes known as *Dhall's effect* (Dhall and Liu, 1978). We describe Dhall's effect through the following example.

Example 2.1 Consider a platform with two processors, $P = \{P_1, P_2\}$ and a task system $\tau_i = (A_i, E_i, D_i) = \{\tau_1 = (0,1,50), \tau_2 = (0,1,50), \tau_3 = (0,100,100)\}$.

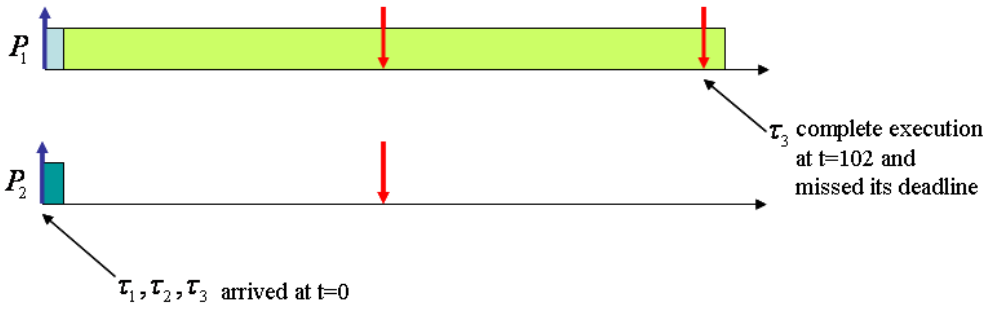

Figure 2.12: An example of Dhall's effect. EDF fails to schedule a task system with utilization 1.04 on two processors.

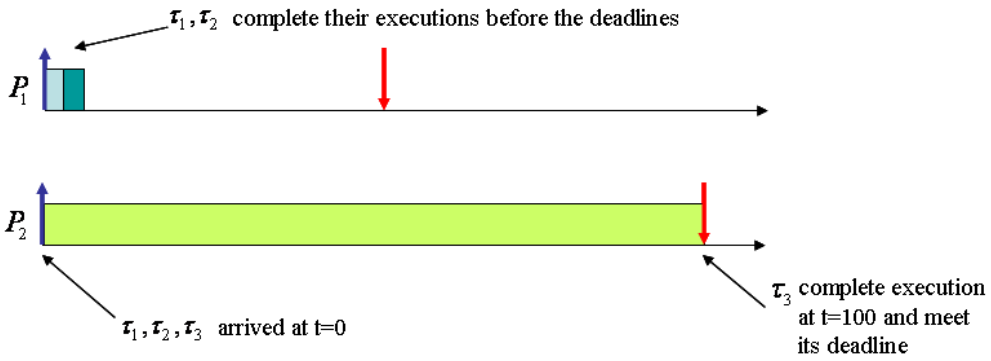

Figure 2.13: An example of two processors platform and a task systems that are schedulable by other scheduling strategies but not schedulable by EDF.

Another situation that is also meaningful to help us understand the behavior of multiprocessor real-time system is that *scheduling anomalies* can happen in multiprocessor scheduling. For instance, Graham (1976) has observed that anomalous results may occur, where increasing the number of processors, reducing the execution times, or weakening precedence constraints, can render a schedulable systems unschedulable.

We would also like to recall a fundamental restriction we mentioned in the earlier chapter. In multiprocessor real-time scheduling, a single job may execute upon at most one processor at any instant in time, even if there are several idle processors available. Perhaps, this restriction makes the scheduling in multiprocessor significantly more complex. Recall that, as Liu observed (Liu, 1969), "*The simple fact that a task can use only one processor even when several processors are free at the same time adds a surprising amount of difficulty to the scheduling of multiprocessors.*" Undoubtedly, the next generation of embedded and real-time systems will demand parallel execution. Looking at this significant need, recently, some researchers have studied extensions to the traditional workload models, to allow for the possibility that a single job may execute simultaneously on multiple processors. We briefly review some of these works in the following section.

2.4 Parallel Execution upon Multiprocessor Real-time Systems

Attempting to address the difficulty in multiprocessor scheduling, some researchers have studied extensions to the workload models traditionally used in real-time scheduling theory, to allow for the possibility that a single job may execute simultaneously on multiple processors. We briefly review a few significant work respects to this attempt.

2.4.1 Dynamic Scheduling Algorithm

Manimaran and Murthy (1998) proposed a dynamic algorithm for non-preemptive scheduling of real-time tasks upon multiprocessor systems. In this work, they introduced an extra notion of a real-time task. Specifically each task τ_i is ***aperiodic*** and is characterized by its arrival time (a_i), ready time (r_i), worst case computation time (c_i^j) and deadline (d_i). c_i^j is the worst case computation time of T_i which is the upper bound on the computation time, when run on j^{th} processors in parallel where $1 \le j \le N$. There are four significant differences of Manimaran and Murthy's work to what we proposed in this thesis:

- They work with *aperiodic* task model.
- When a task is parallelized, all its parallel subtasks (split tasks) have to start at the same time in order to synchronize their executions.
- They equally split a task to j^{th} processors (maximum degree of parallelization permitted that satisfy c_i^j)
- Manimaran and Murthy algorithm is a variant *of myopic algorithm* proposed by Ramamritham et al. (1990). The myopic algorithm is a *heuristic search algorithm* that schedules dynamically arriving real-time tasks with resource constraints.

2.4.2 Work Limited Parallelism

Similarly, Collete et al. (2007, 2008), investigated global scheduling of implicit deadline sporadic task systems with *work-limited* job parallelism upon identical parallel machines. In other words, they allow jobs to be executed on different processors at the very same instant.

In this work, they considered a ***sporadic*** task system. Recall that, in sporadic task model, a task $\tau_i = (C_i, D_i, T_i)$ is characterized by a worst-case execution requirement C_i, a (relative) deadline D_i, and a minimum inter-arrival separation T_i, also known as period. For a task τ_i and m identical processors, they provide an m-tuple of real numbers $\Gamma_i \overset{def}{=} \left(\gamma_{i,1}, \gamma_{i,2}, ...\gamma_{i,m}\right)$ to model job parallelism, with the interpretation that a job of τ_i that executes for t time units on j processors completes $\gamma_{i,j} \times t$ units of execution. They observed that, full parallelism, which corresponds to the case where $\Gamma_i = (1, 2, ..., m)$ is not realistic. That is if full parallelism is allowed, the multiprocessor scheduling problem is equivalent to the uniprocessor one.

Collete et al. assumed that a work-limited job parallelism with the following definition:

Definition (from Collete et al., 2007, 2008): The job parallelism is said to be work-limited if and only if for all Γ_i we have:

$$\forall 1 \le i \le n, \forall 1 \le j < j' \le m, \frac{j'}{j} > \frac{\gamma_{i,j'}}{\gamma_{i,j}}$$

One major difference of Collete et al.'s approach compared to our work is that they assumed an *m*-tuple of real numbers is given to guide them in splitting a job.

2.4.3 Maximum Workload Derivative First with Fragment Elimination (MWF-FE)

Another significant work is by Lee et al. (2003). They proposed an algorithm called "*Maximum Workload derivative First with Fragment Elimination*" (MWF-FE) for the on-line scheduling problem. This algorithm utilizes the property of *scalable* tasks for on-line and real-time scheduling. They assumed tasks are scalable if their computation time can be decreased (up to some limit) as more processors are assigned to their execution. They also defined the total amount of processors time devoted to the execution of a scalable task as the *workload* of the task. They assumed, as the number of processors allocated to a scalable task increases, its computation time decreases but its workload increases because of parallel execution overhead, such as contention, communication, and unbalanced load distribution.

In Lee et al. algorithm, the total workload of all scheduled tasks is reduced by managing processors allocated to the tasks as few as possible without missing their deadlines. As a result, the processors in the system have lesser load to execute the scheduled tasks and can execute more newly arriving tasks before their deadlines. They also defined available processors due to their next allocation assignment as *fragmented* workload and need to eliminate it by allocating more processors to previously scheduled tasks. As more processors are allocated to a task, its computation is completed sooner. Figure 2.14 depicts the idea of minimizing total workload and eliminating a fragmented workload.

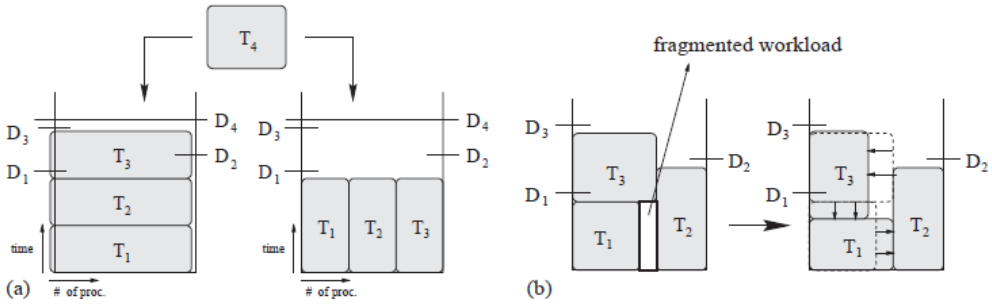

Figure 2.14: (a) Minimizing total workload and (b) eliminating a fragmented workload (Lee et al., 2003)

2.4.4 Divisible Load Theory (DLT)

In a series of papers (2006a, 2006b, 2007a, 2007b, 2007c) Lin et al. have extended the Divisible Load Theory known as DLT (Bharadwaj et al., 2003) to real-time workloads. Since this thesis extends the work of Lin et al. (2006a, 2006b, 2007a, 2007b, 2007c), we will briefly review the fundamental and some current findings of DLT in this section.

"Divisible load theory offers a tractable and realistic approach to scheduling that allows integrated modeling of computation and communication in parallel and distributed computing systems" (Robertazzi, 2003). DLT involves the study of an optimal distribution strategy that distributes loads among a collection of processors that link to each other via a network (Bharadwaj et al., 1996). DLT thus seeks optimal strategies to split divisible loads into chunks/ fractions and send them to the processing nodes with the goal of minimizing the overall completion time (the "makespan").

Loads in DLT can be arbitrarily divided into pieces and distributed among the processors and links in a network system. An arbitrarily divisible load is a load that can be arbitrarily partitioned into any number of load fractions. Applications that have this kind of load are to be found in bioinformatics (e.g protein sequence analysis and simulation of cellular micro physiology), high energy and particle physics (e.g the CMS –Compact Muon Solenoid– and ATLAS –Atoroidal LHC Apparatus– projects), Kalman filtering (Sohn et al., 1998), image processing (Bharadwaj and Ranganath, 2002; Xiaolin et al., 2003), database searching (Drozdwoski, 1997) and multimedia processing (Balafoutis et al., 2003). It is assumed that there are no precedence relations among the distributed loads or to other loads.

Most of the studies done in DLT research involve developing mechanisms for the efficient distribution and allocation of a given load to processors over a network such that all the processors complete processing their assigned sub-loads at the same time – this is sometimes referred to as *the optimality principle* in DLT.

Intuitively, the optimality principle yields a schedule with minimum makespan because any schedule that is not compliant with this principle can be improved by transferring some load from busy processors to idle ones when some processors are idle while others are still busy (Bharadwaj et al., 1996). Optimal strategies for the distribution of loads have been obtained for several network topologies including linear daisy chains (Robertazzi, 1993), star network (Drozdowski and Wolniewicz, 2006), and bus and tree networks (Sohn and Robertazzi, 1993; Bharadwaj et al., 2000; Barlas and Bharadwaj, 2004).

There have been further studies in terms of load distribution policies for two and three dimensional meshes (Drozdowski and Glazek, 1999) and hypercubes (Piriyakumar and Murthy, 1998). In (Sohn and Robertazzi, 1998a, 1998b) the concept of time varying processor speed and link speed are introduced. There also have been study on multi-installment sequential scheduling (Bharadwaj et al., 1995; Barlas and Bharadwaj, 2000; Glazek, 2003; Drozdowski and Lawenda, 2005) and Multi-round algorithms (Yang et al., 2003; Marchal et al., 2005; Yang and Casanova, 2005). Note that none of these deal with real-time workloads.

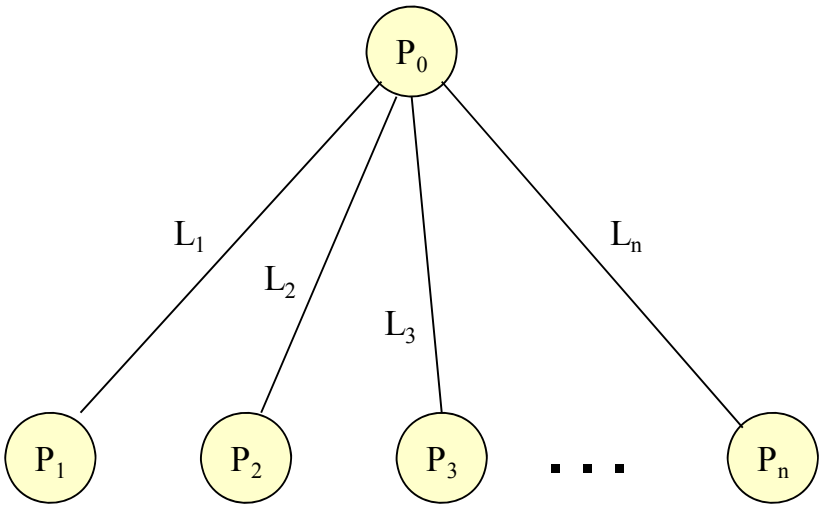

Figure 2.15: Single-Level Tree Network ∑

We will briefly explain the concept of DLT using the single-level tree network. We chose to focus on this network topology due to its wide applicability in real-time systems (due to its applicability, earlier work on RT-DLT, such as the work of Lin et al. (2006a, 2006b, 2007a, 2007b, 2007c)) were also modeled using this network model). Greater details of DLT can be found in Bharadwaj et al. (1996).

As illustrated in Figure 2.15, a single-level tree network, denoted as Σ, consists of (n+1) processors and N links. The root processor P_0 is connected to the child processors $P_1, P_2, ..., P_n$ via links $L_1, L_2, ..., L_n$. The root processor divides the total load into $(n+1)$ fractions, denoted as $\alpha_0, \alpha_1, \alpha_2, ..., \alpha_n$. P_0 will keep its own fraction α_0 and distribute the remaining fractions to the child processors $P_1, P_2, ..., P_n$ in the sequence $1, 2, ..., n$. Each processor will begin execution when it completes receiving its load fraction.

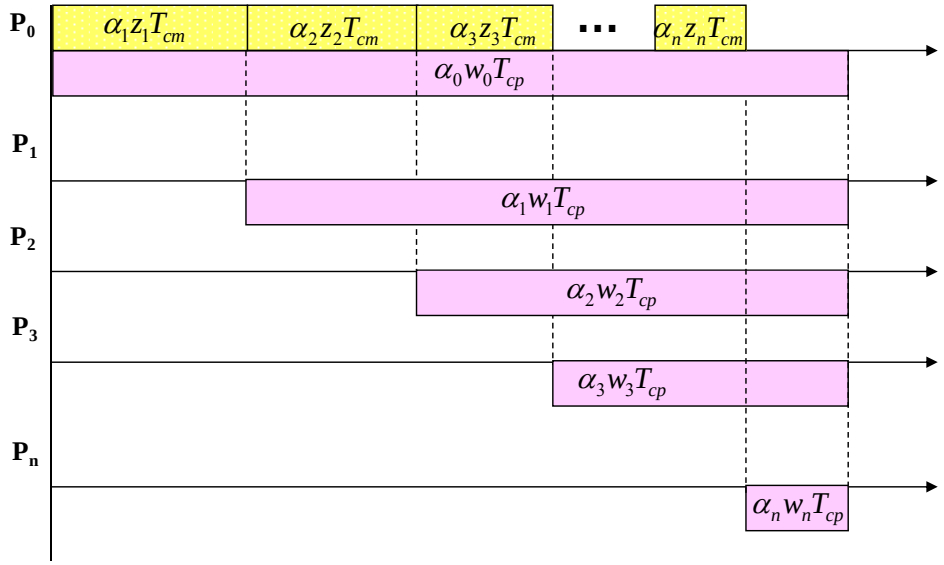

Figure 2.16: Timing Diagram: Single-Level Tree Network with Front-End

In the DLT literature, strategies have been proposed to deal with processors both with and without front-end processing capabilities. In the case of processors with front end processing capabilities, it is assumed that some of the processors in the network are equipped with co-processors so that they are able to process their assigned load and communicate simultaneously. Figure 2.16 shows an example of timing diagram of communication and computation done in a single-level tree network with front-end processing capabilities. Note that, the root processor P_0 starts its own computation of fraction α_0 while sending the remaining fractions to the child processors.

In clusters without front-end processing capabilities, processors can only compute or communicate at any given instant in time. Figure 2.17 depicts an example of timing diagram of communication and computation done in a single-level tree network without front-end processing capabilities. In this network, the root processor P_0 will only start its own computation of fraction α_0 after it has finished sending the other fractions to the child processors.

In most DLT literature, the following notations (Bharadwaj et al., 1996) are used:

$$w_i = \frac{\text{Time taken by processor } P_i \text{ to compute a given load}}{\text{Time taken by a standard processor to compute the same load}}$$

$$T_{cp} = \text{Time taken to process a unit load by the standard processor}$$

$$z_i = \frac{\text{Time taken by link } L_i \text{ to compute a given load}}{\text{Time taken by a standard processor to compute the same load}}$$

$$T_{cm} = \text{Time taken to communicate a unit load on a standard link}$$

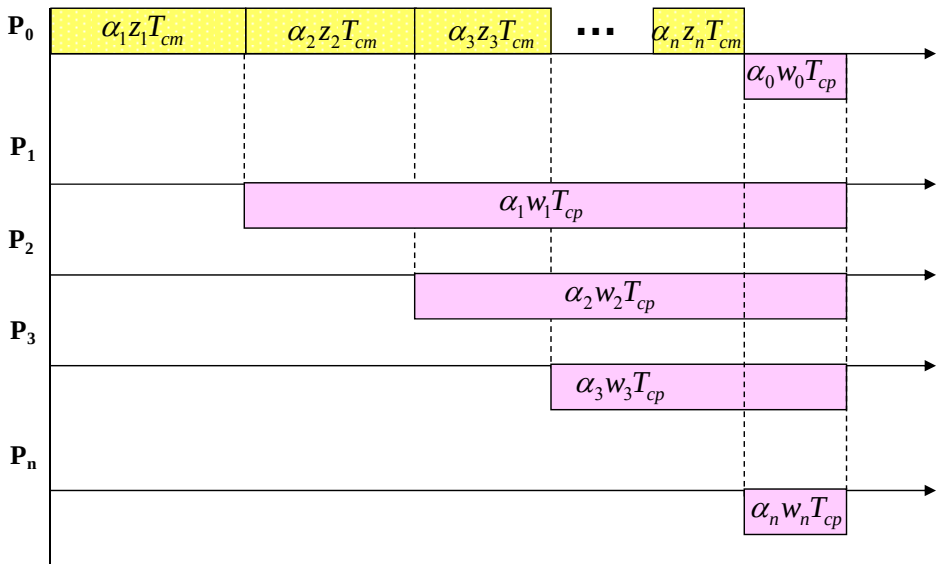

Figure 2.17: Timing Diagram: Single-Level Tree Network without Front-End

2.4.5 Real-time Divisible Load Theory (RT-DLT)

The promising strategy of DLT has recently attracted the attention of the Real-time system research community. Lin, Lu, Deogun, and Goddard (2007a, 2007b, 2007c, 2007d, and 2007e) have applied results from Divisible Load Theory (DLT) to the scheduling of arbitrarily divisible real-time workloads upon multiprocessor. To the best of our knowledge, these are among the earliest attempts to study the application of DLT to real-time computing system. Among other results, they obtained elegant solutions to the following two problems:

i. Given a divisible job and a specified number of processors upon which it may execute, determine how this job should be divided among the assigned processors in order to minimize the time at which it completes execution.

ii. Given a divisible real-time job, determine the minimum number of processors that must be assigned to this job in order to ensure that it complete by its deadline.

In the initial work of Lin et al (2006a, 2006b, 2007a), they investigated the use of DLT to enhance the quality of service (QoS) and provide performance guarantees in cluster computing environments. The main contributions made in this work were in providing the first formal definitions of RT-DLT, and in proposing a scheduling framework integrating DLT and EDF (Earliest Deadline First) scheduling. They conducted series of extensive simulation experiments and based upon the outcome of these experiments, they made some anomaly observations which, they note, seem to disagree with previously-published results in conventional (non real-time) DLT.

Lin et al. (2007b) further extended their work on RT-DLT. They studied a problem where if the required number of processors to process a job are not available and the job waits for some currently running jobs to finish and free additional processors. This essentially causes a waste of processing power as some processors are idle within the waiting period. They refer this problem as Inserted Idle Times (IITs) problem. They proposed a new real-time divisible load scheduling approach that utilizes IITs. Two contributions were made in this work. First, they mapped a cluster with different processor available times to a *heterogeneous* cluster of "virtual processors" which may each have different processing capabilities but all of which are assumed to be available simultaneously. A DLT heterogeneous model is then applied to guide task partitioning, to derive a task execution time function and to approximate the minimum number of processors required to meet a task deadline. Second, they proved that executing the partitioned subtasks in the homogenous cluster at different processor available times leads to completion times no later than the estimates. This result is then applied to develop a new divisible load scheduling algorithm that uses IITs and provides real-time guarantees.

Using the same motivation in (2006a, 2006b, 2007a, 2007b), Lin et al. (2007c) proposed another strategy to further make use of IITs. They claimed that when certain conditions hold, the enhanced algorithm can optimally partition and schedule jobs to fully utilize IITs. Two contributions are made in this work. First, they proposed a new partitioning approach to fully utilize IITs and investigated its applicability constraints. Second, they integrated this with their previous work (2006a, 2006b, 2007a, 2007b) and proposed a new real-time scheduling algorithm.

2.4.6 Extending Real-time Divisible Load Theory (RT-DLT)

We investigated the anomaly observations found in Lin et al. (2006a, 2006b, 2007a) and provide theoretical explanation using recent results from real-time multiprocessor scheduling theory. We presented this work in greater detail in Chapter 3. We also studied the work of Lin et al. (2007b, 2007c) and observed that their scheduling algorithms are inefficient. Thus, we proposed two alternatives algorithms that significantly improved the previous algorithms by Lin et al. (2007b, 2007c). We presented these works in Chapter 4 and Chapter 5. Figure 2.18 depicts the research roadmap of our work in extending RT-DLT of Lin et al. (2006a, 2006b, 2007a, 2007b, 2007c).

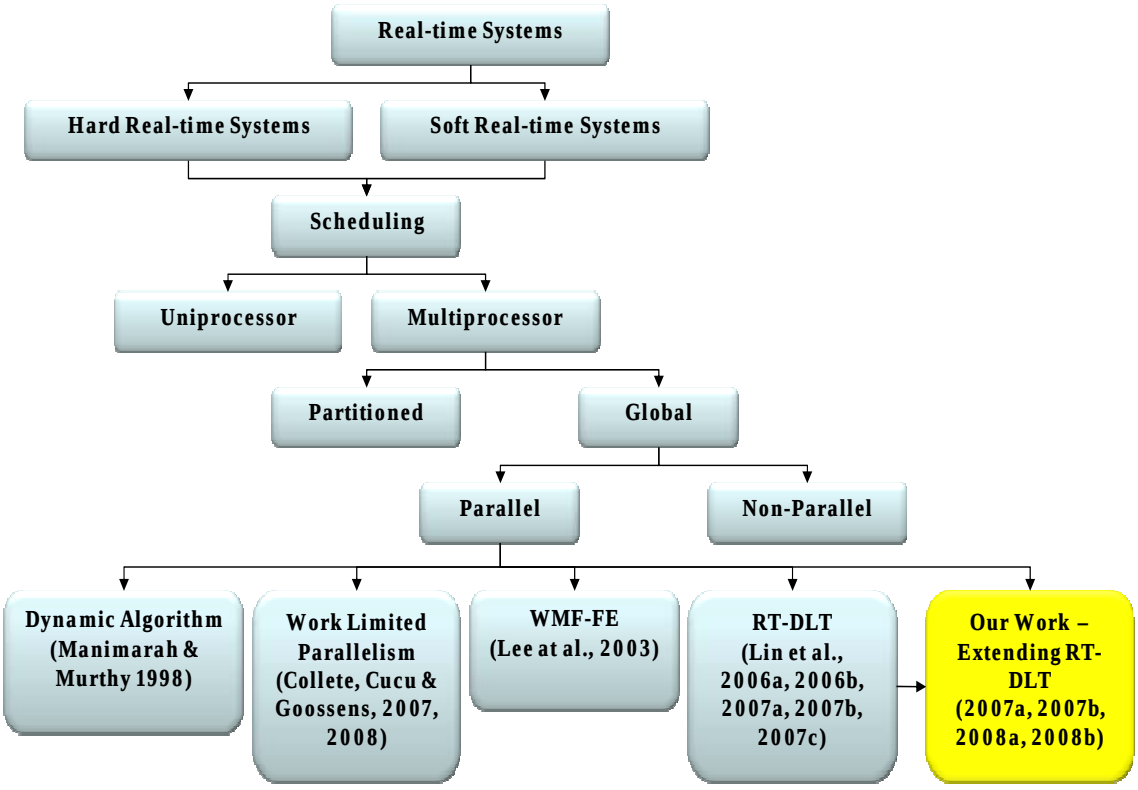

Figure 2.18: Research Roadmap

CHAPTER 3

DEADLINE-BASED SCHEDULING OF DIVISIBLE REAL-TIME LOADS

3.1 Introduction

The previous chapter highlighted the shortcomings of the traditional workload models used in the real-time multiprocessor scheduling. We have also briefly introduced some of the approaches that had been proposed to address these difficulties. Among the significant approaches is the application of DLT to real-time workloads. In this chapter, we will report our first extension of the initial work of Lin et al. (2006a, 2006b, 2007a). Lin et al. (2006a, 2006b, 2007a) proposed a scheduling framework integrating DLT and EDF. They conducted a series of extensive simulation experiments comparing the proposed framework with other approaches. Based upon the outcome of these experiments, they made some observations which, they note, seems to disagree with previously-published results in conventional (non real-time) DLT. Our results here provide a theoretical analysis of some of these observations, and thereby help identify the kinds of systems for which these observations hold.

The remainder of this chapter is organized as follows. In Section 3.2, we discuss the overall picture of how Lin et al. (2006a, 2006b, 2007a) applied DLT to real-time workloads, and their apparently anomalous observations. In Section 3.3 we present our analysis and theoretical explanation by using some fundamental theories of real-time scheduling. We conclude this chapter in Section 3.4.

3.2 Application of DLT to Real-time workloads

Now let us recall the fundamental definitions used in RT-DLT:

Task Model

Each divisible job J_i is characterized by a 3-tuple (A_i, σ_i, D_i), where $A_i \geq 0$ is the arrival time of the job, $\sigma_i > 0$ is the total load size of the job, and $D_i > 0$ is its relative deadline, indicating that it must complete execution by time-instant $A_i + D_i$.

System Model

The computing cluster used in DLT is comprised of a head node denoted P_0, which is connected via a switch to ***n*** processing nodes denoted $P_1, P_2, ..., P_n$. All processing nodes have the same computational power, and all the links from the head to the processing nodes have the same bandwidth.

Assumptions

- The head node does not participate in the computation – its role is to accept or reject incoming jobs, execute the scheduling algorithm, divide the workload and distribute data chunks to the processing nodes.

- Data transmission does not occur in parallel: at any time, the head node may be sending data to at most one processing node. However, computation in different processing nodes may proceed in parallel to each other.

- The head node, and each processing node, is non-preemptive. In other words, the head node completes the dividing and distribution of one job's workload before considering the next job, and each processing node completes executing one job's chunk before moving on to the chunk of any other job that may have been assigned to it.

- Different jobs are assumed to be independent of one another; hence, there is no need for processing nodes to communicate with each other.

According to DLT, linear models are used to represent transmission and processing times (Bharadwaj et al., 2003). The computation time of a load of size σ_i is equal to $\sigma_i \times C_m$, while the processing time is equal to $\sigma_i \times C_p$, where C_m is a cost function for transmitting a unit workload and C_p is a cost function for processing a unit workload. For the kinds of applications considered in (2006a, 2006b, 2007a) the output data is just a short message and is assumed to take negligible time to communicate. To summarize, a computing cluster in DLT is characterized by a 3-tuple (n, C_p, C_m) where $\boldsymbol{n}$ denotes the number of processing nodes, and C_p and C_m denote the amount of time taken to process and transmit a unit of work, respectively. For a given computing cluster, let β be defined as follows:

$$\beta \overset{def}{=} \frac{C_p}{(C_p + C_m)} \tag{3.1}$$

3.2.1 Scheduling Frameworks

To evaluate the applicability of DLT for real-time workloads, Lin et al. (2006a, 2006b, 2007a) proposed a set of scheduling frameworks combining scheduling algorithms, node assignment strategies, and task partitioning strategies. Figure 3.1 depicts an abstraction of these frameworks.

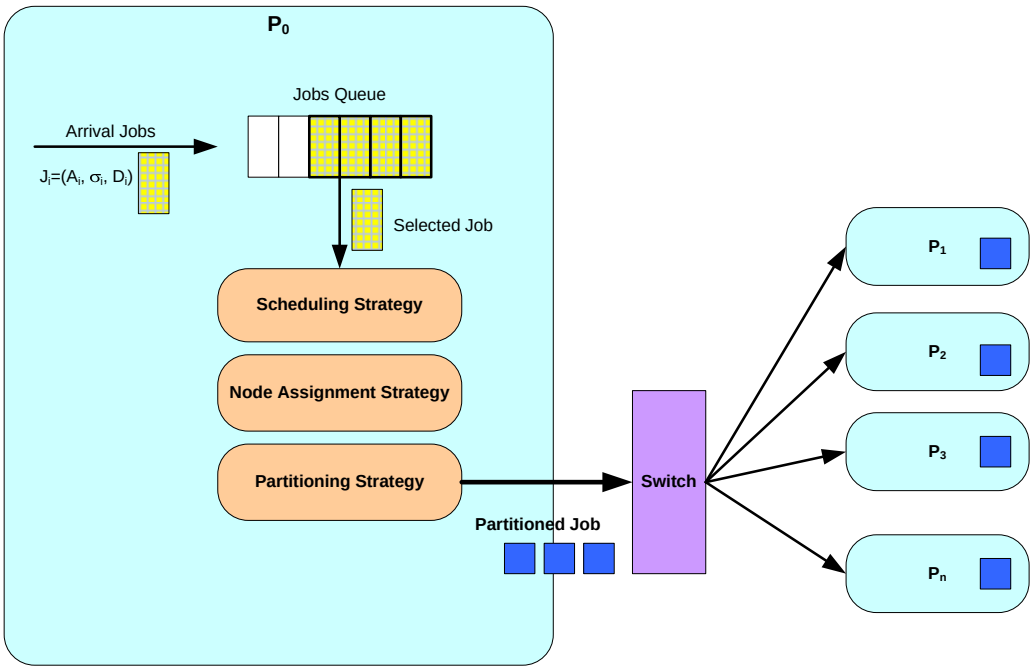

Figure 3.1: The abstraction of RT-DLT framework.

As shown in Figure 3.1, each arrival jobs $J_i = (A_i, \sigma_i, D_i)$ will be placed in the *Jobs Queue*. The *Scheduling Strategy* decide the execution order, the *Node Assignment Strategy* compute the number of processors needed for a task execution and *Partitioning Strategy* distribute a job into subtasks and send each subtask to each processors via a switch.

3.2.1.1 Scheduling Algorithms

The scheduling algorithms investigated in the framework are:

- **First-in First-out (FIFO)** Jobs are considered by the head node in the order in which they arrive: job $J_i = (A_i, \sigma_i, D_i)$ is considered before $J_k = (A_k, \sigma_k, D_k)$ if $A_i < A_k$ (ties broken arbitrarily). Observe that FIFO is not really a “real-time" strategy, in the sense that it does not take into account the temporal constraints on the system.

- **Earliest Deadline First (EDF)** Jobs are considered by the head node in the order of their (absolute) deadlines: if jobs $J_i = (A_i, \sigma_i, D_i)$ and $J_k = (A_k, \sigma_k, D_k)$ are both awaiting service and the head node chooses J_i, then $A_i + D_i < A_k + D_k$ (ties broken arbitrarily).

- **Maximum Workload-derivative First (MWF)** This algorithm (Lee et al., 2003) is evidently one that is used extensively in conventional (non real-time) DLT; since (Lin et al., 2006a, 2006b, 2007a) concluded that this is not particularly suited for real-time computing; we will not discuss it further in this chapter.

3.2.1.2 Node Assignment Strategies

The scheduling algorithm (discussed above) is used to determine which waiting job is next considered for dispatch by the head node. In addition, the head node is responsible for determining how many processing nodes to assign to this selected job --- this is the responsibility of the node assignment strategy.

The node assignment strategies considered by Lin et al. (2006a, 2006b, 2007a) are:

- **All nodes (AN).** All *n* processing nodes in the cluster are assigned to one single job currently selected by the head node. This strategy thus tries to finish the current task as early as possible, and then continue with other waiting jobs.

- **Minimum nodes (MN)**. A job is assigned the minimum number n^{min} of processing nodes needed in order to complete prior to its deadline, thereby saving the remaining processors for other jobs.

In essence, AN tends to treat the cluster as a uniprocessor platform, as one task is assigned to all available processors. In contrast, MN attempts to maximize the degree of parallelism in processing the workload by assigning the fewest possible processors to each job, and a larger number of jobs can be processed simultaneously in parallel.

3.2.1.3 Partitioning Strategies

After selecting the next job to execute and the number of processing nodes to assign to it, the head node next determines how to partition the job's workload among the assigned processors. A naive approach may be to partition the workload equally among the assigned processing nodes -- this is referred to as the *Equal Partitioning Rule* (EPR). However, EPR is provably non-optimal for clusters in which the communication cost C_m is non-zero. Instead, Lin et al. (2006a, 2006b, 2007a) derived a superior partitioning strategy, the *Optimal Partitioning Rule* (OPR).

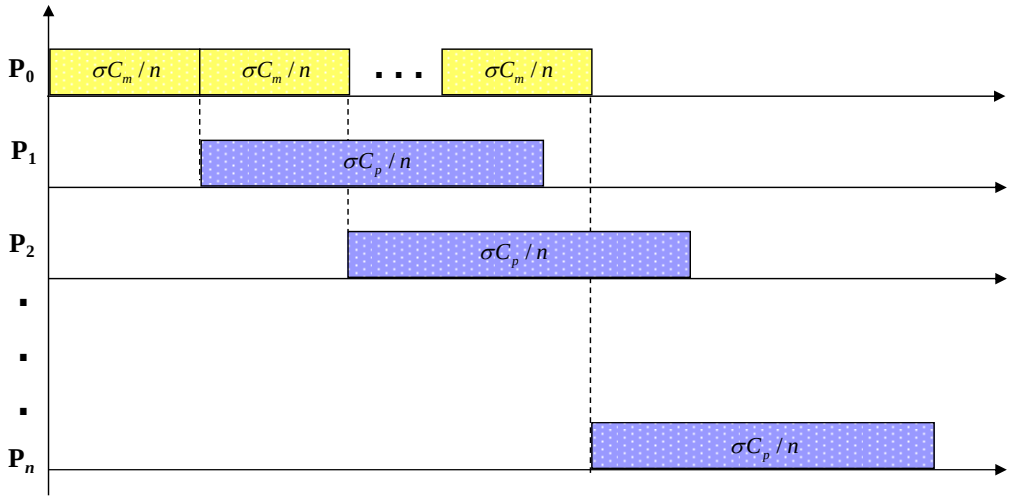

Figure 3.2: Timing diagram for EPR-based partitioning

Figure 3.2 illustrates the time diagram produced by the EPR-based partitioning. This diagram shows the cost functions: the data transmission time on each link and the data processing time on each processing node. The data transmission time on each link is defined as $\sigma C_m / n$ and the data processing time on each node is defined as $\sigma C_p / n$, where σ is the workload size, C_m is the cost of transmitting a unit workload, C_p is the cost of processing a unit workload and n is the total number of processing nodes allocated to the workload. By analyzing this time diagram the workload execution time is defined as:

$$\xi(\sigma, n) = \sigma C_m + \frac{\sigma C_p}{n} \tag{3.2}$$

The time diagram produced by the OPR-based partitioning strategy is shown in Figure 3.3. In DLT it is known that the completion time of a job on a given set of processing nodes is minimized if all the processing nodes complete their execution of the job at the same instant. OPR is based upon this DLT optimality principle.

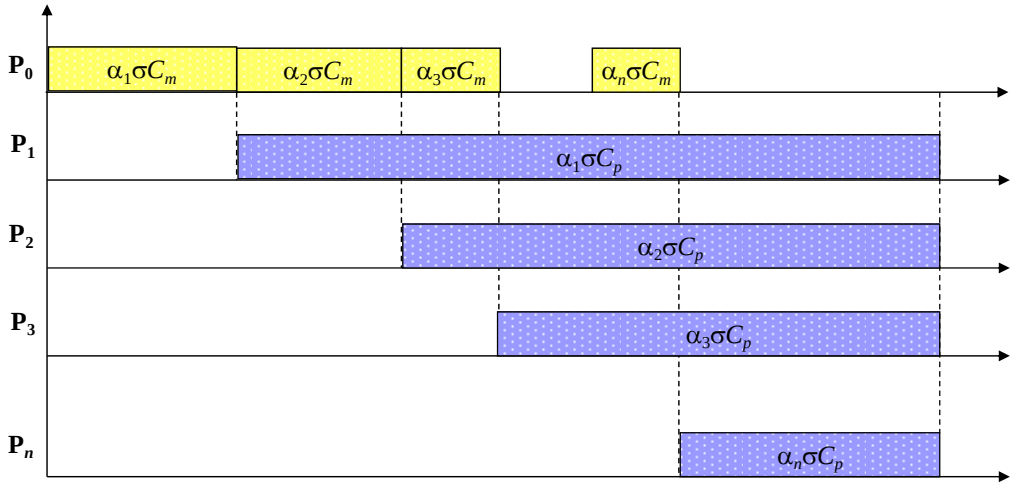

Figure 3.3: Timing diagram for OPR-based partitioning

For a given divisible workload (A_i, σ_i, D_i) and a given number of processing nodes *n*, let α_j denote the fraction of the load assigned to the *j*'th processing node, $0 < \alpha_j \leq 1$, $\sum_{j=1}^{n} \alpha_j = 1$, C_m is the cost of transmitting a unit workload and C_p is the cost of processing a unit workload. Thus, we will have the following cost function: the data transmission time on j^{th} link is $\alpha_j \sigma C_m$ and the data processing time on j^{th} node is $\alpha_j \sigma C_m$. Using the DLT optimality principle, the workload execution time can be defined as follows:

$$\begin{aligned} \xi(\sigma, n) &= \alpha_1 \sigma C_m + \alpha_1 \sigma C_p \\ &= (\alpha_1 + \alpha_2)\sigma C_m + \alpha_2 \sigma C_p \\ &= (\alpha_1 + \alpha_2 + \alpha_3)\sigma C_m + \alpha_3 \sigma C_p \\ &\ldots \\ &= (\alpha_1 + \alpha_2 + \ldots + \alpha_n)\sigma C_m + \alpha_n \sigma C_p \end{aligned} \tag{3.3}$$

Lin et al. (2006a, 2006b, 2007a) then show that an application of the optimality principle results in the following values for the α_i 's:

$$\alpha_1 = \frac{1-\beta}{1-\beta^n} \tag{3.4}$$

$$\alpha_j = \beta^{j-1}\alpha_1, \text{ for j > 1} \tag{3.5}$$

The job execution time is also defined as:

$$\xi(\sigma, n) = \frac{1-\beta}{1-\beta^n}\sigma(C_p + C_m) \tag{3.6}$$

Assuming a job has a start time s , where $s \geq A$, then to meet the job's deadline, it is necessary that:

$$s + \xi(\sigma, n) \leq A + D \tag{3.7}$$

Using the equation (3.5), Lin et al. (2006a, 2006b, 2007a) derived the minimum number of processors $n^{\min}$ needed to complete this job by its deadline:

$$n^{\min} = \left\lceil \frac{\ln \gamma}{\ln \beta} \right\rceil \tag{3.8}$$

Where, $\gamma = 1 - \frac{\sigma C_m}{A + D - s}$ and β is as defined in equation (3.1).

To further understand the application of DLT to real-time cluster-based scheduling, Lin et al. (2006a, 2006b, 2007a) also derived the time function and $n^{\min}$ for EPR-based partitioning. Recall that the job execution time is defined as $\xi(\sigma, n) = \sigma C_m + \frac{\sigma C_p}{n}$. Assuming that a job has a start time s_i, then the task completion time is $C(n) = s_i + \xi(\sigma, n)$, then to meet the job's deadline, it is necessary that $C(n) \leq A + D$. That is:

$$s + \sigma C_m + \frac{\sigma C_p}{n} \leq A + D \tag{3.9}$$

Thus,

$$\frac{\sigma C_p}{A + D - s - \sigma C_m} \leq n \tag{3.10}$$

Using the equation (3.9), Lin et al. (2006a, 2006b, 2007a) derived the minimum number of processors $n^{\min}$ needed to complete this job by its deadline as:

$$n^{\min} = \left\lceil \frac{\sigma C_p}{A + D - s - \sigma C_m} \right\rceil \tag{3.11}$$

3.2.2 An Apparent Anomaly

Using the scheduling framework described above, Lin et al. (2006a, 2006b, 2007a) generated 10 scheduling algorithms: EDF-OPR-MN, EDF-OPR-AN, EDF-EPR-MN, EDF-EPR-AN, FIFO-OPR-MN, FIFO-OPR-AN, FIFO-EPR-MN, FIFO-EPR-AN, MWF-OPR-MN and MWF-EPR-MN. Based upon extensive simulations over a wide range of system parameters, Lin et al. (2006a, 2006b and 2007a) concluded that the **EDF-OPR-AN** combination performs the best from among the 10 different combinations they evaluated. It is reasonable that EDF be the scheduling algorithm in the optimal combination since it is the only “real-time" cognizant scheduling algorithm among the three considered. Similarly, it is not surprising that OPR be the optimal workload partitioning strategy, given that the other strategy evaluated in the simulation experiments – EPR – is provably inferior.

However, they found the conclusion that the **AN** node-assignment strategy is superior to the **MN** strategy somewhat surprising, since prior published results from conventional DLT strongly indicate that MN assigning the minimum number of processing nodes per job and -- hence maximizing parallelism – is a superior strategy to AN.

3.3 A Comparison of EDF-OPR-AN and EDF-OPR-MN

As stated in the previous section, Lin et al. (2006a, 2006b, 2007a) were somewhat surprised to observe that EDF-OPR-AN seems to perform better than EDF-OPR-MN for real-time systems. In this section, we attempt to explain these apparently anomalous findings. Observe that the AN strategy, by assigning all the processing nodes to a single job at each instant in time, reduces to a variant to *uniprocessor* EDF, while the MN strategy, which attempts to maximize inter-job parallelism, more closely resembles *multiprocessor* EDF. Hence it behooves us to first review known results concerning uniprocessor and multiprocessor EDF, which we do in Section 3.3.1 below.

In Section 3.3.2, we take a closer look at the role played by the head node. Regardless of the node assignment strategy used, each job must first be processed by the head node. If a significant fraction of the time taken to process a job is spent at the head node, then the overall processing of the job is more likely to resemble the uniprocessor scheduling of the head node bottleneck. We identify conditions under which such a bottleneck occurs, thereby identifying conditions in which EDF-OPR-AN is a particularly appropriate scheduling framework.

In Section 3.3.3, we study conditions under which the head node is not a bottleneck. Under such conditions, the tradeoff between the AN and MN strategies is highlighted by considering the cost of executing a job under both strategies, where the cost is defined to be the product of the number of processors used and the total execution time. We will see that the AN strategy has a greater cost per job. But, it will turn out that the conditions that result in the head node not becoming a bottleneck are also responsible for ensuring that the cost of the AN strategy does not exceed the cost of the MN strategy by too much. This fact, in conjunction with the well-known superior behavior of uniprocessor EDF as compared to multiprocessor EDF, results in EDF-OPR-AN once again performing better than EDF-OPR-MN.

3.3.1 Uniprocessor and multiprocessor EDF Scheduling of "traditional" jobs

With respect to traditional (as opposed to divisible) workloads, as we mentioned in the earlier chapter, EDF is known to be an excellent scheduling algorithm for uniprocessor platforms under a wide variety of conditions. For preemptive systems of independent jobs, it has been shown (Dertouzos, 1974) to be optimal in the sense that if any scheduling algorithm can schedule a given system to meet all deadlines, then EDF, too, will meet all deadlines for this system.

For non-preemptive systems, too, it has been shown (Jeffay, Stanat and Martel, 1991) to be optimal under specific well defined conditions. Furthermore, efficient tests have been designed for determining whether given systems are successfully scheduled or not, by both preemptive and non-preemptive uniprocessor EDF.

For multiprocessor platforms, EDF is not quite as good an algorithm. It is provably not optimal for scheduling collections of independent jobs. Recall, for instance, Theorem 2.1 in Chapter 2: any system of independent jobs for which a schedule meeting all deadlines exists on an m-processor platform can be scheduled to meet all deadlines by EDF on an m-processor platform in which each processor is $\left(2-\frac{1}{m}\right)$ times as fast. This bound is known to be tight, in the sense that systems have been identified that are feasible on m-processor platforms but which EDF fails to successfully schedule on m-processor platforms in which the individual processors are less $\left(2-\frac{1}{m}\right)$ that times as fast. Similarly pessimistic results are known for EDF-scheduling on multiprocessor platforms in which different processors have different speeds of computing capacities, as well as for non-preemptive scheduling.

A further known fact (Baker and Baruah, 2007) concerning multiprocessor EDF scheduling will prove useful to us. Let us recall the *density* of a (traditional) job $j_i = (A_i, C_i, D_i)$ to be the ratio of its execution requirement to its relative deadline $\delta(j_i) = (C_i / D_i)$**,** and the system density of a system of jobs to be the largest density of any job in the system. It has been observed that the larger the system density, the poorer the performance of multiprocessor EDF tends to be in scheduling that system of jobs. More specifically, it has been observed (Baruah et. al, 1991) that the Effective Processor Utilization (EPU) --- the fraction of the computing capacity of the computing platform that is guaranteed to be devoted to executing jobs that do meet their deadlines --- tends to decrease with increasing system density, when multiprocessor EDF is the scheduling algorithm used.

3.3.2 When the head node is a bottleneck

We now return to the divisible load model. Depending on the node-assignment strategy used, multiple jobs may be executing simultaneously. However, each job J_i, with workload σ_i must first be processed by the head node. Now all that the head node does is distributing the workload to the processing nodes. But since it does so sequentially, the amount of time needed to complete this data distribution is equal to $(\sigma_i \times C_m)$.

It therefore follows that $(\sigma_i \times C_m)$ is a lower bound on the amount of time taken by the bottleneck head node to process a job of workload size σ_i, regardless of which node assignment strategy is used. This leads to the following observation:

Observation 1 *For a given sequence of divisible jobs* $\{(A_i, \sigma_i, D_i)\}_{i\geq 1}$ *to be schedulable under any node assignment strategy, it is necessary that the sequence of traditional jobs* $\{=(A_i, \sigma_i \times C_m, D_i)\}_{i\geq 1}$ *, be schedulable on a uniprocessor.*

Equation 3.6 gives the amount of time $\xi(n)$ needed to execute a job with workload σ upon *n* processors under the OPR partitioning strategy. Using the definition of β in (Equation 3.1) in Equation 3.6, we obtained:

$$\begin{aligned}\xi(n) &= \frac{1-\beta}{1-\beta^n}\sigma(C_p + C_m) \\ &= \frac{\sigma}{1-\beta^n}\left(1 - \frac{C_p}{C_p + C_m}\right)(C_p + C_m) \\ &= \frac{\sigma}{1-\beta^n}\left(\frac{C_m}{C_p + C_m}\right)(C_p + C_m) \\ &= \frac{\sigma C_m}{1-\beta^n} \qquad (3.12)\end{aligned}$$

Since C_m denotes the rate at which data is transmitted from the head node to the processing nodes, σC_m represents a lower bound on the amount of time needed to complete execution of a job of size σ. Equation 3.12 above tells us that the amount of time taken to complete the execution of a job when assigned n processing nodes is equal to this lower bound, inflated by a factor $\frac{1}{(1-\beta^n)}$. Hence if $\frac{1}{(1-\beta^n)}$ is very small, then the time taken to complete the execution of this job is very close to the lower bound. Keeping this observation in mind, it become our interest to evaluate the behavior of EDF-OPR-AN.

In this strategy, at each instant in time all *N* processing nodes are devoted to executing one job – the one currently selected by the head node. Hence the duration of time devoted to executing a job of workload size σ is equal to $(\sigma C_m) \times (1/(1-\beta^n))$, and no other jobs get executed at all during this time. This leads to Observation 2 below:

Observation 2 *For a given sequence of divisible jobs* $\{(A_i, \sigma_i, D_i)\}_{i \geq 1}$ *to be schedulable under the EDF-OPR-AN strategy, it is* <u>*sufficient*</u> *that the sequence of traditional jobs* $\{=(A_i, \sigma_i \times C_m \times (1/(1-\beta^n)), D_i)\}_{i \geq 1}$, *be schedulable on a uniprocessor.*

Recall that in DLT, a platform (also known as a computing cluster) is characterized by the 3-tuple (n, C_p, C_m), with *n* denoting the number of processing nodes in the cluster and C_p and C_m denoting the amount of time taken to process and transmit a unit of work, respectively. On the basic of Observations 1 and 2, we can identify certain computing clusters upon which EDF-OPR-AN is likely to perform particularly well in comparison to other node-assignment strategies.

Lemma 1 *EDF-OPR-AN is a particularly appropriate scheduling framework for computing clusters in which* $\left(\frac{C_p}{C_p + C_m}\right)^n$ *is small.*

Proof: We will show that the sufficient condition of Observation 2 is very close to the necessary condition of Observation 1 when $\left(\frac{C_p}{C_p + C_m}\right)^n$ is small. Indeed, observe that $\beta \overset{def}{=} \left(\frac{C_p}{C_p + C_m}\right)$. Hence, $\frac{1}{(1-\beta^n)}$ decreases with decreasing $\left(\frac{C_p}{C_p + C_m}\right)^n$, becoming arbitrarily close to 1 as $\left(\frac{C_p}{C_p + C_m}\right)^n$ becomes arbitrarily small. And for values of $\frac{1}{(1-\beta^n)}$ close to 1, the traditional jobs referred to in the statement of Observation 2 become almost identical to the ones in the statement of Observation 1.

Since schedulability of the jobs in the statement of Observation 1 is necessary for the DLT system to be schedulable, it follows that the sufficient condition for EDF-OPR-AN is close to the *necessary* condition, and hence EDF-OPR-AN is close to optimal.

To understand the implications of Lemma 1, let us consider computing clusters in which $C_p \ll C_m$ (i.e., C_p is a lot smaller than C_m) or $C_p \approx C_m$ (i.e., the values of C_p and C_m are of comparable magnitude). We note that these seem reasonable scenarios since:

i) Data-transmission rates in computing clusters are typically far slower than processing rates, rather than the other way around.

ii) For the types of DLT systems considered by Lin et al. (2006a, 2006b, 2007a), recall that it is assumed that a linear model is used to represent processing cost – the processing time for a workload of size x is assumed to equal $x \times C_p$. Under such an assumption, it follows that the processing algorithm that is being implemented, and executed upon the processing nodes, is a linear-time algorithm. This argument rules out the possibility that the algorithm being implemented has such a high computational complexity that data-transmission costs are actually dominated by processing costs.

iii) And finally, even for those rare example clusters in which C_p is currently $\gg C_m$, technological trends (as exemplified by Moore's Law) are such that C_p tends to decrease at a far greater rate than C_m. As a consequence, it is highly likely that C_p will become $\approx C_m$ in the not-too-distant future even for these clusters.

For instance if $C_p \leq C_m$, it follows that $\beta \leq 0.5$. Table 3.1 bounds the factor by which the execution requirement of each job gets inflated beyond its lower bound, according to Observation 2:

Table 3.1: Bound on Inflation Factor

n	**BOUND ON INFLATION FACTOR**
3	1.15
4	1.07
5	1.03
10	1.001

As the table above indicates, for a computing cluster with 5 processing nodes $(n = 5)$, the execution requirement of each traditional job to which each divisible job may be considered mapped is within 3% of its lower bound, when EDF-OPR-AN is the scheduling framework used. For n = 10 the inflation factor is negligible – less than one-tenth of one percent. Based upon these numbers and the known optimality of uniprocessor EDF (Jeffay et al., 1991), we may conclude that for computing clusters with $C_p \approx C_m$ and $n \geq 5$, EDF-OPR-AN offers near-optimal performance.

3.3.3 When the head node is not a bottleneck

As stated in Section 3.1.2, Lin et al. (2006a, 2006b, 2007a) were somewhat surprised to observe that, contrary to expectations (based upon results from conventional DLT), the EDF-OPR-AN strategy seems to perform better than the EDF-OPR-MN strategy for real-time systems. In Section 3.2.2 above, we provided part of the explanation for the apparent anomaly identified by Lin et al. (2006a,

2006b, 2007a), by identifying certain conditions in which EDF-OPR-AN is a particularly appropriate scheduling framework.

We now examine the situation when these conditions are not satisfied. For these situations, we obtain a different set of reasons to explain EDF-OPR-AN's superior behavior. Now, the major difference between an “AN”(all nodes) and “MN” (minimum number of nodes) strategy is highlighted by the *cost* of executing a job, where the cost is defined to be the product of the number of processors used and the total execution time:

$$\begin{aligned} x(n) &= \xi(n) \times n \\ &= \sigma C_m \times \frac{n}{1-\beta^n} \end{aligned} \tag{3.13}$$

Although Equation 3.8 showed that increasing the number n of processing nodes assigned to a job decreases the time $\xi(n)$ needed to execute it, Equation 3.13 above illustrates that the cost $x(n)$ of executing a job increases with increasing n. Consequently, an “MN” strategy incurs lower overall cost by minimizing the number of processing nodes assigned to a job, as compared to an “AN” strategy. For example, we would expect an “MN" strategy to use a smaller fraction of the computing capacity of the cluster as compared to an “AN” strategy, in order to actually execute any particular job.

Contrasted against this greater efficiency of the “MN” strategy is the well-known superior performance of uniprocessor EDF as opposed to multiprocessor EDF. As explained in Section 3.2.1, uniprocessor EDF is characterized by a significantly greater Effective Processor Utilization (EPU) than multiprocessor EDF. Since EDF-OPR-AN reduces to uniprocessor EDF while EDF-OPR-MN reduces to multiprocessor EDF, the tradeoff is that while EDF-OPR-MN *incurs lower costs in actually executing a job, the fraction of the platform capacity that is used for such execution is also correspondingly lower than for* EDF-OPR-AN.

In the remainder of this section, we compare the relative effects of these two factors, and use this comparison to explain the apparently anomalous empirical observations of Lin et al. (2006a, 2006b, 2007a).

Table 3.2: Cost, for selected values of β and n (assuming $\sigma C_m = 1$).

n	***β***					
	0.75	**0.8**	**0.9**	**0.95**	**0.99**	**0.999**
1	4.00	5.00	10.00	20.00	100.00	1000.00
3	5.19	6.15	11.07	21.03	101.01	1001.00
4	5.85	6.78	11.63	21.56	101.51	1001.50
5	6.56	7.44	12.21	22.10	102.02	1002.00
10	10.60	11.20	15.35	24.92	104.58	1004.51

§1. The relative cost. The case of interest is when $C_p \gg C_m$ (since otherwise the results in Section 3.2.2 provide adequate explanation for EDF-OPR-AN's superior performance). By the definition of β (Equation 1), it follows that $\beta \to 1$. Table 3.2 lists some values of $\frac{n}{1-\beta^n}$ for different values of *n*, for some values of β close to one. The critical observation is that for large β, the cost does not increase by a large factor with increasing *n*. For instance, in a cluster with 10 processing nodes and $\beta = 0.99$, an all-nodes strategy cannot inflate the cost of a job by a factor greater than 1.05. Even when β is as low as 0.9, the inflation is bounded from above by a 1.54 factor. While the inflation in cost is greater for smaller β, for example, for $\beta = 0.75$ and $n = 10$ the inflation is $10.6 / 4 \approx 2.7$. Recall that for such smaller β the headnode bottleneck is the dominating effect: for $\beta = 0.75$ and $n = 10$, the inflation factor $\frac{1}{(1-\beta^n)}$ of Observation 2 is a mere 1.05%.

These observations are formalized in the following:

Observation 3 *For computing clusters with a large value for* β*, the cost inflation experienced by jobs executed under an AN strategy is small.*

As we will see next, this slight inflation in total cost tends to be more than offset by the decrease in Effective Processor Utilization (EPU) experienced by EDF-OPR-MN vis-à-vis EDF-OPR-AN.

§2. The EPU effect. With respect to any node-assignment strategy, we define a mapping from each divisible job to a "traditional" (non-divisible) job as follows. Suppose that the node-assignment strategy assigns a job (A_i, σ_i, D_i) to n_i processing nodes. Ignoring for the moment the bottleneck head node, from the perspective of the processing nodes we can look upon this assignment as transforming the divisible job into a traditional job $(A_i, \sigma_i \times C_m \times (1/(1-\beta^{n_i})), D_i)$ on a single processor, this single processor being a virtual one obtained from the n_i processing nodes, and that exists only for the duration of the job's execution.

Since EDF-OPR-MN assigns the minimum possible number of processing nodes to each job, it is likely that these jobs will complete very close to their deadlines. More specifically, consider a job $J_i = (A_i, \sigma_i, D_i)$. The number of processors $n_{\min}$ assigned to this job under the EDF-OPR-MN combination of strategies is such that $\sigma_i / (1-\beta^{n_{\min}})$ is relatively large compared to D_i. Recalling (Section 3.2.1) that the ratio of the execution requirement of a traditional job to its relative deadline parameter is called its density, this immediately yields Observation 4 below:

Observation 4 *Under EDF-OPR-MN scheduling, divisible jobs tend to be mapped on to traditional jobs of high density.*

As stated in Section 3.2.1, it has been observed that multiprocessor EDF on traditional jobs exhibits poor Effective Processor Utilization (EPU) on high-density systems:

Observation 5 *The scheduling component (i.e., EDF) of the EDF-OPR-MN scheduling framework exhibits poor performance, in the sense that it is unable to make effective use of a significant fraction of the computing capacity of the platform.*

Recall that we had identified a tradeoff of increased cost versus poorer EPU. Observations 3 and 5 provide estimates for comparing these two effects, yielding the following lemma:

Lemma 2 *For computing clusters with largeβ, EDF-OPR-MN is a poor scheduling framework compared to EDF-OPR-AN.*

Proof: Observation 3 above asserts that the cost inflation suffered by individual jobs is relatively small under the EDF-OPR-MN strategy upon such computing clusters, while Observation 5 states that the loss of EPU is high. Taken together, these two factors yield the lemma.

3.4 Summary

In this chapter, we have taken a close theoretical look at the kind of real-time divisible loads studied by Lin et al. (2006a, 2006b and 2007a). Using recent results from “traditional” multiprocessor scheduling theory, we have provided satisfactory explanations for the apparently anomalous observation reported that an “all nodes” (AN) node-assignment strategy appears to significantly out-perform a “minimum number of nodes" (MN) strategy in the scheduling of real-time divisible workloads.

CHAPTER 4

SCHEDULING DIVISIBLE REAL-TIME LOADS ON CLUSTER WITH VARYING PROCESSOR START TIMES

4.1 Introduction

We have described the foundations of the initial work on RT-DLT (Lin et al., 2006a, 2006b, 2007a) in the previous chapters. While scheduling a particular divisible job, Lin et al. (2006a, 2006b, 2007a) assumed in this initial work that all the processors under consideration are simultaneously available to the job. Later, Lin et al. (2007b, 2007c) extended their work on RT-DLT to address the problem of distributing arbitrarily parallelizable real-time workloads among processors which become available at different instants in the future. They proposed an approximation algorithm to determine the smallest number of processors needed to complete the divisible job by its deadline. We consider the same problem in this chapter, and improve upon their results by providing with an efficient algorithm for solving this problem.

In the following section, we will describe in greater detail the issues that motivate consideration of this problem. In Section 4.3 and 4.4, we discuss the prior approaches that have been proposed for solving it. In Section 4.5, we present our efficient algorithm. We performed a series of simulations to compare the performance of these algorithms; we discuss the results of these experiments in Section 4.6. Finally we, conclude this work in Section 4.7.

4.2 Motivation

The initial work on RT-DLT assumed that all the processors to be allocated to the divisible job are simultaneously available. However, this is often not the case since some processors may have been allocated to previously-admitted (and scheduled) jobs — such processors will only become available once the jobs to which they have been allocated have completed execution upon them. When scheduling a given job, if a sufficient number of processors are available then the processors are allocated and the job is started. But if the required number of processors are not available, prior techniques required that the job be delayed until currently running jobs have finished and freed up an adequate number of additional processors. This causes a waste of computing capacity since some processors are idle even though there are waiting jobs; in the real-time context, such wastage can lead to missed deadlines.

In an attempt to lessen the deleterious effects of such waste on overall system performance, Lin et al. extended RT-DLT in (2007b, 2007c) to be applicable in a more general framework, in which each processor only becomes available to the job at a specified instant in the future. Their approach was to model such a cluster as a *heterogeneous* cluster, comprised of processors that are all available immediately, but have different computing capacities.

In this manner, they transformed the problem of different ready times for the processors to one of identical ready times but different computing capacities, which they subsequently solved (approximately) by extending the strategies that had previously been used for the analysis of homogeneous clusters.

In this chapter, we study the problem of determining the minimum number of processors that must be assigned to a job in order to guarantee that it meets its deadline, on clusters in which all processors are not simultaneously available. We provide efficient solutions to this problem, thereby improving on the approximate solutions of Lin et al. (2007b, 2007c). Our approach is very different from the approach of Lin et al. (2007b, 2007c), in that we have chosen to directly work with identical processors and different ready times (rather than first transforming to the heterogeneous cluster model).

4.3 Foundation

4.3.1 Processor Ready Times

As we mentioned earlier, the initial work on RT-DLT assumed that all processors are simultaneously made available to a job. In recent work of Lin et al. (2007b, 2007c), they further extend this model to allow for the possibility that all the processors are not immediately available. In this extended model, at any instant in time at which the head-node is determining whether to accept an incoming job or not (and if so, how to divide the job and allocate the pieces to the processors), there is a vector $\langle r_1, r_2, ...r_n \rangle$ of positive real numbers, with r_i, called the *ready time of* P_i, denoting the earliest time-instant (at or after the present) at which the *i*'th processing node becomes available. In this chapter, we retain the assumption made by Lin et al. (2007b, 2007c) that P_i can only participate in data transmission and/ or computation of the job currently under consideration at or after time-instant r_i.

Since this work extends the work of Lin et al. (2007b, 2007c), we briefly review some of the results from the previous chapter. We start out with the simpler model — all processor have the same ready time, and then proceed to the more challenging model in which different processors become available at different times in the future.

4.3.2 Processors with Equal Ready Times

In (Lin et al., 2006a, 2006b, 2007a), it is assumed that all the processors, upon which a particular job will be distributed by the head node, are available for that job over the entire time-interval between the instant that the head-node initiates data transfer to any one of these nodes, and the instant that it completes execution upon all the nodes. Under this model of processor availability, it is known that the completion time of a job on a given set of processing nodes is minimized if all the processing nodes complete their execution of the job at the same instant.

This makes intuitive sense – if some processing node completes before the others for a given distribution of the job's workload, then a different distribution of the workload that transfers some of the assigned work from the remaining processing node to this one would have an earlier completion time. Figure 4.1 depicts the data transmission and execution time diagram when processors have equal ready times.

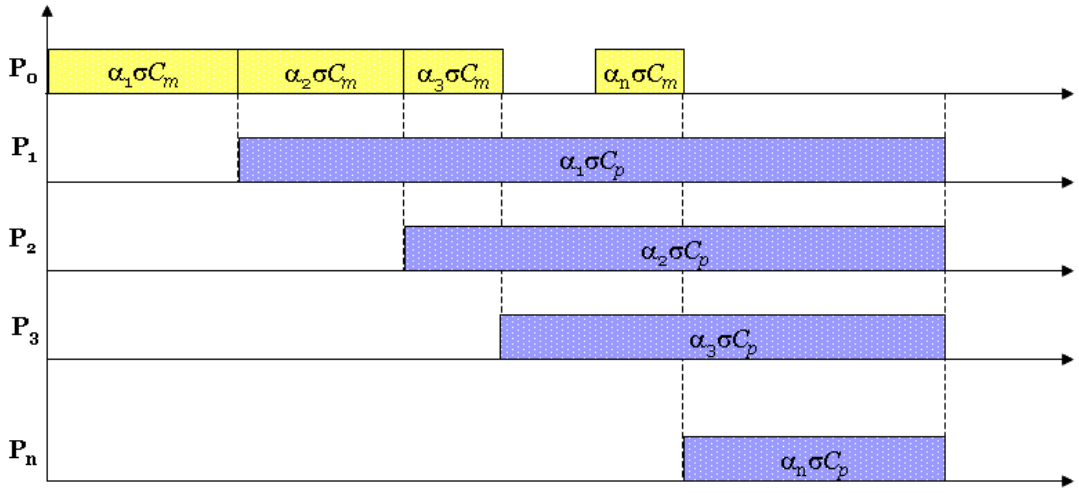

Figure 4.1: Data transmission and execution time diagram when processors have equal ready times

For a given job (A, σ, D) and a given number of processing nodes *n*, let $\sigma \times \alpha_i$ denote the amount of the load of the job that is assigned to the *j*'th processing node, $1 \le j \le n$. Since data-transmission occurs sequentially, the *i*'th node P_i can only receive data after the previous $(i-1)$ nodes have completed receiving their data. Hence, each P_i receives its data over the interval $\left[C_m \sum_{j=1}^{i-1} \alpha_j \sigma, C_m \sum_{j=1}^{i} \alpha_i \sigma\right)$ and therefore completes execution at time-instant $C_m \sum_{j=1}^{i} \alpha_j \sigma + C_p \alpha_i \sigma$.

By the optimality principle, P_i and P_{i+1} complete execution at the same time-instant. We therefore have:

$$C_m \sum_{j=1}^{i} \alpha_j \sigma + C_p \alpha_i \sigma = C_m \sum_{j=1}^{i+1} \alpha_j \sigma + C_p \alpha_{i+1} \sigma$$
$$\equiv C_m \sum_{j=1}^{i} \alpha_j \sigma + C_p \alpha_i \sigma = C_m \sum_{j=1}^{i} \alpha_j \sigma + C_m \alpha_{i+1} \sigma + C_p \alpha_{i+1} \sigma$$
$$\equiv C_p \alpha_i \sigma = \left(C_p + C_m \right) \alpha_{i+1} \sigma$$
$$\equiv \alpha_{i+1} = \left(\frac{C_p}{C_p + C_m} \right) \alpha_i$$
$$\equiv \alpha_{i+1} = \beta \alpha_i$$

That is, the fractions $\alpha_1, \alpha_2, ..., \alpha_n$ form a geometric series; furthermore, this series sums to one. Using the standard formula for the sum of an *n*-term geometric series, we require that

$$\frac{\alpha_1 (1 - \beta^n)}{1 - \beta} = 1$$

$$\equiv \alpha_1 = \left(\frac{1 - \beta}{1 - \beta^n} \right) \tag{4.1}$$

Letting $\xi(\sigma, n)$ denote the time-instant at which the job completes execution, and observing that this completion time is given by the sum of the data-transmission and processing times on P_i, we have

$$\xi(\sigma, n) = \alpha_1 C_m \sigma + \alpha_1 C_p \sigma$$

$$\equiv \xi(\sigma, n) = \frac{1 - \beta}{1 - \beta^n} \sigma (C_m + C_p) \tag{4.2}$$

4.3.3 Processors with Different Ready Times

The derivations in Section 4.2.2 above all assume that all *n* processors are immediately available. In (2007b, 2007c), Lin et al. allow for the possibility that all the processors are not immediately available. To determine the completion time of a job upon a given number of processors in this more general setting, Lin et al. (2007b, 2007c) adopt a *heuristic* approach that aims to partition a job so that the allocated processors could start at different times but finish computation (almost) simultaneously.

To achieve this, they first map the given homogenous cluster with different processor available times $r_1, r_2, ...r_n$ (with $r_i \leq r_{i+1} \forall_i$) into a *heterogeneous* model where all *n* assigned nodes become available simultaneously at the time-instant r_n, but different processors may have different computing capacities. Intuitively speaking, the $i'th$ processor has its computing capacity inflated to account for the reality that it is able to execute over the interval $[r_i, r_n)$ as well. Figure 4.2 depicts the data transmission and execution time diagram when processors have different ready times.

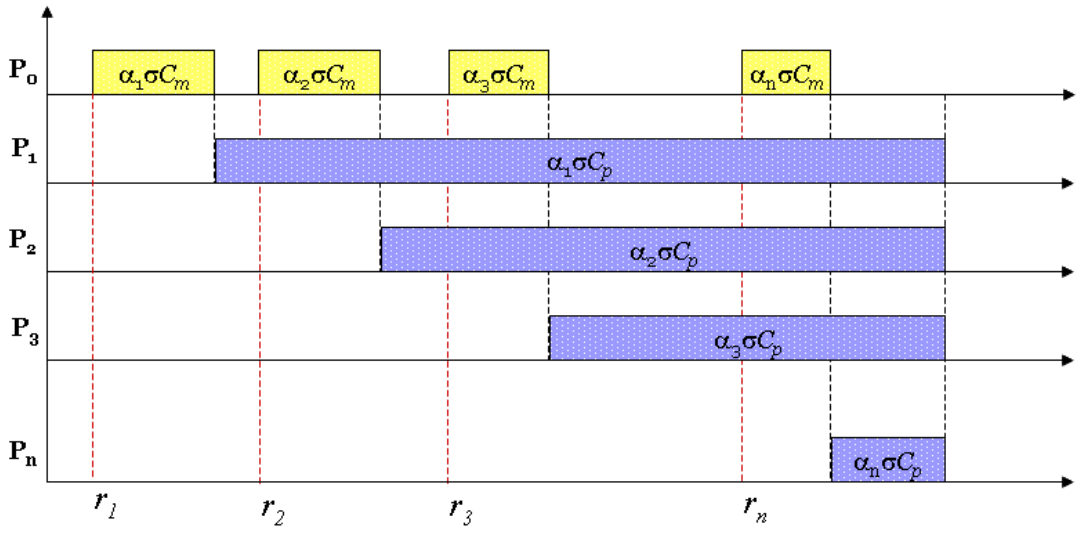

Figure 4.2: Data transmission and execution time diagram when processors have different ready times

In (Lin et al., 2007b, 2007c), this heterogeneity is modeled by associating a different constant C_{pi} with each processor P_i, with the interpretation that it takes C_{pi} time to complete one unit of work on the processor P_i. The formula for determining C_{pi}, as given in (Lin et al., 2007b, 2007c), is

$$C_{pi} = \frac{\xi(\sigma, n)}{\xi(\sigma, n) + r_n - r_i}, \tag{4.3}$$

where $\xi(\sigma, n)$ denotes the completion time if all processors are immediately available in the original (homogenous) cluster— see Equation 4.2. In (Lin et al., 2007b, 2007c), these C_{pi} values are used to derive formulas for computing the fractions of the workload that are to be allocated to each heterogeneous processor such that all processors complete at approximately the same time, and for computing this completion-time. These formulas are further discussed below in Section 4.4.

4.4 Determining the required minimum number of processors

When allocating resources in order to meet a divisible job's deadline, a scheduling algorithm must know the minimum amount of resources required by the job. Previous work by Lin et al. described how to compute this when all the processors are simultaneously allocated to a job (Lin et al., 2006a, 2007b, 2007a), and when processors can be allocated to a job at different times (Lin et al., 2007b). When all the processors are allocated simultaneously, recall that the completion time is given by Equation 4.2. The minimum number of processors needed is easily computed from Equation 4.2, by setting this completion time to the job's deadline (A + D) in Equation 4.2, and making "*n*" — the number of processors — the variable. (Since the number of processors is necessary integral, it is actually the ceiling of this value that is the minimum number of processors.)

When the processors have different ready times, using this same approach to determine the minimum number of processors needed is more challenging. Recall that the approach given in (Lin et al., 2007b) approximates the completion time of a job on a given number of processors by first transforming the cluster to a heterogeneous cluster in which all processors are available simultaneously but each processor may have a different computing capacity — these computing capacities are defined according to Equation 4.3. Using these computing capacities (the C_{pi}'s), it is easy to derive an expression for the exact completion time on such a heterogeneous platform (which, Lin et al. (2007b, 2007c) asserts, is an approximation of the completion time on the original homogeneous system with different processor ready times). Such a formula for the completion time on a heterogeneous platform is given in Lin et al. (2007b, (Equation 6)).

However, it is difficult to use this (approximate) completion-time formula to determine the minimum number of processors needed, for the following reason. In order to compute the right-hand side of Equation 4.3, we must already know the number of processors being used (since both $\xi(\sigma, n)$ and r_n depend upon this number). Thus, there is a circularity of reasoning going on here — the number of processors actually used must be known in order to compute the minimum number of processors needed.

We have been informed (in a personal email communication from the authors) that this dilemma is tackled in Lin et al. (2007b) by iterating over the possible values of $n - n = 1,2,\ldots$, until the minimum number of processors computed using that particular value of n is equal to the value used in computing the right-hand side of Equation 4.3. The approach in (Lin et al., 2007b) further approximates the behavior of the heterogeneous system by a homogeneous system with the same number of processors — (Lin et al., 2007b (Equations 9 and 10)) — when computing the minimum number of processors needed. In essence, they are determining the number of processors needed to meet the job's deadline assuming that all the processors become available at time-instant r_n, where r_n is the ready time of the n'th processor for some n guessed to be no smaller than the minimum number of processors needed.

As formally proved in Lin et al. (2007b), such an approximation is a safe one, in that while it may overestimate the number of processors needed, it is guaranteed to not underestimate it and hence deadlines are guaranteed to be met. However, it is not difficult to construct scenarios in which the degree of pessimism, as measured by the ratio of the actual minimum number of processors needed and the number computed by this approach, is arbitrarily large.

4.5 Computing the exact required minimum processors

We have adopted an altogether different approach to circumvent this circularity of reasoning. Rather than first deriving a formula for computing the completion time on a given number of processors and then using this formula to determine the minimum number of processors needed to meet a deadline, we instead compute the minimum number of processors directly, from first principles. Our approach is presented in pseudo-code form in Figure 4.3.

The general idea is as follows. Starting out with no processors, we will repeatedly add processors until we have either added enough to complete the job (line 3 in the pseudo-code), or we determine that it is not possible to complete this job by its deadline (line 4 in the pseudo-code). We now discuss the pseudo-code in greater detail. We are given the size of the workload (σ), the amount of time between the current instant and the deadline (Δ), the cluster parameters C_p and C_m, and the processor ready times $r_1, r_2, ...r_n$ in sorted order. We will determine the minimum number of processors needed ($n_{\min}$), the shares allocated to each processor (the α_i's), and the time at which each processor will begin receiving data from the head node P_0 (the s_i's).

MINPROCS(σ, Δ)

1 $s_1 \leftarrow r_1; alloc \leftarrow 0; i \leftarrow 1$

2 **while** (true) **do**

3 **if** ($alloc \geq 1$) break **end if**

4 **if** ($s_i \geq \Delta$) break **end if**

5 $\alpha_i \leftarrow (\Delta - s_i) \div (\sigma \times (C_m + C_p))$

6 $s_{i+1} \leftarrow \max(r_{i+1}, s_i + (\alpha_i \times \sigma \times C_m))$

7 $alloc \leftarrow alloc + \alpha_i$

8 $i \leftarrow i + 1$

end while

9 **if** ($alloc \geq 1$) **then** ▷ success!!

10 $n_{\min} \leftarrow i$

else ▷ cannot meet the deadline, regardless of the number of processors used

11 $n_{\min} \leftarrow \infty$

end if

Figure 4.3: Computing $n_{\min}$

The pseudo-code uses two additional variables — *alloc*, denoting the fraction of the workload that has already been allocated, and *i*, indicating that P_i is being considered. The main body of the pseudo-code is an infinite **while** loop, from which the only exist is be one of two **break** statements. The **break** in line 3 indicates that we have allocated the entire job, while executing the **break** in line 4 means that we need to execute beyond the deadline (i.e., there are not enough processors with ready times prior to this job's deadline for us to be able to meet its deadline).

If neither **break** statement executes, we compute α_i, the fraction of the job that is allocated to processor P_i. The value is computed by observing that allocating a fraction α_i of the load require this node to be receiving data for $C_m\alpha_i\sigma$ time units and then executing this data for $C_p\alpha_i\sigma$ time units. In keeping with the optimality rule, we would like to have this processor complete execution at the job deadline (i.e., at time-instant Δ); since P_i may only begin receiving data at time-instant s_i, we require that $s_i + C_m\alpha_i\sigma + C_p\alpha_i\sigma = \Delta$, from which we derive the value of α_i given in line 5.

Once P_i's share is computed, we can compute the time at which P_{i+1} may begin execution. This is the later of its ready time and the time at which P_i has finished receiving data (and the head-node is thus able to commence data-transmission to P_{i+1}). This computation of s_{i+1} is done in line 6. Lines 7 and 8 update the values of the fraction of the workload that has already been allocated, and the index of the processor to be considered next.

Properties. It should be evident that the schedule generated by this algorithm is both correct — the job will indeed complete by its deadline on the computed number of processors, according to the schedule that is implicitly determined by the algorithm, and optimal — the number of processors used is the minimum possible. Making reasonable assumptions on the problem representation (e.g., that the processor ready times are provided in sorted order), it is also evident that the run-time of this algorithm is linear in the number of processors used. Hence, since the output of such an algorithm must explicitly include the processor shares (the α_i's) in order to be useful for actual scheduling and dispatching, it is asymptotically optimal from the perspective of run-time computational complexity.

4.6 Simulation Results

We have conducted extensive simulation experiments to estimate the degree by which our optimal algorithm outperforms the non-optimal approach of (Lin et al., 2007b). In this section, we describe these experiments, present some of the results, and draw some conclusions regarding under which conditions it is most beneficial to adopt our approach in preference to the one in (Lin et al., 2007b).

The outcomes of our experiments are plotted in Figure 4.4 through Figure 4.11. For greater detail, we also present the results data in Table 4.1 through Table 4.8. All the graphs plot the minimum number of processors ($n_{\min}$) needed to complete a given real-time workload by its specified deadline, when this minimum number of processors is computed by our algorithm (depicted in the graphs by filled circles) and when it is computed by the algorithm in (Lin et al., 2007d) (depicted in the graphs by filled squares). As can be seen in all the graphs, the performance of our algorithm is never inferior to, and typically better than, the performance of the algorithm in (Lin et al., 2007b) —this is not surprising since our algorithm is optimal while the one in (Lin et al., 2007b) is not.

We now describe the experiments in greater detail. We determined the minimum number of processors as computed using both algorithms, under a variety of system and workload conditions. The system and workload is characterized by: the number of processors *N* and the processor release times $r_1, r_2, ..., r_N$; the cluster parameters C_p and C_m denoting the data-processing and communication rates respectively; and the real-time workload characterized by arrival-time, size, and deadline. In each experiment, all the parameters were kept constant and one parameter varied, thereby allowing us to evaluate the relative performance of the two algorithms with respect to the varying parameter.

4.6.1 Increasing Deadlines

The first two sets of experiments evaluate the relative performance of the two algorithms as the deadline of the workload is increased, for clusters of 16 and 32 processors. The results are shown in Figure 4.4 and Figure 4.5 respectively. The performance improvement is observed to be very significant for tight deadlines; as the deadline increases, the performance penalty paid by the algorithm in (Lin et al., 2007d) drops off. Table 4.1 and Table 4.2 show the minimum number of processors as determined by both algorithms with respect to load deadlines. Observe that, particularly in Table 4.1, not only does the approximation algorithm generate significantly larger numbers than our efficient algorithm, but the required number of processors often exceeds the cluster capacity. In such situations, the cluster will simply reject the job as it does not have the capacity to execute the job. As shown in this particular simulation, with the approximation algorithm, only 30% of jobs will tend to be executed upon this cluster, compared to 100% of jobs by the efficient algorithm.

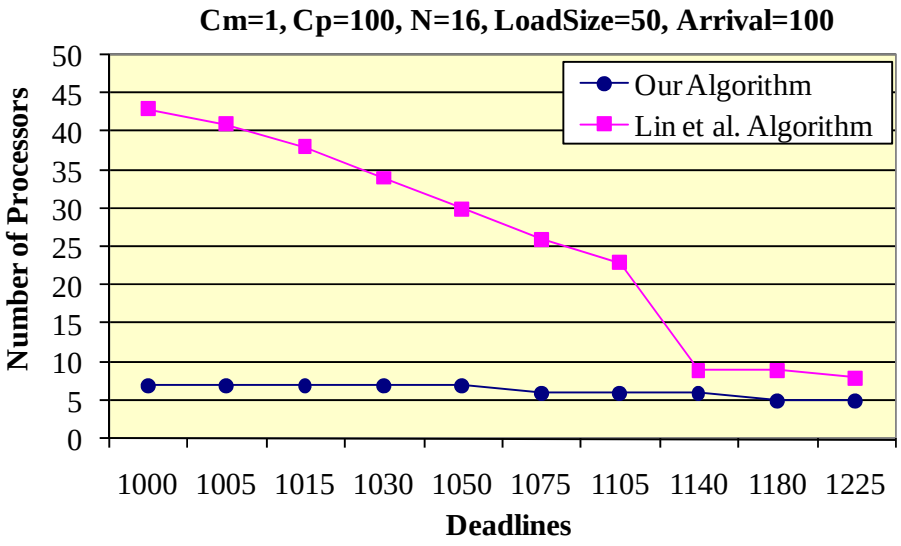

Figure 4.4: Comparison of generated n_{min} with increasing deadline and a cluster of n=16 processors

Table 4.1: Comparison of generated n_{min} with increasing deadline, and a cluster of n=16 processors

Deadlines	Our Algorithm	Lin et al. Algorithm
1000	7	**43**
1005	7	**41**
1015	7	**38**
1030	7	**34**
1050	7	**30**
1075	6	**26**
1105	6	**23**
1140	6	9
1180	5	9
1225	5	8

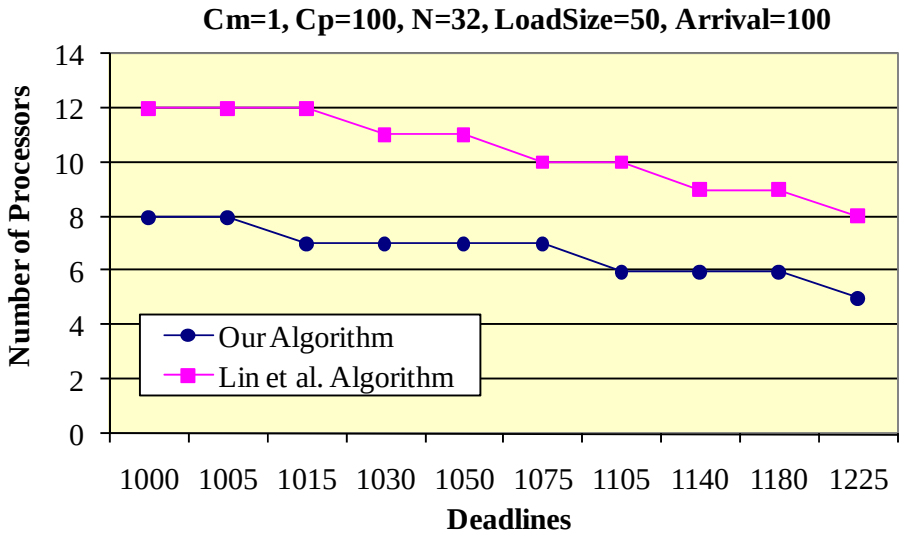

Figure 4.5: Comparison of generated n_{min} with increasing deadline, and a cluster of n=32 processors

Table 4.2: Comparison of generated $n_{\min}$ with increasing deadline, and a cluster of n=32 processors

Deadlines	Our Algorithm	Lin et al. Algorithm
1000	8	12
1005	8	12
1015	7	12
1030	7	11
1050	7	11
1075	7	10
1105	6	10
1140	6	9
1180	6	9
1225	5	8

4.6.2 Increasing Communication Cost

The two graphs shown in Figure 4.6 and Figure 4.7 evaluate the relative performance of the two algorithms as the communication cost parameter of the cluster – C_m – is increased, for clusters of 16 and 32 processors respectively. Table 4.3 and Table 4.4 show the generated $n_{\min}$ in greater detail. As can be seen, the performance improvement for our algorithm increases as C_m increases.

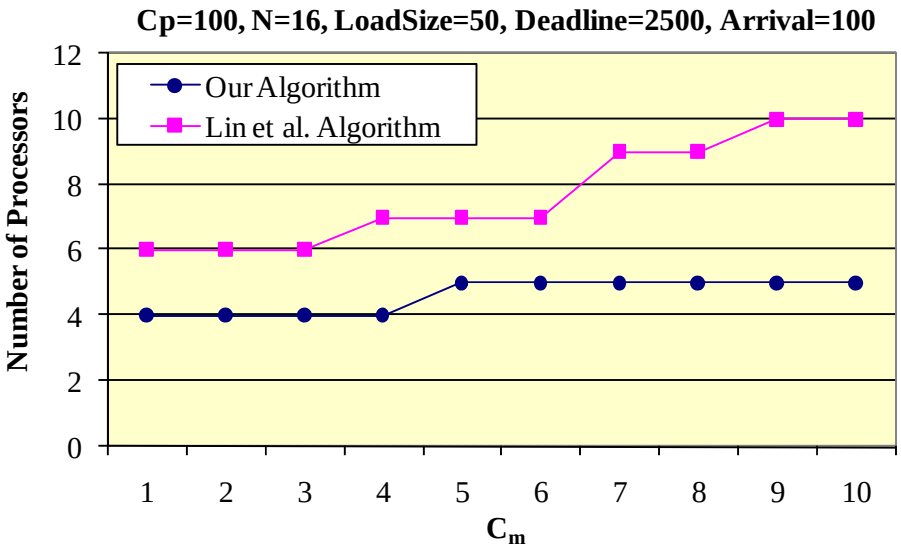

Figure 4.6: Comparison of generated n_{min} with increasing C_m and a cluster of n=16 processors

Table 4.3: Comparison of generated n_{min} with increasing communication cost C_m, and a cluster of n=16 processors

Cm	Our Algorithm	Lin et al. Algorithm
1	4	6
2	4	6
3	4	6
4	4	7
5	5	7
6	5	7
7	5	9
8	5	9
9	5	10
10	5	10

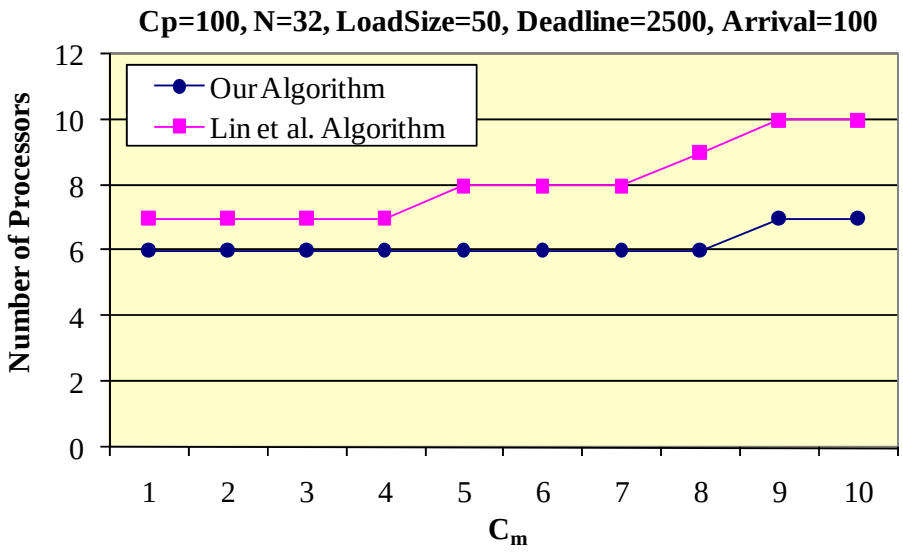

Figure 4.7: Comparison of generated n_{min} with increasing C_m, and a cluster of n=32 processors

Table 4.4: Comparison of generated n_{min} with increasing C_m, and a cluster of n=32 processors

4.6.3 Increasing Computation Cost

Figures 4.8 and Figure 4.9 show the relative performance of the two algorithms as the processing cost parameter of the cluster – C_p – is increased, for clusters of 16 and 32 processors respectively. The performance improvement seen by our algorithm once again increases with increasing C_p.

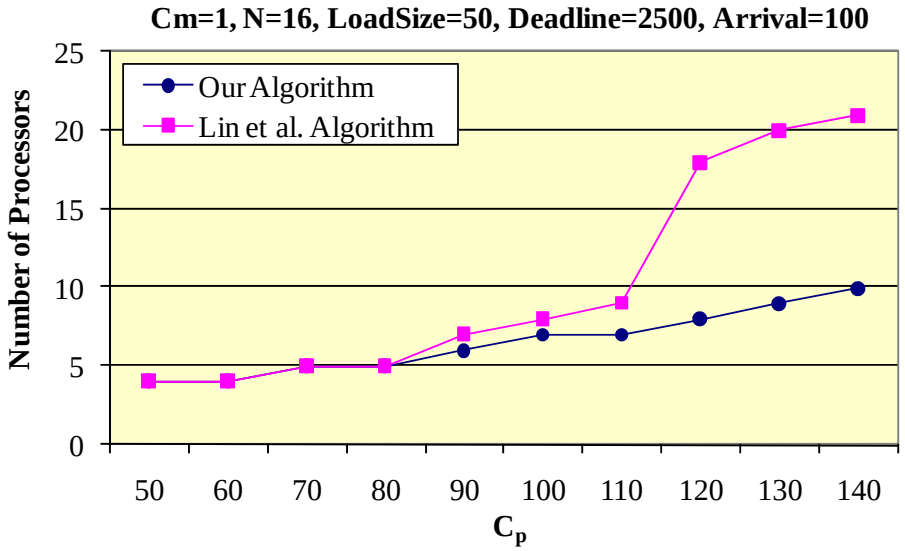

Figure 4.8: Comparison of generated n_{min} with increasing C_p, and a cluster of n=16 processors

Table 4.5: Comparison of generated n_{min} with increasing C_p, and a cluster of n=16 processors

Cp	Our Algorithm	Lin et al. Algorithm
50	4	4
60	4	4
70	5	5
80	5	5
90	6	7
100	7	8
110	7	9
120	8	18
130	9	20
140	10	21

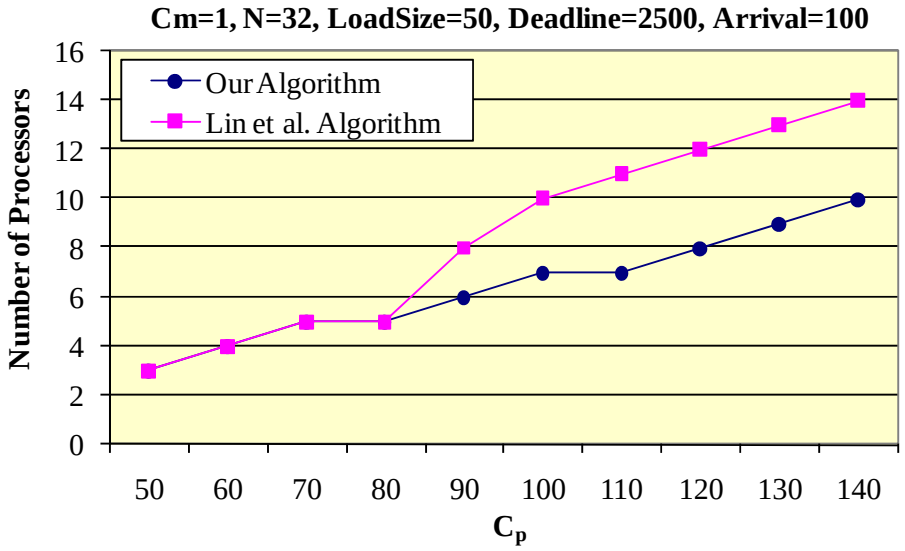

Figure 4.9: Comparison of generated n_{min} with increasing C_p, and a cluster of n=32 processors

Table 4.6: Comparison of generated n_{min} with increasing C_p, and a cluster of n=32 processors

Cp	Our Algorithm	Lin et al. Algorithm
50	3	3
60	4	4
70	5	5
80	5	5
90	6	8
100	7	10
110	7	11
120	8	12
130	9	13
140	10	14

Table 4.5 and table 4.6 shows the minimum number of processors generated by both algorithms with respect to increasing communication cost upon a cluster of 16 and 32 processors. Once again, observe that the approximation algorithm tends to compute a number of processors that exceeds the cluster capacity, particularly within 16 processors cluster; thus it will tend to reject more jobs than necessary.

4.6.4 Increasing Workload Size

The graphs in Figure 4.10 and Figure 4.11 evaluate the relative performance of the two algorithms as the size of the workload is increased, for clusters of 16 and 32 processors respectively. The performance improvement is observed to be negligible or very small for small loads; but as the load size increases, the performance penalty paid by the algorithm in (Lin et al., 2007b) becomes more significant.

For greater detail, we show the minimum number of processors generated by both algorithms in Table 4.7 and Table 4.8. Once again, as shown in both tables, the approximation algorithm tends to compute a number of processors that exceeds the cluster capacity; thus it will tend to reject more jobs than necessary.

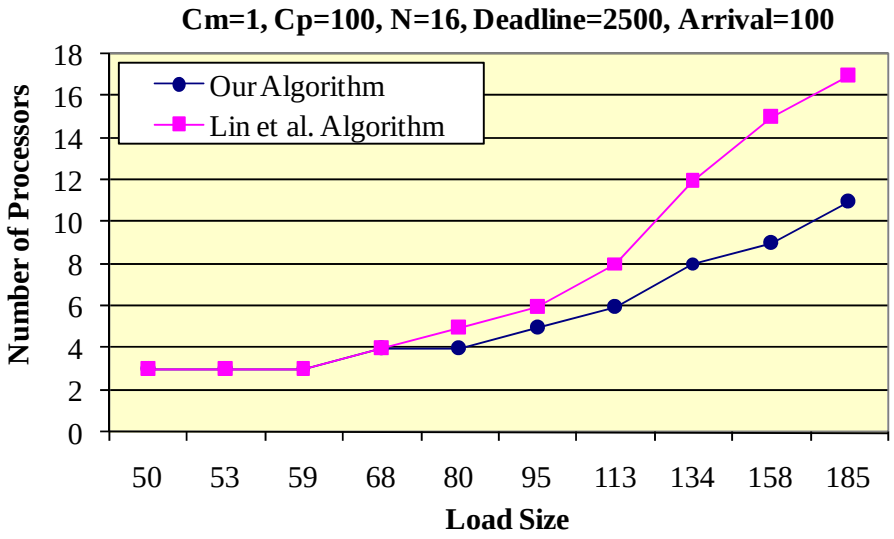

Figure 4.10: Comparison of generated n_{min} with increasing load size, and a cluster of n=16 processors

Table 4.7: Comparison of generated n_{min} with increasing workload size, and a cluster of n=16 processors

Load Size	Our Algorithm	Lin et al. Algorithm
50	3	3
53	3	3
59	3	3
68	4	4
80	4	5
95	5	6
113	6	8
134	8	12
158	9	15
185	11	17

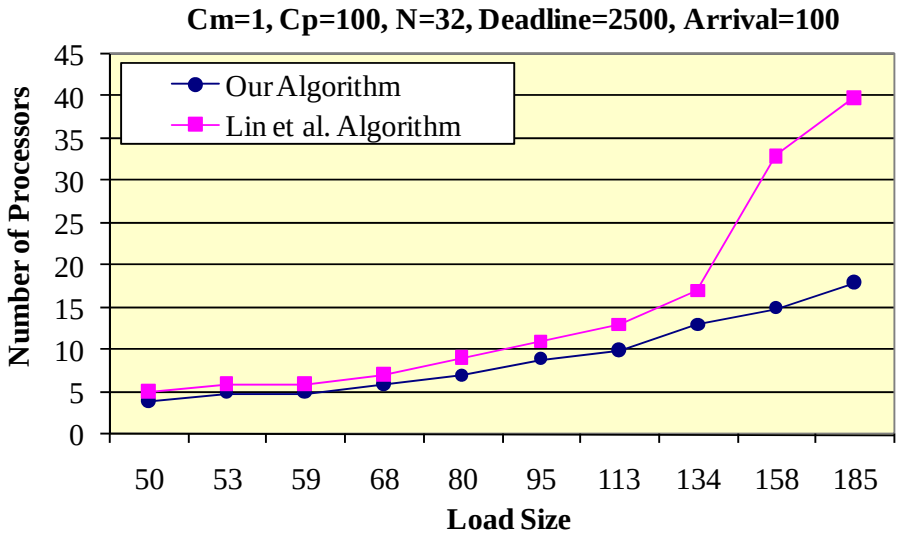

Figure 4.11: Comparison of generated n_{min} with increasing workload size, and a cluster of n=32 processors

Table 4.8: Comparison of generated n_{min} with increasing workload size, and a cluster of n=32 processors

Load Size	Our Algorithm	Lin et al. Algorithm
50	4	5
53	5	6
59	5	6
68	6	7
80	7	9
95	9	11
113	10	13
134	13	17
158	15	33
185	18	40

The high-level conclusions to be drawn from these experiments are that the previous algorithm (the one in (Lin et al., 2007b)) is acceptable upon clusters in which the communication and computation overheads are very small, and on workloads that do not “stress” the system (i.e., they are small jobs, and/or have large relative deadlines). In other circumstances, our optimal algorithm performs significantly better.

4.7 Summary

In this chapter, we have studied scheduling problems in RT-DLT when applied to clusters in which different processors become available at different time-instants. We proposed an algorithm that efficiently determines the minimum number of processors that are required to meet a job deadline. Through extensive experimental evaluation, we have shown that this efficient formula significantly improves on the heuristic approximations proposed by Lin et al. (2007b, 2007c).

CHAPTER 5

A LINEAR PROGRAMMING APPROACH FOR SCHEDULING DIVISIBLE REAL-TIME LOADS

5.1 Introduction

In the previous chapter, we studied the problem of distributing arbitrarily parallelizable real-time workloads among processors which become available at different instants of time. We also presented an efficient algorithm to determine the minimum number of processors needed to complete this job by its deadline upon such cluster environments, and through a series of simulations, we showed that our algorithm always produced an efficient number and comfortably out-performed the approximate algorithms found in (Lin et al., 2007b, 2007c).

In this chapter, we will study another aspect of the problem: *Given* a divisible job of a particular size and a fixed number of processors, which may become available at different times, upon which to execute it, *determine* the earliest completion time for the job. We formulate this problem as a linear program (LP), and thereby provide a polynomial-time algorithm for solving the problem.

The remainder of this Chapter is organized as follows. In Section 5.2, we describe in greater detail the motivation for this work and demonstrate the non-optimality of previously proposed approaches. In Section 5.3, we present our LP formulation. We performed a series of simulations to compare the performance of these approaches: we discuss the design and results of these simulations in Section 5.4 and Section 5.5.

5.2 Computing completion time on a given set of processors

As discussed in Chapter 4, for the case when the processors have different ready times, Lin et al. proposed an approach in (2007b, 2007c) via the abstraction of *heterogeneous* clusters — clusters in which all *n* processors become available at the same instant but different processors may have different computing capacities. Specifically, the algorithm in (Lin et al., 2007b, 2007c) assumes that all n processors become available at time-instant r_n and the *i*'th processor P_i takes $C_{pi} \times x$ time to process *x* units of data, where the C_{pi}'s are as given in Equation 4.3 (reproduced below):

$$C_{pi} = \frac{\xi(\sigma, n)}{\xi(\sigma, n) + r_n - r_i}, \tag{5.1}$$

Here, $\xi(\sigma, n)$ denotes the completion time if all processors are immediately available in the original cluster, as given by Equation 4.2. Using these processor computing capacities, the approach of Lin et al. (2007b) adopts a strategy very similar to the one in 4.2.1 of the previous chapter, to derive formulas for computing the fractions of the workload that must be assigned to each (hypothetical heterogeneous) processor in order that all the processors complete at the same instant (Lin et al., 2007b, (Equations 4 and 5)), and for computing this completion time (Lin et al., 2007b, (Equations 6)). We illustrated these formulas via the following example.

Example 1: Consider a cluster in which $C_m = C_p = 1$, and consider a job of size $\sigma = 30$ which arrives at time-instant zero, and is assigned two processors P_1 and P_2 in this cluster, with ready-times $r_1 = 0$ and $r_2 = 21$ respectively. We describe how to convert this cluster to a heterogeneous cluster of two processors in which both become available at time-instant 21 (i.e., at r_2).

First, we need to compute $\xi(\sigma, 2)$ according to Equation 5.2 – the completion time of this job was to be scheduled optimally upon two homogeneous processors that are always available. It may be verified that Equation 5.1 yields $\alpha_1 = \frac{2}{3}$ (two-thirds of the job is assigned to processor P_1 and the remaining one-third to P2); consequently, P_1 participates in data-transmission over [0; 20) and computation over [20; 40) while P_2 participates in data-transmission over [20; 30) and computation over [30; 40) for an eventual completion-time of $\xi = 40$. Using this value in Equation 5.3 we get $C_{p1} = 1$ and $C_{p2} = \frac{40}{61}$ as the processor computing capacities in the heterogeneous cluster.

We now describe how to compute the fractions α'_1 and α'_2 of the job allocated to the two (hypothetical) heterogeneous processors. The idea is to apply the optimal partitioning rule to the heterogeneous platform – determine the values of α'_1 and α'_2 such that if the first processor were to be assigned a load $\alpha_1\sigma$ and the second a load $\alpha_2\sigma$ (both starting at the same time-instant), both processors would complete at the same instant.

That is, we need values for α'_1 and α'_2 that sum to one ($\alpha'_1 + \alpha'_2 = 1$) and satisfy:

$$C_m\alpha'_1\sigma + C_{p1}\alpha'_1\sigma = C_m(\alpha'_1 + \alpha'_2)\sigma + C_{p2}\alpha'_2\sigma \qquad (5.2)$$

Solving, we obtain the values $\alpha'_1 = \frac{101}{162}$ and $\alpha'_2 = \frac{61}{162}$. Mapping these shares back to the original cluster (homogeneous processors, but with different processor ready times), we obtain the following schedule:

Processor P_1 participates in data-transmission over the time-interval $\left[0, 18\frac{19}{27}\right)$, and processes this data over the time-interval $\left[18\frac{19}{27}, 37\frac{11}{27}\right)$; hence, P_1's completion time is $37\frac{11}{27}$.

Processor P_2 participates in data-transmission over the time-interval $\left[21, 32\frac{8}{27}\right)$, and processes this data over the time-interval $\left[32\frac{8}{27}, 43\frac{16}{27}\right)$; hence, P_2's completion time is $43\frac{16}{27}$.

Taking the larger of the two individual processor completion times, we see that the overall completion time is equal to $43\frac{16}{27}$.

Non-optimality of the (Lin et al., 2007b) approach. This approach is easily seen to be non-optimal. For the situation considered in Example 1 it may be verified that if P_1 and P_2 (in the original cluster) were assigned fractions $\alpha_1 = \frac{27}{40}$ and $\alpha_2 = \frac{13}{40}$ of the load respectively, then P_1 would receive data over interval [0; 20.25) and process this data over [20.25, 40.5) for a completion-time of 40.5; meanwhile, P_2 would receive data over the interval [21, 30.75) and process this data over the interval [30.75, 40.5) for an overall completion time of 40.5 (which is earlier than the $43\frac{16}{27}$ completion time of the schedule in the example).

In fact, examples are easily constructed in which the completion-time bound obtained using the approach of (Lin et al., 2007b) is arbitrarily worse than the optimal. (Consider a simple modification to our two-processor cluster in Example 1 above that increased r_2 to some value $\rho > 30$ but leaves everything else unchanged. The optimal schedule – the one with earliest completion time – would execute the entire load on the first processor for a completion time of 30. However, the approach of first transforming to a heterogeneous platform would necessarily assign non-zero load to the second processor (see (Lin et al., 2007b, (Equations 4 and 5))), and hence have a completion-time $> \rho$. As $\rho \to \infty$, the performance of this approach therefore becomes arbitrarily bad as compared to the optimal approach.

5.3 Linear Programming Formulation

We now describe how the problem of computing the earliest completion time may be formulated as a linear programming problem. This would immediately allow us to conclude that the earliest completion time can be computed exactly in polynomial time, since it is known that a Linear Program (an LP) can be solved in polynomial time by the ellipsoid algorithm (Khachiyan, 1979) or the interior point algorithm (Karmakar, 1984). In addition, the exponential-time simplex algorithm (Dantzig, 1963) has been shown to perform extremely well in practice and is often

the algorithm of choice despite its exponential worst-case behavior. It is also well-known that LP problems can be efficiently solved (in polynomial time), and that excellent libraries (and several stand-alone tools) exist for solving LP's extremely efficiently in practice.

---

minimize ξ

subject to the following constraints:

(1) $\alpha_1 + \alpha_2 + ... + \alpha_n = 1$

(2) $0 \le \alpha_i \qquad 1 \le i \le n$

(3) $r_i \le s_i, \qquad 1 \le i \le n$

(4) $s_i + \alpha_i \sigma C_m \le s_{i+1}, \qquad 1 \le i \le n$

(5) $s_i + \alpha_i \sigma (C_m + C_p) \le \xi, \qquad 1 \le i \le n$

Figure 5.1: Computing the completion time – LP formulation

Given the workload size σ, the cluster parameters C_m and C_p, and the *n* processor ready-times $r_1, r_2, ..., r_n$ we construct a linear program (Figure 5.1) on the following (2n + 1) variables:

- The *n* variables $\{\alpha_1, \alpha_2, ..., \alpha_n\}$, with α_i denoting the fraction of the workload to be assigned to the *i*'th processor;

- The *n* variables $\{s_1, s_2, ..., s_n\}$ with s_i denoting the time-instant at which the head node begins transmitting data to the *i*'th processor; and

- The variable ξ, denoting the completion time of the schedule (i.e., the objective function of the minimization problem).

The constraints that must be satisfied by these variables are as follows:

(1) The entire workload must be allocated; i.e., the α_i's must sum to 1. This is represented by equality (1) of the LP in Figure 5.1.

(2) The fraction of the workload allocated to each processor must be non-negative. This is represented by the n inequality (2) of the LP in Figure 5.1.

(3) Each processor may begin receiving data only after its ready time—we must have $s_i \geq r_i$ for all *i*. This is represented by the n inequality (3) of the LP in Figure 5.1.

(4) Data-transmission is sequential, which means that data-transmission to the (i + *1*)'th processor may commence (at time-instant s_i) only after data-transmission to the i'th processor has completed (at time-instant $s_i + \alpha_i \sigma C_m$. This is represented by the n inequality (4) of the LP in Figure 5.1.

(5) The completion time on the *i*'th processor (i.e., ($s_i + \alpha_i \sigma C_m + \alpha_i \sigma C_p$)) is, by definition, no larger than the completion time ξ of the entire schedule. This is represented by the n inequality (5) of the LP in Figure 5.1.

Recall that, when all processors have equal ready time, by using the DLT optimality principle, the workload execution time $\xi(\sigma, n)$ as given in Equation 3.3 (reproduced here):

$$\begin{aligned}\xi(\sigma,n) &= \alpha_1\sigma C_m + \alpha_1\sigma C_p \\ &= (\alpha_1+\alpha_2)\sigma C_m + \alpha_2\sigma C_p \\ &= (\alpha_1+\alpha_2+\alpha_3)\sigma C_m + \alpha_3\sigma C_p \\ &\ldots \\ &= (\alpha_1+\alpha_2+\ldots+\alpha_n)\sigma C_m + \alpha_n\sigma C_p \end{aligned} \quad (5.3)$$

And the values for the α_i are computed by using:

$$\alpha_1 = \frac{1-\beta}{1-\beta^n} \quad (5.4)$$

$$\alpha_j = \beta^{j-1}\alpha_1, \text{ for j > 1} \quad (5.5)$$

Where β is defined as:

$$\beta \overset{def}{=} \frac{C_p}{(C_p + C_m)} \quad (5.6)$$

When processors have different ready time, we use the same principle to compute the value of fraction α_i and the fraction's execution time must not exceed the overall schedule completion time ξ, as represented by the inequality (5)'s of the LP in Figure 5.1.

$$s_i + \alpha_i\sigma C_m + \alpha_i\sigma C_p \le \xi \quad (5.7)$$

In addition, since processors have different ready time, we need to compute different s_i for each fraction α_i by setting the constraints as represented by the inequality (3)'s and (4)'s of the LP in Figure 5.1.

The following two lemmas formally assert that the problem of obtaining a feasible solution to the LP in Figure 5.1 is equivalent to the problem of obtaining a schedule for the workload on the n processors.

Lemma 1 *Given feasible solution to the LP in Figure 5.1 that assigns value* ξ_o *to the variable* ξ*, we can construct a schedule for a workload of size* σ *with completion-time* ξ_o.

Proof: Any feasible solution to the LP assigns nonnegative values to each of the α_i's and s_i's. By assigning a workload $\sigma\alpha_i$ to the *i*'th processor, it immediately follows from the construction of the LP that the total workload is assigned to the n processors. Furthermore ξ_o, the value of ξ in the feasible solution, clearly represents an upper bound on the completion time of the schedule.

Lemma 2 *Given a schedule for a workload of size* σ *with completion-time* ξ_1 *that there is a feasible solution to the LP in Figure 5.1 that assigns the variable* ξ *the value* ξ_1.

Proof: Let α'_i denote the amount of the workload executed on the *i*'th processor, and s'_i the time-instant at which transmission of this workload commences to the *i*'th processor. It is once again evident from the manner of construction of the LP that the variable assignment

$$\alpha_i \leftarrow \alpha'_i, \qquad 1 \le i \le n$$
$$s_i \leftarrow s'_i, \qquad 1 \le i \le n$$
$$\xi \leftarrow \xi_1$$

constitutes a feasible solution to the LP. We now present our main result in this section, demonstrating the equivalence of scheduling to minimize completion time to Linear Programming.

Theorem 1 *Computing the earliest completion time of a given divisible workload of size* σ *on a cluster with parameters* C_m *and* C_p*, and n identical processors with (non-decreasing) available times* $r_1, r_2, ..., r_n$ *is equivalent to solving the linear programming problem given in Figure 5.1.*

Proof: It immediately follows from Lemma 1 and Lemma 2 that the smallest value of ξ satisfying the LP in Figure 5.1 is exactly the desired earliest completion time.

As a parenthetical side-note, we observe that this LP is easily modified to compute the earliest completion time upon more general heterogeneous platforms as well, in which there may be a different C_{mi} and a different C_{pi} associated with the data-communication and computing capacity of each processor P_i. The only modifications are that each C_m in the third constraints is replaced by C_{mi} for each *i,* and that each C_m and C_p in the fourth constraints are replaced by C_{mi} and C_{pi} respectively for each *i*. We will demonstrate this heterogeneous transformation in the following section.

5.4 Simulation Design

We have performed extensive simulation experiments comparing the performance of our (proved efficient) LP solution technique to the approximate technique of (Lin et al., 2007b, 2007c). Our experiments were performed in MATLAB, using the **linprog**, a linear-programming solver that is available with MATLAB to solve our LPs. Figure 5.2 depicts the design of these simulation programs.

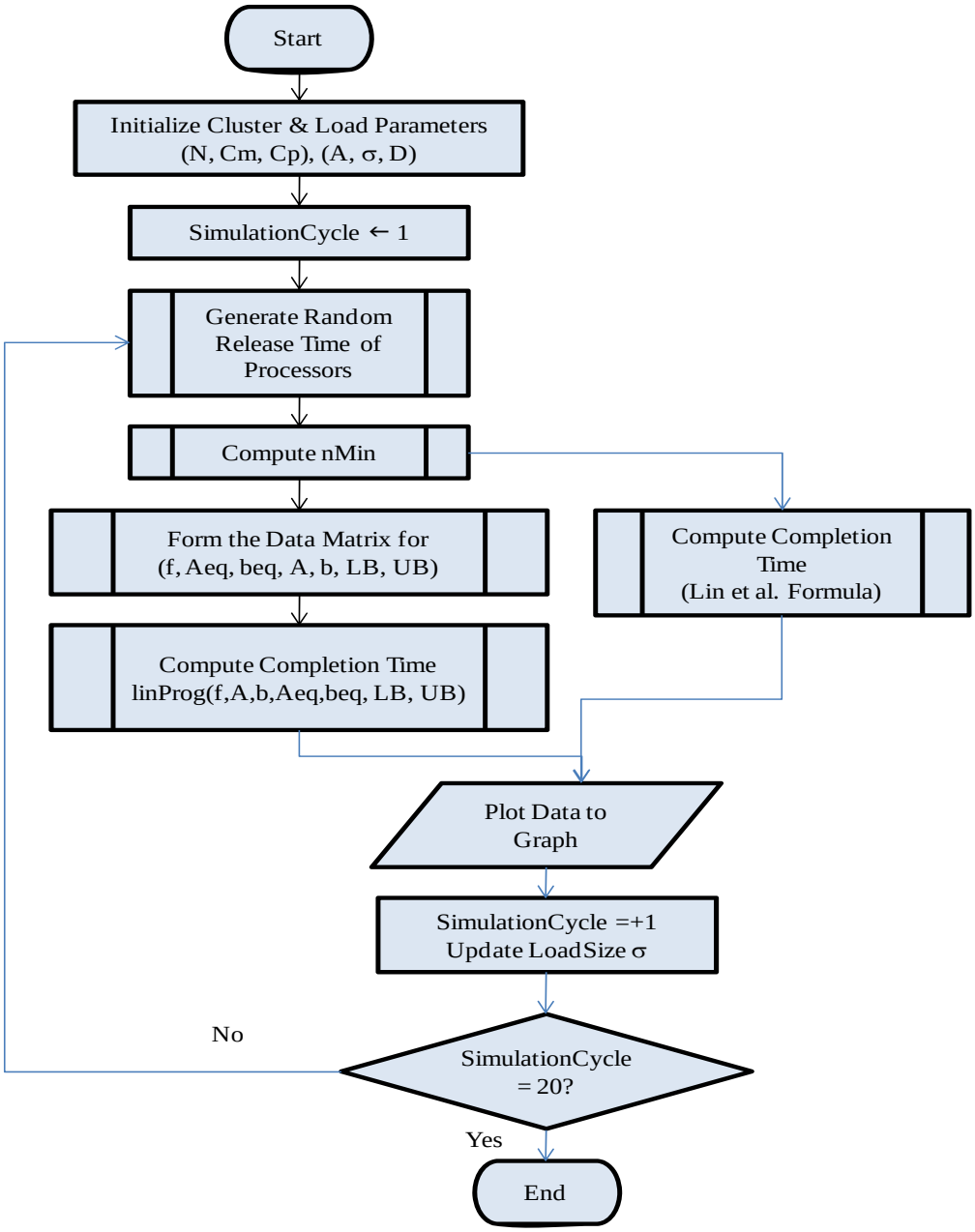

Figure 5.2: The Simulation Design

As shown in Figure 5.2, We now briefly describes the simulation design:

- **Initialize Cluster Parameters**

  We performed 6 sets of simulations to compare the performance of our LP-based formula to the Lin et al. (2007b, 2007c) solution. For these simulations, we set the cluster parameters $C_m = 1$ and $C_p = 100$. In each set, we considered 6 values for the number of processors ***n***: $n = 4$, $n = 6$, $n = 8$, $n = 12$, $n = 16$ and $n = 20$. For example for the first set, we initialize the cluster parameter as ($C_m = 1, C_p = 100$, $n = 4$).

  We also studied the behavior of our LP-based approach with different cluster parameters. Thus, for these purpose, we set different values of C_m and C_p.

- **Initialize Load Parameters**

  In this procedure, we initialize the workload parameters (A, σ, D) with the initial values.

- **Simulation Cycle**

  This step is to initialize the variable *SimulationCycle*=1, which later will be use to control the simulation cycle.

- **Generate Random Release Time**

  This procedure will randomly generate the processor release times – the r_i's. For example, for $n = 8$, it will generate 8 processor release times values.

- **Compute the** $n^{\min}$

  Assuming $n^{\min}$ has been generated by some formula described in the previous chapter; this procedure will use $n^{\min}$ value for the next computation.

- **Form the Data Matrix**

  This procedure will form the data matrix to be used in the computation of the completion time. In the following, we present an example of generated data matrix for a cluster ($C_m = 1, C_p = 100$, $n = 3$) and a workload ($A = 0$, $\sigma = 20$, $D = 1000$).

  Example:

  Given an objective function:

  $$\text{Minimize } Z = 0.\alpha_1 + 0.\alpha_2 + 0.\alpha_3 + 0.s_1 + 0.s_2 + 0.s_3 + \xi$$

  Subject to the following constraints:

  $$\begin{aligned}
  \alpha_1 + \alpha_2 + \alpha_3 &= 1 \\
  -s_1 &\le -r_1 \\
  -s_2 &\le -r_2 \\
  -s_3 &\le -r_3 \\
  \alpha_1 \sigma C_m + s_1 - s_2 &\le 0 \\
  \alpha_2 \sigma C_m + s_2 - s_3 &\le 0 \\
  \alpha_1 \sigma (C_m + C_p) + s_1 - \xi &\le 0 \\
  \alpha_2 \sigma (C_m + C_p) + s_2 - \xi &\le 0 \\
  \alpha_3 \sigma (C_m + C_p) + s_3 - \xi &\le 0
  \end{aligned}$$

  The generated matrix would be:

  $$f = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

  $$Aeq = [1 \quad 1 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0]; \; beq = [1]$$

$$A=\begin{bmatrix} 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ \sigma C_m & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & \sigma C_m & 0 & 0 & 1 & -1 & 0 \\ \sigma(C_m + C_p) & 0 & 0 & 1 & 0 & 0 & -1 \\ 0 & \sigma(C_m + C_p) & 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & \sigma(C_m + C_p) & 0 & 0 & 1 & -1 \end{bmatrix}; \; b=\begin{bmatrix} -r_1 \\ -r_2 \\ -r_3 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$LB=\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$UB=\begin{bmatrix} \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \end{bmatrix}$$

- **Compute Completion Time**

  To compute the completion time, this procedure will call the **linprog**, a linear programming solver that is available in MATLAB:

  E=linprog(f, A, b, Aeq, beq, LB, UB);

- **Compute Completion Time (Lin et al. formula)**

  This procedure will compute the completion time of a job execution according to Lin et al. formula (2007b, 2007c).

- **Plot Data to Graph**

  The completion time generated by both formulas will be plotted on the same graph.

- **Update Simulation Cycle & Load Size σ**

  Simulation cycle will be updated in steps of 1 and stopped at *SimulationCycle*=20. For each of the simulation cycle, we increased the load size in steps of 5.

5.5 Experimental Evaluation

5.5.1 Performance Comparison

In this section, we compare the completion time computed by our LP with the completion time computed by the approximation approach proposed in (Lin et al., 2007d). As shown in Figure 5.3 through Figure 5.8, in all cases the LP based formula computed a lower completion time compared to the one computed by the approximation approach of (Lin et al., 2007b, 2007c).

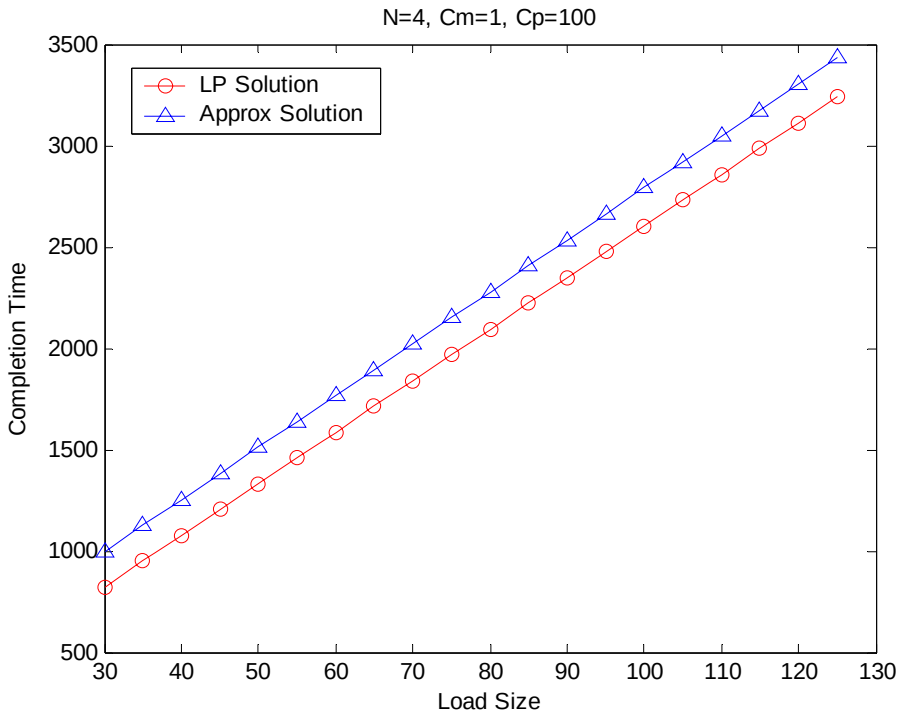

Figure 5.3: Comparison - computed completion time when $n = 4$

Observe that as the number of processors n increases, the relative improvement of our approach to the approximation approach increases. These observations may be explained as follows. Recall that the approximation approach uses the processor availability to estimate the completion time. As n increases it is more likely that takes on a larger value.

For example, these are the release times we used for $n=16$: [19,111, 111, 255, 321, 321, 321, 763, 763, 774, 907, 935, 1016, 1054, 1168, 1390]; this means that the r_n used when $n=16$ is 1390. In (2007b and 2007c) Lin et al. derived an upper bound of minimum number of processors for a job to complete before its deadline by proving that:

$$\hat{\xi}(\sigma,n) \le \frac{1-\beta}{1-\beta^n}\sigma(C_m + C_p) \tag{5.8}$$

$$\hat{\xi}(\sigma,n) \le \xi(\sigma,n) \tag{5.9}$$

And use the equation to estimates the completion time as:

$$C(n) = r_n + \hat{\xi}(\sigma,n) \le r_n + \frac{1-\beta}{1-\beta^n}\sigma(C_m + C_p) \tag{5.10}$$

We have shown via Example 1 that the performance of this approximation approach is non-optimal since their computation time is inflated by the value of r_n and $r_n \to \infty$. Thus, as shown in Figure 5.3 through Figure 5.8, as n increases, Lin et al. formula (2007b, 2007c) is more likely to takes on a larger value of r_n and uses these values to estimates the completion time. Consequently, this formula will compute a larger completion time.

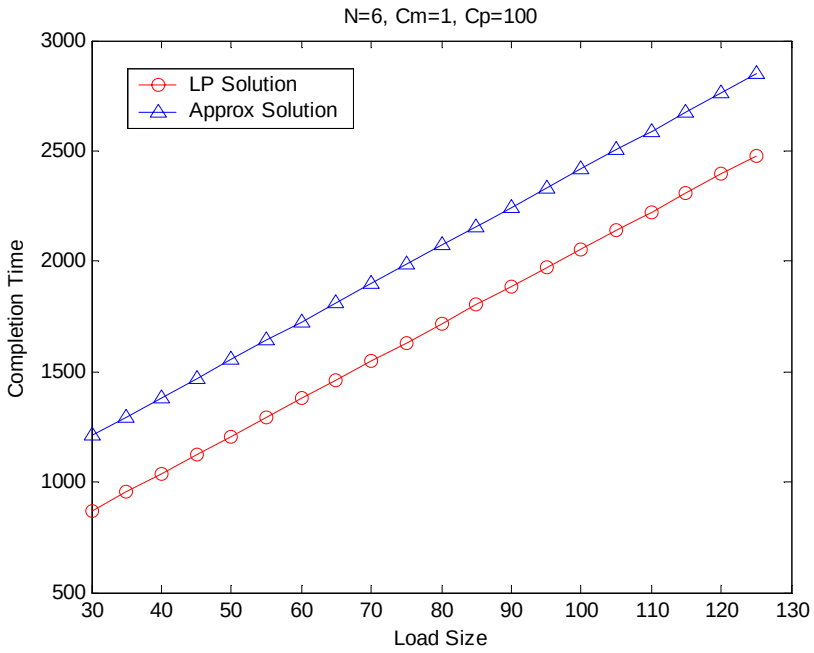

Figure 5.4: Comparison - computed completion time when $n=6$

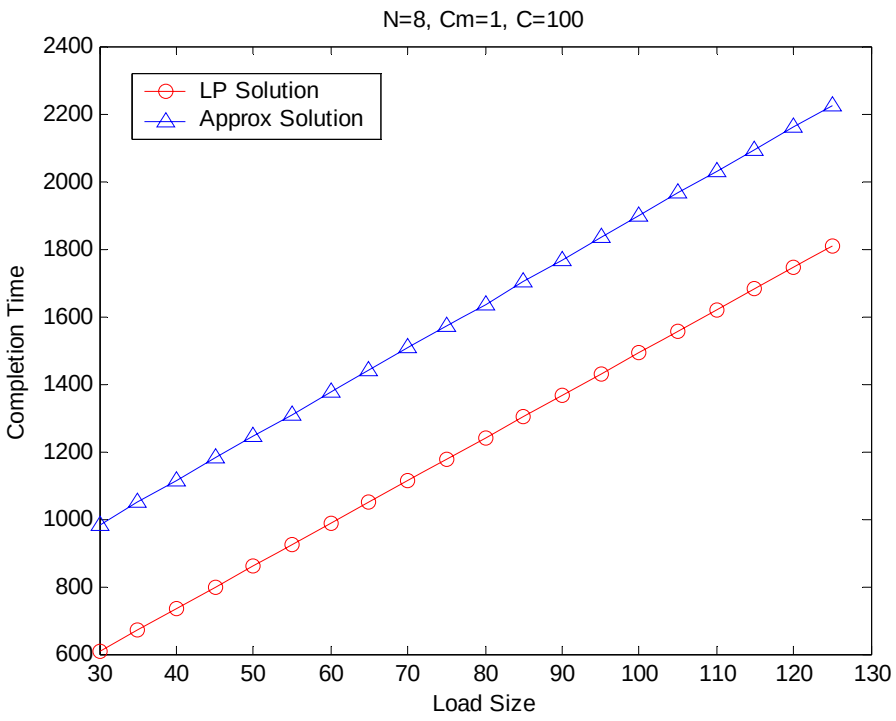

Figure 5.5: Comparison - computed completion time when $n = 8$

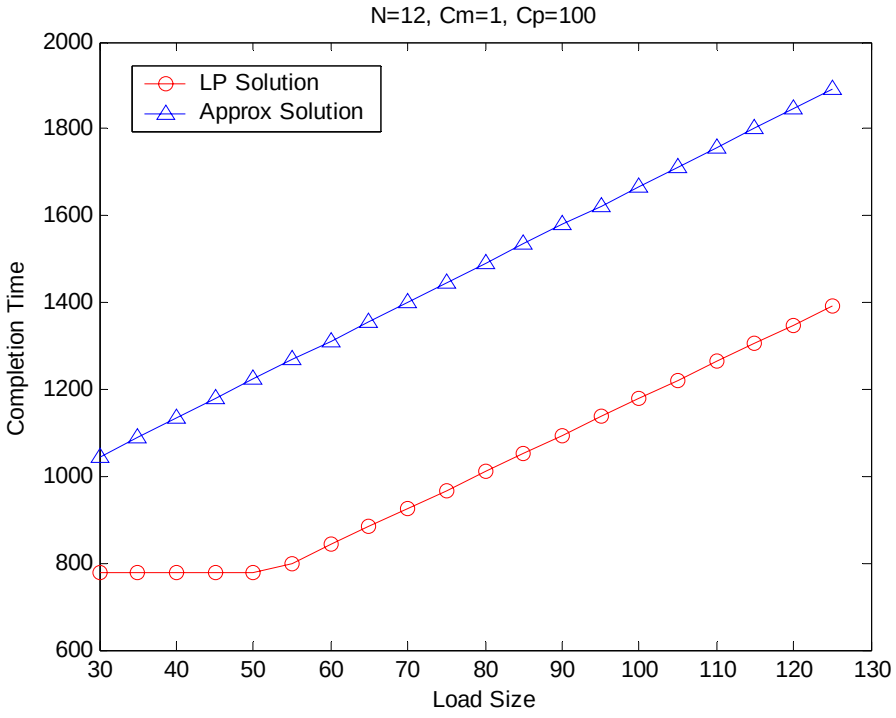

Figure 5.6: Comparison - computed completion time when $n = 12$

For greater detail, let us consider the following examples: Given a load size $\sigma = 60$, $C_m = 1$, $C_p = 100$, $n = 8$ with each processor release time as $r_1 = 194$, $r_2 = 207$, $r_3 = 207$, $r_4 = 365$, $r_5 = 381$, $r_6 = 428$, $r_7 = 524$ and $r_8 = 524$. Using this values in Equation 5.10, the completion time computed by Lin et al. formula (2007b, 2007c) is:

$$\begin{aligned} C(n) &= r_n + \frac{1-\beta}{1-\beta^n}\sigma(C_m + C_p) \\ &= 524 + \frac{1-0.9901}{1-0.9235} \times 60 \times (1+100) \\ &= 1308 \end{aligned}$$

Using the same values in our LP-based algorithm and run the simulation program, the computed completion time $\xi = 1113$. The following table shows the exact values of fractions α_i computed by our LP-based algorithm, the processors's release time r_i, the start time s_i computed by the LP-based algorithm, the sending and computation time of each fractions ($\alpha_i \sigma C_m + \alpha_i \sigma C_p$) assigned to each processors and the completion time ξ in the final column. Particularly in this example, the completion time computed by our algorithm is approximately 15% lesser than the one computed by Lin et al. formula (Lin 2007b, 2007c).

Table 5.1: Fraction α_i values and calculations of completion time ξ

i^{th} Processor	α_i	r_i	s_i	$\alpha_i \sigma C_m + \alpha_i \sigma C_p$	$\xi = s_i + \alpha_i \sigma (C_m + C_p)$
1	0.1517	194	194	919	1113
2	0.1495	207	207	906	1113
3	0.1480	207	216	897	1113
4	0.1234	365	365	748	1113
5	0.1208	381	381	732	1113
6	0.1131	428	428	685	1113
7	0.0972	524	524	589	1113
8	0.0962	524	530	583	1113

Based on the experiments, we observed that our algorithm is superior for use in admission control – i.e., in determining whether to accept or reject incoming jobs based on whether they will meet their deadlines or not. This follows since the completion time computed by our approach is much smaller than the one computed by the approximation approach of (Lin et al., 2007b, 2007c), and it is hence far more likely that our approach will meet any given job's deadline.

Although we do not present any simulation results concerning the likelihood of meeting or missing deadlines, it is easy to observe why this is likely to be the case from our graphs when $n = 12$, $n = 16$ and $n = 20$. Consider, for example the $n = 20$ case, and consider jobs of the different sizes all with a deadline of 1500. None of these jobs would be deemed to meet their deadlines by the approximation approach. As a result, the scheduler will simply reject these jobs even though, as indicated by our efficient test, they would in fact have met their deadlines.

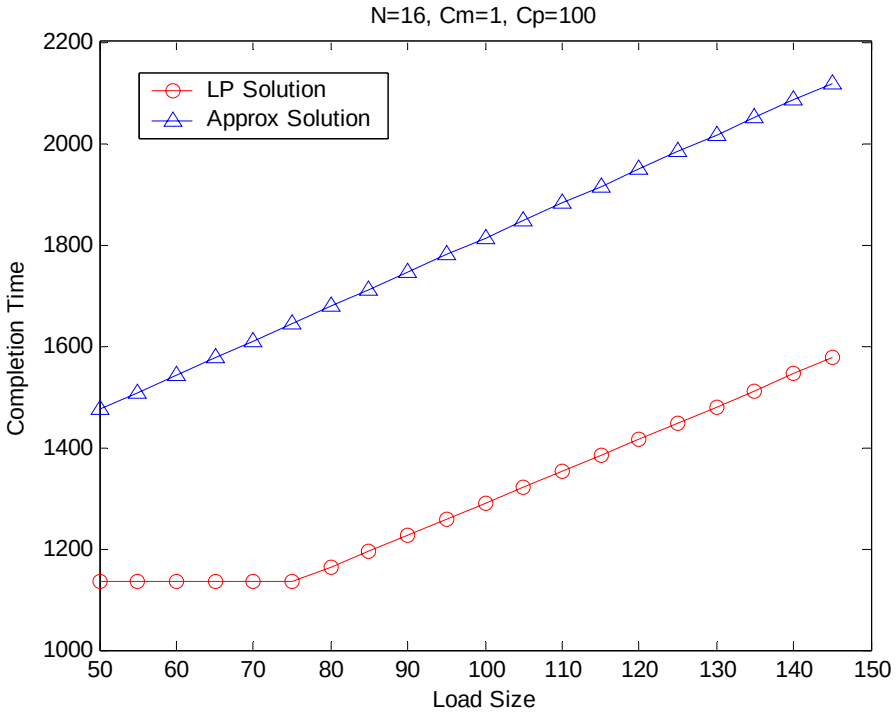

Figure 5.7: Comparison - computed completion time when $n = 16$

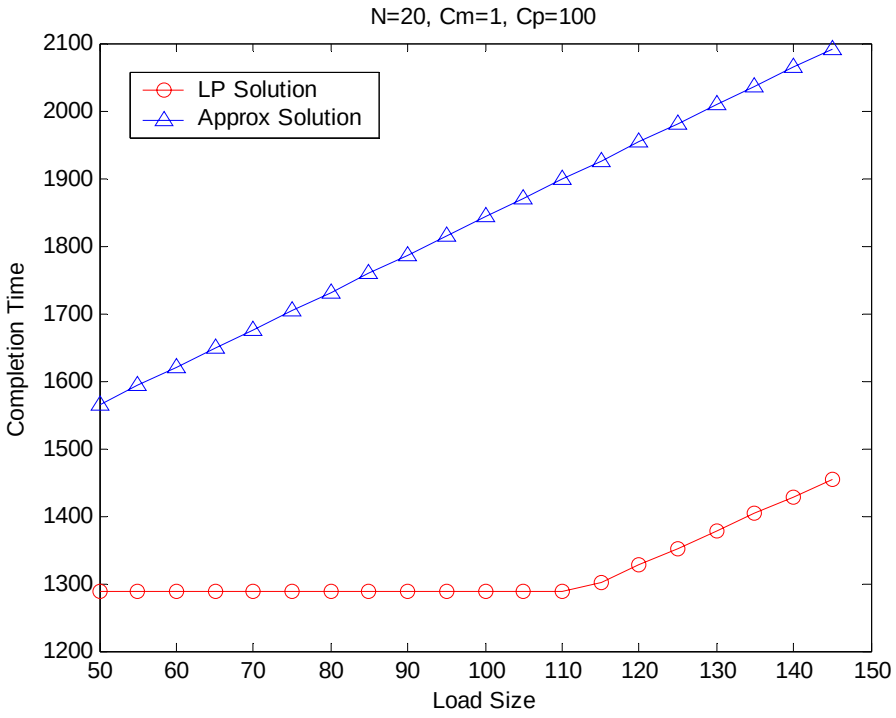

Figure 5.8: Comparison - computed completion time when $n = 20$

We also studied the behavior of our LP-based approach with different cluster parameters. In the simulations depicted in Figure 5.9, we set C_m to values of 1, 3, 5, 7, 9 while maintaining $C_p = 100$. And for each set of this simulation we increased the load size in steps of 10 units. In the simulations depicted in Figure 5.10 we ran sets of simulations with C_p values set to 100, 110, 120, 130, 140 while maintaining $C_m = 1$. As observed, the completion time increases as C_m or C_p increases.

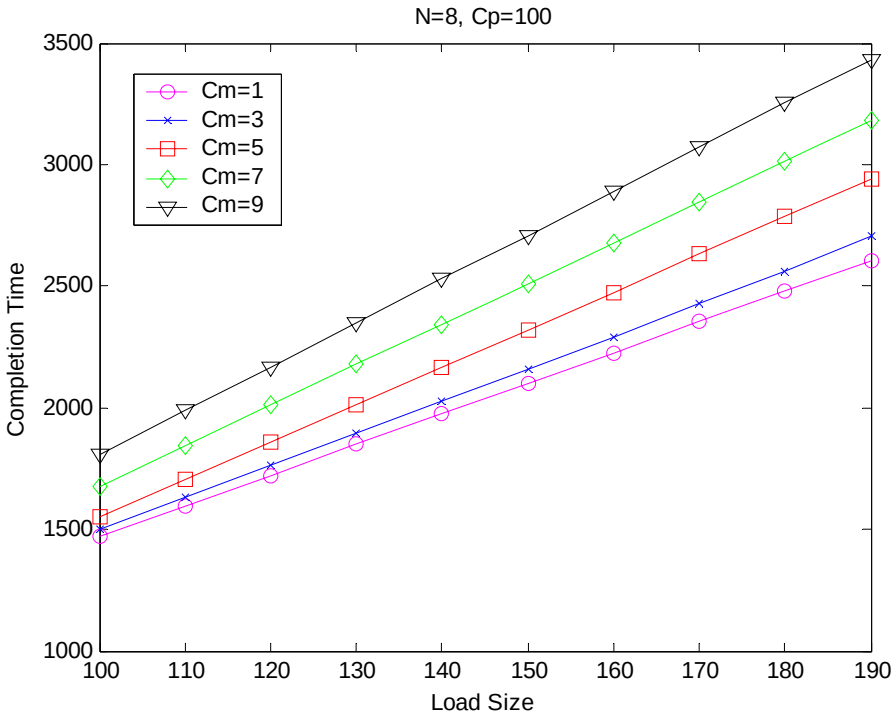

Figure 5.9: Computed completion time with various C_m values

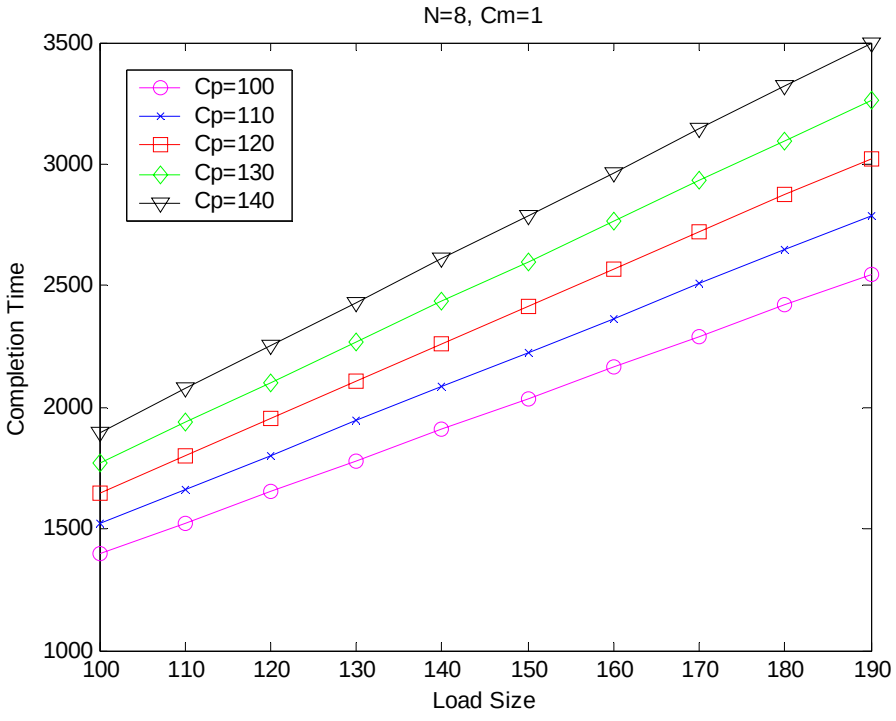

Figure 5.10: Computed completion time with various C_p values

5.5.2 Heterogeneous Platforms

As we mentioned in section 5.3, our LP is easily modified to compute the earliest completion time upon more general heterogeneous platforms. Following is an example of the LP specification for heterogeneous clusters of $n = 4$.

(1) $$\alpha_1 + \alpha_2 + \alpha_3 + \alpha_4 = 1$$

(2)
$$r_1 \le s_1$$
$$r_2 \le s_2$$
$$r_3 \le s_3$$
$$r_4 \le s_4$$

(3)
$$s_2 \ge s_1 + \alpha_1 \sigma C_{m1}$$
$$s_3 \ge s_2 + \alpha_2 \sigma C_{m2}$$
$$s_4 \ge s_3 + \alpha_3 \sigma C_{m3}$$

(4)
$$s_1 + \alpha_1 \sigma (C_{m1} + C_{p1}) \le \xi$$
$$s_2 + \alpha_2 \sigma (C_{m2} + C_{p2}) \le \xi$$
$$s_3 + \alpha_3 \sigma (C_{m3} + C_{p4}) \le \xi$$
$$s_4 + \alpha_4 \sigma (C_{m4} + C_{p4}) \le \xi$$

Note that, to specify a heterogeneous platform, the only modifications are in the Constraint numbers (3) and (4), where C_m and C_p values are involved. Recall that C_m is the data communication cost and C_p is the computation cost for each processor in a cluster.

5.5.3 Effect of Number of Processors

In this section we study the behavior of this LP approach for different numbers of processors in the cluster. We used the cluster parameters $C_m = 1$ and $C_p = 100$, and ran 5 sets of simulations with $n = 4$, $n = 8$, $n = 12$, $n = 16$ and $n = 20$. In each set we increased the load size in steps of 10 units. As shown in Figure 5.11, the completion time decreases significantly with increasing n, particularly for larger load sizes. And when $n = 20$, the completion time for each job does not increase much with increasing load size. For a certain kind of system, where there are jobs with hard deadlines to be met, it makes sense to allocate more processors to these jobs so that they will finish their executions earlier, and release the processors for other incoming jobs. This reinforces a finding first reported in (Chuprat and Baruah, 2007) and described in Chapter 3 of this dissertation, that a scheduling framework which allocates all processing nodes to one job at a time performs very well when compared to a scheduling framework which allocates the minimum number of nodes needed to just meet the job's deadline

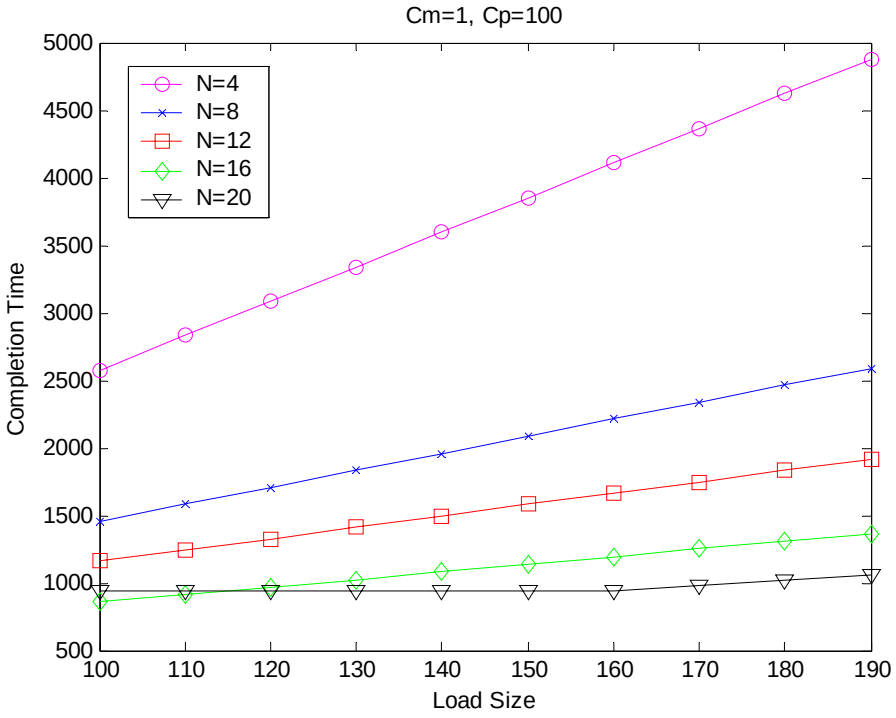

Figure 5.11: Computed completion time with various n values

5.6 Summary

In this chapter, we studied scheduling problems in RT-DLT when applied to clusters in which different processors become available at different time-instants. We proposed an LP based formula to efficiently determine the earliest completion time for the job on a given processors. Through extensive experimental evaluation, we have shown that this LP based formula significantly improves on the heuristic approximations that were the only techniques previously known for solving these problems.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

6.1 Summary

Many future computing applications such as automated manufacturing systems, military systems, high speed telecommunication systems, flight control systems, etc., have significant real-time components. Such real-time application systems demand complex and significantly increased functionality and are increasingly being implemented upon multiprocessor platforms, with complex synchronization, data-sharing and parallelism requirements.

However, current formal models of real-time workloads were designed within the context of uniprocessor real-time systems; hence, they are often not able to accurately represent salient features of multiprocessor real-time systems. Furthermore, they may impose additional restrictions (“additional" in the sense of being mandated by the limitations of the model rather than the inherent characteristics of the platform) upon system design and implementation. One particular restriction that has been extended from uniprocessor models to multiprocessor ones is that they do not allow task parallel execution.

Researchers have recently attempted to overcome this shortcoming by applying workload models from Divisible Load Theory (DLT) to real-time systems. The resulting theory, referred to as Real-time Divisible Load Theory (RT-DLT). RT-DLT holds great promise for modeling an emergent class of massively parallel real-time workloads. However, the theory needs strong formal foundations before it can be widely used for the design and analysis of hard real-time safety-critical applications.

This thesis presents our works in obtaining such formal foundations, by generalizing and extending recent results and concepts from multiprocessor real-time scheduling theory. We summarize the contributions presented in this thesis in Section 6.2. We list a number of open questions that together comprise a future research agenda arising from this thesis in Section 6.3.

6.2 Contributions and Significance

The research performed as part of this thesis significantly advances the state of the art of RT-DLT. Most prior work has been simulation-based and hence not applicable to the design and analysis of hard-real-time systems (where even a single deadline failure is unacceptable and hence guarantees are required during system design time that all timing constraints will be met); to our knowledge, the work reported in this thesis is the first to be able to make such guarantees. As a consequence, we have enabled the use of RT-DLT for the design and analysis of safety-critical systems.

The specific technical contributions obtained during this research are:

i. We have investigated the application of Divisible Load Theory (DLT) models to real-time workloads. Specifically, we have examined the initial work of Lin et al. (2006a, 2006b and 2007a) and their apparently anomalous findings with respect to a scheduling framework integrating DLT and EDF.

ii. We have used recent results from traditional multiprocessor scheduling theory to provide satisfactory explanations for the apparently anomalous observations identified by Lin et al. in (2006a, 2006b and 2007a).

iii. We have investigated the application of DLT to real-time scheduling and report the findings in Chapter 4 and 5. Specifically, we addressed the scheduling problems in RT-DLT when applied to clusters in which different processors become available at different time-instants.

iv. We have devised an efficient algorithm to determine the minimum number of processors that must be assigned to a job in order to guarantee that it meets its deadline — on clusters in which all processors are not simultaneously available. We have also shown that our solution significantly improved the approximate algorithms found in (Lin et al., 2007b, 2007c).

v. We have formulated the problem of determining the completion time of a given divisible job upon a specified number of processors as a Linear Programming (LP) problem. Based on this LP approach, we have improved the non-optimal approximate algorithms found in (Lin et al., 2007b, 2007c).

We summarized the research the contributions made and list of publications in Figure 6.1.

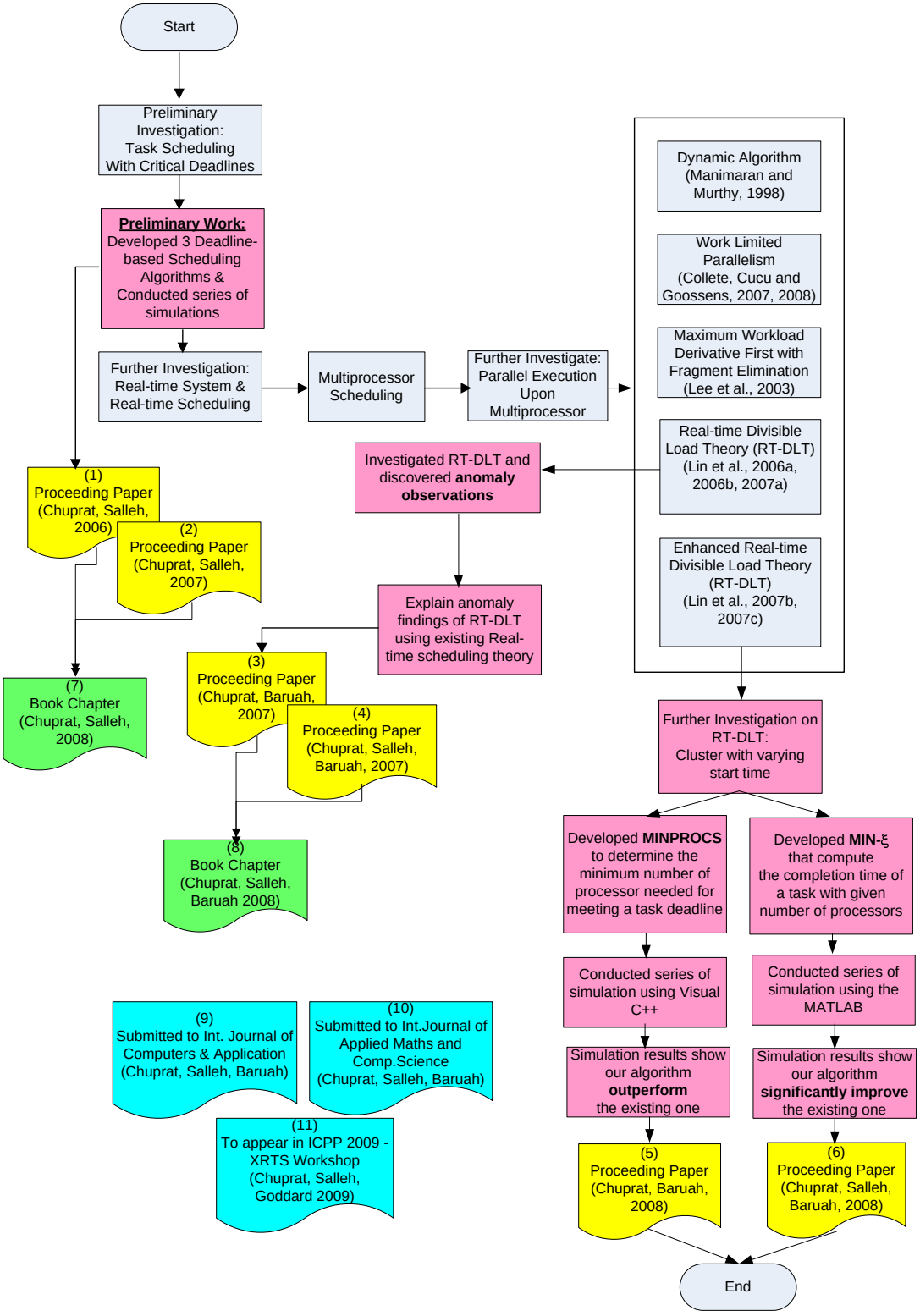

Figure 6.1: Summary of Contributions and Publications

6.3 Future Research

Real-time divisible load theory (RT-DLT), as developed by Lin et al. (2006a, 2006b, 2007a, 2007b, 2007c), has the potential to provide a solid theoretical foundation for the provision of real-time performance guarantees while executing arbitrarily divisible workloads on parallel computing clusters. However this is an emerging area of research; therefore, there are many open areas of research only some of which were addressed in this thesis. In this subsection, we briefly list some potential avenues for future research extending the results of this thesis.

i. Task model

In this thesis we have only considered the *sporadic* task model. *Periodic* and *aperiodic* task models are among other widely used task models in real-time systems. It would be worth extending this work to be applicable for these additional models as well.

ii. Real-time Scheduling Algorithms

We have mainly discussed the EDF scheduling algorithm and integrated this algorithm into the RT-DLT scheduling framework. There are many other algorithms that can potentially be integrated; among these are the Earliest Deadline Zero Laxity (EDZL) and Deadline Monotonic (DM) scheduling algorithms.

iii. Multi-round DLT

We have so far investigated and developed a single-round DLT algorithm. We observe that, some more complex problems such as scheduling with blocking/reservation (Mamat et al., 2008) may be efficiently solvable with Multi-round DLT algorithm.

iv. Network model

In this thesis, we have restricted our attention to the *single-level tree* topology. However, there are several different network topologies, such as stars, meshes, and trees that have been extensively studied in DLT. We believe exploring these additional network topologies in the context of RT-DLT offers vast opportunities for further research.

v. Network with Front-end Processor

Recall that, one assumption in Lin et al. (2006a, 2006b, 2007a) which we note is a bit different from the original work of DLT is that the head node is assumed to lack of front-end processing capabilities and hence not participate in the computation. It would be interesting to extend this work to allow for head-nodes with front-end processing capabilities.

REFERENCES

Anderson, J. and Srinivasan, A. (2004). Mixed Pfair/ ERfair scheduling of asynchronous periodic tasks. *Journal of Computer and System Sciences*, 68(1):157–204.

ATLAS (AToroidal LHC Apparatus) Experiment, CERN (European Lab for Particle Physics). *ATLAS Web Page*. http://atlas.ch/.

Baker, T.P. and Baruah, S. (2007). Schedulability analysis of multiprocessor sporadic task systems. *Handbook of Real-Time and Embedded Systems*. Chapman Hall/ CRC Press.

Baker, T. P., Cirinei, M. and Bertogna, M. (2008). EDZL scheduling analysis. *Real-Time Systems*. 40(3): 264-289.

Balafoutis, E., Paterakis, M., Triantafillou, P., Nerjes, G., Muth, P. and Weikum, G. (2003). Clustered Scheduling Algorithms for Mixed Media Disk Workloads in a Multimedia Server. *Special issue of Cluster Computing on Divisible Load Scheduling*, Kluwer Academic Publishers, 6(1):75-86.

Barlas, G. and Bharadwaj, V. (2000). Theoretical and Practical aspects of multi-installment distribution for the processing of multiple divisible loads on bus networks. *Proceedings of International Conference on Computing and Information*.

Barlas, G. and Bharadwaj, V. (2004). Quantized Load Distribution for Tree and Bus-connected Processors. *Parallel Computing*, 30(7): 841-865.

Baruah, S., Mok, A. and Rosier, L. (1990). Preemptively scheduling hard-real-time sporadic tasks on one processor. *Proceedings of the 11th Real-Time Systems Symposium*, pages 182–190, Orlando, Florida. IEEE Computer Society Press.

Baruah, S., Koren, G., Mao, D., Mishra, B., Raghunathan, A., Rosier, L., Shasha, D. and Wang, F. (1991). On the competitiveness of on-line real-time task scheduling. *Real-Time Systems*, 4:125-144.

Baruah, S., Gehrke, J. and Plaxton, G. (1995). Fast scheduling of periodic tasks on multiple resources. *Proceedings of the Ninth International Parallel Processing Symposium*, pages 280–288. IEEE Computer Society Press.

Baruah, S., Cohen, N., Plaxton, G. and Varvel, D. (1996). Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625.

Baruah, S. (2002). Robustness results concerning EDF scheduling upon uniform multiprocessors. *Proceedings of the EuroMicro Conference on Real-Time Systems*, 95–102, Vienna, Austria, June, IEEE Computer Society Press.

Baruah, S. and Goossens, J. (2003). Rate-monotonic scheduling on uniform multiprocessors. *IEEE Transactions on Computers*, 52(7):966–970.

Baruah, S. (2004). Optimal Utilization Bounds for the Fixed-Priority Scheduling of Periodic Task Systems on Identical Multiprocessors. *IEEE Transactions on Computers*, 53(6):781-784.

Baruah, S. (2006). The Non-preemptive Scheduling of Periodic Tasks upon Multiprocessors. *Real-Time Systems*, 32, 9–20.

Bharadwaj, V., Ghose, D. and Mani, V. (1995). Multi-installment Load Distribution in Tree Networks with Delays. *IEEE Transactions on Aerospace & Electronic Systems*, 31(2): 555-567.

Bharadwaj, V., Ghose, D., Mani, V. and Robertazzi, T.G. (1996). *Scheduling Divisible Loads in Parallel and Distributed Systems*, IEEE Computer Society Press, Los Alamitos CA, September.

Bharadwaj, V., Li, X. and Chung, K.C. (2000). On the Influence of Start-up Costs in Scheduling Divisible Loads on Bus Networks. *IEEE Transactions on Parallel and Distributed Systems*. 11(12): 1288-1305.

Bharadwaj, V. and Ranganath, S. (2002). Theoretical and Experimental Study of Large Size Image Processing Applications using Divisible Load Paradigm on Distributed Bus Networks. *Image and Vision Computing*, Elsevier Publishers, 20(13-14):917-936.

Bharadwaj, V., Ghose, D. and Robertazzi, T. G. (2003). Divisible load theory: A new paradigm for load scheduling in distributed systems. *Cluster Computing*, 6(1):7-17.

Buttazzo, G., Lipari, G., Abeni, L. and Caccamo, M. (2005). Soft Real-Time Systems: Predictability vs. Efficiency. Springer Science and Business Media, Inc., 233 Spring Street, New York, NY 10013, USA.

Buttazzo, G. (2004). *Hard Real-time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer Science and Business Media, Inc., New York, NY 10013, USA.

Chuprat, S. and Baruah, S. (2007). Deadline-based scheduling of divisible real-time loads. *Proceedings of the ICSA International Conference on Parallel and Distributed Computing Systems*, Las Vegas, Nevada.

Chuprat, S. and Baruah, S. (2008). Scheduling Divisible Real-Time Loads on Clusters with Varying Processor Start Times. *Proceedings of 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA, 15-24.

Chuprat, S., Salleh, S. and Baruah, S. (2008). Evaluation of a linear programming approach towards scheduling divisible real-time loads. *Proceedings of the International Symposium on Information Technology* (co-sponsored by the IEEE), Kuala Lumpur, Malaysia.

Cirinei, M. and Baker, T. P. (2007). EDZL Scheduling Analysis. *Proceedings of the 19th Euromicro Conference on Real-Time Systems (ECRTS)*, IEEE Computer Society, Washington, DC, 9-18.

Collette, S., Cucu, L. and Goossens, J. (2008) Integrating Job Parallelism in Real-Time Scheduling Theory. *Information Processing Letters*, vol. 106(5): 180-187, May.

Collette, S., Cucu, L., and Goossens, J. (2007). Algorithm and complexity for the global scheduling of sporadic tasks on multiprocessors with work-limited parallelism. *Proceedings of the 15th International Conference on Real-Time Systems*, Nancy, France, March.

Compact Muon Solenoid (CMS) Experiment for the Large Hadron Collider at CERN (European Lab for Particle Physics). *CMS Web Page*. http://cmsinfo.cern.ch/Welcome.html/.

Cottet, F., Delacroix, J., Kaiser, C. and Mammeri, Z. (2002). *Scheduling in Real-Time Systems*. England: John Wiley & Sons Ltd.

Dantzig, G. B. (1963). *Linear Programming and Extensions*. Princeton University Press, 1963.

Dertouzos, M. L. (1974). Control robotics: The procedural control of physical processes. *Information Processing*. 74.

Dertouzos, M. L. and Mok, A.K. (1989). Multiprocessor on-line scheduling of hard real-time tasks. *Transactions on Software Engineering* 15(12):1497-1506.

Dhall, S. K. and Liu, C. L. (1978). On a real-time scheduling problem. *Operations Research*, 26:127–140.

Drozdowski, M. and Glazek, W. (1999). Scheduling Divisible Loads in a Three-Dimensional Mesh of Processors. *Parallel Computing*, 25(4): 381-404.

Drozdowski, M. and Lawenda, M. (2005). On Optimum Multi-installment Divisible Load Processing. Euro-Par 2005 Parallel Processing, *Lecture Notes in Computer Science 3648*, Springer, 231-240.

Drozdowski, M. and Wolniewicz, P. (2006). Optimum divisible load scheduling on heterogeneous stars with limited memory. *European Journal of Operational Research*, Elsevier, 172(2): 545-559.

Glazek, W. (2003). A Multistage Load Distribution Strategy for Three Dimensional Meshes. *Special issue of Cluster Computing on Divisible Load Scheduling*, Kluwer Academic Publishers, 6(1):31-40.

Goossens, J., Funk, S. and Baruah, S. (2003). Priority Driven Scheduling of Periodic Task Systems on Multiprocessors. *Journal of Real-Time System*, 25, 187-205.

Graham, R.L. (1976). Bounds on the performance of scheduling algorithms. *Computer and Job Scheduling Theory*, 165-227. John Wiley and Sons.

Jeffay, K., Stanat, D., and Martel, C. (1991). On non-preemptive scheduling of periodic and sporadic tasks. *Proceedings of the 12th Real-Time Systems Symposium*, 129–139, San Antonio, Texas. IEEE Computer Society Press.

Karmakar. N. (1984). A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395.

Khachiyan, L. (1979). A polynomial algorithm in linear programming. *Dokklady Akademiia Nauk SSSR*, 244:1093–1096.

Lee, W., Hong, S. J. and Kim, J. (2003). On-line scheduling of scalable real-time tasks on multiprocessor systems. *Journal of Parallel and Distributed Computing*, 63(12):1315-1324.

Leung, J. and Whitehead, J. (1982). On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2:237–250.

Lin, X., Lu, Y., Deogun, J. and Goddard, S. (2006a). Real-time divisible load scheduling for cluster computing. *Technical Report UNL-CSE-2006-0016, Department of Computer Science and Engineering*, The University of Nebraska at Lincoln.

Lin, X., Lu, Y., Deogun, J. and Goddard, S. (2006b). Real-time divisible load scheduling for clusters. *Proceedings of the Real-Time Systems Symposium – Work-In-Progress Session*, pages 9–12, Rio de Janerio, December.

Lin, X., Lu, Y., Deogun, J. and Goddard, S. (2007a) Real-time divisible load scheduling for cluster computing. *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS)*.

Lin, X., Lu, Y., Deogun, J. and Goddard, S. (2007b). Real-Time Divisible Load Scheduling with Different Processor Available Times. *Proceedings of International Conference on Parallel Processing (ICPP)*, Xian, China, September.

Lin, X., Lu, Y., Deogun, J. and Goddard, S. (2007c). Enhanced Real-Time Divisible Load Scheduling with Different Processor Available Times. *Proceedings of 14th International Conference on High Performance Computing (HiPC)*, Goa, India, December.

Liu, C. and Layland, J. (1973). Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of ACM*. 20(1) 46-61.

Liu, C. L. (1969). Scheduling algorithms for multiprocessors in a hard real-time environment. *JPL Space Programs Summary*, 37-60, II:28-31.

Liu, J. W. S. (2000). *Real-Time Systems*. Prentice-Hall, Inc., Upper Saddle River, New Jersey 07458.

Lopez, J., Diaz, J. and Garcia, L. (2004). Utlization Bounds for EDF Scheduling on Real-Time Multiprocessor Systems. *Real-Time Systems*, 28:39-68.

Mamat, A., Lu, Y., Deogun, J. and Goddard, S. (2008). Real-Time Divisible Load Scheduling with Advance Reservation. *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, IEEE Computer Society, Washington, DC, 37-46.

Manimaran, G. and Murthy, C. S. R. (1998). An efficient dynamic scheduling algorithm for multiprocessor real-time systems. *IEEE Transaction on Parallel and Distributed Systems*, 9(3):312–319.

Marchal, L., Yang, Y., Casanova, H. and Robert, Y. (2005). A Realistic Network/Application Model for Scheduling Divisible Loads on Large-Scale Platforms. Proceedings of *International Parallel and Distributed Processing Symposium (IPDPS)*.

Mok, A. K. (1983). Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment. *PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology*.

Phillips, C. A., Stein, C., Torng, E. and Wein, J. Optimal time-critical scheduling via resource augmentation. (1997). *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, pages 140-149, El Paso, Texas, 4-6 May.

Phillips, C. A., Stein, C., Torng, E., and Wein, J. (2002). Optimal time-critical scheduling via resource augmentation. *Algorithmica*, 32(2):163–200.

Piriyakumar, D.A.L. and Murthy C.S.R. (1998). Distributed Computation for a Hypercube Network of Sensor-Driven Processors with Communication Delays Including Setup Time. *IEEE Transactions on Systems, Man and Cybernetics-Part A: Systems and Humans*, 28(2): 245-251.

Ramamritham, K., Stankovic, J.A. and Shiah, P.F. (1990). Efficient scheduling algorithms for real-time multiprocessor systems. *IEEE Transaction on Parallel and Distributed Systems*, 1(2):184-194, April.

Robertazzi, T.G. (1993). Processor Equivalence for a Linear Daisy Chain of Load Sharing Processors. *IEEE Transactions on Aerospace and Electronic Systems*, 29(4): 1216-1221.

Robertazzi, T. G. (2003). Ten reasons to use divisible load theory. *Computer*, 36(5):63–68.

Robertazzi, T.G. (2008). *Divisible Load Scheduling Research Websites*: www.ee.sunysb.edu/~tom/dlt.html.

Salleh, S., Zomaya, A.Y, Olariu, S. and Sanugi, B. (2005). *Numerical simulations and case studies using Visual C++.Net*. Wiley-Interscience, Hoboken, NJ, USA.

Sohn, J. and Robertazzi, T.G. (1993). Optimal Load Sharing for a Divisible Job on a Bus Network. *Proceedings of the 1993 Conference on Information Sciences and Systems*, The Johns Hopkins University, Baltimore MD, 835-840.

Sohn, J. and Robertazzi, T.G. (1998a). An Optimal Load Sharing Strategy for Divisible Jobs with Time-Varying Processor Speeds. *Proceedings of the Eighth International (ISCA) Conference On Parallel and Distributed Computing Systems*, September, Orlando, Florida, 27-32.

Sohn, J. and Robertazzi, T.G. (1998b). An Optimal Load Sharing Strategy for Divisible Jobs with Time-Varying Processor Speeds. *IEEE Transactions on Aerospace and Electronic Systems*, 34(3): 907-923.

Srinivasan, A. and Baruah, S., (2002). Deadline-based scheduling of periodic task systems on multiprocessors. *Information Processing Letters* . 84, 93-98.

Stankovic, J.A. and Ramamritham, K., (1985). Evaluation of a Flexible Task Scheduling Algorithm for Distributed Hard Real-Time Systems. *IEEE Transactions on Computers*, C-34(12):1130-1143.

Yang. Y. and Casanova, H. (2003). UMR: A Multi-Round Algorithm for Scheduling Divisible Workloads. *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*.

Yang. Y. and Casanova, H. (2005). Multi-round algorithms for scheduling divisible workloads. *IEEE Transactions on Parallel and Distributed Systems* (TPDS), 16(11):1092–1102.

Xiaolin, L., Bharadwaj, V. and Ko, C.C. (2003). Distributed Image Processing on a Network of Workstations. *International Journal of Computers and Applications*, ACTA Press, 25(2): 1-10.

APPENDIX A

PAPERS PUBLISHED DURING THE AUTHOR'S CANDIDATURE

From the material in this thesis there are, at the time of submission, papers which have been published, presented or submitted for publication or presentation as following:

Papers published

Suriayati Chuprat, Shaharuddin Salleh, and Sanjoy Baruah. Evaluation of a linear programming approach towards scheduling divisible real-time loads. Proceedings of the International Symposium on Information Technology (co-sponsored by the IEEE), Kuala Lumpur, Malaysia. August 2008.

Suriayati Chuprat, Sanjoy Baruah. Scheduling Divisible Real-Time Loads on Clusters with Varying Processor Start Times. Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2008), Kaohsiung, Taiwan, August 2008. (26% acceptance rate)

Suriayati Chuprat, Shaharuddin Salleh, and Sanjoy Baruah. Deadline-based Scheduling of Divisible Real-Time Loads with Setup costs and Load balancing Considered, In Work-In-Progress Proceedings of the 28th IEEE Real-Time Systems Symposium (RTSS 2007), Tucson, Arizona, USA, December 2007.

Suriayati Chuprat and Sanjoy Baruah. Deadline-based scheduling of divisible real-time loads. In Proceedings of the ICSA International Conference on Parallel and Distributed Computing Systems, Las Vegas, Nevada, September 2007.

Suriayati Chuprat and Shaharuddin Salleh. A Deadline-Based Algorithm For Dynamic Task Scheduling With Precedence Constraints. In Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN2007), Innsbruck, Austria, February 2007.

Suriayati Chuprat and Shaharuddin Salleh. Scheduling Algorithms in the Soft Real-Time Systems. Presented at the Computer Science and Mathematics Symposium 2006 (CSMS 2006), Terengganu, Malaysia, November 2006.

Book Chapters

Suriayati Chuprat and Shaharuddin Salleh. Deadline-Based Algorithms for Dynamic Scheduling In Soft Real-Time Systems. In Book Chapters of Advances in Planning, Scheduling and Timetabling Volume 2, Universiti Teknologi Malaysia, 2008.

Suriayati Chuprat, Shaharuddin Salleh and Sanjoy Baruah. Applying Divisible Load Theory in Real-Time Multiprocessor Scheduling. In Book Chapters of Advances in Planning, Scheduling and Timetabling Volume 2, Universiti Teknologi Malaysia, 2008.

Paper Accepted for Publication

Suriayati Chuprat, Shaharuddin Salleh and Steve Goddard. Real-time Divisible Load Theory: A Perspective. *To appear in Proceedings of the Workshop of Real-time Systems on Multicore Platforms: Theory and Practice (To be held in conjunction with ICPP'09 - The 2009 International Conference on Parallel Processing).*

Papers Submitted

Suriayati Chuprat, Shaharuddin Salleh, and Sanjoy Baruah. A Linear Programming Approach for Scheduling Divisible Real-time Workloads. *Submitted to International Journal of Applied Mathematics and Computer Science.*

Suriayati Chuprat, Shaharuddin Salleh, and Sanjoy Baruah. Scheduling Divisible Real-Time Workloads on Clusters with Arbitrary Processor Release Times. *Submitted to International Journal of Computers and Applications.*