Summer 7-24-2011

REAL-TIME DIVISIBLE LOAD SCHEDULING FOR CLUSTER COMPUTING

Anwar Mamat
University of Nebraska-Lincoln, anwarmamat@gmail.com

REAL-TIME DIVISIBLE LOAD SCHEDULING FOR CLUSTER COMPUTING

by

Anwar Mamat (Niwaer Ai)

A DISSERTATION

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfillment of Requirements

For the Degree of Doctor of Philosophy

Major: Computer Science

Under the Supervision of Professor Ying Lu

Lincoln, Nebraska

July, 2011

REAL-TIME DIVISIBLE LOAD SCHEDULING FOR CLUSTER COMPUTING

Anwar Mamat (Niwaer Ai), Ph.D.

University of Nebraska, 2011

Advisor: Ying Lu

Cluster computing has become an important paradigm for solving large-scale problems. However, as the size of a cluster increases, so does the complexity of resource management and maintenance. Therefore, automated performance control and resource management are expected to play critical roles in sustaining the evolution of cluster computing. The current cluster scheduling practice is similar in sophistication to early supercomputer batch scheduling algorithms, and no consideration is given to desired quality-of-service (QoS) attributes. To fully avail the power of computational clusters, new scheduling algorithms that provides high performance, QoS assurance, fault-tolerance, energy savings and streamlined management of the cluster resources needs to be developed.

The challenge, however, in developing real-time scheduling algorithms for cluster and grid computing is to support various types of applications. Broadly speaking, computational loads submitted to a cluster can be categorized into three types: sequential, modularly divisible and arbitrarily divisible. An arbitrarily divisible workload model is a good approximation of many real-world applications, e.g., distributed search for a pattern in text, audio, graphical, and database files; distributed processing of big measurement data files; and many simulation problems. All elements in such an application often demand an identical type of processing, and relative to the huge total computation, the processing on each individual element is infinitesimally small. As such applications become a major type of cluster workloads and

thus providing QoS to arbitrarily divisible loads becomes a significant problem for cluster-based research computing facilities.

The problem of providing performance guarantees to divisible load applications has not been studied systematically. The objective of this dissertation is to provide assured QoS performance to cluster and grid applications through the development of new real-time scheduling theory and algorithms, particularly, real-time divisible load scheduling algorithms for cluster computing. We develop and apply real-time scheduling algorithms for cluster computing, providing QoS for the gird and High Performance Computing (HPC) applications. In this dissertation, we address the aforementioned challenges by investigating and developing 1) real-time scheduling algorithms for divisible loads, 2) a real-time scheduling algorithm for divisible loads with advance resource reservation, 3) an efficient real-time divisible load scheduling algorithm for large clusters and 4) feedback-control based real-time divisible load scheduling algorithms that provide predictable performance in unpredictable environments.

ACKNOWLEDGEMENTS

This dissertation would not be possible without the help, support, guidance, and efforts of a lot of people. First, I would like to thank my advisor Dr. Ying Lu for her constant support, patience, and guidance throughout my Ph.D. She has motivated and inspired me to do my research. I have been fortunate to have her as my advisor. I would like to thank Dr. Steve Goddard for his encouragement and inspiration. I am grateful to him for holding me to a high research standard and enforcing strict validations for each research result. I would like to thank Dr. Jitender Deogun for his help with my critical thinking and theoretical analysis. He sets high standards for students and encourages students to think deeper. I learned a lot from his algorithms courses, which in turn contributed significantly to my research. I would also like to express my gratitude to Dr. David Swanson and the Holland Computing Center staff. They have been very helpful to provide essential computing environment for my simulations. They provided me access to the log files of the clusters, which greatly helped my research. I would like to thank to my committee members Dr. Sanjoy Baruah and Dr. Xiao Cheng Zeng for serving in my committee. I would like to thank my colleagues Xuan Lin and Duc Luong for their help and sharing their knowledge. This dissertation is dedicated to my loving wife Railgul and daughter Zilala, my parents Mamatjan and Roshengul. Without their support and love, I would not be able to complete my ph.D.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cluster computing has become an important paradigm for solving large-scale problems. However, as the size of a cluster increases, so does the complexity of resource management and maintenance. Therefore, automated performance control and resource management are expected to play critical roles in sustaining the evolution of cluster computing. Current cluster batch scheduling algorithms do not consider desired quality-of-service (QoS) attributes. To fully avail the power of computational clusters, new scheduling algorithms that provides high performance, QoS assurance, and streamlined management of cluster resources needs to be developed.

Real-time scheduling theory has been very successful in providing deterministic QoS in desktop systems [11, 13, 49]. A significant challenge in developing real-time scheduling algorithms for cluster computing, however, is to support various types of cluster applications. Broadly speaking, computational loads submitted to a cluster can be structured in three primary ways: *indivisible*, *modularly divisible*, and *arbitrarily divisible*. An indivisible load is essentially a sequential job which cannot be further divided, and thus must be assigned to a single processor. *Modularly divisible* loads can be divided a priori into a certain number of subtasks and are often described by a task (or processing) graph. *Arbitrarily divisible* loads, also called embarrassingly parallel workloads, can be partitioned into an arbitrarily large number of independent load fractions. This workload model is a good approximation of many real-world applications [26], e.g., distributed search for a pattern in text, audio, graphical, and database files; distributed processing of big measurement data files; and many simulation problems. Quite a few scientific applications conform to this divisible load task model. Examples of arbitrarily divisible loads can be easily found in high energy and particle physics as well as biometrics. For example, the CMS (Compact Muon Solenoid) [29] and ATLAS (A Toroidal LHC Apparatus) [6] projects, which are associated with the LHC (Large Hadron Collider) at CERN (European Laboratory for Particle Physics), execute cluster-based applications with arbitrarily divisible loads. Usually all elements in such computational loads demand an identical type of processing, and relative to the huge total computation, the processing on each individual element is infinitesimally small. The problem of providing QoS or real-time guarantees for sequential and modularly divisible jobs in distributed systems has been studied extensively. However, despite the increasing importance of arbitrarily divisible applications [68], to the best of our knowledge, the real-time scheduling of arbitrarily divisible loads has not been systematically investigated.

Scheduling of arbitrarily divisible loads represents a problem of great significance for cluster-based research computing facilities such as the U.S. CMS Tier-2 sites [76]. For example, one of the management goals at the University of Nebraska-Lincoln (UNL) Holland Computing Center (a CMS Tier-2 site) is to provide a multi-tiered QoS scheduling framework in which applications "*pay*" according to the response time requested for each job [76]. By monitoring the CMS mailing-list, we have learned that CMS users always want to know task response times when they submit tasks to clusters. However, without a good QoS mechanism, current cluster sites cannot provide these users good response time estimations. Existing real-time cluster scheduling algorithms assume the existence of a task graph for all applications, which are not appropriate for arbitrarily divisible loads. To better manage these high-end clusters and control their performance, we propose to develop new real-time scheduling algorithms that support arbitrarily divisible applications.

Divisible Load Theory (DLT) provides an in-depth study of distribution strategies for arbitrarily divisible loads [68, 9, 79]. The goal of DLT is to exploit parallelism in computational data so that the workload can be partitioned and assigned to several processors such that execution completes in the shortest possible time [9]. DLT has been previously applied to and implemented in Grid computing [84, 42, 78]. Complimentary to that work, we apply DLT in the design of real-time scheduling algorithms for cluster computing; specifically, DLT is applied in the partitioning of applications, such as CMS [29] and ATLAS [6], that execute on a large cluster.

Recently, there has been some study on real-time divisible load scheduling. Lin et al. proposed a real-time divisible load scheduling algorithm and investigated the problem of providing deterministic QoS to arbitrarily divisible applications executing in cluster environments in [45]. They applied DLT to guide task partitioning, to derive task execution function, and to compute the minimum number of processors required to meet its deadline. The proposed algorithm EDF-DLT-MN can optimally partition the workload on allocated nodes so that all subtasks of a task complete at the same time. This algorithm requires that all allocated nodes are available at the same time. If the required number of processors are not available, the task waits for some currently running jobs to finish and free additional processors. This causes a waste of processing power as some processors are idle when the system is waiting for enough processors to become available to start the waiting task. This is called the Inserted Idle Time (IIT) problem. Lin et al. investigated this drawback in [46], where they solved the IIT problem by casting the homogeneous cluster to a heterogeneous cluster. Lee et al. investigated scheduling algorithms for "scalable real-time tasks" running on a multiprocessor system and proposed MWF (Maximum Workload Derivative First) algorithm in [43]. Like divisible load, it assumes that a task can be executed on multiprocessors and as more processors are allocated its pure execution time decreases. Chuprat and Baruah proposed an algorithm that can utilize the IIT, and employed a linear programming approach to compute task execution times [17, 18].

The proposed algorithms in [45, 46, 43, 17, 18], however, have the following limitations:

1. These approaches did not consider the setup cost of divisible loads. The setup cost could come from the delay for starting a remote process; It may also include the time to initiate a network connection and physical network latency etc. It has also been shown that the setup cost for computation can be up to 25 seconds in practice, which is significant for small tasks. When there are setup costs, task execution time no longer decreases monotonically as the number of allocated processors increases. Therefore, in order to avoid waste of resources, the scheduling algorithm has to decide the optimal number of allocated processors that minimizes execution time of a task. Existing approaches did not consider these setup costs and their effects.

2. Previous approaches did not consider the advance reservation of resources. For grid applications that require simultaneous access to multi-site resources, supporting advance reservations in a cluster is important. At the cluster level, some debugging applications, or interactive applications require a specified number of processors to be available at predefined time intervals. Scheduled maintenance and processor down times can also be treated as advance reservations. Existing

Table 1.1: Sizes of OSG Clusters.

Host Name	No. of CPUs
fermigrid1.fnal.gov	41863
osgserv01.slac.stanford.edu	9103
lepton.rcac.purdue.edu	7136
cmsosgce.fnal.gov	6942
osggate.clemson.edu	5727
grid1.oscer.ou.edu	4169
osg-gw-2.t2.ucsd.edu	3804
osg.rcac.purdue.edu	3535
pg.ihepa.ufl.edu	3324
cmsgrid01.hep.wisc.edu	3297
u2-grid.ccr.buffalo.edu	2104
red.unl.edu	1140

divisible load scheduling algorithms do not consider the scenarios where some processors are not available for some period of time due to advance reservations.

3. Previous algorithms do not scale well. Clusters are becoming increasingly bigger and busier. In Table 1.1, we list the sizes of some OSG (Open Science Grid) clusters. As we can see, all of these clusters have more than one thousand CPUs, with the largest providing over 40 thousand CPUs. Figures 1.1 and 1.2 show the number of tasks waiting in the OSG cluster at University of California, San Diego for a 20-hour period, demonstrating that at times there could be as many as 37 thousand tasks in the waiting queue of a cluster. As the cluster size and workload increase, so does the scheduling overhead. For a cluster with thousands of nodes and/or thousands of waiting tasks, the scheduling overhead could be substantial and existing divisible load scheduling algorithms are no longer applicable due to the lack of scalability. For example, to schedule the bursty workload in Figure 1.1, the previously best-known real-time scheduling algorithm [17] takes more than 11 hours to make admission control decisions on the 14,000 tasks that arrive in an hour. This is certainly not acceptable.

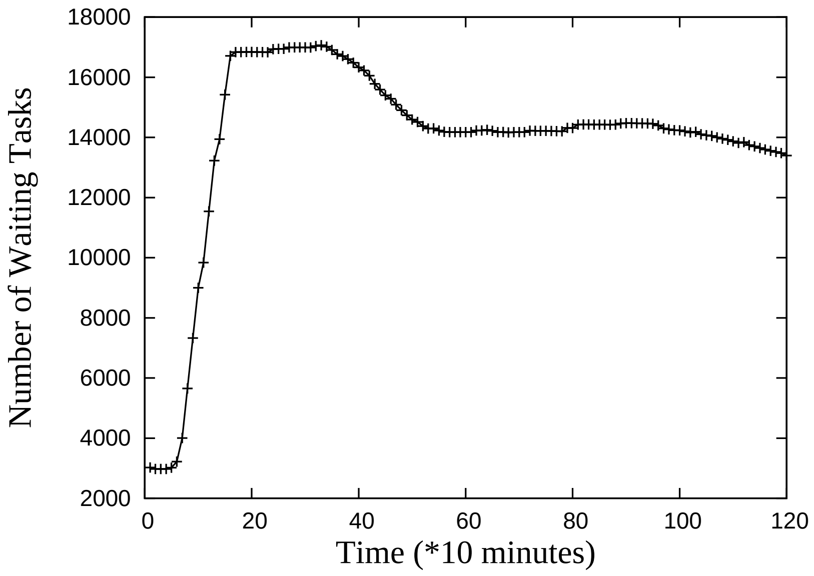


Figure 1.1: Status of a UCSD Cluster (Bursty Arrival).

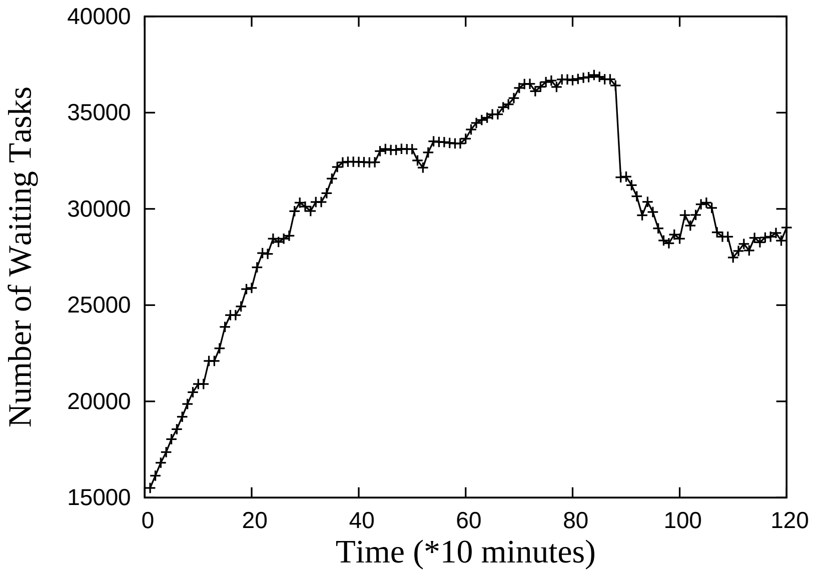


Figure 1.2: Status of a UCSD Cluster (Large Queue).

4. Existing algorithms assume that the task execution time is accurately known or can be derived based on the data size prior to execution. Furthermore, these are "open-loop" scheduling algorithms. Once schedules are created, they are not adjusted based on continuous feedback from the system. While they perform well in predictable environments, their performance in open and dynamic environments could be unacceptably poor. In an open environment like a general purpose cluster, where workloads are unknown and may vary at run-time, we need adaptive solutions that can maintain desired performance by handling system variations dynamically.

To address aforementioned limitations, in this dissertation, we propose to investigate:

- Real-time divisible load scheduling with setup costs, where we investigate the algorithms that schedule arbitrarily divisible load with setup costs and analyzed the effects of setup costs on the scheduling decisions and their performance.
- Real-time divisible load scheduling that supports advance reservations, where we develop a multi-stage algorithm that can schedule both advance reservation tasks and regular aperiodic tasks. We not only enforce the real-time agreement but also address the under-utilization concerns raised by advance reservations. We systematically study the impact of advance reservations on system performance.
- Efficient real-time divisible load scheduling algorithms, where we provide a scalable scheduling algorithm by decoupling the admission controller and the dispatcher. The proposed algorithm is linear in the number of processors in the cluster and the number of waiting tasks and incurs little scheduling overhead even on large clusters with long waiting queues.

- Feedback control based real-time divisible load scheduling algorithm, where we integrate control theory into the real-time scheduling of divisible loads. By dynamically handling workload and system variations, our algorithm provides predictable QoS guarantees for soft real-time divisible loads in unpredictbale environments.

By integrating the real-time theory and divisible load theory into cluster scheduling, the proposed algorithms in this dissertation can provide QoS assurance and fault tolerance and facilitate automated management of cluster resources. The results can also be extended to a grid of clusters, and integrated into grid-level schedulers to provide grid-level service guarantees. This research contributes significantly to the area of real-time divisible load scheduling.

This dissertation is organized as follows: In chapter 2, we discuss the related work and in chapter 3, we present task and system models. In chapter 4, we describe the real-time divisible load scheduling algorithm that considers setup costs and the real-time divisible load scheduling algorithm that supports advance reservations is presented in chapter 5. In chapter 6 we present the efficient real-time divisible load scheduling algorithm. In chapter 7, we present the feedback-control based real-time divisible load scheduling. Chapter 8 concludes the dissertation.

Chapter 2

Related Work

The previous chapter briefly introduces the four challenges that we address in this dissertation. In this chapter, we summarize the work related to these four challenges. In Section 2.1, we discuss existing real-time divisible load scheduling algorithms for cluster computing. We describe the related work to the real-time divisible load scheduling with advance reservations in Section 2.2. In Section 2.3, we discuss the complexity of existing real-time divisible load scheduling algorithms. In Section 2.4, we summarize the related work on the feedback-control based real-time scheduling.

2.1 Real-time Divisible Load Scheduling

The real-time scheduling models investigated for distributed or multiprocessor systems often (e.g., [67, 66, 38, 1, 64, 34, 4, 41]) assume periodic or aperiodic sequential jobs that must be allocated to a single resource and executed by their deadlines. In recent years, researchers have begun to investigate real-time scheduling of parallel applications on clusters [85, 65, 27, 2, 3]. However, most of these studies assume the existence of some form of task graph to describe communication and precedence relations between computational units called subtasks (i.e., nodes in the task graph).

Netto and Buyya [62] consider the scheduling of parallel bag-of-tasks applications, where each application is formed of a bag of independent sequential tasks that need to be completed by a deadline. Because bag-of-tasks applications are not arbitrarily divisible, they are different from the divisible loads investigated in our research.

The most closely related work [43] to this research is scheduling algorithms for "scalable real-time tasks" running on a multiprocessor system. In that work, like divisible loads, it is assumed that a task can be executed on more than one processor and as more processors are allocated, its pure computation time decreases monotonically. The paper notes that the decision on the number of processors allocated to tasks is an important factor in the design of parallel scheduling algorithms. However, the simulations described in the paper are limited. Their conclusions on comparing their proposed MWF schemes with the EDF and FIXED algorithms [61, 7] hold true only in certain scenarios [45]. The work on scheduling "moldable jobs" [8, 19, 35, 69, 74] is also related, but only He et al. [35] have considered QoS support.

Lin et al. developed a real-time divisible load scheduling algorithm and investigated the problem of providing deterministic QoS to arbitrarily divisible applications executing in cluster environments in [45]. Their work in this research differs significantly from previous work in real-time as well as cluster computing in both the task model assumed and in the comprehensiveness of their study. In that work, unlike previous study in [43], they do not assume task execution times are known a priori. Instead, DLT is applied to guide task partitioning, to derive its execution function, and to compute the minimum number of processors required to meet its deadline. They identified three important and necessary design decisions: 1) workload partitioning, 2) node assignment, and 3) task execution order. They also systematically studied the effects of the different design parameters. The algorithm in [45], however, ignores setup costs of divisible loads. We significantly extend that work, where we propose and evaluate new algorithms that can handle setup costs of divisible loads.

2.2 Real-time Divisible Load Scheduling with Advance Reservations

Real-time scheduling of parallel applications on a cluster has been studied extensively [38, 85, 65, 27, 2, 3]. However, they either do not consider arbitrarily divisible loads or have no support for advance reservations. Due to the increasing importance of arbitrarily divisible applications [68], a few researchers [43, 17, 18, 35] have investigated the real-time divisible load scheduling. Lin et al. applied divisible load theory [79] and developed several scheduling algorithms for real-time divisible loads [45, 46, 44]. However, they do not support advance reservations.

To offer QoS support, researchers have investigated resource reservations for networks [22, 28, 82], CPUs [15, 72], and co-reservations for resources of different types [20, 48]. The most well-known architectures that support resource reservations include GRAM [21], GARA [31, 32] and SNAP [20]. These research efforts mainly focus on resource reservation protocols and QoS support architectures. Our work, on the other hand, focuses on scheduling mechanisms to meet specific QoS objectives, which could be integrated into architectures like GARA [31, 32] to satisfy Grid users' QoS requirements.

Advance reservation and resource co-allocation in Grids [72, 37, 30, 60, 32] assume the support of advance reservations in local cluster sites. Cluster schedulers like PBS PRO, Maui and LSF [57] support advance reservations. However, they are not widely applied in practice due to under-utilization concerns. In [57, 12], backfilling is used to improve system utilization. However, these results still show a significant waste of system resources when advance reservations are supported. Furthermore, these schedulers do not provide real-time guarantees to regular tasks.

In this thesis, we investigate real-time divisible load scheduling algorithms that support advance reservations, where we provide QoS guarantees for both advance reservation and regular tasks. We also investigate the effect of advance reservations on the system performance.

2.3 Efficient Real-time Divisible Load Scheduling

Real-time divisible load scheduling has been investigated in [45, 46, 43, 17, 18]. Focusing on QoS, real-time guarantees, and better utilization of cluster resources, existing approaches place little emphasis on scheduling efficiency. They assume that scheduling takes much less time than the execution of a task, and thus ignore the scheduling overhead. However, clusters are becoming increasingly bigger and busier. As the cluster size and workload increase, so does the scheduling overhead. If we use N to represent the number of processors and n to denote the number of tasks waiting in the system, the time complexity of the most efficient algorithms proposed in [43] (i.e., MWF-FA and EDF-FA) is $O(n^2 + nN)$. The time complexity of algorithms proposed in [17, 18] is $O(nNlogN)$ and the algorithm in [46] has a time complexity of $O(nN^3)$. For a cluster with thousands of nodes or thousands of waiting tasks, the scheduling overhead could be substantial and existing divisible load scheduling algorithms are no longer applicable due to the lack of scalability.

In this dissertation, we address this deficiency of existing approaches and develop an efficient algorithm for real-time divisible load scheduling. Our algorithm has a time complexity that is linear in the number of tasks in the queue and the number of nodes in the cluster. It is efficient and scales well to large clusters. In addition, the algorithm performs similar to algorithms in [46, 17, 18], and it eliminates IITs.

2.4 Feedback-Control Based Real-time Divisible Load Scheduling

The existing real-time divisible load scheduling algorithms in [58, 44, 43, 17, 46, 45, 35] are all "open-loop" scheduling algorithms. Once the schedules are created, they are not adjusted according to the system status. They are designed based on worst-case workload parameters to ensure task deadlines, which often result in extremely low system utilization due to pessimistic estimates. Most real-time divisible applications have soft real-time requirements: they have stringent timeliness requirements but can nevertheless tolerate deadline misses to a certain pre-specified degree. Therefore, for such applications, it is more cost effective not to design algorithms for the worst case. Instead, we should make a proper tradeoff between the deadline guarantee and the system utilization.

In an open environment like a computing cluster, workload changes and system variations should be handled dynamically. To address this challenge, we need a feedback-control based approach. Control theory provides us a scientific foundation for designing feedback-control based computing systems [40, 14, 81, 39, 36]. Diao et al. [24, 25] designed a controller to balance the resource demands in a database management system. Liu et al. [50] developed an adaptive multivariate controller to provide service differentiation in a multi-tier web site. QoS-driven workload management was presented using a nested feedback controller [86], where the inner loop regulates the CPU utilization of a virtual container and the outer loop maintains the application-level response time at its target. Control theory was also applied in feedback-control based real-time scheduling of sequential tasks [75, 51, 53]. Similar approaches were used for e-mail server queue management [63], web cache hit ratio control [56, 55], and CPU utilization control in data centers [83]. Guarantees were made on power dissipation [71] with a control theoretic microprocessor thermal management. In distributed real-time and embedded systems, researchers designed model predictive controllers to regulate CPU utilization [54] and power consumption [80]. Block et al. [10] applied control theory to design an adaptive framework for multiprocessor real-time systems.

In recent years, the soft real-time applications in cluster environments have been growing rapidly. Examples include web servers and real-time data base systems. For those system, it is difficult to accurately model the workloads, and there is a tradeoff between system utilization and deadline miss. It is more cost effective not to design for the worst case, even if deadlines could be missed occasionally. Existing feedback-control based approaches, however, focus on either single server systems or sequential tasks. They are not applicable to control divisible loads. To provide QoS guarantees for soft real-time divisible applications whose execution times cannot be accurately derived from the data size, we need to find creative ways to apply feedback control theory to the real-time divisible load scheduling. In this dissertation, we investigate feedback control based real-time divisible load scheduling algorithms that maintain low deadline miss ratios and high utilizations despite dynamic workload changes.

Chapter 3

Models

In this chapter we describe our task and system models and state assumptions related to these models. All of our work is based on theses models, unless specified otherwise.

3.1 Task Model

In this dissertation, we assume the workload may consist of two types of tasks: regular tasks and reservation tasks.

3.1.1 Regular Task

For a regular task, a real-time aperiodic task model is assumed, in which each aperiodic task T_i consists of a single invocation specified by (A_i, σ_i, D_i), where A_i is the task arrival time, σ_i is the total data size of the task, and D_i is its relative deadline [47]. The task absolute deadline is given by $A_i + D_i$. Assuming T_i is arbitrarily divisible, the task execution time is thus dynamically computed based on the total data size σ_i, resources allocated (i.e., processing nodes and bandwidth) and the partitioning method applied to parallelize the computation. There are many applications [26] conforming to this divisible load task model, e.g., distributed search for a pattern in text, audio, graphical, and database files; distributed processing of big measurement data files; and many simulation problems. For such applications, we often derive their data sizes σ_i based on their input file sizes.

3.1.2 Reservation Task

A reservation task R_i is specified by the tuple $(R_a^i, R_s^i, n_i, R_e^i, IO_{ratio}^i)$, where R_a^i is the arrival time of the reservation request, R_s^i and R_e^i are respectively the start time and the finish time of the reservation, n_i is the number of nodes to be reserved in $[R_s^i, R_e^i]$ interval, and IO_{ratio}^i specifies the data transmission time relative to the length of reservation. It is assumed that for a reservation, data transmission happens at the beginning and computation follows. Let $R_{io}^i = R_s^i + (R_e^i - R_s^i) \times IO_{ratio}^i$. We have data transmission in the interval $[R_s^i, R_{io}^i]$ and computation in the interval $[R_{io}^i, R_e^i]$.

3.2 System Model

A cluster consists of a head node, denoted by P_0, connected via a switch to N processing nodes, denoted by P_1, P_2, ..., P_N. We assume that all processing nodes have the *same* computational power and all links from the switch to the processing nodes have the *same* bandwidth. The system model assumes a typical cluster environment in which the head node does not participate in computation. The role of the head node is to accept or reject incoming tasks, execute the scheduling algorithm, divide the workload and distribute data chunks to the processing nodes. Since different nodes process different data chunks, the head node sequentially sends every data chunk to corresponding processing node via the switch. We assume that data transmission does not happen in parallel. For arbitrarily divisible loads, tasks and subtasks are independent. Therefore, when executing such applications processing nodes do not communicate with each other.

According to the divisible load theory (DLT), linear models are used to represent processing and transmission times [79]. In the simplest scenario, the computation time of a load σ is calculated by a cost function $Cp(\sigma) = \sigma\chi$, where χ represents the time to compute a unit of workload on a single processing node. The transmission time of a load σ is calculated by a cost function $Cm(\sigma) = \sigma\tau$, where τ is the time to transmit a unit of workload from the head node to a processing node. For many applications the output data is just a short message and is negligible, particularly considering the very large size of the input data. Therefore, in this thesis we only model transfer of application input data but not the transfer of output data. The extension to consider the output data transfer using DLT is straightforward.

The following notations, partially adopted from [79], are used in the thesis.

- $T = (A, \sigma, D)$: A divisible task, where A is the arrival time, σ is the data size, and D is the relative deadline.

- $\alpha = (\alpha_1, \alpha_2, ..., \alpha_n)$: Data distribution vector, where n is the number of processing nodes allocated to the task, α_j is the data fraction allocated to the j^{th} node, i.e., $\alpha_j\sigma$ is the amount of data that is to be transmitted to the j^{th} node for processing, $0 < \alpha_j \leq 1$ and $\Sigma_{j=1}^{n}\alpha_j = 1$.

- τ: Cost of transmitting a unit workload.

- χ: Cost of processing a unit workload.

- θ_{cm}: The setup time (cost) for the head node to initialize a communication on a link.

- θ_{cp}: The setup time (cost) for a processing node to initialize a computation.

Chapter 4

Real-Time Divisible Load Scheduling with Setup Costs

4.1 Introduction

Arbitrarily divisible applications form an important category among the computational loads submitted to a cluster. Providing QoS or real-time guarantees for arbitrarily divisible applications executing in a cluster environment not only significantly improves the user experience, but also reinforces the system performance. Lin et al. investigated the problem of providing deterministic QoS for arbitrarily divisible application for cluster computing [45]. They identified that when developing such a scheduling algorithm, we need to make three important decisions: 1) scheduling policy that determines the task execution order; 2) the number n of processors that are allocated to each task; and 3) a strategy that partitions the task workload among n allocated processors. However, the proposed approach did not consider the setup costs of the divisible loads. The delay of starting a remote process or the time to initiate a network connection etcetera all contribute to the setup costs. It has been shown that the setup costs are significant in some scenarios. As a result, task execution times no longer monotonically decreases as the number of processors increases, which introduces new challenges to the real-time divisible load scheduling problem.

In this chapter, we significantly extend that work in [45], where we propose and evaluate new algorithms that can handle setup costs of divisible loads. We also conduct the analysis and experiments on large clusters to investigate the effects of multiple design decisions and system parameters. Next, we present our algorithms. In this chapter, all tasks follow the regular task model described in Section 3.1.1.

4.2 Algorithms

This section presents real-time scheduling algorithms for divisible loads with setup costs. To develop the algorithms, we need to make three important decisions. The first is to adopt a scheduling policy to determine the order of execution for tasks (Section 4.2.1). The second decision is to choose a strategy to partition the task (Section 4.2.2), that is, to partition the task data among a given number of computing resources. The last decision is to determine the number n of processing nodes to assign to each task (Section 4.2.3). Basically, for a real-time divisible task, the number of the processing nodes assigned to it can be between the minimum number of nodes for it to complete before its deadline and all available processing nodes in the system.

4.2.1 Scheduling Policies

Three scheduling policies to determine the execution order of tasks are investigated: FIFO (First In First Out), EDF (Earliest Deadline First) and MWF (Maximum Workload derivative First) [43]. The FIFO scheduling algorithm executes tasks following their order of arrival and is a common practice adopted by cluster administrators to manage a task queue. EDF, a well-known real-time scheduling algorithm, orders tasks by their absolute deadlines. As a real-time scheduling algorithm for divisible tasks, the main rules of MWF [43] are: 1) a task with the highest workload derivative (δw_i) is scheduled first; and 2) the number of nodes allocated to a task is kept as small as possible (n^{min}) without violating its deadline. Node assignment is described in Section 4.2.3. Here, we review how MWF determines task execution order and defines the workload derivative metric, δw_i.

$$\delta w_i = w_i(n_i^{min} + 1) - w_i(n_i^{min}), \tag{4.1}$$

where $w_i(n)$ represents the workload (cost) of a task T_i when n processing nodes are assigned to it. That is, $w_i(n) = n \times \mathcal{E}(\sigma_i, n)$, where $\mathcal{E}(\sigma_i, n)$ denotes the task execution time (see Section 4.3 for $\mathcal{E}$'s calculation). Therefore, δw_i is the derivative of the task workload $w_i(n)$ at n_i^{min} (the minimum number of nodes needed by T_i to meet its deadline).

4.2.2 Task Partitioning Methods

We apply a task partitioning method to divide a task among its allocated processing nodes. Two different partitioning methods are investigated: *Optimal Partitioning Rule* (OPR), and *Equal Partitioning Rule* (EPR). OPR is based on divisible load theory (DLT), which states that the optimal execution time is obtained when all nodes allocated to a task complete their computation at the same time [79]. For comparison, we propose EPR, based on a common practice of dividing a task into n equal-sized subtasks when the task is to be processed by n nodes. When different partitioning methods are applied to parallelize a task's computation, the task will experience different execution time and may require varied minimum number n^{min}

of nodes. In Section 4.3, we provide detailed analysis on these partitioning methods and derive the task execution time and n^{min} for each of them.

4.2.3 Node Assignment Policies

Node assignment determines the number of processing nodes allocated to a task. In this chapter, we study two primary strategies for node assignment. First, assign a task all N or n^* (i.e., $\min(N, n^*)$) nodes to finish it as early as possible (see Section 4.4 for n^*'s description). Second, assign a task the minimum number n^{min} of nodes it needs to meet its deadline and thereby save resources for new tasks. To guarantee that a task finishes by its deadline, the real-time scheduler must know the minimum number of nodes required by the task. Since n^{min} is determined by not only the task data size, deadline and execution start time but also the applied partitioning method, we derive n^{min} in Section 4.3 when partitioning methods are thoroughly analyzed.

4.2.4 Algorithm Framework

As is typical for dynamic real-time scheduling algorithms [66, 23, 59], when a new task arrives, the scheduler dynamically determines if it is feasible to schedule the task without compromising the guarantees for previously admitted tasks. The general framework for a schedulability test is shown in Figure 4.1. It can be configured to generate various real-time divisible load scheduling algorithms by giving the design decisions on: 1) scheduling policy (FIFO, EDF or MWF), 2) task partitioning rule (OPR or EPR), and 3) node assignment method (assigning a task $\min(N, n^*)$ or n^{min} nodes). Upon completion of the test, if all tasks are schedulable a feasible schedule is developed and the new task is accepted; otherwise, it is rejected.

By following the aforementioned framework, we generate ten algorithms: EDF-OPR-MN, EDF-OPR-AN, EDF-EPR-MN, EDF-EPR-AN, FIFO-OPR-MN, FIFO-

Data Structure:

- $n_i^{min}(t)$ - the minimum number of processing nodes needed to finish T_i before its deadline, assuming it is dispatched at time t.
- AvailableNodesList < t_k, AN_k > - a list of the number of available nodes along with the time, where t_k is the time and AN_k is the number of available nodes.

Pseudocode:

```
boolean Schedulability-Test(T)

   TempTasksList ← T + AdmittedTasksQueue
   order TempTasksList    /* EDF, FIFO or MWF (Decision 1) */
   generate AvailableNodesList    /* Obtain the available nodes information */
   ScheduledTaskList ← $\phi$    /* Initialization */

   while TempTaskList $! = \phi$

      /* Trying to assign a task $n^{min}$ or $\min(N, n^*)$ nodes (Decision 3)*/
      identify the first task $T_i$ and the earliest time $t_k$ where the available nodes
      $AN_k \geq n_i^{min}(t_k)$ or identify the earliest time $t_k$ when $AN_k \geq N$

      remove $T_i(A_i, \sigma_i, D_i)$ from TempTasksList

      $s_i \leftarrow t_k$    /* Set the scheduled starting time */
      $n_i \leftarrow n_i^{min}(t_k)$ or $n_i \leftarrow \min(N, n_i^*)$

      /* According to the chosen partitioning rule: OPR or EPR (Decision 2), set the
      expected completion time following Eq. 8 or Eq. 19 */
      $e_i \leftarrow \mathcal{E}(\sigma_i, n_i) + s_i$

      if $e_i > A_i + D_i$
         return false /* Deadline misses */

      put $T_i(A_i, \sigma_i, D_i, s_i, n_i, e_i)$ into ScheduledTaskList

      update AvailableNodesList

   end while

   /* All tasks in the cluster are schedulable */
   AdmittedTasksQueue← ScheduledTaskList

   return true

end Schedulability Test()
```

Figure 4.1: Schedulability Test for the Algorithms.

OPR-AN, FIFO-EPR-MN, FIFO-EPR-AN, MWF-OPR-MN, and MWF-EPR-MN. The nomenclature of the algorithms includes three parts corresponding to the three design decisions. The first part denotes the scheduling policy adopted: EDF, FIFO or MWF. The second part represents the choice of the partitioning rule: DLT-based

OPR or heuristic EPR. In the third portion of the name, MN means the algorithm assigns a task the minimum number of nodes needed to meet its deadline and AN means the algorithm assigns all N or n^* number of nodes. Since MWF always allocates a task n^{min} nodes, the algorithm only has the MN version.

4.3 Analysis of Task Partitioning Methods

Section 4.2.2 has introduced the two partitioning methods that we will investigate in this chapter. In this section, we analyze these methods in detail. Since different partitioning methods lead to different task executions, we derive the task execution time and n^{min} for each of these methods. These analysis provide essential ingredients for the real-time scheduling algorithms (Figure 4.1).

In the analysis, depending on whether the task setup costs (i.e., θ_{cm} and θ_{cp}) are negligible or not, we have two different scenarios. Similar to the previous work on divisible loads [79], linear models are used to represent processing and transmission times. When setup costs are negligible, the data transmission (or communication) time on the j^{th} link is $C_m(\alpha_j\sigma) = \alpha_j\sigma\tau$ and the data processing time on the j^{th} node is $C_p(\alpha_j\sigma) = \alpha_j\sigma\chi$; and when setup costs are significant, the data transmission and processing costs are $\theta_{cm} + \alpha_j\sigma\tau$ and $\theta_{cp} + \alpha_j\sigma\chi$ respectively. Lin et al. [45] have considered the scenario where setup costs for initializing data transmission and data processing are negligible. In the following two sections, scenarios with setup costs are analyzed for the two partitioning methods (i.e., OPR and EPR). To analyze the task execution time for OPR, a method proposed by Bharadwaj et al. [9] is adopted.

4.3.1 Optimal Partitioning Rule (OPR) with Setup Costs

In this section, we present the analysis for the case where setup costs are significant. For a given task, let $\mathcal{E}$ denote the *Task Execution Time*, which is a function of σ and n. We first analyze the execution time function, $\mathcal{E}(\sigma, n)$, assuming n nodes are to be allocated to process a total data size of σ. Then, we use it to derive the minimum number, n^{min}, of nodes needed to meet the task deadline.

The setup cost of communication comes from physical network latencies, network protocol overhead, or middleware overhead. In the TeraGrid project [77], the network speed can be up to 40Gbps with a latency of around 100ms. That is, the latency contributes to about 1/3 of the time required to send 1GB of data. It has also been shown that the setup cost for computation can be up to 25 seconds in practice [16], which is significant for small tasks.

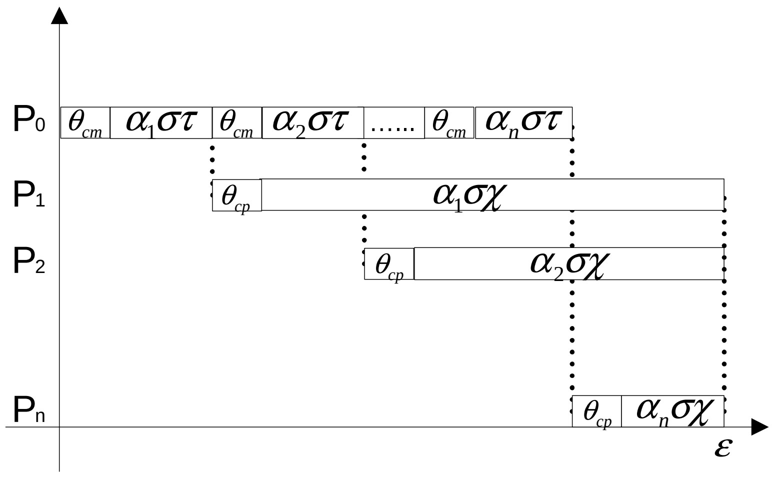


Figure 4.2: Time Diagram for OPR-Based Partitioning with Setup Costs.

1-a) Task Execution Time Analysis: Taking the setup costs into consideration, the data transmission time on the j^{th} link is modeled as $C_m(\alpha_j\sigma) = \theta_{cm} + \alpha_j\sigma\tau$, and the data processing time on the j^{th} node is $C_p(\alpha_j\sigma) = \theta_{cp} + \alpha_j\sigma\chi$. Figure 4.2 shows an example task execution time diagram following OPR when n nodes are allocated to a task and setup costs are modeled. Analyzing the time diagram, we derive the *Task Execution Time* $\mathcal{E}$ as follows

$$\begin{aligned}
\mathcal{E}(\sigma, n) &= (\theta_{cm} + \alpha_1\sigma\tau) + (\theta_{cp} + \alpha_1\sigma\chi) && (4.2)\\
&= 2\theta_{cm} + (\alpha_1 + \alpha_2)\sigma\tau + (\theta_{cp} + \alpha_2\sigma\chi) && (4.3)\\
&= 3\theta_{cm} + (\alpha_1 + \alpha_2 + \alpha_3)\sigma\tau + (\theta_{cp} + \alpha_3\sigma\chi) && (4.4)\\
&\dots\\
&= n\theta_{cm} + (\alpha_1 + \alpha_2 + \alpha_3 + \ldots + \alpha_n)\sigma\tau + (\theta_{cp} + \alpha_n\sigma\chi).
\end{aligned}$$

From Eq. 4.2 and Eq. 4.3, we have $\alpha_2 = \alpha_1\beta - \phi$, where

$$\beta = \frac{\chi}{\tau + \chi} \quad \text{and} \quad \phi = \frac{\theta_{cm}}{\sigma(\tau + \chi)}. \tag{4.5}$$

Similarly, from Eq. 4.3 and Eq. 4.4, we get $\alpha_3 = \alpha_2\beta - \phi$, and therefore $\alpha_3 = \alpha_1\beta^2 - \beta\phi - \phi$, leading to the general formula

$$\begin{aligned}
\alpha_j &= \alpha_1\beta^{j-1} - \Sigma_{k=0}^{j-2}\beta^k\phi. \qquad \text{Thus}\\
\alpha_j &= \alpha_1\beta^{j-1} - \frac{1 - \beta^{j-1}}{1 - \beta}\phi, \; for\ j = 2, 3, \cdots n.
\end{aligned}$$

Now, substituting α_j with $(\alpha_1\beta^{j-1} - \frac{1-\beta^{j-1}}{1-\beta}\phi)$ in equation $\sum_{j=1}^{n}\alpha_j = 1$, we get

$$\begin{aligned}
\alpha_1 + \Sigma_{j=2}^{n}(\alpha_1\beta^{j-1} - \frac{1 - \beta^{j-1}}{1 - \beta}\phi) &= 1\\
\text{i.e.,}\;\; \alpha_1 + \Sigma_{j=1}^{n-1}(\alpha_1\beta^{j} - \frac{1 - \beta^{j}}{1 - \beta}\phi) &= 1.
\end{aligned}$$

A solution to the above equation leads to

$$\alpha_1 = \frac{1-\beta}{1-\beta^n} + \frac{n\phi}{1-\beta^n} - \frac{\phi}{1-\beta}.$$

Let $\quad \mathcal{B}(n) = \frac{1-\beta}{1-\beta^n} + \frac{n\phi}{1-\beta^n} - \frac{\phi}{1-\beta}$,

it follows that

$$\mathcal{E}(\sigma, n) = \theta_{cm} + \theta_{cp} + \sigma(\tau + \chi)\mathcal{B}(n). \tag{4.6}$$

1-b) Derivation of n^{min}*:* If task $T = (A, \sigma, D)$ has a start time s, then to meet its deadline, $\mathcal{E}(\sigma, n) \leq A + D - s$ must be satisfied. That is

$$\theta_{cm} + \theta_{cp} + \sigma(\tau + \chi)\mathcal{B}(n) \leq A + D - s. \tag{4.7}$$

This constraint can be solved numerically. The smallest integer n that satisfies the constraint is the minimum number n^{min} of nodes that need to be assigned to task T at time s to meet its deadline.

Note that the model without setup costs (Lin et al. [45]) is a special case of this model, where $\theta_{cm} = \theta_{cp} = 0$ and accordingly $\phi = \frac{\theta_{cm}}{\sigma(\tau+\chi)} = 0$. Thus, the constraint Eq. 4.7 reduces to $\sigma(\tau + \chi)\frac{1-\beta}{1-\beta^n} \leq A + D - s$.

4.3.2 Equal Partitioning Rule (EPR) with Setup Costs

In this section, we present the analysis of EPR when setup costs are significant.

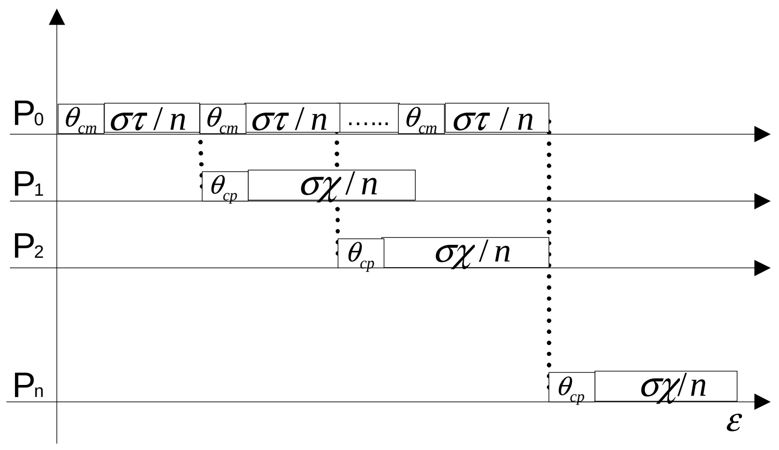


Figure 4.3: Time Diagram for EPR-Based Partitioning with Setup Costs.

2-a) Task Execution Time Analysis: Figure 4.3 shows an example task execution time diagram following EPR when n nodes are allocated to a task and setup costs are modeled. By analyzing the time diagram, we have $\mathcal{E}(\sigma, n) = n\theta_{cm} + \sigma\tau + \theta_{cp} + \alpha_n\sigma\chi$, where $\alpha_n = \frac{1}{n}$. Thus

$$\mathcal{E}(\sigma, n) = n\theta_{cm} + \sigma\tau + \theta_{cp} + \frac{\sigma\chi}{n}. \tag{4.8}$$

2-b) Derivation of n^{min}: Assuming that the task $T = (A, \sigma, D)$ has a start time s, then the task completion time $C(n) = s + \mathcal{E}(\sigma, n)$ must satisfy the constraint $C(n) \leq A + D$. That is

$$s + n\theta_{cm} + \sigma\tau + \theta_{cp} + \frac{\sigma\chi}{n} \leq A + D. \tag{4.9}$$

Let $\omega = A + D - s - \sigma\tau - \theta_{cp}$. We have

$$\theta_{cm}n^2 - \omega n + \sigma\chi \leq 0. \tag{4.10}$$

Since $\theta_{cm} > 0$, $Y = \theta_{cm}n^2 - \omega n + \sigma\chi$ is a parabola that opens upward. Figure 4.4 shows three representative positions of the parabola, when $\omega^2 - 4\sigma\chi\theta_{cm}$ exhibits different signs. Thus, to derive n^{min} three cases need to be considered.

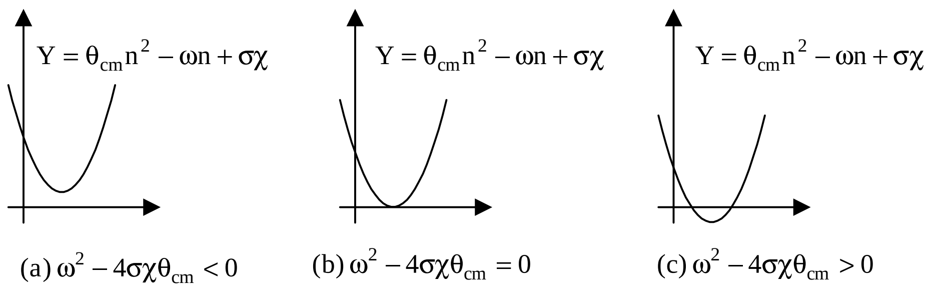

$Y = \theta_{cm} n^2 - \omega n + \sigma\chi$
$Y = \theta_{cm} n^2 - \omega n + \sigma\chi$
$Y = \theta_{cm} n^2 - \omega n + \sigma\chi$
(a) $\omega^2 - 4\sigma\chi\theta_{cm} < 0$
(b) $\omega^2 - 4\sigma\chi\theta_{cm} = 0$
(c) $\omega^2 - 4\sigma\chi\theta_{cm} > 0$

Figure 4.4: Derivation of n^{min}: $Y = \theta_{cm} n^2 - \omega n + \sigma\chi$ Positions.

In the first case, when $\omega^2 - 4\sigma\chi\theta_{cm} < 0$, the parabola has no real axis intercepts, which implies that $Y = \theta_{cm} n^2 - \omega n + \sigma\chi$ will always be greater than 0. Therefore constraint Eq. 4.10 cannot be satisfied for any real number n, meaning it is impossible to meet the task deadline at time s.

In the second case, when $\omega^2 - 4\sigma\chi\theta_{cm} = 0$, the parabola has only one real axis intercept where $n = \frac{\omega}{2\theta_{cm}}$. This is the only possible value of n that satisfies constraint Eq. 4.10. In addition, n, the number of processing nodes, must be a positive integer. Thus, the task can meet its deadline at time s if and only if $n = \frac{\omega}{2\theta_{cm}}$ is a positive integer.

In the third case, when $\omega^2 - 4\sigma\chi\theta_{cm} > 0$, the parabola has two real axis intercepts. From Figure 4.4, we can see that in order to satisfy constraint Eq. 4.10, the value of n should fall between the two real roots of equation $\theta_{cm} n^2 - \omega n + \sigma\chi = 0$. That is

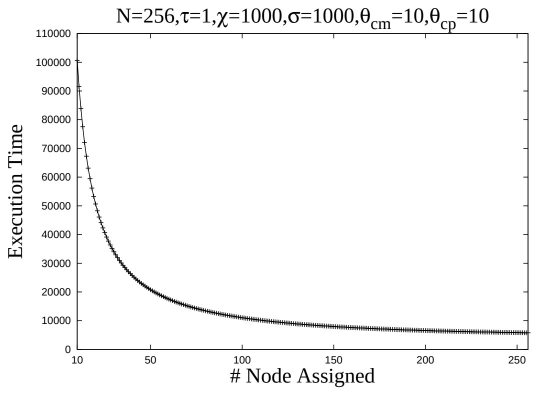
$$\frac{\omega - \sqrt{\omega^2 - 4\theta_{cm}\sigma\chi}}{2\theta_{cm}} \leq n \leq \frac{\omega + \sqrt{\omega^2 - 4\theta_{cm}\sigma\chi}}{2\theta_{cm}}.$$

Since n must be a positive integer, in this case the minimum number of nodes needed for the task to complete before its deadline is

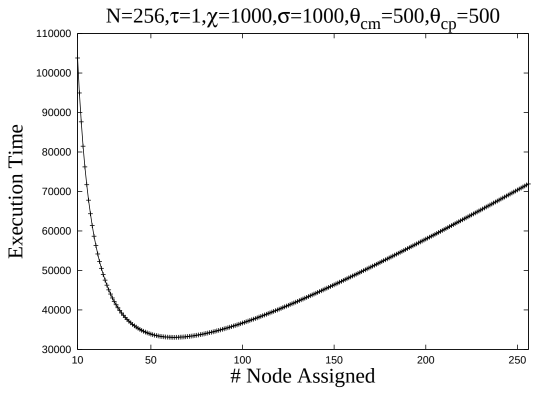
$$n^{min} = \begin{cases} N/A & \text{if } \frac{\omega+\sqrt{\omega^2-4\theta_{cm}\sigma\chi}}{2\theta_{cm}} < 1; \\ 1 & \text{if } \frac{\omega-\sqrt{\omega^2-4\theta_{cm}\sigma\chi}}{2\theta_{cm}} < 1 \text{ and } \frac{\omega+\sqrt{\omega^2-4\theta_{cm}\sigma\chi}}{2\theta_{cm}} \geq 1; \\ \lceil\frac{\omega-\sqrt{\omega^2-4\theta_{cm}\sigma\chi}}{2\theta_{cm}}\rceil & \text{if } \frac{\omega-\sqrt{\omega^2-4\theta_{cm}\sigma\chi}}{2\theta_{cm}} \geq 1. \end{cases}$$

4.4 Analysis of Node Assignment Policies

While scheduling, the number of nodes assigned to a real-time divisible task could be between *the minimum number n^{min} of nodes the task needs to meet its deadline* and *all available N nodes*. The two plots in Figure 4.5 show the relationship between the task execution time $\mathcal{E}$ (Eq. 4.6) and n, the number of nodes assigned, when setup costs are different. As demonstrated in Figure 4.5a, when setup costs are small assigning a greater number of nodes to a task will always reduce its execution time. However, Figure 4.5b shows when the setup costs are significant the execution time of a task is no longer a monotonically decreasing function of the number of nodes assigned. That is, there exists an optimal number n^* such that when a task is assigned n^* nodes the task execution time is the shortest. For example, in Figure 4.5b $n^* = 63$.



(a) When setup costs are negligible.



(b) When setup costs are significant.

Figure 4.5: Task Execution Time vs. Node Assignment.

In this thesis, for the node assignment strategies, we only investigate the two extreme cases, that is, assigning n^{min} or $\min(N, n^*)$ nodes to a task. When the setup costs are negligible, to assign $\min(N, n^*)$ nodes means to allocate all available N nodes to a task. On the other hand, when setup costs are significant (such that $n^* < N$) the strategy to assign all N nodes to a task is not a useful strategy. Instead, assigning n^* nodes can save system resources as well as minimize the task execution time.

4.5 Performance Evaluation

In previous sections, we have proposed and analyzed various real-time cluster-based scheduling algorithms for divisible loads. In this section, their performance relative to each other and to changes of configuration parameters are experimentally evaluated.

We have developed a discrete simulator, called DLSim, to simulate real-time divisible load scheduling in clusters. This simulator, implemented in Java, is a component-based tool, where the main components include a workload generator, a cluster configuration component, a real-time scheduler component, a task dispatcher, and a logging component. The real-time scheduler component is implemented following our algorithm framework proposed in Section 4.2.4, which can be configured to simulate different scheduling algorithms with varied policies on task execution order, workload partitioning and node assignment.

For each simulation, five parameters, N, τ, χ, θ_{cm} and θ_{cp} are specified for a cluster. In this chapter, to evaluate the algorithms performance in processing different streams of tasks, we generate synthetic workloads with parameters varying in wide ranges. To generate task $T_i = (A_i, \sigma_i, D_i)$, similar to the work by Lee et al. [43], we assume that the interarrival times follow an exponential distribution with a specified mean of $1/\lambda$ and task data sizes σ_i are normally distributed with a specified mean of $Avg\sigma$ and a standard deviation equal to the mean. Task relative deadlines are assumed to be uniformly distributed in $[\frac{AvgD}{2}, \frac{3AvgD}{2}]$ range, where $AvgD$ is the mean relative deadline. To specify $AvgD$, a new term $DCRatio$ is introduced. It is defined as the ratio of mean deadline to mean execution time (cost), that is $\frac{AvgD}{\mathcal{E}(Avg\sigma,N)}$, where $\mathcal{E}(Avg\sigma, N)$ is the task execution time computed with Eq. 4.6 assuming the task has a data size equal to $Avg\sigma$ and runs on all N processing nodes. Given $DCRatio$, the cluster size N and the average data size $Avg\sigma$, $AvgD$ is implicitly specified as $DCRatio \times \mathcal{E}(Avg\sigma, N)$. In this way, by $DCRatio$, task relative deadlines are specified relating to the average task execution time. In addition, a task relative deadline D_i is chosen to be larger than its execution time $\mathcal{E}(\sigma_i, N)$.

Similar to the work by Lee et al. [43], we define another metric $SystemLoad$ to represent how loaded a cluster is:

$$SystemLoad = \frac{\mathcal{E}(Avg\sigma, 1)\lambda}{N}, \tag{4.11}$$

where $\mathcal{E}(Avg\sigma, 1)$ is the execution time of an average size task running on a processing node and λ/N is the average task arrival rate per node. Sometimes, we specify $SystemLoad$ for a simulation instead of average interarrival time $1/\lambda$. Configuring $(N, \tau, \chi, \theta_{cm}, \theta_{cp}, SystemLoad, Avg\sigma, DCRatio)$ is equivalent to specifying $(N, \tau, \chi, \theta_{cm}, \theta_{cp}, 1/\lambda,\ Avg\sigma, DCRatio)$, because

$$1/\lambda = \frac{\mathcal{E}(Avg\sigma, 1)}{SystemLoad \times N}. \tag{4.12}$$

To evaluate the real-time performance, we use two metrics: *Task Reject Ratio* and *System Utilization*. Task reject ratio is the ratio of the number of task rejections to the number of task arrivals. The smaller the ratio, the better the performance. In contrast, the greater the system utilization, the better the performance.

For all figures in this chapter, a point on a curve corresponds to the average performance value of ten simulations. In the ten runs, the same parameters (N,τ, χ, θ_{cm}, θ_{cp}, $1/\lambda$, $Avg\sigma$, DCRatio) are specified but different random numbers are generated for task arrival times A_i, data sizes σ_i, and deadlines D_i. For each simulation, the total simulation time is 10,000,000 time units, which is sufficiently long.

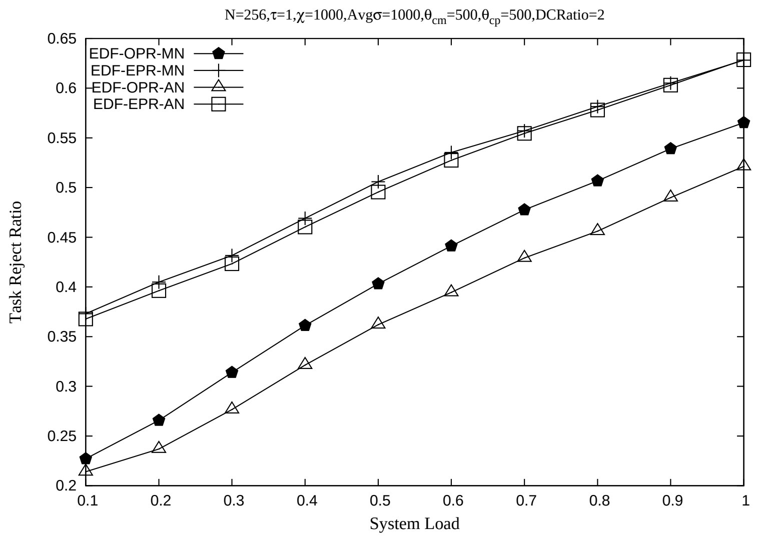
We have identified three important scheduling decisions: *Task Partitioning*, *Node Assignment*, and *Scheduling Policy* in designing real-time, cluster-based scheduling algorithms for divisible loads (see Section 4.2). In the next three subsections, we evaluate the effects of these decisions, compare the algorithms proposed in Section 5.2, and respectively investigate the scenarios where each of these three decisions matters.

4.5.1 OPR vs. EPR Partitioning

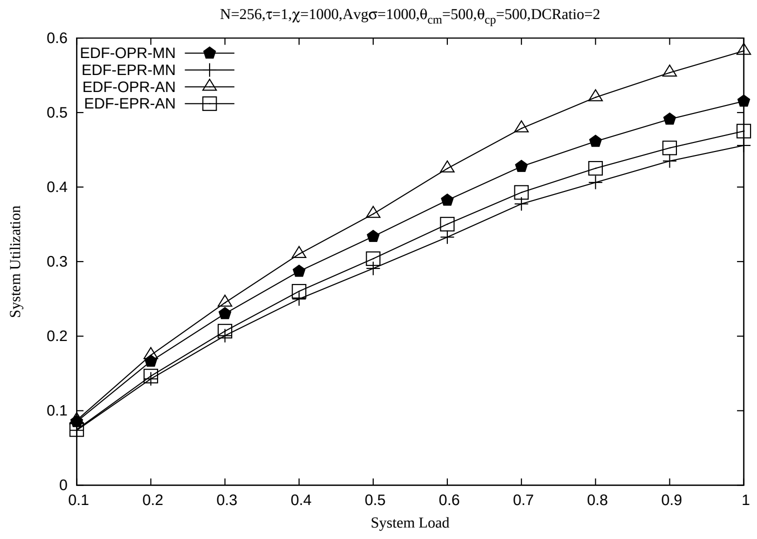
We first evaluate the performance of the following real-time scheduling algorithms with respect to the two proposed partioning rules (OPR and EPR): EDF-OPR-MN vs. EDF-EPR-MN, EDF-OPR-AN vs. EDF-EPR-AN, FIFO-OPR-MN vs. FIFO-EPR-MN, FIFO-OPR-AN vs. FIFO-EPR-AN, and MWF-OPR-MN vs. MWF-EPR-MN. We only present the comparisons of EDF-OPR-MN vs. EDF-EPR-MN and EDF-OPR-AN vs. EDF-EPR-AN here. The performance results for the other pairs are similar.

Simulation Modeling. For our basic simulation model we chose the following parameters: number of processing nodes in the cluster $N = 256$; unit data transmission time $\tau = 1$; unit data processing time $\chi = 1000$; transmission setup cost $\theta_{cm} = 500$; processing setup cost $\theta_{cp} = 500$; $SystemLoad$ changes in $\{0.1, 0.2, \cdots, 1.0\}$ range; Average data size $Avg\sigma = 1000$; and the ratio of the average deadline to the average execution time $DCRatio = 2$. Our simulation has a three-fold objective. *First*, we want to verify our hypothesis that it is advantageous to apply DLT in real-time cluster-based scheduling. *Second*, we study the effects of *DCRatio*, and *third*, we want to investigate the effects of the processing speed.

Merits of DLT for Cluster Scheduling



(a) Task Reject Ratio



(b) System Utilization

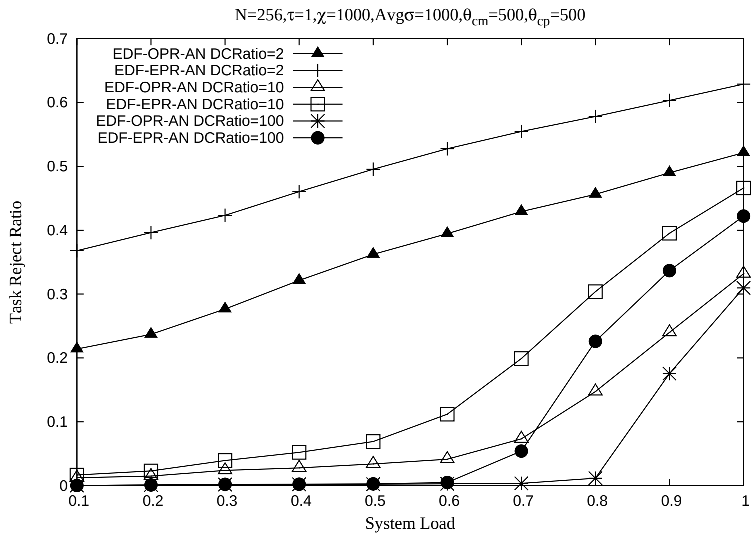
Figure 4.6: OPR vs. EPR: Merits of DLT.

To study the merits of DLT we employ our basic simulation model without any change. Figure 4.6 shows *Task Reject Ratio* and *System Utilization* of the four algorithms: EDF-OPR-MN, EDF-EPR-MN, EDF-OPR-AN, and EDF-EPR-AN. Observe that EDF-OPR-MN always leads to a lower *Task Reject Ratio* and a higher *System Utilization* than EDF-EPR-MN. Similarly, EDF-OPR-AN always performs better than EDF-EPR-AN. These simulation results confirm our hypothesis that it is advantageous to apply DLT in real-time cluster-based scheduling algorithms. The reason is, compared to the partitioning heuristic EPR, the DLT-based OPR provides an optimal task partitioning, which leads to minimum task execution times. As a result, with an OPR scheduling algorithm (i.e., EDF-OPR-MN or EDF-OPR-AN), the cluster can satisfy a larger number of task deadlines and be better utilized.

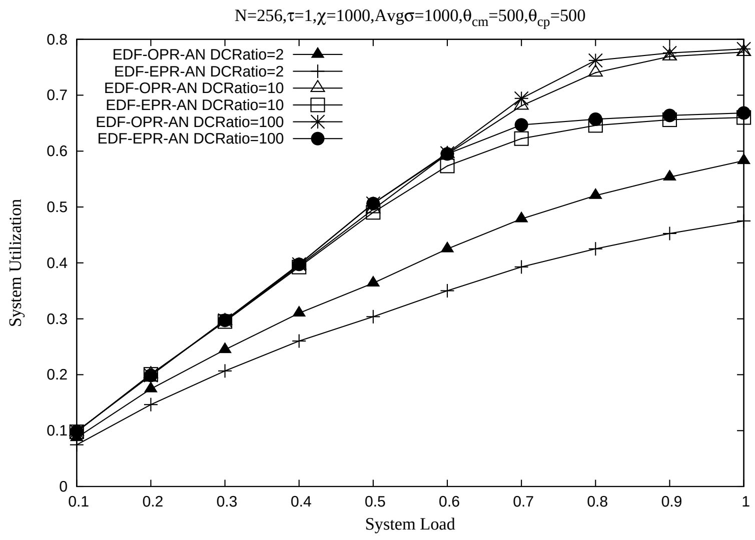
We carried out the same type of simulations by changing the following cluster or workload parameters one at a time: *cluster size* N and *average data size* $Avg\sigma$. Results are similar to Figure 4.6, where algorithms with OPR partitioning always perform better than algorithms with EPR partitioning.

Effects of *DCRatio*

To study the effects of the *DCRatio*, we use the same configuration as the basic simulation model except that we vary the *DCRatio* over $\{2, 4, 6, 10, 20, 50, 100\}$ range. For the sake of readability, Figure 4.7 only shows the performance of EDF-OPR-AN and EDF-EPR-AN with DCRatio $= 2$, 10, and 100. Corresponding to different combinations of algorithm and DCRatio, six curves are produced. Again, Figure 4.7 shows that the algorithm with OPR partitioning performs better. In addition, we can see when SystemLoad is low (i.e., when SystemLoad < 0.8), the performance of EDF-EPR-AN becomes closer to that of EDF-OPR-AN as *DCRatio* increases. This is because the higher the *DCRatio*, the looser the task deadlines are. Consequently, when SystemLoad is low and cluster resources are plenty, the worse execution times caused by a non-optimal partitioning rule, like EPR, will have less impact on the algorithm's performance.

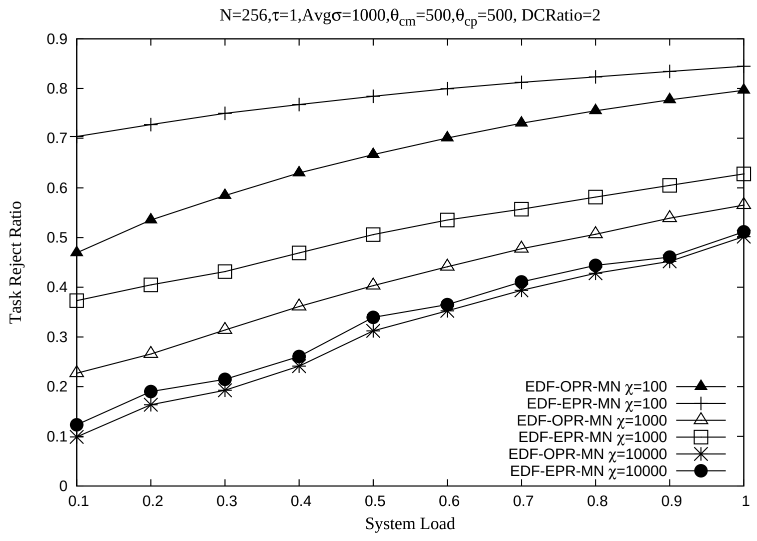


(a)

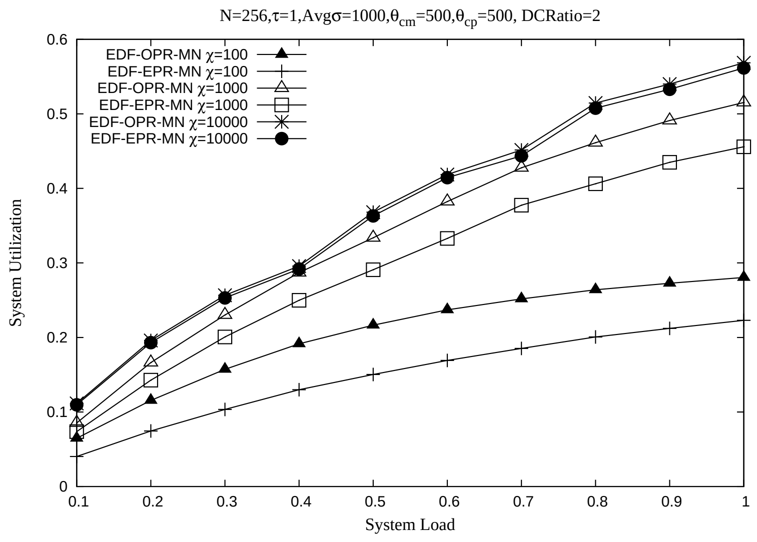


(b)

Figure 4.7: OPR vs. EPR: Effects of *DCRatio*.



(a)



(b)

Figure 4.8: OPR vs. EPR: Effects of Processing Speed.

Effects of Processing Speed

To study effects of the processing speed, we vary χ over $\{100, 500, 1000, 5000, 10000\}$ range. The larger the χ, the slower the computation. Figure 4.8 shows the results of EDF-OPR-MN and EDF-EPR-MN with $\chi = 100$, 1000, and 10000 respectively.

We observe that the OPR partitioning algorithm EDF-OPR-MN still outperforms the EPR partitioning algorithm EDF-EPR-MN. However, as the processing speed decreases, i.e., χ increases, the differences between the two algorithms become less significant. In particular, when the computation is extremely slow ($\chi = 10000$), the curves for the two algorithms are almost the same, indicating non-differentiable *Task Reject Ratios* and *System Utilization.* To demonstrate this point, let us assume χ is so large that the ratio of τ to χ is approaching 0. As a result, β from Eq. 4.5 will approach 1, causing the data fractions allocated to processing nodes $\alpha_1, \alpha_2, \cdots, \alpha_n$, to all be close to $\frac{1}{n}$ for OPR. Therefore, OPR and EPR will perform the same in this case.

Summary. From the aforementioned intensive experiments, we have the following conclusions: a) No matter what the system parameters are, the algorithms with DLT-based partitioning (OPR) always perform better than those with the equal-sized partitioning heuristic (EPR). This demonstrates that it is beneficial to apply DLT (divisible load theory) in real-time cluster-based scheduling; b) When SystemLoad is low, the difference between OPR and EPR becomes smaller as *DCRatio* (i.e., deadline) increases; and c) As χ increases, that is, as node processing speed decreases, the difference between OPR and EPR becomes negligible.

4.5.2 n^* vs. n^{min} Node Assignment

In this subsection, we compare and analyze the real-time scheduling algorithms with different node assignment methods. We investigate the performance difference in algorithms assigning all N or n^* nodes to every task (ALG-AN) vs. those assigning the minimum number n^{min} of nodes needed to meet a task deadline (ALG-MN). The relative performance of EDF-OPR-MN vs. EDF-OPR-AN is systematically studied. It is noteworthy that in contrast to the results by Lee et al. [43] comparing MWF(-

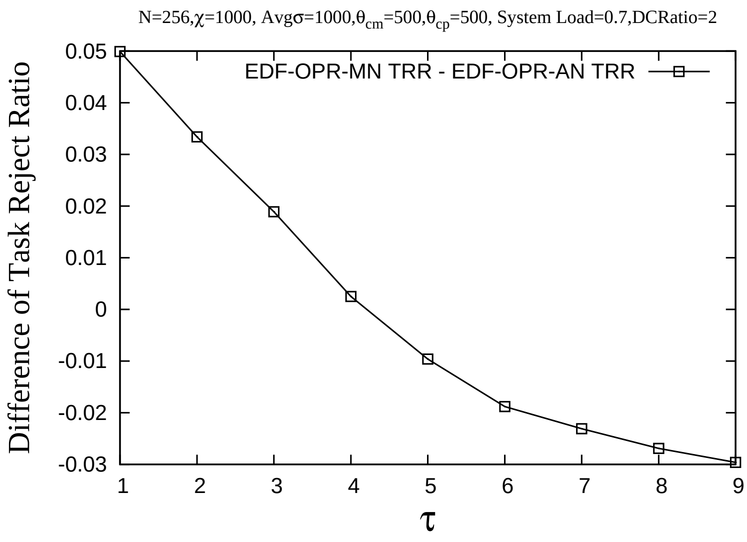
MN) and FIXED(-AN) algorithms, our initial data (see Figure 4.6) seem to indicate that EDF-OPR-AN outperforms EDF-OPR-MN most of the time.

Effects of Transmission Cost

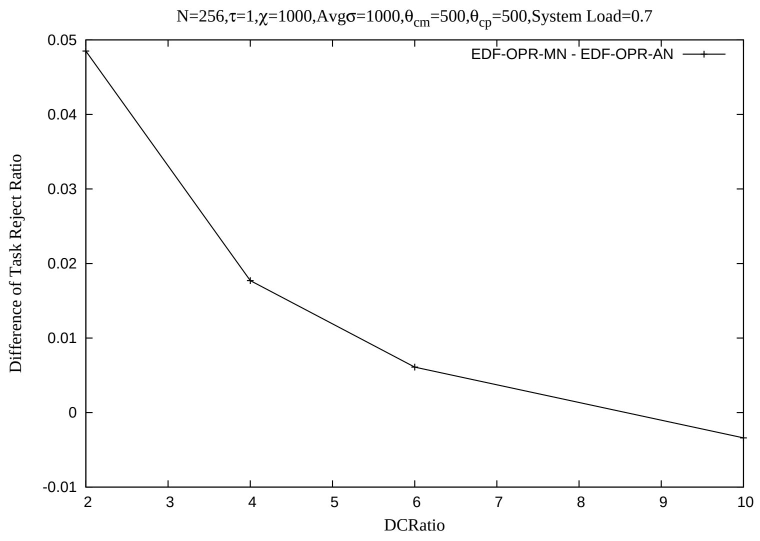
Figure 4.9a shows the relative performance of the two algorithms, i.e, *Task Reject Ratio (TRR)* of EDF-OPR-MN − *Task Reject Ratio (TRR)* of EDF-OPR-AN. In this simulation, we gradually increase the transmission cost τ. As we can see, when τ is small EDF-OPR-MN leads to a bigger *Task Reject Ratio* than EDF-OPR-AN and as τ increases EDF-OPR-MN begins to have a smaller *Task Reject Ratio* than EDF-OPR-AN. This indicates that the relative performance of EDF-OPR-MN vs. EDF-OPR-AN improves as τ gets larger.

In Section 4.4, we have discussed the rational behind the two different node assignment strategies: *an algorithm of type ALG-AN* tries to finish the current task as soon as possible by assigning more processing nodes to a task, while *an algorithm of type ALG-MN* tries to conserve resources for new tasks. For an ALG-AN, the problem is it causes higher parallel execution overheads than the ALG-MN counterpart, e.g., EDF-OPR-AN leads to higher overheads than EDF-OPR-MN. As shown in Figure 4.2, the node idle time due to data transmission is one type of parallel execution overhead. For the cluster model investigated (see Chapter 3), the higher the transmission cost (τ) the greater the overhead. That explains why in the aforementioned simulation we observe that as τ increases, the performance of EDF-OPR-AN is affected more than that of EDF-OPR-MN and EDF-OPR-MN begins to perform better than EDF-OPR-AN.

The results shown in Figures 4.6 and 4.9a contradict the conclusion drawn by Lee et al. [43] that the n^{min} node assignment strategy (ALG-MN) performs better than the maximum node assignment strategy (ALG-AN). As we can see in Figure 4.9a,



(a) Effects of transmission cost τ.



(b) Effects of $DCRratio$.

Figure 4.9: n^* vs. n^{min}.

there are scenarios where ALG-MN performs better than ALG-AN, while in the other scenarios the reverse is true.

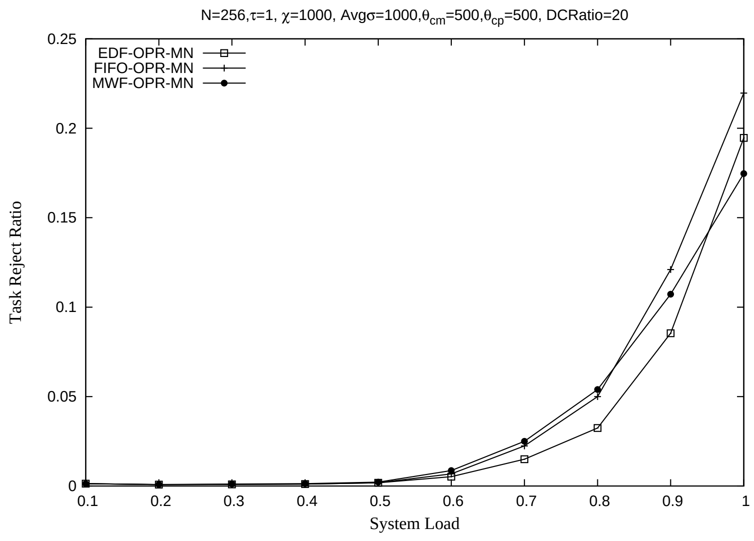
Effects of $DCRatio$

In this subsection, we study effects of changing deadlines, where we vary the $DCRatio$ from 2 to 10. By increasing the $DCRatio$, we have longer relative deadlines compared to the mean execution time. For an ALG-MN, a longer deadline leads to a smaller n^{min} of nodes allocated to a task, thus smaller parallel execution overhead. While for an ALG-AN, its node assignment and resulting overhead will not be affected by deadlines, since a task is always assigned $\min(N, n^*)$ number of nodes. Therefore, we believe, as $DCRatio$ increases and ALG-MN's overhead decreases, ALG-MN's performance relative to that of ALG-AN is going to improve. Figure 4.9b validates our hyphothesis, where we observe that by increasing $DCRatio$ from 2 to 10, the *Task Reject Ratio* difference of EDF-OPR-MN and EDF-OPR-AN gets smaller.

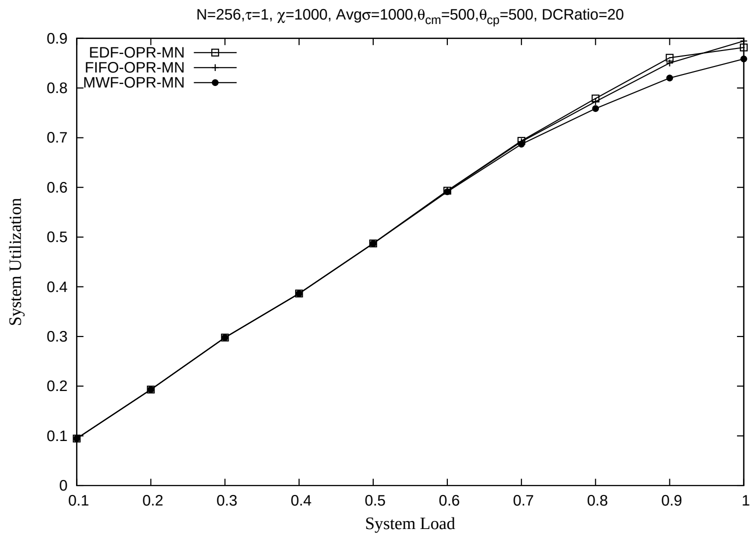
Summary. From the aforementioned intensive experiments, we have the following conclusions: a) Both ALG-MN and ALG-AN have their own advantages. One outperforms the other under certain system configurations and workload scenarios; b) As the transmission cost τ increases, the relative performance of ALG-MN vs. ALG-AN improves; and c) As $DCRatio$ increases and task deadlines become less tight, ALG-MN's performance gets better relative to that of ALG-AN.

4.5.3 FIFO, EDF vs. MWF Scheduling Policies

In this subsection, we examine different execution order policies and compare algorithms FIFO-OPR-MN, EDF-OPR-MN vs. MWF-OPR-MN.

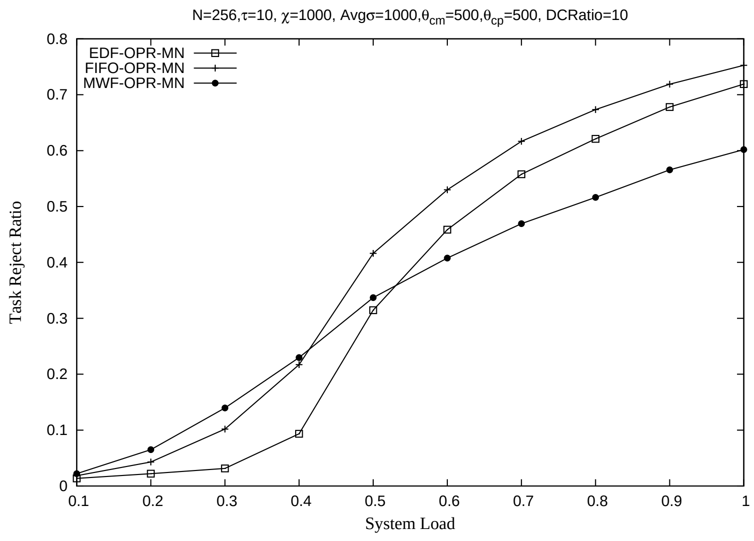


(a)

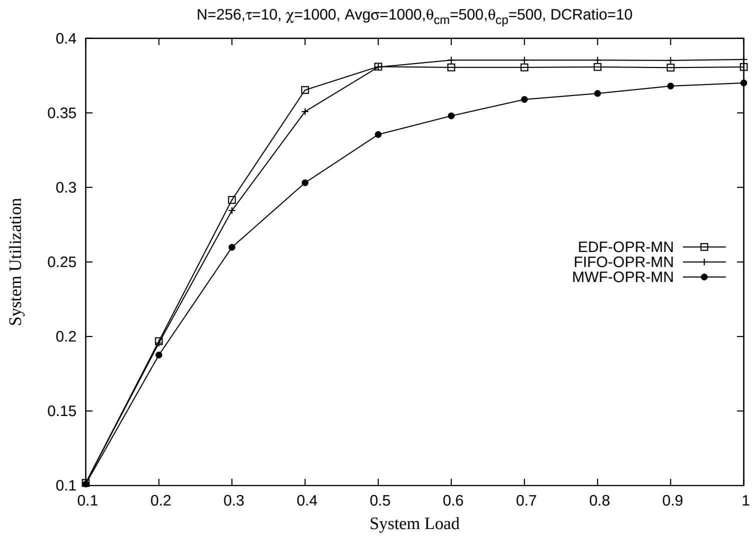


(b)

Figure 4.10: FIFO, EDF vs. MWF: When $\tau = 1$

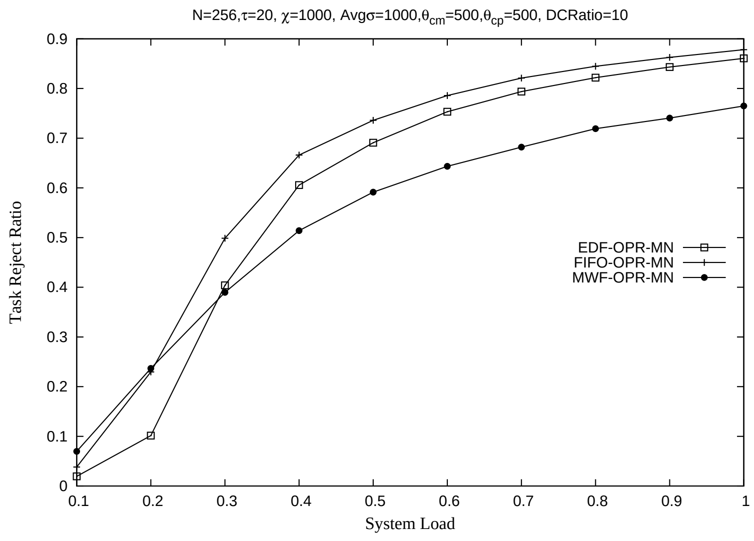


(a)

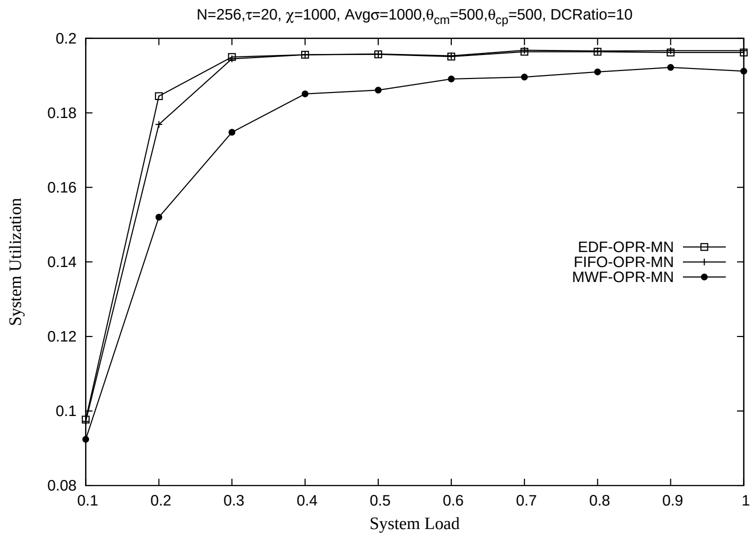


(b)

Figure 4.11: FIFO, EDF vs. MWF: When $\tau = 10$



(a)



(b)

Figure 4.12: FIFO, EDF vs. MWF: When $\tau = 20$

Recall that the MWF (Maximum Workload derivative First) algorithm proposed by Lee et al. [43] executes the task with the highest workload derivative (δw_i) first and thus reduces the total workload (cost) of all scheduled tasks. In their paper [43] MWF is compared with EDF and shown that MWF performs better than EDF. Moreover, the authors claim that MWF is likely to be the best choice for on-line scheduling of divisible tasks.

We conducted intensive simulations and a systematic study of the three execution order strategies. Our data cast some doubts on the conclusion drawn by Lee et al. [43] that the MWF algorithm is the best choice. Our hypothesis is that MWF performs well when task parallel execution overhead (workload) is significant compared to pure task computation time. To test our hypothesis, a group of simulations is designed to study how changing parallel overhead affects the performance of scheduling algorithms. In the 20 simulations, we gradually change the data transmission cost (τ) from 1 to 20, while keeping the data processing cost (χ) constant. Since the bigger the τ the higher the parallel execution overhead, for the 20 simulations with τ changing from 1 to 20 the task overhead increases. According to our theory, MWF should perform better than EDF and FIFO when τ increases.

Figures 4.10, 4.11, and 4.12 show the results for simulations where $\tau = 1$, 10 and 20 respectively. As observed, when τ is small, the *Task Reject Ratio* curve of EDF-OPR-MN lies below that of MWF-OPR-MN, indicating EDF execution order performs better. As τ increases, the relative performance of the two algorithms begins to change. When τ increases to 20, MWF-OPR-MN outperforms EDF-OPR-MN, leading to smaller *Task Reject Ratios* for most SystemLoad conditions. These data match our analysis and verify our hypothesis that MWF performs better than EDF and FIFO as workload parallelization overhead increases.

Interestingly, for all 20 simulations with τ changing from 1 to 20, EDF-OPR-MN always leads to smaller *Task Reject Ratios* than FIFO-OPR-MN. Another quite interesting phenomenon is that among the three algorithms, MWF-OPR-MN always results in the worst *System Utilizations.* MWF policy tries to schedule tasks with bigger workload derivatives δw_i first (see Eq. 4.1 for δw_i's calculation). The larger the task size σ_i, the bigger δw_i tends to be. Thus, this scheduling policy tries to schedule large tasks first. However, inserting those large tasks before small tasks often causes deadline violations of small tasks. As a result, with MWF policy, large tasks usually cannot pass the schedulability test and likely be rejected, which explains why MWF-OPR-MN leads to the worst *System Utilizations* among the three algorithms, even for cases when MWF-OPR-MN has the best *Task Reject Ratios*.

4.6 Summary

From the discussion above, we conclude that: a) the best choice of execution order policy depends on the particular system and workload conditions; b) our results seem to show that most of the time algorithms using EDF policy perform better than algorithms using FIFO policy; c) when communication cost (τ) is small, algorithms using MWF policy do not have any advantage, while τ increases, MWF algorithms begin to perform better than their EDF and FIFO counterparts; and d) MWF algorithms tend to reject large tasks and thus lead to smaller system utilizations.

Chapter 5

Real-Time Divisible Load Scheduling with Advance Reservations

5.1 Introduction

For the grid applications that require simultaneous access to multi-site resources, supporting advance reservations in a cluster is important. In the cluster level, some debugging and interactive applications require a specified number of processors to be available at predefined time intervals. The scheduled maintenance and processor down times can also be treated as advance reservations. In a large-scale cluster, the resource management system (RMS), which provides real-time guarantees or QoS, is central to its operation. To support real-time applications at a Grid level, advance reservations of cluster resources play a key role. However, advance reservations in a cluster environment have been largely ignored due to the under-utilization concerns and lack of support for agreement enforcement [70]. In this chapter, we investigate real-time divisible load scheduling with advance reservations and tackle its challenges. In a cluster with no provision for reservations, resources are allocated to tasks until they finish processing. If, however, advance reservations are supported in a cluster, computing nodes and the communication channel could be reserved for a period of time and become unavailable for regular tasks. Due to these constraints, it becomes a very difficult task to efficiently count the available resources and schedule real-time tasks.

We made two major contributions. First, we proposed a multi-stage real-time divisible load scheduling algorithm that supports advance reservations. The novelty of our approach is that we consider reservation blocks on both computing nodes and communication channels. According to [73], many applications have huge deployment overheads, which require large and costly file staging before applications start. To provide real-time guarantees, it is therefore essential to take the reservation's data transmission into account. Second, the effects of advance reservations on system performance are thoroughly investigated. Our study demonstrates that with our proposed algorithm and appropriate advance reservations, we could avoid under-utilizing the real-time cluster.

This chapter is organized as follows. The real-time scheduling with advance reservation is investigated in Section 5.2. In Section 5.3, we establish correctness of our approach. We evaluate and analyze the system performance in Section 5.4.

5.2 Scheduling with Advance Reservations

This section presents a real-time scheduling algorithm that supports advance reservations in a cluster. Here, tasks follow the regular task and the reservation task models described in section 3.1.

In [47], we investigated the problem of real-time divisible load scheduling in clusters. Our previous work, however, does not address the challenges of supporting advance reservations. Without advance reservations, computing resources are allocated to tasks until they finish computation. If, however, advance reservations are supported in a cluster, a computing node could be reserved for a period of time and become unavailable for regular tasks. The reservations thus block the processing of regular tasks and cast severe constraints on the real-time scheduling. In the following, we use an example to illustrate the challenge.

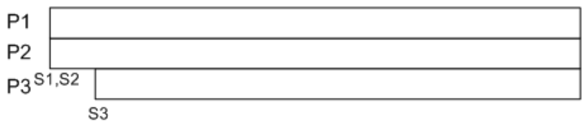


Figure 5.1: Cluster Nodes with no Reservation.

In Figure 5.1, we show three processing nodes P_1, P_2 and P_3 available at time points S_1, S_2 and S_3 respectively. Once available, they could be allocated to execute a new task. Upon arrival of a new task, the real-time scheduler considers the system status and determines if enough processing power is available to finish the task before its deadline. The decision process is simple when there is no reservation block: for node P_i, any time between S_i and the task deadline could be allocated to the new task. It, however, becomes a complicated process when there are advance reservations. For instance, as shown in Figure 5.2, there is an advance reservation R occupying node P_2 from time R_s to time R_e. During the reserved period, the time from R_s to R_{io} is used for transmitting data to node P_2 and the time from R_{io} to R_e is used for computation. Because of the reservation, node P_2 becomes unavailable in the time period $[R_s, R_e]$. Furthermore, the reservation interferes with activities on other nodes. During the time period $[R_s, R_e]$, nodes P_1 and P_3 could be used to compute tasks. However, data transmission to P_1 or P_3 is not allowed in the interval $[R_s, R_{io}]$ when data is transmitted to node P_2. Because of these constraints, it becomes a challenge to efficiently count the available processing power and schedule real-time tasks. The remainder of this section discusses how we overcome this challenge and design an algorithm that supports advance reservations in a real-time cluster.

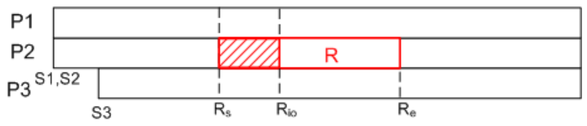


Figure 5.2: Cluster Nodes with a Reservation.

5.2.1 Admission Control Algorithm

As is typical for dynamic real-time scheduling algorithms [23, 59, 66], when a task arrives, the scheduler dynamically determines if it is feasible to schedule the new task without compromising the guarantees for previously admitted tasks. The pseudocode for the schedulability test is shown in Algorithm 1.

According to the newly arrived task's type, the algorithm invokes an admission test. For a reservation, it (Algorithm 2) first checks if enough processing nodes are available to accommodate the reservation. Because data transmission does not happen in parallel, we must then ensure that the new reservation will not cause any I/O conflict. The function IO_Overlap(R_k, R) verifies if data transmissions for R_k and R overlap. If so, new reservation R is rejected. If the admission test is successful, it proves that accepting R will not compromise the guarantees for previously accepted reservations. Its impact on previously accepted regular tasks is yet to be analyzed, which is the third step of the algorithm. For a regular task T, the admission test (Algorithm 3) checks if T is schedulable with the accepted reservations. When a new regular task T arrives, it is added to the waiting queue of regular tasks. We adopt the EDF (Earliest Deadline First) scheduling algorithm and order the queue by

Algorithm 1 boolean Schedulability_Test(T)

```
1: if isResv(T) then
2:     // Reservation admission control
3:     if !ResvAdmTest(T) then
4:         return false
5:     end if
6: end if
7: // Reservation list
8: TempResvList ← ResvQueue
9: // Regular task list
10: TempTaskList ← TaskWaitingQueue
11: if isResv(T) then
12:     TempResvList.add(T)
13: else
14:     TempTaskList.add(T)
15: end if
16: // EDF scheduling of regular tasks
17: order TempTaskList by task absolute deadline
18: order TempResvList by reservation start time
19: while TempTaskList != $\phi$ do
20:     TempTaskList.remove(T)
21:     // Regular task admission control
22:     if !AdmTest(T) then
23:         return false
24:     else
25:         TempScheduleQueue.add(T)
26:     end if
27: end while
28: TaskWaitingQueue ← TempScheduleQueue
29: ResvQueue ← TempResvList
30: return true
```

task absolute deadlines. The schedulability test (Algorithm 1) invokes the admission test (Algorithm 3) for each task in the queue. If they are all successful, it proves that accepting T will not compromise the guarantees for previously accepted tasks including all reservations and regular tasks.

As mentioned, Algorithm 3 tests the schedulability of a regular task. We have N processing nodes in the cluster, available at time points $S_1, S_2, \cdots, S_N$. Assume

Algorithm 2 boolean ResvAdmTest($R(R_a, R_s, n, R_e, IO_{ratio})$)

```
 1: // Check if the number of available nodes
 2: // in [$R_s$, $R_e$] time period is less than $n$
 3: if MinAvailableNode($R_s$, $R_e$) $< n$ then
 4:     return false
 5: end if
 6: for $R_k \in$ ResvQueue do
 7:     if IO_Overlap($R_k$, R) then
 8:         return false
 9:     end if
10: end for
11: reserve $n$ available nodes for R
12: return true
```

reservations are made on these nodes for specific periods of time. To determine whether or not a regular task T(A, σ, D) is schedulable, the N nodes' total processing time that could be allocated to task T by its absolute deadline $A + D$ is computed. If the total time is enough to process the task, deadline D can be satisfied and task T is schedulable. To derive the processing time, we first compute the blocking time when a processing node cannot be utilized.

Blocking happens because data cannot be transmitted in parallel. Transmission to a processing node blocks transmissions to all other nodes. There are three types of blocking: 1) the blocking is caused by a reservation's data transmission; 2) among nodes allocated to a task, data transmission to a node blocks the other transmissions of the same task; 3) the blocking is caused by another task's data transmissions.

To count the available processing power and decide a task's schedulability, we have to consider all these blocking factors. However, the degree of blocking varies, depending on node available times, task start times and reservation lengths. For instance, a reservation with a long data transmission could lead to lengthy blocking. If a reservation and a task start at the same time, the task is blocked during the reservation's data transmission. Computing the exact blocking time is complicated. Thus, to simplify the computation, the worst-case blocking scenario is initially assumed in the admission test and only if a task is determined schedulable, will it be sent to the task partition procedure to accurately compute the task's schedule.

The worst-case blocking happens when all nodes become available at the same time as the reservation's data transmission. In this case, all nodes have to be blocked the whole time during the reservation's data transmission. Assuming this worst-case blocking scenario, Algorithm 3 derives the amount of data σ_{sum} that can be processed in the total available time. When σ_{sum} is larger than the task data size σ, enough nodes are found to finish the task.

The algorithm sorts the N nodes in a non-decreasing order of node available time, i.e., making $S_1 \leq S_2 \leq \cdots \leq S_N$. Following this order, each node's available processing time μ_i is computed. First, μ_i is initialized to be $A + D - \max(S_i, A)$, the longest time that P_i could be allocated to task T by its absolute deadline $A + D$. Then considering the effects of reservations, μ_i is adjusted. For the worst-case blocking, a node P_i is assumed not utilized when data is transmitted for reservations. Let $t_0 = \max(A, S_1)$. The total time (ResvIO) consumed by the reservation's I/O in the interval $[t_0, A + D]$ is computed, which is the blocking time relevant to task T's schedule. If a reservation is on node P_i, task T cannot utilize P_i during the reservation's computation ($ResvCP_i$). The algorithm thus reduces μ_i by ResvIO and $ResvCP_i$. After considering the type 1 blocking caused by reservations, the algorithm considers the other two blocking factors. B_{i-1} counts the type 2 blocking time on node P_i, which is caused by the same task's data transmissions to previously assigned nodes $(P_1, P_2, \cdots, P_{i-1})$. Again, the worst-case blocking is assumed, i.e., P_i is assumed to be blocked for a time period of transmitting data to the first $i - 1$ nodes. μ_i is, therefore, reduced by B_{i-1}. This gives the final value of μ_i. The algorithm then derives the size of data that can be processed in μ_i amount of time. The total sum of data that can be processed by the first i nodes are recorded in σ_{sum}. Once σ_{sum} is larger than the task data size σ, enough nodes n^{min} are found for the task. The algorithm concludes that task T is schedulable, assigns n^{min} number of nodes to it and invokes the task partition procedure (MSTaskPartition, Algorithm 4) to accurately schedule the task. Processing task T may block tasks scheduled in the future, which leads to type 3 blocking. Considering this factor, the algorithm properly adjusts node available time S_i in MSTaskPartition procedure (Algorithm 4).

5.2.2 Task Partitioning Algorithm

The previous section discusses how the real-time scheduling algorithm makes the admission control decision. As mentioned, upon admitting a regular task $T(A, \sigma, D)$, a certain number (n) of nodes are allocated to it at certain time points $(S_1, S_2, \cdots, S_n)$. According to the admission controller, these nodes can finish processing T by deadline $A + D$. This section presents the next step of the scheduling algorithm: the task partition procedure. How a task is partitioned and executed on the allocated n nodes is described.

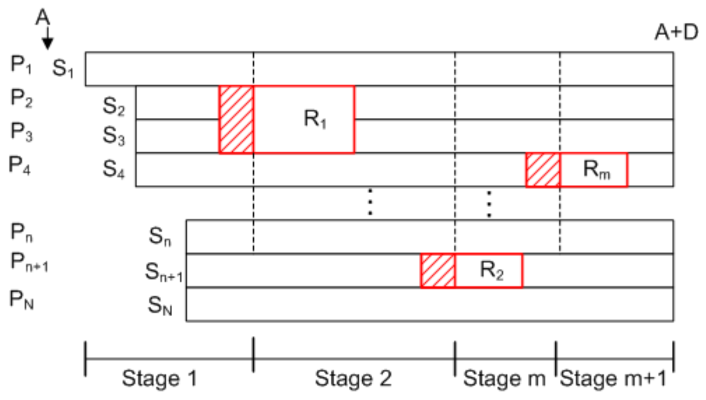


Figure 5.3: Multi-Stage Scenario 1.

Without loss of generality, the n nodes are assumed to be sorted in a non-decreasing order of their available times. i.e., $S_1 \leq S_2 \leq \cdots \leq S_n$. Task $T(A, \sigma, D)$'s

Algorithm 3 boolean AdmTest(T(A, σ, D))

```
1: $\sigma_{sum} = 0$
2: If a node $P_i$ becomes available during a reservation's data transmission, $P_i$'s avail-
   able time $S_i$ is reset at the finish time of the data transmission
3: sort nodes in non-decreasing order of node available time
4: $t_0 = \max(A, S_1)$
5: // Compute the total reservation I/O time (ResvIO) and the reservation compu-
   tation time ($ResvCP_i$) on node $P_i$ in the period [$t_0$, A+D]
6: ResvIO = 0
7: **for** i ← 1:N **do**
8:   $ResvCP_i = 0$
9: **end for**
10: **for** R($R_a$, $R_s$, n, $R_e$, $IO_{ratio}$) ∈ ResvQueue **do**
11:   **if** $R_s > t_0$ **then**
12:     $R_{io} = R_s + (R_e - R_s) \times IO_{ratio}$
13:     **if** $R_{io} > A + D$ **then**
14:       ResvIO += A + D - $R_s$
15:     **else**
16:       ResvIO += $R_{io}$ - $R_s$
17:     **end if**
18:     **for** $P_i \in$ {nodes assigned to R} **do**
19:       $ResvCP_i$ += $R_e$ - $R_{io}$
20:     **end for**
21:   **end if**
22: **end for**
23: $B_0 = 0$
24: **for** i ← 1:N **do**
25:   // Compute node $P_i$'s available processing time
26:   $\mu_i$ = A + D - $\max(S_i, A)$
27:   $\mu_i = \mu_i$ - ResvIO - $ResvCP_i$ - $B_{i-1}$
28:   // Compute the size of data that can be processed
29:   $\sigma_i = \frac{\mu_i}{\tau+\chi}$
30:   $\sigma_{sum} = \sigma_{sum} + \sigma_i$
31:   **if** $\sigma_{sum} \geq \sigma$ **then**
32:     $n^{min} = i$
33:     assign the first $n^{min}$ nodes to $T$ at their corresponding available times
        $S_1, S_2, \cdots, S_{n^{min}}$
34:     MSTaskPartition(T(A,$\sigma$,D,$n^{min}$,$S_1, S_2, \cdots, S_{n^{min}}$))
35:     **return** true
36:   **end if**
37:   $\upsilon_i = \sigma_i \times \tau$
38:   $B_i = B_{i-1} + \upsilon_i$
39: **end for**
40: **return** false
```

processing on the n nodes should not interfere with reservations in the interval [t_0, $A+D$], where $t_0 = \max(A, S_1)$. Once accepted, a reservation $R(R_a, R_s, k, R_e, IO_{ratio})$ is guaranteed a certain number (k) of nodes at the specified start time (R_s). On the reserved nodes, the processing of regular tasks must stop before the reservation starts at R_s. If the reservation requires data transmission in the interval $[R_s, R_{io}]$, where $R_{io} = R_s + (R_e - R_s) \times IO_{ratio}$, data transmissions to other nodes cannot be scheduled in the same interval. The proposed algorithm considers these constraints when partitioning and processing a task. To be applicable to a broader range of systems, our solution does not require the support of task preemption.

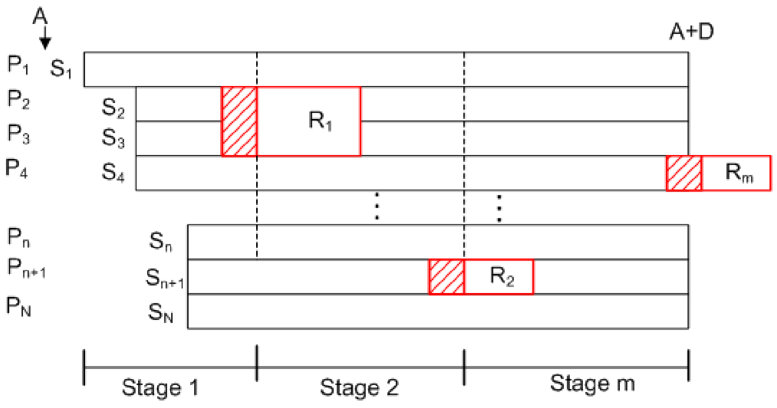


Figure 5.4: Multi-Stage Scenario 2.

Assume during the interval [t_0, $A + D$] there are m reservations, $R_1, R_2, \cdots, R_m$, in the cluster. According to each of these reservations' data transmission interval (denoted by $[R_s^i, R_{io}^i]$, where $R_{io}^i = R_s^i + (R_e^i - R_s^i) \times IO_{ratio}^i$), we divide the interval [t_0, $A + D$] into M stages. The first stage starts at t_0 and ends at R_{io}^1 (the data transmission finish time of reservation R_1). The second stage starts at R_{io}^1 and ends at R_{io}^2. In general, interval $[R_{io}^{i-1}, R_{io}^i]$ is the i^{th} stage when i is not the first or last stage. The last M^{th} stage ends at the task deadline $A + D$. If $A + D$ is not in the last reservation's data transmission interval, i.e., $A + D > R_{io}^m$, $M = m + 1$ stages are generated, and there is no reservation's data transmission in the last stage

(Figure 5.3); otherwise, $M = m$ and the last stage ends in the middle of R_m's data transmission (Figure 5.4).

After dividing the interval $[t_0, A + D]$, we form M stages and each stage includes at most one reservation's data transmission interval (i.e., the interval $[R_s^i, R_{io}^i]$), which will occur at the the end of the stage. When partitioning task T into subtasks T_j^i for the j^{th} node in the i^{th} stage, where $j = 1, 2, \cdots, n$ and $i = 1, 2, \cdots, M$, the following constraints must be satisfied. If reservation R_i is on node P_j, subtask T_j^i must finish its data transmission and computation before the reservation starts at R_s^i. On the other hand, if R_i is not on P_j, subtask T_j^i can continue its computation until the end of the stage R_{io}^i but T_j^i must finish its data transmission before R_s^i, when R_i's data transmission starts. The multi-stage task partition procedure is shown in Algorithm 4.

5.3 Proof of Algorithm Correctness

In this section, we prove the correctness of the proposed real-time scheduling algorithm. We start with proving the algorithm correctness in special scenarios and then generalize the proof.

Case 1: A Group of Processors Are Available Simultaneously

For this case, we assume that a group of l ($l \leq N$) processors $\{P_1, \cdots, P_l\}$ are available at the same time x (See Figure 5.5). During the closed time interval $[R_s, y]$, a reservation R is transmitting data. We consider admission and partitioning of a task among l processors in the time interval $[x, y]$ and prove that the workload admitted to execute on the l processors in the time interval $[x, y]$ can be completed.

Lemma 5.3.1 *If the allocated processors are available at the same time, then the task will meet the deadline.*

Algorithm 4 MSTaskPartition(T(A,σ,D,n,$S_1, S_2, \cdots, S_n$))

```
1:  // Input: task $T$ and its allocated nodes M: number of stages in the interval [$t_0$, $A + D$], where $t_0 = \max(A, S_1)$
2:  // Output: task partition vector $a[n, M]$, where $0 \leq a[j, i] \leq 1$ is the fraction of data allocated to the $j^{th}$ node in the $i^{th}$ stage, and $\sum_{j=1}^{n} \sum_{i=1}^{M} a[j, i] = 1$
3:  $\sigma_{sum} = 0$
4:  $\sigma_{last} = \sigma$
5:  // Initialize $t_j$, $P_j$'s start time in current stage
6:  for $j \leftarrow 1 : n$ do
7:      $t_j = S_j$
8:  end for
9:  for $i \leftarrow 1 : M$ do
10:     sort the $n$ nodes by their start times
11:     for $j \leftarrow 1 : n$ do
12:         if $P_j \in$ {nodes assigned to $R_i$} then
13:             // Fraction of data that can be processed before $R_s^i$
14:             $a[j, i] = \frac{R_s^i - t_j}{\sigma(\tau + \chi)}$
15:         else
16:             // Fraction of data that can be transmitted
17:             // before $R_s^i$ and processed before $R_{io}^i$
18:             $tmp = \min(R_s^i - t_j, (R_{io}^i - t_j)\frac{\tau}{\tau + \chi})$
19:             $a[j, i] = \frac{tmp}{\sigma\tau}$
20:         end if
21:         // Update $P_{j+1}$'s start time, considering blocking
22:         $t_{j+1} = \max(t_j + a[j, i]\sigma\tau, t_{j+1})$
23:         // Compute $P_j$'s start time in next stage
24:         if $P_j \in$ {nodes assigned to $R_i$} then
25:             $t_j = R_e^i$
26:         else
27:             $t_j = R_{io}^i$
28:         end if
29:         $\sigma_{sum} = \sigma_{sum} + a[j, i]\sigma$
30:         if $\sigma_{sum} \geq \sigma$ then
31:             $a[j, i] = \frac{\sigma_{last}}{\sigma}$
32:             /* Record these multi-stage data transmissions in the $n$ nodes. Update the other nodes'available times considering the blocking caused by $T$'s first stage data transmission. When scheduling other tasks in the future, a later stage data transmission of $T$ will be treated the same as a reservation's data transmission. */
33:             $UpdateNodeStatus()$
34:             return
35:         end if
36:         $\sigma_{last} = \sigma - \sigma_{sum}$
37:     end for
38: end for
```

Proof It is easy to see that the workload admitted by the admission controller is no more than the workload that can be dispatched by the partitioning algorithm.

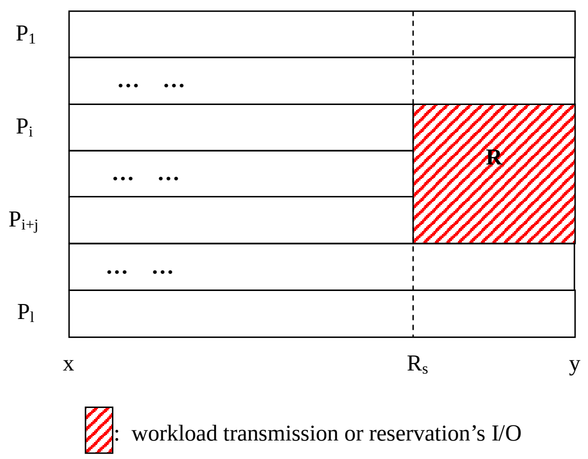


Figure 5.5: Single-Stage Case 1: l Processors with the Same Available Time.

According to the admission control algorithm (Section 5.2.1), we estimate that in the time interval $[x, y]$, the available processing time on node P_k, $k \leq l$, is $\mu_k = y - x - ResvIO - B_{k-1}$, where $ResvIO$ represents the time consumed by the reservation's I/O and B_{k-1} counts the blocking time caused by the regular task's data transmissions to nodes $\{P_1, \cdots, P_{k-1}\}$, leading to $\mu_k = R_s - x - B_{k-1}$. That is, due to the reservation's I/O in $[R_s, y]$, the admission control algorithm assumes that processors are only available for processing regular tasks in the time interval $[x, R_s]$.

We use σ^{est} and σ^{act} to respectively denote the workload estimated by the admission control algorithm and the actual workload that can be processed by the l processors in the time interval $[x, y]$. As we have derived in our earlier work [45], when simultaneously allocating l processors to a divisible task T of size σ, the task execution time is

$$\mathcal{E} = \frac{1-\beta}{1-\beta^l}\sigma(\tau + \chi) \tag{5.1}$$

where

$$\beta = \frac{\chi}{\tau + \chi} \tag{5.2}$$

In addition, since the admission control algorithm assumes that the l processors are only available in the time interval $[x, R_s]$, we have

$$\sigma^{est} = \frac{R_s - x}{\frac{1-\beta}{1-\beta^l}(\tau + \chi)} \tag{5.3}$$

In contrast, the task partition algorithm leverages the fact that in the time interval $[R_s, y]$, although the data transmission cannot be done for regular tasks due to the conflict with the reservation's I/O, idle processors can still run computations for regular tasks. Following the task partitioning algorithm, we distribute workload to the l processors. There could be two typical cases for the actual workload processing.

Subcase 1.1: The transmission time of regular tasks is $\geq R_s$ (See Figure 5.6). In this case, data has to be transmitted continueously till R_s. We want to prove that $\sigma^{act} \geq \sigma^{est}$.

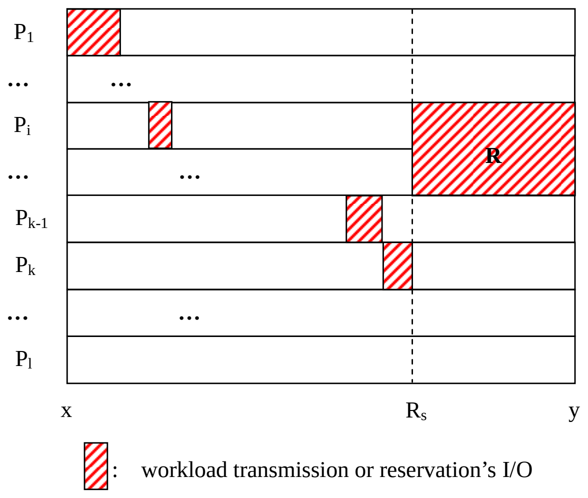


Figure 5.6: Actual Workload Processing: Typical Case 1.

The actual workload σ^{act} dispatched to the l processors is the maximum amount of workload that can be transmitted during the time interval $[x,\, R_s]$. Thus, we have

$$\sigma^{act} = \frac{R_s - x}{\tau} \tag{5.4}$$

According to Equations (5.3) and (5.2), we get

$$\sigma^{est} = \frac{R_s - x}{\frac{\tau}{1-\beta^l}} \tag{5.5}$$

From Equation (5.2), we know $0 < \beta < 1$. Thus, we have

$$\begin{aligned} 1 &> 1-\beta^l && (5.6)\\ \Rightarrow \quad \tau &< \frac{\tau}{1-\beta^l} && (5.7)\\ \Rightarrow \quad \frac{R_s - x}{\tau} &> \frac{R_s - x}{\frac{\tau}{1-\beta^l}} && (5.8) \end{aligned}$$

That is

$$\sigma^{act} > \sigma^{est} \tag{5.9}$$

Subcase 1.2: The data transmission time of regular tasks is $< R_s$. (See Figure 5.7). This indicates that enough regular workload has been transmitted to fully utilize idle processors in the time interval $[x,\, y]$. Consider a node $P_k, k \leq l$, and that P_k is not part of the reservation. Then P_k can process data in the entire interval $[x, y]$. For the node P_k, we denote the amount of workload that is processed in the time interval $[R_s, y]$ as σ_{Δ_k} and its data transmission time as t_{Δ_k}. Among the workload σ_k for the processor P_k, the σ_{Δ_k} is processed in the interval $[R_s, y]$, as demonstrated in Figure 5.8. Then we have

$$\sigma_{\Delta_k} = \frac{y - R_s}{\chi} \tag{5.10}$$

$$\text{and} \quad t_{\Delta_k} = \sigma_{\Delta_k} * \tau \tag{5.11}$$

If a node P_k is part of the reservation, in the closed interval $[R_s, \ y]$ (like nodes $P_i, \cdots, P_{i+j}$ in Figure 5.8), we have $\sigma_{\Delta_k} = 0$.

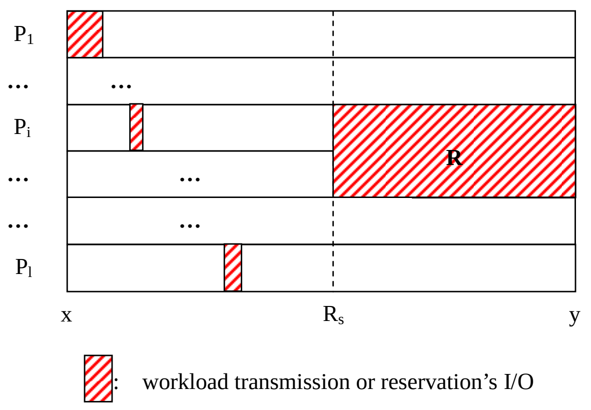


Figure 5.7: Actual Workload Processing: Typical Case 2.

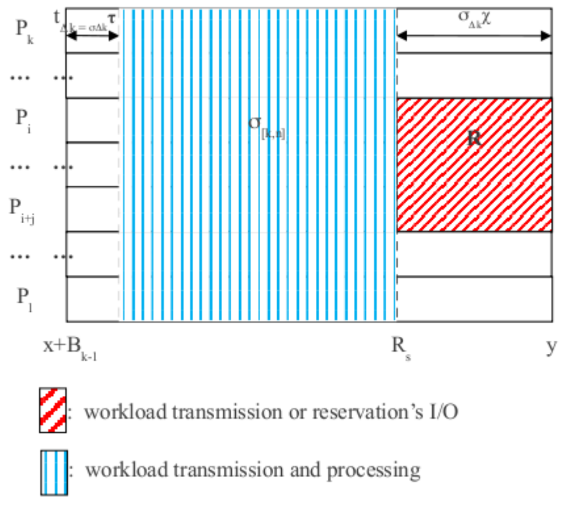

t_{Δk = σΔk τ}
σ_{Δk}χ
Pk
…
Pi
…
Pi+j
…
Pl
σ[k,l]
R
x+Bk-1
Rs
y
▨: workload transmission or reservation's I/O
▥: workload transmission and processing

Figure 5.8: Demonstration of σ_{Δ_k}, t_{Δ_k} and $\sigma_{[k,l]}$.

We also define the following terms

- σ_k^{act}: the workload dispatched to node P_k;
- $\sigma_{[k,l]}^{est}$: the workload that can be processed by nodes $\{P_k, P_{k+1}, \cdots, P_l\}$ in the time interval $[x + B_{k-1},\ R_s]$;
- $\sigma_{[k,l]}$: the workload that can be processed by nodes $\{P_k, P_{k+1}, \cdots, P_l\}$ in the time interval $[x + B_{k-1} + t_{\Delta_k}, R_s]$;
- $\sigma_{[k,l]}^{sum}$: $\sigma_{[k,l]} + \sigma_{\Delta_k}$.

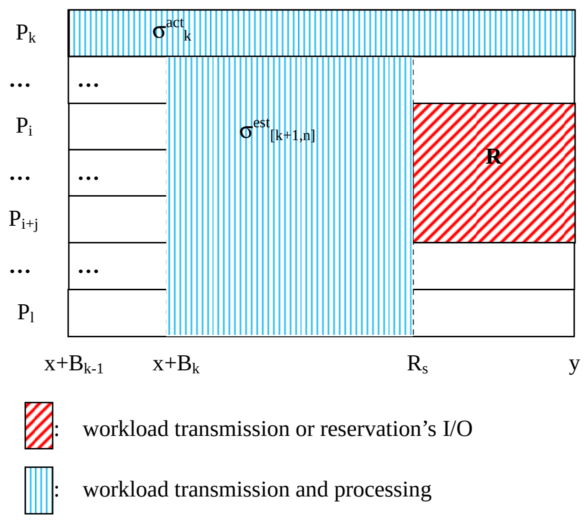

P_k
σ^act_k
...
P_i
σ^est_[k+1,n]
R
...
P_i+j
...
P_l
x+B_k-1
x+B_k
R_s
y
▨: workload transmission or reservation's I/O
▥: workload transmission and processing

Figure 5.9: Demonstration of σ_k^{act} and $\sigma_{[k+1,l]}^{est}$.

We have

$$\begin{aligned}
\sigma_{[k,l]}^{sum} &= \sigma_{[k,l]} + \sigma_{\Delta_k} \\
&= \frac{R_s - (x + B_{k-1} + t_{\Delta_k})}{\frac{1-\beta}{1-\beta^{l-k+1}}(\chi + \tau)} + \frac{t_{\Delta_k}}{\tau} \\
&= \frac{R_s - (x + B_{k-1})}{\frac{1-\beta}{1-\beta^{l-k+1}}(\chi + \tau)} - \frac{t_{\Delta_k}}{\frac{\tau}{1-\beta^{l-k+1}}} \\
&+ \frac{t_{\Delta_k}}{\tau} \\
&= \sigma_{[k,l]}^{est} - \frac{t_{\Delta_k}}{\frac{\tau}{1-\beta^{l-k+1}}} + \frac{t_{\Delta_k}}{\tau} \\
&> \sigma_{[k,l]}^{est} \qquad (5.12)
\end{aligned}$$

In addition, since $\sigma_{[k,l]}^{sum} = \sigma_{[k,l]} + \sigma_{\Delta_k} = \sigma_{[k+1,l]}^{est} + \sigma_k^{act}$, we get

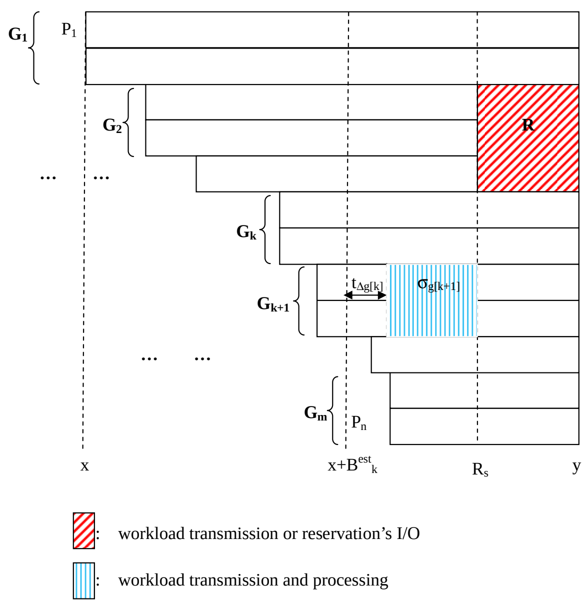
$$\begin{aligned}
\sigma_{[k+1,l]}^{est} + \sigma_k^{act} &> \sigma_{[k,l]}^{est} \Rightarrow \qquad (5.13) \\
\sigma_k^{act} &> \sigma_{[k,l]}^{est} - \sigma_{[k+1,l]}^{est} \qquad (5.14)
\end{aligned}$$

It follows that

$$\begin{aligned}
\sigma^{act} &= \sum_{k=1}^{l} \sigma_k^{act} \\
&> \sum_{k=1}^{l} (\sigma_{[k,l]}^{est} - \sigma_{[k+1,l]}^{est}) \\
Thus \qquad \sigma^{act} &> \sigma_{[1,l]}^{est} - \sigma_{[l+1,l]}^{est} \qquad (5.15)
\end{aligned}$$

Since we are considering the workload allocation to nodes $\{P_1, P_2, \cdots, P_l\}$, $\sigma_{[l+1,l]}^{est} = 0$. Substituting $\sigma^{est} = \sigma_{[1,l]}^{est}$ into Equation (5.15), we have

$$\sigma^{act} > \sigma^{est} \qquad (5.16)$$



G1
P1
G2
R
...
Gk
Gk+1
tΔg[k]
σg[k+1]
Gm
Pn
x
x+B^est_k
Rs
y
workload transmission or reservation's I/O
workload transmission and processing

Figure 5.10: Single-Stage Case 2: Multiple Same-Available-Time Processors Groups.

In the above, we have proved the algorithm correctness for Case 1 (Figure 5.5) when l processors are available at the same time, by showing that the workload admitted to execute on the l processors (i.e., σ^{est}) is never more than that can be processed by the l processors (i.e., σ^{act}).

Case 2: A Group of Processors Are Available at Different Times

Lemma 5.3.2 *If the allocated processors are available at different times, then the task will meet the deadline.*

Proof We prove Case 2, in a general scenario, where there are many processors groups and each of which have same available time (Figure 5.10). That is, we prove

$$\sum_{i=1}^{m} \sigma_{g[i]}^{act} \geq \sum_{i=1}^{m} \sigma_{g[i]}^{est} \tag{5.17}$$

The proof is based on induction. **Base Case**: since $\sigma^{act} \geq \sigma^{est}$ holds for Case 1 (Figure 5.5), we know the following inequality is true

$$\sigma_{g_{[1]}}^{act} \geq \sigma_{g_{[1]}}^{est} \tag{5.18}$$

Induction: We assume that Equation (5.17) holds for all $j \leq k$, that is,

$$\sum_{i=1}^{j} \sigma_{g_{[i]}}^{act} \geq \sum_{i=1}^{j} \sigma_{g_{[i]}}^{est}, \quad j = 1, 2, ...k \tag{5.19}$$

Now we want to prove Equation (5.17) also holds for $k + 1$ groups. From Equation (5.19) when $j = k$, we have

$$\sigma_{\Delta_{g_{[k]}}} = \sum_{i=1}^{k} \sigma_{g_{[i]}}^{act} - \sum_{i=1}^{k} \sigma_{g_{[i]}}^{est} \geq 0 \tag{5.20}$$

This workload difference $\sigma_{\Delta_{g_{[k]}}}$ is caused by the admission control algorithm's pessimistic behavior assuming that

1. Even though all processors are available but can not be used for regular tasks in the time interval $[R_s, y]$ and that
2. a processor experiences the worst-case blocking time, i.e, when nodes $\{P_1, P_2, \cdots, P_{k-1}\}$ are transmitting data for regular workload, node P_k is blocked and thus assumed unavailable.

On the other hand, the partition algorithm utilizes these idle resources and processes $\sigma_{\Delta_{g_{[k]}}}$ extra workload on the first k groups of processors. Due to this workload increase, the blocking time encountered by group G_{k+1} increases and its processors' available time reduces by $t_{\Delta_{g_{[k]}}} = \sigma_{\Delta_{g_{[k]}}} \times \tau$.

We define the following terms:

- $B_{g_{[k]}}^{est}$: the first k groups' data transmission, i.e., group G_{k+1}'s blocking time estimated by the admission control algorithm;

- $\sigma_{g[k+1]}$: the workload that can be processed by group G_{k+1} in the time interval $[x + B^{est}_{g_{[k]}} + t_{\Delta_{g_{[k]}}}, R_s]$ (see Figure 5.10);
- $l_{g_{[k]}}$: the number of processors in group G_k.

Since $\sigma_{\Delta_{g_{[k]}}} = \frac{t_{\Delta_{g[k]}}}{\tau}$, we have

$$\begin{aligned} \sigma_{\Delta_{g_{[k]}}} &\geq \frac{t_{\Delta_{g_{[k]}}}}{\frac{Cms}{1-\beta^{l_{g_{[k+1]}}}}} && (5.21) \\ \text{i.e.,} \quad \sigma_{\Delta_{g_{[k]}}} &\geq \frac{t_{\Delta_{g_{[k]}}}}{\frac{1-\beta}{1-\beta^{l_{g_{[k+1]}}}}(\tau + C_{ps})} && (5.22) \\ & && (5.23) \end{aligned}$$

Again, since $\sigma_{g_{[k+1]}} = \frac{R_s - (x + B^{est}_{g_{[k]}} + t_{\Delta_{g_{[k]}}})}{\frac{1-\beta}{1-\beta^{l_{g_{[k+1]}}}}(\tau+\chi)}$, we get

$$\begin{aligned} \sigma_{\Delta_{g_{[k]}}} + \sigma_{g_{[k+1]}} &\geq \frac{R_s - (x + B^{est}_{g_{[k]}})}{\frac{1-\beta}{1-\beta^{l_{g_{[k+1]}}}}(C_{ms} + C_{ps})} \\ \text{i.e.,} \quad \sigma_{\Delta_{g_{[k]}}} + \sigma_{g_{[k+1]}} &\geq \sigma^{est}_{g_{[k+1]}} && (5.24) \end{aligned}$$

Considering that group G_{k+1} has $l_{g_{[k+1]}}$ processors in the time interval $[x + B^{est}_{g_{[k]}} + t_{\Delta_{g_{[k]}}}, y]$ and leveraging the result we have proved for Case 1 (Figure 5.5), we get

$$\sigma^{act}_{g_{[k+1]}} \geq \sigma_{g_{[k+1]}} \qquad (5.25)$$

From Equations (5.24) and (5.25), it follows that

$$\sigma^{act}_{g_{[k+1]}} \geq \sigma^{est}_{g_{[k+1]}} - \sigma_{\Delta_{g_{[k]}}} \qquad (5.26)$$

Substituting $\sigma_{\Delta_{g_{[k]}}}$ by $\sum_{i=1}^{k} \sigma^{act}_{g_{[i]}} - \sum_{i=1}^{k} \sigma^{est}_{g_{[i]}}$, it becomes

$$\sigma^{act}_{g[k+1]} \geq \sigma^{est}_{g[k+1]} - (\sum_{i=1}^{k} \sigma^{act}_{g[i]} - \sum_{i=1}^{k} \sigma^{est}_{g[i]}) \quad (5.27)$$

$$\text{i.e.,} \quad \sum_{i=1}^{k+1} \sigma^{act}_{g[i]} \geq \sum_{i=1}^{k+1} \sigma^{est}_{g[i]} \quad (5.28)$$

This completes the algorithm correctness proof, i.e., the induction-based proof of $\sum_{i=1}^{m} \sigma^{act}_{g[i]} \geq \sum_{i=1}^{m} \sigma^{est}_{g[i]}$, for Case 2 (see Figure 5.10). We conclude that $\sigma^{act} \geq \sigma^{est}$ holds for the single-stage Case 2.

Combining lemma 5.3.1 and lemma 5.3.2, we have the following lemma.

Lemma 5.3.3 *In a stage, tasks will meet their deadlines.*

Case 3: Multi-stage Scenario Next, we present the proof for the multi-stage case and show that $\sigma^{act} \geq \sigma^{est}$ holds for the multi-stage scenario, i.e., Case 3 shown in Figure 5.11.

Lemma 5.3.4 *The multi-stage dispatch algorithm's actual dispatched workload is no less than the estimated workload by the admission controller.*

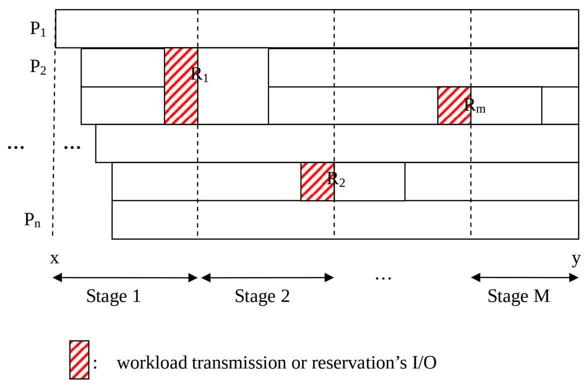


Figure 5.11: Case 3: Multi-Stage Scenario.

According to the proved result for the single-stage (Figure 5.10), we know that for each stage k in Figure 5.11, we have $\sigma^{act}_{s[k]} \geq \sigma^{est}_{s[k]}$. Therefore

$$\sum_{k=1}^{M} \sigma^{act}_{s_{[k]}} \geq \sum_{k=1}^{M} \sigma^{est}_{s_{[k]}} \tag{5.29}$$

$$\text{i.e.,} \quad \sigma^{act} \geq \sum_{k=1}^{M} \sigma^{est}_{s_{[k]}} \tag{5.30}$$

We now prove that $\sigma^{est} = \sum_{k=1}^{M} \sigma^{est}_{s_{[k]}}$ holds. if $\sigma^{est} = \sum_{k=1}^{M} \sigma^{est}_{s[k]}$ were true, we can conclude $\sigma^{act} \geq \sigma^{est}$.

According to the admission control algorithm (Algorithm 3), $\sigma^{est} = \sum_{i=1}^{n} \sigma^{est}[i]$ where $\sigma^{est}[i]$ is the estimated workload for node P_i in the time interval $[x, y]$. We use $\sigma^{est}[i, k]$ to represent the estimated workload for node P_i in stage k and prove by induction that $\forall j \in \{1, 2, \cdots, n\}$, $\sum_{i=1}^{j} \sigma^{est}[i] = \sum_{i=1}^{j} \sum_{k=1}^{M} \sigma^{est}[i, k]$ is true.

We define the following terms

- $I_{s[k]}$: the reservation I/O in stage k;
- $C[i, k]$: the reservation's computation on node P_i in stage k;
- $\mu[i, k]$: node P_i's available time in stage k;
- $L : y - x$.

According to Algorithm 3, we have $\sigma^{est}[i] = \frac{\mu_i}{\tau + \chi}$ and

$$\begin{aligned} \mu_i &= y - x - ResvIO - ResvCP_i - B_{i-1} \\ &= L - \sum_{k=1}^{M} (I_{s[k]} + C[i, k]) - B_{i-1} \end{aligned} \tag{5.31}$$

Base case: since $B_0 = 0$, we get

$$\begin{aligned} \mu_1 &= L - \sum_{k=1}^{M} (I_{s[k]} + C[1,k]) \\ \Rightarrow \mu_1 &= \sum_{k=1}^{M} \mu[1,k] \\ \Rightarrow \frac{\mu_1}{\tau + \chi} &= \frac{\sum_{k=1}^{M} \mu[1,k]}{\tau + \chi} \\ \text{i.e., } \sigma^{est}[1] &= \sum_{k=1}^{M} \sigma^{est}[1,k] \end{aligned} \tag{5.32}$$

Induction: we assume

$$\sum_{i=1}^{h} \sigma^{est}[i] = \sum_{i=1}^{h} \sum_{k=1}^{M} \sigma^{est}[i,k] \tag{5.33}$$

We prove $\sum_{i=1}^{h+1} \sigma^{est}[i] = \sum_{i=1}^{h+1} \sum_{k=1}^{M} \sigma^{est}[i,k]$ holds. Algorithm 3 assumes that $B_h = \sum_{i=1}^{h} \sigma^{est}[i] \times \tau$. According to the assumption made in Step 2, we have

$$B_h = \sum_{i=1}^{h} \sum_{k=1}^{M} \sigma^{est}[i,k] \times \tau \tag{5.34}$$

Substituting the above equation into Equation (5.31), it becomes

$$\begin{aligned}
\mu_{h+1} = L \quad &- \quad \sum_{k=1}^{M}(I_{s[k]} + C[h+1,k] \\
&+ \quad \sum_{i=1}^{h} \sigma^{est}[i,k] \times \tau) \qquad (5.35) \\
\Rightarrow \mu_{h+1} \quad &= \quad \sum_{k=1}^{M} \mu[h+1,k] \\
\Rightarrow \frac{\mu_{h+1}}{\tau+\chi} \quad &= \quad \frac{\sum_{k=1}^{M} \mu[h+1,k]}{\tau+\chi} \\
\text{i.e., } \sigma^{est}[h+1] \quad &= \quad \sum_{k=1}^{M} \sigma^{est}[h+1,k] \qquad (5.36)
\end{aligned}$$

Adding Equations (5.33) and (5.36), we have

$$\sum_{i=1}^{h+1} \sigma^{est}[i] = \sum_{i=1}^{h+1} \sum_{k=1}^{M} \sigma^{est}[i,k] \qquad (5.37)$$

which completes the induction-based proof of $\forall j \in \{1, 2, \cdots, n\}$, $\sum_{i=1}^{j} \sigma^{est}[i] = \sum_{i=1}^{j} \sum_{k=1}^{M} \sigma^{est}[i,k]$. Therefore,

$$\begin{aligned}
\sum_{i=1}^{n} \sigma^{est}[i] \quad &= \quad \sum_{i=1}^{n} \sum_{k=1}^{M} \sigma^{est}[i,k] \\
\text{i.e., } \sigma^{est} \quad &= \quad \sum_{i=1}^{n} \sum_{k=1}^{M} \sigma^{est}[i,k] \\
\Rightarrow \quad \sigma^{est} \quad &= \quad \sum_{k=1}^{M} \sigma^{est}_{s[k]} \qquad (5.38)
\end{aligned}$$

With Equations (5.30) and (5.38), we conclude that $\sigma^{act} \geq \sigma^{est}$ holds for Case 3 and the multi-stage real-time scheduling algorithm is correct.

Theorem 5.3.5 *The proposed multi-stage algorithm is correct.*

Proof The proof follows from lemmas 5.3.1, 5.3.2, and 5.3.3.

5.4 Performance Evaluation

In the previous section, we presented a real-time scheduling algorithm that supports advance reservations. In this section, we experimentally evaluate the performance of the algorithm.

5.4.1 Simulation Configurations

We develop a discrete simulator to simulate a wide range of clusters. Three parameters, N, τ and χ are specified for every cluster.

For a set of regular tasks $T_i = (A_i, \sigma_i, D_i)$, A_i, the task arrival time, is specified by assuming that the interarrival times follow an exponential distribution with a mean of $1/\lambda$; task data sizes σ_i are assumed to be normally distributed with the mean and the standard deviation equal to $Avg\sigma$; task relative deadlines (D_i) are assumed to be uniformly distributed in the range $[\frac{AvgD}{2}, \frac{3AvgD}{2}]$, where $AvgD$ is the mean relative deadline. To specify $AvgD$, we use the term $DCRatio$ defined in [45]. It is defined as the ratio of mean deadline to mean minimum execution time (cost), that is $\frac{AvgD}{\mathcal{E}(Avg\sigma, N)}$, where $\mathcal{E}(Avg\sigma, N)$ is the execution time assuming the task has an average data size $Avg\sigma$ and is allocated to run on N fully-available nodes simultaneously [45]. Given a $DCRatio$, the cluster size N and the average data size $Avg\sigma$, $AvgD$ is implicitly specified as $DCRatio \times \mathcal{E}(Avg\sigma, N)$. Thus, task relative deadlines are related to the average task execution time. In addition, a task's relative deadline D_i is chosen to be larger than its minimum execution time $\mathcal{E}(\sigma_i, N)$. In summary, we specify the following parameters for a simulation: N, τ, χ, $1/\lambda$, $Avg\sigma$, and $DCRatio$.

To analyze the cluster load for a simulation, we use the metric $SystemLoad$ [43].

It is defined as, $SystemLoad = \frac{\mathcal{E}(Avg\sigma,1)\times\lambda}{N}$, which is the same as, $SystemLoad = \frac{TotalTaskNumber\times\mathcal{E}(Avg\sigma,1)}{TotalSimulationTime\times N}$. For a simulation, we can specify $SystemLoad$ instead of average interarrival time $1/\lambda$. Configuring (N, τ, χ, $SystemLoad$, $Avg\sigma$, $DCRatio$) is equivalent to specifying (N, τ, χ, $1/\lambda$, $Avg\sigma$, $DCRatio$), because, $1/\lambda = \frac{\mathcal{E}(Avg\sigma,1)}{SystemLoad\times N}$.

Algorithm 5 $RegTask2Resv(T(A,\sigma,D),\Delta)$

```
1: // **Input:**
2: // Regular task $T(A,\sigma,D)$ and
3: // advance factor $\Delta$
4: // **Output:**
5: // Reservation $R(R_a, R_s, n, R_e, IO_{ratio})$
6: // Make ResvStartTime = RegTaskArrivalTime
7: $R_s = A$
8: // Compute ResvLength ($R_e$ - $R_s$) based
9: // on RegTaskExecutionTime $\mathcal{E}(\sigma, n^{min})$
10: $\gamma = 1 - \frac{\sigma\tau}{D}$
11: $\beta = \frac{\chi}{\chi+\tau}$
12: $n^{min} = \lceil\frac{ln\gamma}{ln\beta}\rceil$
13: $\mathcal{E} = \frac{1-\beta}{1-\beta^{n^{min}}}\sigma(\tau+\chi)$
14: // ResvLength = RegTaskExecutionTime $\mathcal{E}(\sigma, n^{min})$
15: $R_e = R_s + \mathcal{E}$
16: // Nodes reserved = minimum nodes required by
17: // RegTask at $A$ to complete before its deadline $D$
18: $n = n^{min}$
19: // Make Resv $IO_{ratio}$ =
20: // RegTaskTranmissionTime / RegTaskExecutionTime
21: $IO_{ratio} = \frac{\sigma Cms}{\mathcal{E}}$
22: // The request for Resv arrives $\Delta$ time unit in advance
23: $R_a = R_s - \Delta$
```

To generate reservations, some regular tasks are selected from the aforementioned workload and converted to reservations. To study the algorithm's performance under varied conditions, different percents of the workload are converted to reservations. Algorithm 5 describes the procedure of converting a regular task to a reservation. To ensure that the newly generated workload, mixed with reservations and regular tasks, leads to the same $SystemLoad$ as the original workload, the reservation start time is made equal to the regular task's arrival time. In addition, the number of nodes reserved equals to the minimum number of nodes n^{min} required to finish the regular task before its deadline. It follows that the reservation length is equal to the regular task's execution time $\mathcal{E}(\sigma, n^{min})$; and the reservation's IO_{ratio} is defined according to the regular task's I/O ratio.

Advance Factor for Reservation Since a reservation is often made in advance, we use Δ, called the *advance factor*, to specify the time difference between the arrival of the reservation request (R_a) and the reservation start time (R_s), i.e., $\Delta = R_s - R_a$. Figure 5.12 shows an example where task T_9 is selected and converted to reservation R_9, which is then assumed to arrive before task T_4. Therefore, for the new workload, tasks T_1, T_2, T_3 and reservation R_9 will be scheduled first, followed by tasks $T_4, T_5, \cdots, T_8$ and T_{10}. The earlier a reservation is made, the greater its chance of being accepted. To study an advance reservation's impact on system performance, different advance factors are simulated.

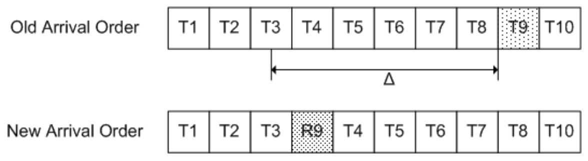


Figure 5.12: An Example of Mixed Workload Generation.

The simulation in this chapter is divided into several experiments. Each experiment consists of several runs, where each run is further divided into ten tests. The parameters N, τ, χ remain constant over all runs. However, *SystemLoad*, *Avg*σ, *DCRatio*), Δ and reservation percentage vary from run to run. For each test, different random numbers are generated for task arrival times A_i, data sizes σ_i and deadlines D_i. For all figures in this section, a point on a curve corresponds to the average performance of ten tests in a specific run of an experiment. For each simulation, the $TotalSimulationTime$ is 10,000,000 time units, which is sufficiently long.

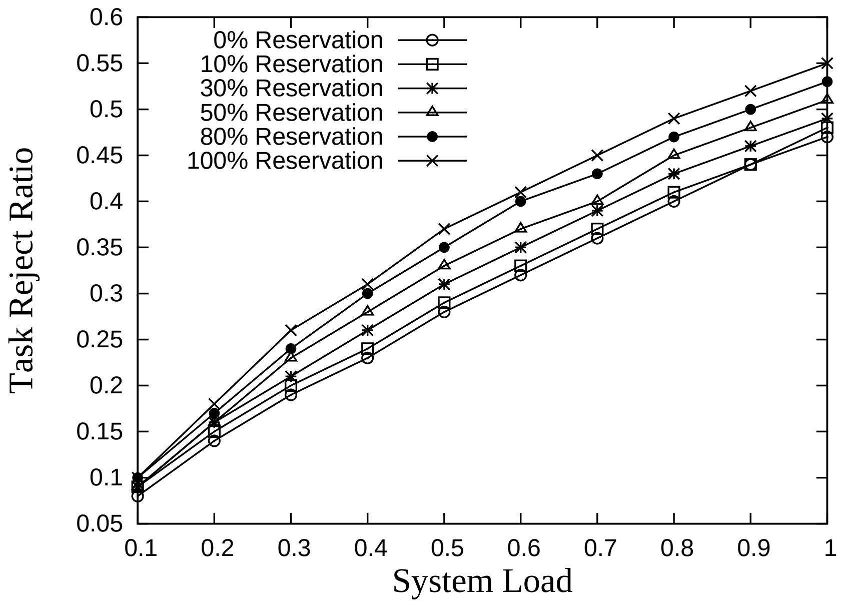
Baseline Configuration. For our basic simulation model we chose the following parameters: number of processing nodes in the cluster $N = 256$; unit data transmission time $\tau = 1$; unit data processing time $\chi = 1000$; $SystemLoad$ changes in the range $\{0.1, 0.2, \cdots, 1.0\}$; Average data size $Avg\sigma = 2000$; and the ratio of the average deadline to the average execution time $DCRatio = 2$. Our simulation has a threefold objective. First, we verify the correctness of the proposed algorithm. Second, we study the effects of reservation percentage, and third, we want to investigate effects of advance factor Δ.

5.4.2 Simulation Results

To validate that the proposed algorithm works correctly, we check all simulation results to verify that real-time requirements of every accepted task are satisfied. There are enough resources to guarantee reservations start and finish at the specified times. Once accepted, tasks are successfully processed by their deadlines.

Effects of Reservation Percentage. We conducted experiments with the baseline configuration. To study the effects of reservation percentage, 0%, 10%, 30%, 50%, 80% and 100% of the workload were set to be reservations respectively.

In the first experiment, we set the advance factor $\Delta = 0$. That is, all reservations request to be started immediately. Figures 5.13a and 5.14a show the simulation results. We can see that for a workload of all regular tasks (i.e., 0% reservation), the scheduler rejects the least number of tasks (TRR) and leads to the highest system utilization (UTIL). As the reservation percentage of the workload increases from 0% to 100%, the TRR increases and the UTIL decreases. These results follow the common intuition that making reservations can reduce system performance. Reservations must start at the requested time and execute continuously until completion. Reservation

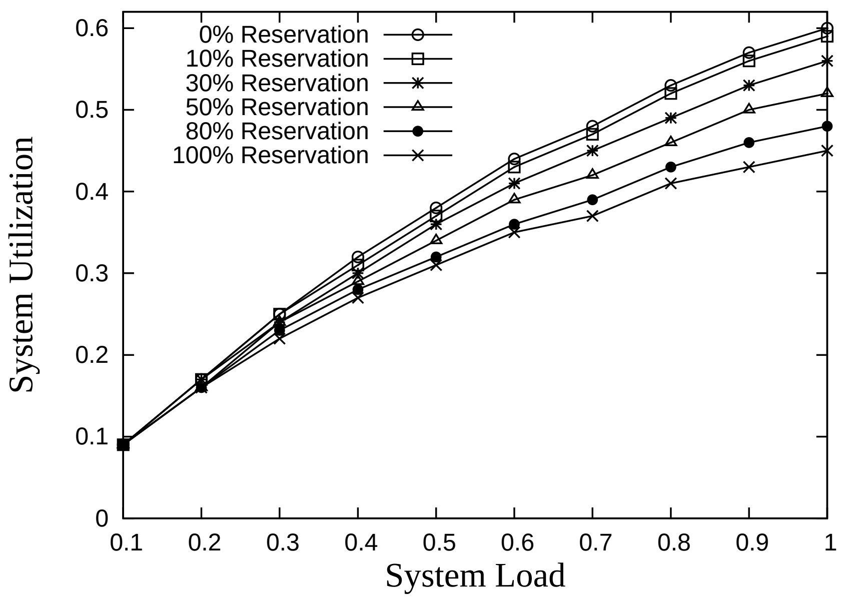


(a) Task reject ratio (Δ=0).

Figure 5.13: Effects of Reservation Percentage (Task Reject Ratio (Δ=0)).

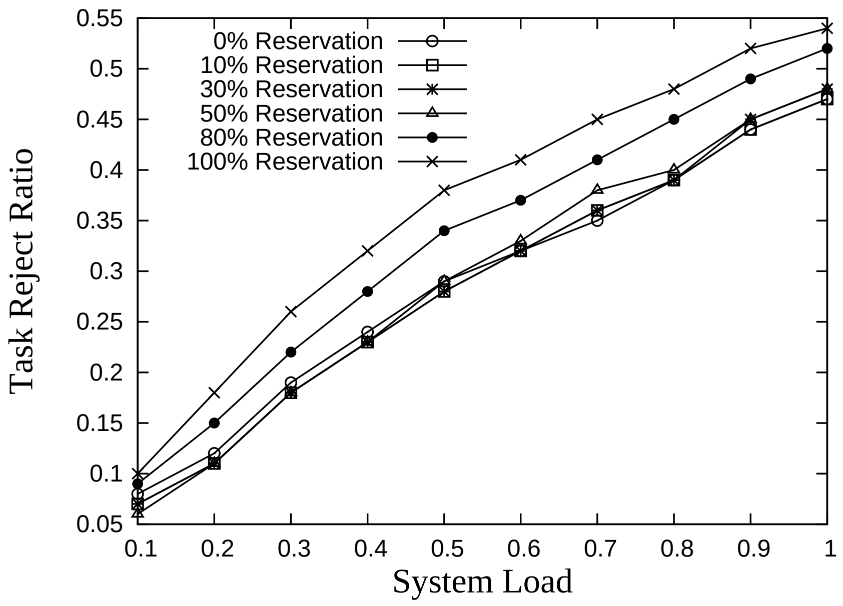
tasks offer little flexibility to the scheduler. In contrast, regular tasks are flexible. That is, they can start at any time as long as they finish before their deadlines. Some parallel tasks, are also flexible in their required number of nodes, allowing the scheduler to dictate the allocated amount of resources. The scheduler can start them earlier with fewer nodes or later with more nodes. In particular, the arbitrarily divisible tasks considered in this chapter, give the scheduler the maximum flexibility. Such tasks can be divided into subtasks to utilize any available processing times in the cluster. These factors explain why the system performs the best with no reservation in the workload.

In the second experiment, we instead let the advance factor equal to the average task interarrival time: $\Delta = 1/\lambda$. That is, all reservations are made $1/\lambda$ time units in advance of their start times. Figures 5.15a and 5.16a show the simulation results. From Figure 5.15a, we can see that the scheduler achieves similar TRRs for workloads with 0%, 10%, 30% and 50% reservations, while the TRR for the workloads with 80%



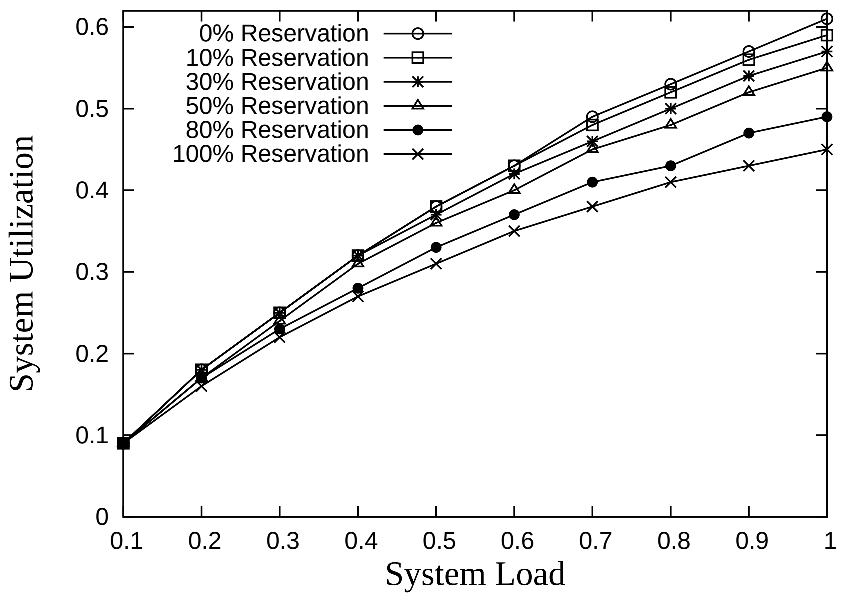
(a) System utilization (Δ=0).

Figure 5.14: Effects of Reservation Percentage (System utilization (Δ=0)).



(a) Task reject ratio (Δ=1/λ).

Figure 5.15: Effects of Reservation Percentage (Task Reject Ratio (Δ=1/λ)).



(a) System utilization (Δ=1/λ).

Figure 5.16: Effects of Reservation Percentage (System Utilization (Δ=1/λ)).

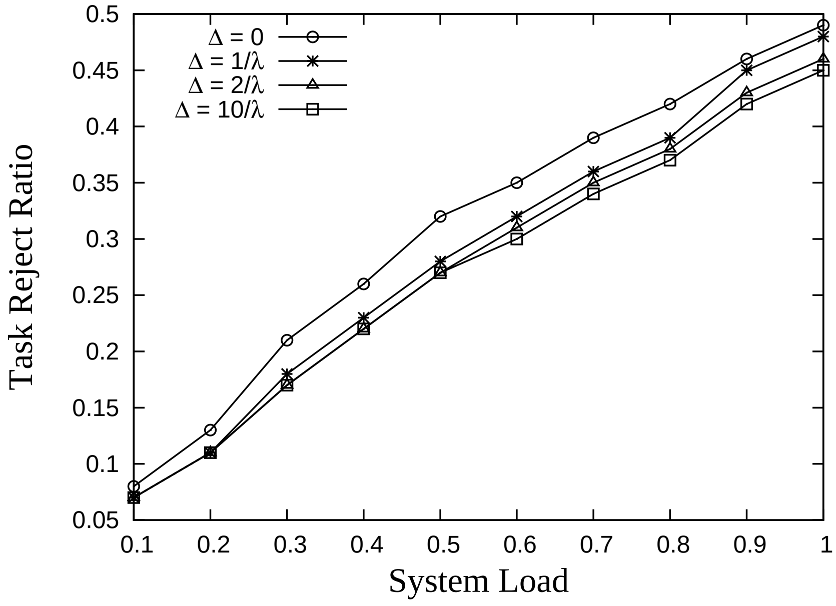


Figure 5.17: Effects of Advance Factor (Reservation Tasks 30%).

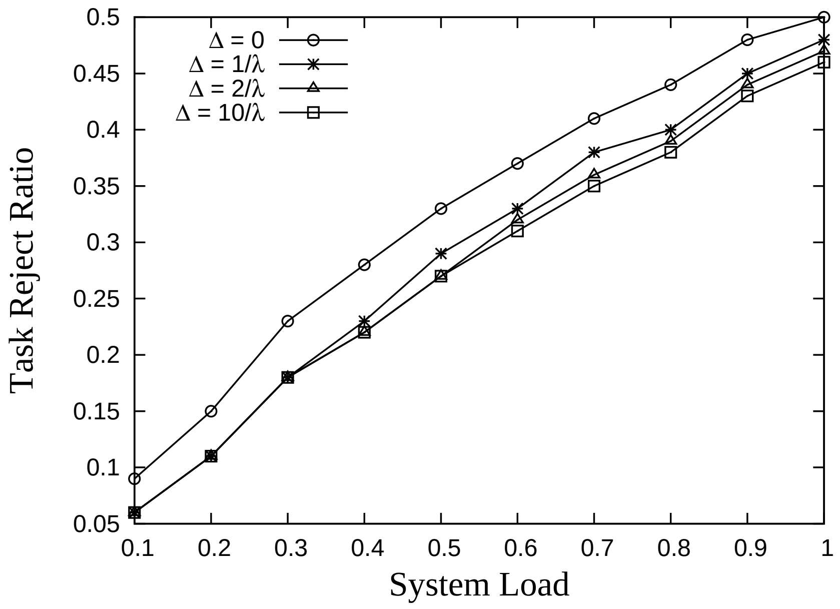


Figure 5.18: Effects of Advance Factor (Reservation Tasks 50%).

and 100% reservations increase significantly. contradicts the intuitive belief of previous researches that reservation decreases system performance. Figure 5.16a shows that UTIL obtained with a workload of no reservation is higher than those obtained with mixed workloads. However, the utilization differences between workloads of 0%, 10%, 30% and 50% reservations are quite small. These results are due to the fact that reservations are made plenty of time in advance. Since an advance reservation requests for some resources in the future, the earlier the reservation is made, the more likely the required resources have not been occupied. Therefore, advance reservations are more likely to be accepted. After cluster resources are booked by reservations, less resources are left to serve regular tasks arriving in the future. As a result, more regular tasks are rejected. However, thanks to the flexibility in scheduling regular tasks, many of them can still be accepted. This explains why the overall system performance (i.e., TRR and UTIL) does not deteriorate with the percentage increase of advance reservations from 0% to 50%.

Next, to understand how much earlier a reservation should be made, we investigate the effects of advance factor Δ.

Effects of Advance Factor. We again conducted experiments with the baseline configuration, where either 30% or 50% of the workload was set aside for reservations. To study the effects of advance factor, we set $\Delta = 0, 1/\lambda, 2/\lambda$, and $10/\lambda$ respectively. Figures 5.17 and 5.18 show the simulation results.

From both figures, we observe that when Δ increases, the TRR decreases. The improvement is significant until $\Delta = 2/\lambda$, and the TRR with advance factors $\Delta = 2/\lambda$ and $\Delta = 10/\lambda$ are similar. Since the UTIL curves have the same trend, we omit them to save space.

In the following, we use an example to illustrate how the advance factor affects the task acceptance. Figure 5.19 shows a regular task T_i arriving at A_i and a reservation

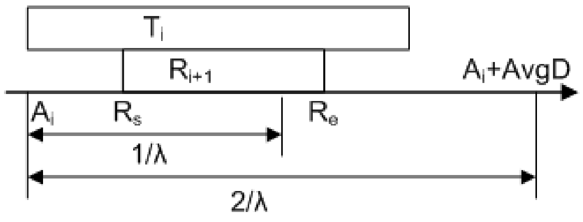


Figure 5.19: Advance Factor Effect.

R_{i+1} requesting to start at R_s. In this simple example, we assume there is only one processing node. If R_{i+1} arrives after A_i, it is rejected because the node is allocated to T_i. If R_{i+1} arrives before T_i, R_{i+1} is booked on the node before T_i arrives. Upon T_i's arrival, the scheduler may still accept T_i and let it utilize the node before and after R_{i+1}, while still finishing before its deadline. In general, since a reservation does not affect a regular task as much as a regular task affects a reservation, it is beneficial to make reservations in advance so that the scheduler can consider them before all *competing* regular tasks.

On average, when the advance reservation factor is equal to or greater than the average task interarrival time (i.e., $\Delta \geq 1/\lambda$), the competition for resources between reservations and regular tasks is less constraining and results in improved performance. When a reservation R arrives, the scheduler decides if it is feasible to schedule R without compromising the guarantees for previously admitted tasks. Therefore, R only competes with reservations and regular tasks that have already been committed to by the system. Moreover, among admitted regular tasks, only those whose deadlines are later than R's start time are actually competing with R for resources. Consequently, if the advance reservation factor is at least as large as the average task deadline (i.e., $\Delta \geq AvgD$), the competition for resources between reservations and regular tasks is almost negligible. For the simulated workloads, since $SystemLoad = \mathcal{E}(Avg\sigma, N) \times \lambda$ and $AvgD = DCRatio \times \mathcal{E}(Avg\sigma, N)$, we have $AvgD = DCRatio \times SystemLoad \times 1/\lambda \leq 2/\lambda$. This explains why we observe significant performance improvements as Δ increases until $\Delta = 2/\lambda$, and the curves for workloads with $\Delta \geq 2/\lambda$ are close to each other with less performance improvement. If a reservation is rejected, it is most likely due to conflicts with other advance reservations. As reservations compete for resources with each other, the task accept ratio decreases significantly, which explains why as the reservation percentage increases beyond 50%, system performance degrades drastically (Figures 5.15a and 5.16a).

5.5 Summary

In this chapter, we investigated the challenging problem of real-time divisible load scheduling with advance reservations in a cluster. To address the under-utilization concerns, we extensively studied the effects of advance reservations. A multi-stage real-time scheduling algorithm is proposed. Simulation results show: 1) our algorithm works correctly and provides real-time guarantees to accepted tasks; and 2) proper advance reservations could avoid the system performance degradation.

Chapter 6

An Efficient Algorithm for Real-Time Divisible Load Scheduling

6.1 Introduction

While existing real-time divisible load scheduling algorithms have focused on satisfying QoS, providing real-time guarantees, and better utilizing cluster resources, these algorithms place little emphasis on scheduling efficiency. The algorithms assume that scheduling takes much less time than the execution of a task, and thus ignore the scheduling overhead. However, clusters are becoming increasingly bigger and busier. In Table 6.1, we list the sizes of some OSG (Open Science Grid) clusters. As we can see, all of these clusters have more than one thousand CPUs, with the largest providing over 40 thousand CPUs. Figure 6.2 shows the number of tasks waiting in the OSG cluster at University of California, San Diego for two 20-hour periods, demonstrating that at times there could be as many as 37 thousand tasks in the waiting queue of a cluster. As the cluster size and workload increase, so does the scheduling overhead. For a cluster with thousands of nodes or thousands of waiting tasks, as will be demonstrated in Section 6.4, the scheduling overhead could be substantial and existing divisible load scheduling algorithms are no longer applicable due to lack of scalability. For example, to schedule the bursty workload in Figure 6.2a, the best-known real-time algorithm [17] prior to our algorithm, takes more than 11 hours to make admission control decisions on the 14,000 tasks that arrived in an hour, while our new algorithm needs only 37 minutes.

Table 6.1: Sizes of OSG Clusters.

Host Name	No. of CPUs
fermigrid1.fnal.gov	41863
osgserv01.slac.stanford.edu	9103
lepton.rcac.purdue.edu	7136
cmsosgce.fnal.gov	6942
osggate.clemson.edu	5727
grid1.oscer.ou.edu	4169
osg-gw-2.t2.ucsd.edu	3804
u2-grid.ccr.buffalo.edu	2104
red.unl.edu	1140

In this chapter, we address the deficiency of existing approaches and present an efficient algorithm for real-time divisible load scheduling. The time complexity of the proposed algorithm is linear in the maximum of the number of tasks in the waiting queue and the number of nodes in the cluster. In addition, the algorithm performs similarly to previous algorithms in terms of providing real-time guarantees and utilizing cluster resources.

Next, we discuss the real-time scheduling algorithm in Section 6.2 and evaluate the algorithm performance in Section 6.4.

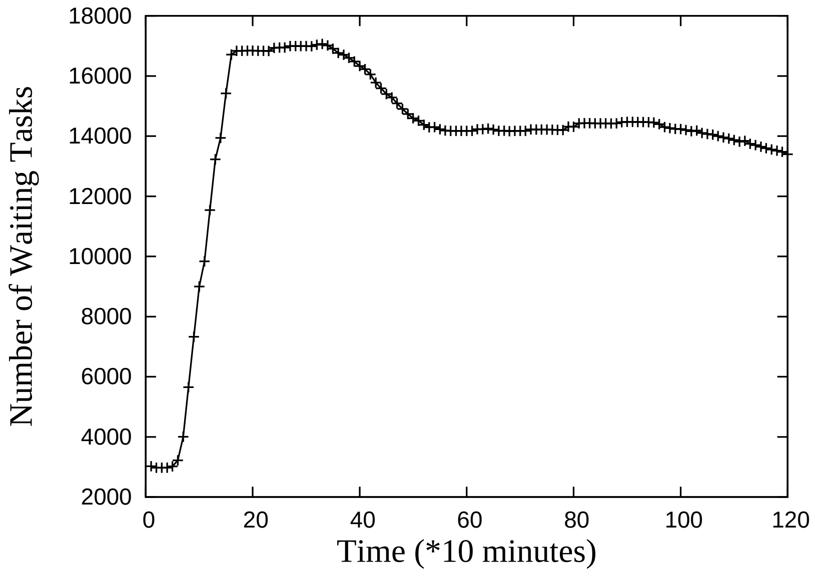


Figure 6.1: Status of a UCSD Cluster (Bursty Arrival).

6.2 Algorithm

In this section, we present our new algorithm for scheduling real-time divisible loads in clusters. We adopt the regular task model in section 3.1.1, system model and notations in section 3.2. Due to their special property, when scheduling arbitrarily divisible loads, the algorithm needs to make three important decisions: task execution order, the number n of processing nodes that should be allocated to each task and a strategy to partition the task among the allocated n nodes.

As is typical for dynamic real-time scheduling algorithms [23, 59, 66], when a task arrives, the scheduler determines if it is feasible to schedule the new task without compromising the guarantees for previously admitted tasks. Only those tasks that pass this schedulability test are allowed to enter the *task waiting queue* (TWQ). This decision module is referred to as the *admission controller*. When processing nodes become available, the *dispatcher* partitions each task and dispatches subtasks to execute on processing nodes.

For existing divisible load scheduling algorithms [17, 18, 43, 46, 45], in order to

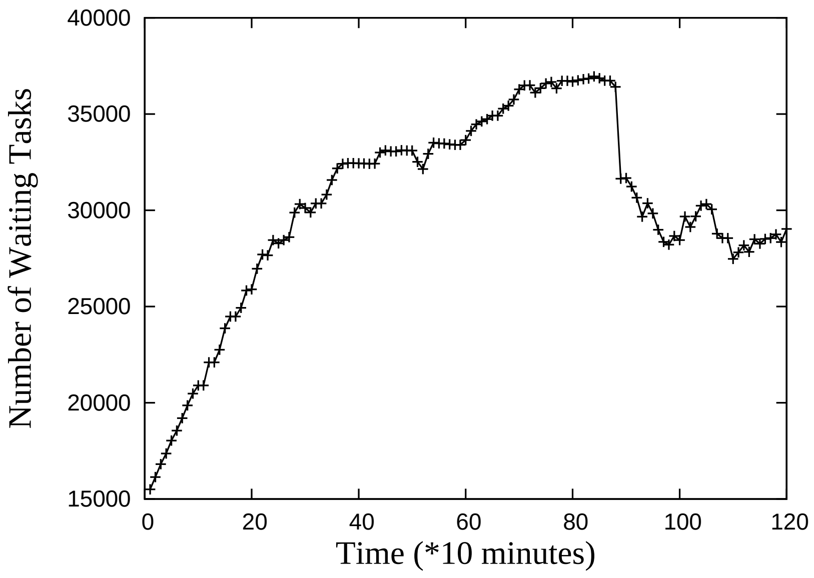


Figure 6.2: Status of a UCSD Cluster (Large Queue).

perform the schedulability test, the admission controller generates a new schedule for the newly arrived task and all tasks waiting in TWQ. If the schedule is feasible, the new task is accepted; otherwise, it is rejected. For these algorithms, the dispatcher acts as an execution agent, which simply implements the feasible schedule developed by the admission controller. There are two factors that contribute to large overheads of these algorithms. First, to make an admission control decision, they reschedule tasks in TWQ. Second, they calculate in the admission controller the minimum number n^{min} of nodes required to meet a task's deadline so that it guarantees enough resources for each task. The later a task starts, the more nodes are needed to complete it before its deadline. Therefore, if a task is rescheduled to start at a different time, the n^{min} of the task may change and needs to be recomputed. This process of rescheduling and recomputing n^{min} of waiting tasks introduces a big overhead.

To address the deficiency of existing approaches, we develop a new scheduling algorithm, which relaxes the tight coupling between the admission controller and the dispatcher. As a result, the admission controller no longer generates an exact schedule, avoiding the high overhead. To carry out the schedulability test, instead of computing n^{min} and deriving the exact schedule, the admission controller assumes that tasks are executed one by one with all processing nodes. This simple and efficient *all nodes assignment* (ANA) policy speeds up the admission control decision. The ANA is, however, impractical. In a real-life cluster, resources are shared and each task is assigned just enough resources to satisfy its needs. For this reason, when dispatching tasks for execution, our dispatcher needs to adopt a different node assignment strategy. If we assume ANA in the admission controller and let the dispatcher apply the *minimum node assignment* (MNA) policy, we reduce the real-time scheduling overhead but still allow the cluster to have a schedule that is appealing in the practical sense. Furthermore, our dispatcher dispatches a subtask as soon as a processing node and the head node become available, eliminating IITs.

Due to the superior performance of EDF-based divisible load scheduling [45], our new algorithm schedules tasks in EDF order as well. Although in this chapter, we describe the algorithm assuming EDF scheduling, the idea is applicable to other divisible load scheduling such as MWF-based scheduling algorithms [43]. In the following, we describe in detail the two modules of the algorithm: admission controller (Section 6.2.1) and dispatcher (Section 6.2.2). Since the two modules follow different rules, sometimes an adjustment of the admission controller is needed to resolve their discrepancy so that task real-time properties can always be guaranteed (Section 6.2.3). Section 6.2.4 proves the correctness of our algorithm.

6.2.1 Admission Controller

When a new task arrives, the admission controller determines if it is feasible to schedule the new task without compromising the guarantees for previously admitted tasks. In the previous work [17, 18, 43, 46, 45, 58], the admission controller follows a bruteforce approach, which inserts the new task into TWQ, reschedules each task and generates a new schedule. Depending on the feasibility of the new schedule, the new task is either accepted or rejected. As we can see, both accepting and rejecting a task involve generating a new schedule.

In this chapter, we make two significant changes in order to develop a new admission control algorithm. First, to determine the schedulability of a new task, we only check the information recorded with the two adjacent tasks (i.e., the preceding and succeeding tasks). Unlike the previous work, our new algorithm could reject a task without generating a new schedule. This significantly reduces the scheduling overhead for heavily loaded systems. Second, we separate the admission controller from the dispatcher, and to make admission control decisions, an ANA policy is assumed.

The new admission control algorithm is called AC-FAST. Algorithm 6 presents its pseudo code. The admission controller assumes an ANA policy. We use $\mathcal{E}$ and C to respectively denote the task execution time and the task completion time. AC-FAST partitions each task following the divisible load theory (DLT), which states that the optimal execution time is obtained when all nodes allocated to a task complete their computation at the same time [79]. Applying this optimal partitioning, we get the execution time of running a task $\tau(A, \sigma, D)$ on N processing nodes as [45],

$$\mathcal{E}(\sigma, N) = \frac{1-\beta}{1-\beta^N}\sigma(\tau+\chi), \tag{6.1}$$

$$where \quad \beta = \frac{\chi}{\tau+\chi}. \tag{6.2}$$

When a new task τ arrives, the algorithm first checks if the head node P_0 will be available early enough to at least finish τ's data transmission before τ's absolute deadline. If not so, task τ is rejected (lines 1-4). As the next step, task τ is tentatively inserted into TWQ following EDF order and τ's two adjacent tasks τ_s and τ_p

(i.e., the succeeding and the preceding tasks) are identified (lines 5-6). By using the information recorded with τ_s and τ_p, the algorithm further tests the schedulability. First, to check whether accepting τ will violate the deadline of any admitted task, the algorithm compares τ's execution time $\tau.\mathcal{E}$ with its successor τ_s's $slack_{min}$, which represents the minimum slack of all tasks scheduled after τ. Next, we give the formal definition of $slack_{min}$. Let S denote the task start time. A task's slack is defined as,

$$slack = A + D - (S + \mathcal{E}), \tag{6.3}$$

which reflects the scheduling flexibility of a task. Starting a task *slack* time units later does not violate its deadline. Therefore, as long as τ's execution time is no more than the slack of any succeeding task, accepting τ will not violate any admitted task's deadline. We define $\tau_i.slack_{min}$ as the *minimum* slack of all tasks scheduled after τ_{i-1}. That is,

$$\tau_{i \cdot slack_{min}} = \min(\tau_{i \cdot slack}, \tau_{i+1 \cdot slack}, \cdots, \tau_{n \cdot slack}). \tag{6.4}$$

If τ's execution time is less than its successor τ_s's $slack_{min}$, accepting τ will not violate any task's deadline (lines 7-10).

The algorithm then checks if task τ's deadline can be satisfied or not. That is, to check if $\tau.(A + D - S) \geq \tau.\mathcal{E}$, where the task start time $\tau.S$ is the preceding task's completion time $\tau_p.C$ or τ's arrival time $\tau.A$ (lines 11-31). If there is a task in TWQ, then the cluster is busy. For a busy cluster, we do not need to resolve the discrepancy between the admission controller and the dispatcher and the task real-time properties are still guaranteed (see Section 6.2.4 for a proof). However, if TWQ becomes empty, the available resources could become idle and the admission controller must consider this *resource idleness*. As a result, in our AC-FAST algorithm, when a new task

τ arrives into an empty TWQ, an adjustment is made (lines 15-17). The purpose is to resolve the discrepancy between the admission controller and the dispatcher so that the number of tasks admitted will not exceed the cluster capacity. For a detailed discussion of this adjustment, please refer to Section 6.2.3. Once a new task τ is admitted, the algorithm inserts τ into TWQ and modifies the $slack_{min}$ and the estimated completion time of tasks scheduled after τ (lines 22-31).

Time Complexity Analysis. In our AC-FAST algorithm, the schedulability test is done by checking the information recorded with the two adjacent tasks. Since TWQ is sorted, locating τ's insertion point takes $O(log(n))$ time and so do functions $getPredecessor(\tau)$ and $getSuccessor(\tau)$. Function adjust(τ) runs in $O(N)$ time (see Section 6.2.3) and it only occurs when TWQ is empty. The time complexity of function $updateSlacks$ is $O(n)$. Therefore, algorithm AC-FAST has a linear i.e., $O(\max(N, n))$ time complexity.

6.2.2 Dispatcher

The dispatching algorithm is rather straightforward. When a processing node and the head node become available, the dispatcher takes the first task $\tau(A, \sigma, D)$ in TWQ, partitions the task and sends a subtask of size $\hat{\sigma}$ to the node, where $\hat{\sigma} = \min\left(\frac{A+D-CurrentTime}{\tau+\chi}, \sigma\right)$. The remaining portion of the task $\tau(A, \sigma - \hat{\sigma}, D)$ is left in TWQ. As we can see, the dispatcher chooses a proper size $\hat{\sigma}$ to guarantee that the dispatched subtask completes no later than the task's absolute deadline $A + D$. Following the algorithm, all subtasks of a given task complete at the task absolute deadline, except for the last one, which may not be big enough to occupy the node until the task deadline. By dispatching the task as soon as the resources become available and letting the task occupy the node until the task deadline, the dispatcher allocates the minimum number of nodes to each task.

Algorithm 6 AC-FAST($\tau(A, \sigma, D)$, TWQ)

```
1: //check head node's available time
2: if ($\tau.(A + D) \leq P_0$.AvailableTime + $\tau.\sigma\tau$) then
3:     return false
4: end if
5: $\tau_p$ = getPredecessor($\tau$)
6: $\tau_s$ = getSuccessor($\tau$)
7: $\tau.\mathcal{E} = \mathcal{E}(\tau.\sigma, N)$
8: if ($\tau_s \neq$ null && $\tau.\mathcal{E} > \tau_s.slack_{min}$) then
9:     return false
10: end if
11: if ($\tau_p ==$ null) then
12:     $\tau.S = \tau.A$
13: else
14:     $\tau.S = \tau_p.C$
15:     if (TWQ == $\emptyset$) then
16:         adjust($\tau$)
17:     end if
18:     $\tau.S = max(\tau.S, \tau.A)$
19: end if
20: if $\tau.(A + D - S) < \tau.\mathcal{E}$ then
21:     return false
22: else
23:     $\tau._{slack} = \tau.(A + D - S - \mathcal{E})$
24:     $\tau.C = \tau.(S + \mathcal{E})$
25:     TWQ.insert($\tau$)
26:     updateSlacks($\tau$, TWQ)
27:     for ($\tau_i \in$ TWQ && $\tau_i.(A + D) > \tau.(A + D)$) do
28:         $\tau_i.C+ = \tau.\mathcal{E}$
29:     end for
30:     return true
31: end if
```

Algorithm 7 updateSlacks($\tau(A, \sigma, D)$,TWQ)

```
1: for ($\tau_i \in$ TWQ ) do
2:   if ($\tau_i.(A + D) > \tau.(A + D)$) then
3:     $\tau_{i \cdot slack} = \tau_{i \cdot slack} - \tau._{\mathcal{E}}$
4:   end if
5: end for
6: i = TWQ.length;
7: $\tau_i.slack_{min} = \tau_i.slack$
8: for (i = TWQ.length - 1; $i \geq 1$; $i - -$) do
9:   $\tau_i.slack_{min} = \min(\tau_i.slack, \tau_{i+1}.slack_{min})$
10: end for
```

To illustrate by an example, if two tasks τ_1 and τ_2 are put into TWQ, from the admission controller's point of view, they will execute one by one using all nodes of the cluster (see Figure 6.3a); in reality, they are dispatched and executed as shown in Figure 6.3b, occupying the minimum numbers of nodes needed to meet their deadline requirements.

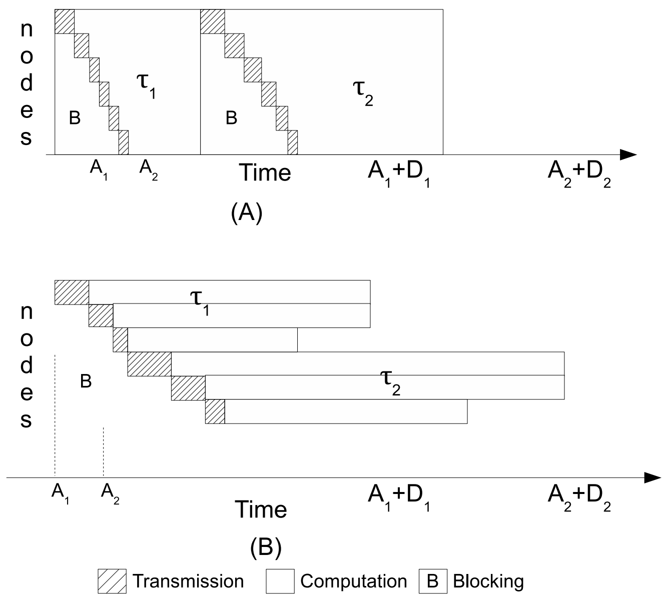


Figure 6.3: An Example Scenario (a) Admission Controller's View (b) Actual Task Execution.

6.2.3 Admission Controller Adjustment

As discussed in previous sections, the admission controller assumes a different schedule than the one adopted by the dispatcher. If TWQ is not empty, the resources are always utilized. In this case, the admission controller can make correct decisions assuming the ANA policy without detailed knowledge of the system. The admitted tasks are dispatched following the MNA policy and are always successfully completed by their deadlines. However, if TWQ is empty, some resources may be idle until the next task arrival. At that point, the admission controller has to know the system status so that it takes resource idleness into account to make correct admission control decisions.

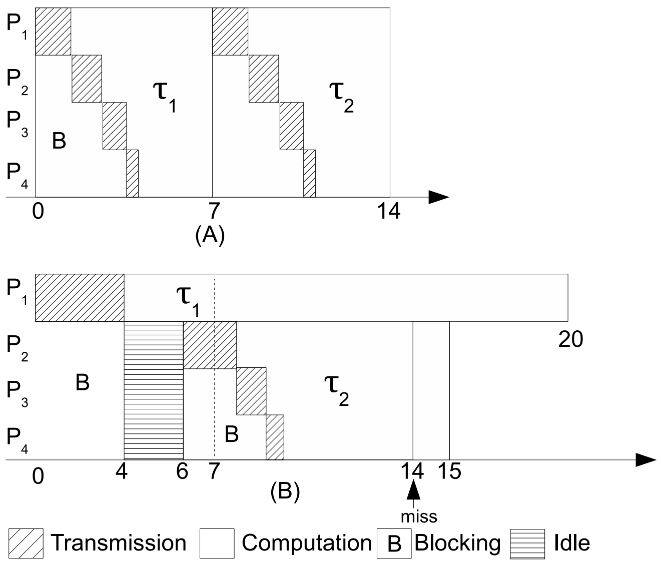


Figure 6.4: An Illustration of the Problem (a) Admission Controller's View (b) An Incorrect Task Execution.

We illustrate this problem in Figure 6.4. τ_1 arrives at time 0. The admission controller accepts it and estimates it to complete at time 7 (Figure 6.4a). However, because τ_1 has a loose deadline, the dispatcher does not allocate all four nodes but the minimum number, one node, to τ_1 and completes it at time 20 (Figure 6.4b). Task τ_2 arrives at an empty TWQ at time 6 with an absolute deadline of 14. The nodes P_2, P_3, P_4 are idle during the time interval $[4, 6]$. If the admission controller were not to consider this resource idleness, it would assume that all four nodes are busy processing τ_1 during the interval $[4, 6]$ and are available during the interval $[7, 14]$. And thus, it would wrongly conclude that τ_2 can be finished with all four nodes before its deadline. However, if τ_2 were accepted, the dispatcher cannot allocate all four nodes to τ_2 at time 6, because node P_1 is still busy processing τ_1. With just three nodes available during the interval $[6, 20]$, τ_2 cannot complete until time 15 and misses its deadline.

To solve this problem, when a new task arrives at an empty TWQ, the admission controller invokes Algorithm 8 to compute the idle time and make a proper adjustment. The algorithm first computes the workload (σ_{idle}) that could have been

Algorithm 8 adjust(τ)

```
1: TotalIdle = 0
2: for $(i = 0; i < N; i++)$ do
3:     $r = max(P_i$.AvailableTime, $P_0$.AvailableTime$)$
4:     TotalIdle += max$(A - r, 0)$
5: end for
6: $\sigma_{idle} = \frac{TotalIdle}{\tau+\chi}$
7: $w = \frac{1-\beta}{1-\beta^N}\sigma_{idle}(\tau + \chi)$
8: $\tau.S+ = w$
```

processed using the idled resources (lines 1-6). According to Eq (7.1), we know, with all N nodes, it takes $w = \frac{1-\beta}{1-\beta^N}\sigma_{idle}(\tau + \chi)$ time units to execute the workload σ_{idle} (line 7). To consider this idle time effect, the admission controller inserts an idle task of size σ_{idle} before τ and postpones τ's start time by w (line 8).

6.2.4 Correctness of the Algorithm

In this section, we prove all tasks that have been admitted by the admission controller can be dispatched successfully by the dispatcher and finished before their deadlines. For simplicity, in this section, we use A_i, σ_i, and D_i to respectively denote the arrival time, the data size, and the relative deadline of task τ_i. We prove by contradiction that no admitted task misses its deadline. Let us assume τ_m is the first task in TWQ that misses its deadline at $d_m = A_m + D_m$. We also assume that tasks $\tau_0, \tau_1, \cdots, \tau_{m-1}$ have been executed before τ_m. Among these preceding tasks, let τ_b be the latest that has arrived at an empty cluster. That is, tasks $\tau_{b+1}, \tau_{b+2}, \cdots, \tau_m$ have all arrived at times when there is at least one task executing in the cluster. Since only tasks that are assumed to finish by their deadlines are admitted, tasks execute in EDF order, and $\tau_b, \tau_{b+1}, \cdots, \tau_m$ are all admitted tasks, we know that the admission controller has assumed that all these tasks can complete by τ_m's deadline d_m. Let σ^{AN} denote the total workload that has been admitted to execute in the time interval $[A_b, d_m]$. We have,

$$\sigma^{AN} \geq \sum_{i=b}^{m} \sigma_i. \tag{6.5}$$

Since all dispatched subtasks are guaranteed to finish by their deadlines (Section 6.2.2), task τ_m missing its deadline means at time d_m a portion of τ_m is still in TWQ. That is, the total workload σ^{MN} dispatched in the time interval $[A_b, d_m]$ must be less than $\sum_{i=b}^{m} \sigma_i$. With Eq (6.5), we have,

$$\sigma^{AN} > \sigma^{MN}. \tag{6.6}$$

Next, we prove that Eq (6.6) cannot hold.

As mentioned earlier, tasks $\tau_{b+1}, \tau_{b+2}, \cdots, \tau_m$ have all arrived at times when there is at least one task executing in the cluster. However, at their arrival times, TWQ could be empty. As described in Section 6.2.3, when a task arrives at an empty TWQ, an *adjustment* function is invoked to allow the admission controller to take resource idleness into account. Following the function (Algorithm 8), the admission controller properly postpones the new task τ's start time by w, which is equivalent to the case where the admission controller "admits" and inserts before τ an idle task τ_{idle} of size σ_{idle} that completely "occupies" the idled resources present in the cluster. Let us assume that $\bar{\tau}_1, \bar{\tau}_2, \cdots, \bar{\tau}_v$ are the idle tasks "admitted" by the admission controller adjustment function to "complete" in the interval $[A_b, d_m]$.

We define $\hat{\sigma}^{AN}$ as the total workload, including those $\bar{\sigma}_i, i = 1, 2, \cdots, v$ of idle tasks, that has been admitted to execute in the time interval $[A_b, d_m]$. $\hat{\sigma}^{MN}$ is the total workload, including those $\bar{\sigma}_i, i = 1, 2, \cdots, v$ of idle tasks, that has been dispatched in the time interval $[A_b, d_m]$. Then, we have,

$$\hat{\sigma}^{AN} = \sigma^{AN} + \sum_{i=1}^{v} \bar{\sigma}_i, \tag{6.7}$$

$$\hat{\sigma}^{MN} = \sigma^{MN} + \sum_{i=1}^{v} \bar{\sigma}_i. \tag{6.8}$$

Next, we first prove that $\hat{\sigma}^{MN} \geq \hat{\sigma}^{AN}$ is true.

Computation of $\hat{\sigma}^{AN}$: $\hat{\sigma}^{AN}$ is the sum of workloads, including those $\sum_{i=1}^{v} \bar{\sigma}_i$ of idle tasks, that are admitted to execute in the time interval $[A_b, d_m]$. To compute $\hat{\sigma}^{AN}$, we leverage the following lemma.

Lemma 6.2.1 *For an admission controller that assumes the ANA policy, if h admitted tasks are merged into one task T, task T's execution time is equal to the sum of all h tasks' execution times. That is,*

$$\mathcal{E}(\sum_{i=1}^{h} \sigma_i, N) = \sum_{i=1}^{h} \mathcal{E}(\sigma_i, N). \tag{6.9}$$

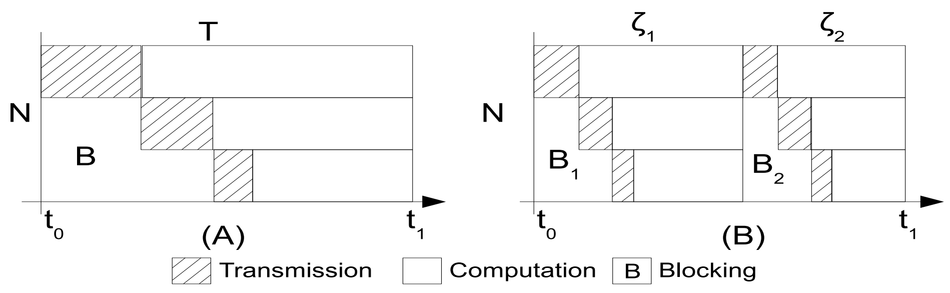


Figure 6.5: Merging Multiple Tasks into One Task.

Proof If we run a single task of size σ on N nodes, the execution time is

$$\mathcal{E}(\sigma, N) = \frac{1-\beta}{1-\beta^N}\sigma(\tau+\chi) \tag{6.10}$$

If multiple tasks of size $\sigma_1, \sigma_2, \cdots, \sigma_h$ execute on N nodes in order, their total execution time is

$$\begin{aligned}\sum_{i=1}^{h}\mathcal{E}_i(\sigma_i, N) &= \sum_{i=1}^{h}\left(\frac{1-\beta}{1-\beta^N}\sigma_i(\tau+\chi)\right) \\ &= \frac{1-\beta}{1-\beta^N}\sum_{i}^{h}\sigma_i(\tau+\chi) \end{aligned} \tag{6.11}$$

Therefore, we have,

$$\sum_{i=1}^{h}\mathcal{E}(\sigma_i, N) = \mathcal{E}(\sum_{i=1}^{h}\sigma_i, N). \tag{6.12}$$

Since $\hat{\sigma}^{AN} = \sigma^{AN} + \sum_{i=1}^{v}\bar{\sigma}_i$, according to the lemma, we have $\mathcal{E}(\hat{\sigma}^{AN}, N) = \mathcal{E}(\sigma^{AN}, N) + \sum_{i=1}^{v}\mathcal{E}(\bar{\sigma}_i, N)$, which implies that the sum of workloads $\hat{\sigma}^{AN}$ admitted to execute in the interval $[A_b, d_m]$, equals to the size of the single workload that can be processed by the N nodes in $[A_b, d_m]$. According to Eq (7.1), we have

$$\hat{\sigma}^{AN} = \frac{d_m - A_b}{\frac{1-\beta}{1-\beta^N}(\chi+\tau)}. \tag{6.13}$$

In addition, it is the sum of workloads assumed to be assigned to each of the N nodes in the interval $[A_b, d_m]$. We use σ_{p_k} to denote the workload fraction assumed to be processed by node P_k in the interval $[A_b, d_m]$. P_1 is always transmitting or computing during $[A_b, d_m]$. Therefore, the workload of node P_1 is:

$$\sigma_{p_1} = \frac{d_m - A_b}{\tau + \chi} \tag{6.14}$$

Because the data transmission does not occur in parallel, other nodes are blocked by P_1's data transmission. We use B_{p_k} to denote the blocking time on node P_k. The node P_2's workload is:

$$\sigma_{p_2} = \frac{d_m - A_b - \sigma_{p_1}\tau}{\tau + \chi} = \frac{d_m - A_b - B_{p_2}}{\tau + \chi} \tag{6.15}$$

In general, we have,

$$\sigma_{p_k} = \frac{d_m - A_b - \sum_{j=1}^{k-1} \sigma_{p_j}\tau}{\tau + \chi} = \frac{d_m - A_b - B_{p_k}}{\tau + \chi} \tag{6.16}$$

Thus, as shown in Figure 6.6, we have,

$$\hat{\sigma}^{AN} = \sum_{k=1}^{N} \sigma_{p_k}. \tag{6.17}$$

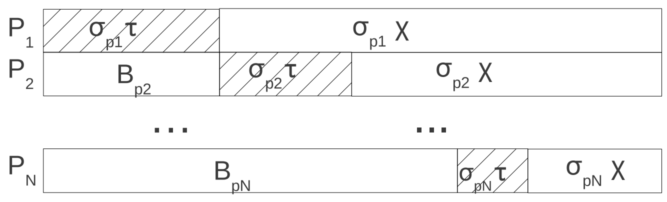


Figure 6.6: All Node Assignment Scenario.

Computation of $\hat{\sigma}^{MN}$: $\hat{\sigma}^{MN}$ denotes the total workload processed in the time interval $[A_b, d_m]$. With idle tasks $\bar{\tau}_1, \bar{\tau}_2, \cdots, \bar{\tau}_v$ completely "occupying" the idled resources during the interval $[A_b, d_m]$, there are no gaps between "task executions" and the cluster is always "busy" processing $\hat{\sigma}^{MN} = \sigma^{MN} + \sum_{i=1}^{v} \bar{\sigma}_i$. which means there are no gaps between task executions and nodes are always busy during $[t_0, t_0 + d_m]$ interval.

Unlike the admission controller, the dispatcher applies MNA policy. When a processing node becomes available, the dispatcher starts to execute a task on the node until the task's deadline. Therefore, a task is divided into subtasks, which can be dispatched to processing nodes at different times. As illustrated by an example in Figure 6.7, the σ_{31} of task 3 is dispatched to P_1 after σ_1 of task 1 finishes and the remaining workload σ_{32} of task 3 is dispatched to P_2 after σ_{22} of task 2 finishes. As we can see, MNA dispatcher leads to a complicated node allocation scenario and it makes it difficult to compute the exact value of $\hat{\sigma}^{MN}$. Therefore, we compute the lower bound of $\hat{\sigma}^{MN}$. If the lower bound of $\hat{\sigma}^{MN}$ is no less than $\hat{\sigma}^{AN}$, we prove that $\hat{\sigma}^{MN}$ is alway no less than $\hat{\sigma}^{AN}$.

Similar to computing $\hat{\sigma}^{AN}$, we calculate how much workloads are processed by each of the N nodes in the given interval. We use $\sigma^{'}_{p_k}$ to denote the sum of workloads that are processed by node P_k in the interval $[A_b, d_m]$. We have,

$$\hat{\sigma}^{MN} = \sum_{k=1}^{N} \sigma^{'}_{p_k}. \tag{6.18}$$

To compute the lower bound of $\hat{\sigma}^{MN}$, we first consider the case, where computing nodes have priorities that are indicated by their node numbers. The node P_1 has the highest priority, while P_N has the lowest priority. We also assume only high priority nodes can block low priority nodes. We use $B^{'}_{p_k}$ to denote the actual blocking due to the data transmission. In this case, since computing nodes have priorities, P_1 is never blocked in $[A_b, d_m]$. Thus the actual workload on P_1 in $[A_b, d_m]$ is:

$$\sigma'_{p_1} = \frac{d_m - A_b}{\tau + \chi} \tag{6.19}$$

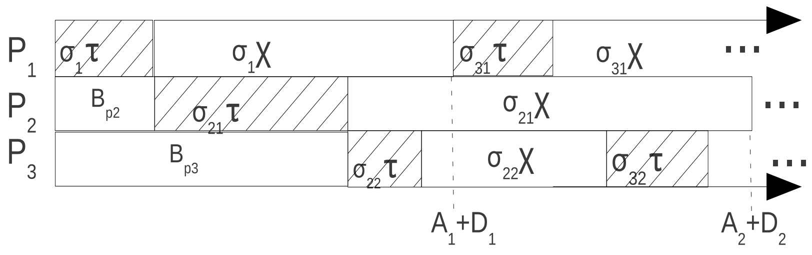


Figure 6.7: A Minimum Node Assignment Scenario.

As shown in Figure 6.7, P_1 could have multiple data transmissions. However, not all data transmissions on P_1 block the effective use of P_2. In Figure 6.7, the second data transmission on P_1 does not block and cause P_2 idle, because P_1's data transmission overlaps with P_2's computation. Therefore, the actual blocking time B'_{p2} is equal or less than the sum of data transmission time on P_1. That is:

$$B'_{p_2} \leq \sigma'_{p_1}\tau \tag{6.20}$$

Therefore,

$$\begin{aligned} \sigma'_{p_2} &= \frac{d_m - A_b - B'_{p_2}}{\tau + \chi} \\ &\geq \frac{d_m - A_b - \sigma'_{p_1}\tau}{\tau + \chi} \end{aligned} \tag{6.21}$$

In general:

$$
\begin{aligned}
B'_{p_k} &\leq \sum_{j=1}^{k-1} \sigma'_{p_j}\tau \quad k = 2, 3, \cdots, N && (6.22)\\
\sigma'_{p_k} &= \frac{d_m - A_b - B'_{p_k}}{\tau + \chi} \\
&\geq \frac{d_m - A_b - \sum_{j=1}^{k-1} \sigma'_{p_j}\tau}{\tau + \chi} && (6.23)
\end{aligned}
$$

So far, we have presented the estimated and actual workloads that are allocated on each node by the admission controller and the dispatcher. We now show that the actual dispatched workload $\hat{\sigma}^{MN}$ is always no less than the estimated workload $\hat{\sigma}^{AN}$ admitted by the admission controller.

From Equations (6.14),(6.15),(6.19), and (6.21), we have,

$$
\begin{aligned}
&\sigma'_{p_1} = \sigma_{p_1} && (6.24)\\
\text{and} \quad &\sigma'_{p_2} \geq \sigma_{p_2} && (6.25)
\end{aligned}
$$

From Eq(6.25), we can see that the actual workload that is dispatched could be more that the estimated workload on P_2. If workload on P_2 increases, it increases the blocking time of the following nodes. In general, if $\sigma'_{p_i} > \sigma_{p_i}$ for any node P_i, the increased workload $\sigma_{\Delta_i} = (\sigma'_{p_i} - \sigma_{p_i})$ increases the blocking time on the following nodes P_{i+1} to P_N by $\sigma_{\Delta_i}\tau$, as shown in Figure 6.8.

But we can show that the increased workload σ_{Δ_i} on P_i is no less than the workload that can be processed in increased blocking time $B_{\Delta_{i+1}=}\sigma_{\Delta_i}\tau$ using all nodes. Therefore, an increased workload on any node contributes to an increase of the accumulated workload $\hat{\sigma}^{MN}$.

Next, we prove this claim. If $\sigma'_{p_i} > \sigma_{p_i}$ for node P_i, then the increased blocking

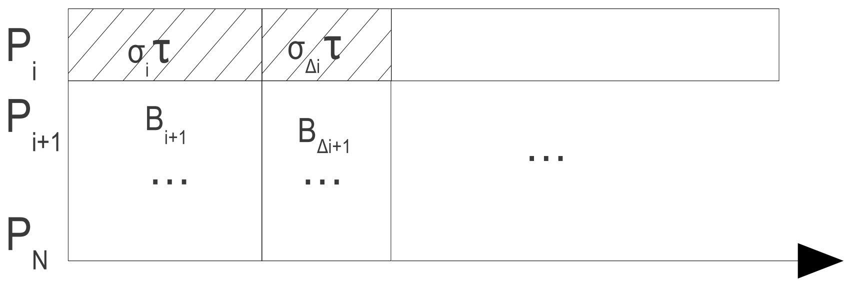


Figure 6.8: Increased Blocking Time.

time is,

$$B_{\Delta_{i+1}} = (\sigma'_{p_i} - \sigma_{p_i})\tau \tag{6.26}$$

The workload that can be processed during an interval t using N nodes is,

$$\sigma = \frac{t}{\frac{1-\beta}{1-\beta^N}(\tau + \chi)} \tag{6.27}$$

Therefore, the workload that can be precessed in $B_{\Delta_{i+1}}$ time using N nodes is:

$$\begin{aligned}
\frac{B_{\Delta_{i+1}}}{\frac{1-\beta}{1-\beta^N}(\tau + C_{ps})} &= \frac{(\sigma'_{p_i} - \sigma_{p_i})\tau}{\frac{1-\beta}{1-\beta^N}(\tau + \chi)} \\
&= \frac{(\sigma'_{p_i} - \sigma_{p_i})\tau}{\frac{\tau}{1-\beta^N}} \\
&= (\sigma'_{p_i} - \sigma_{p_i})(1-\beta^N) \\
&\leq (\sigma'_{p_i} - \sigma_{p_i})
\end{aligned}$$

That is:

$$\frac{B_{\Delta_{i+1}}}{\frac{1-\beta}{1-\beta^N}(\tau + \chi)} \leq \sigma_{\Delta_i} \tag{6.28}$$

From Eq(6.28), we can see that the increased workload σ_{Δ_2} on P_2 is no less than the workload that could be processed in $\sigma_{\Delta_2}\tau$ time units on all following nodes. Next, we prove by induction that for the first i nodes, the actual accumulated workload is no less than the estimated workload.

Base: From Equations (6.24) and (6.25), we have,

$$\sum_{k=1}^{2} \sigma'_{p_k} \geq \sum_{k=1}^{2} \sigma_{p_k} \tag{6.29}$$

We assume

$$\sum_{k=1}^{l} \sigma'_{p_k} \geq \sum_{k=1}^{l} \sigma_{p_k} \tag{6.30}$$

We use σ_l^{inc} to denote the increase of the accumulated workload on the first l nodes. That is

$$\sigma_l^{inc} = \sum_{k=1}^{l} \sigma'_{p_k} - \sum_{k=1}^{l} \sigma_{p_k} \tag{6.31}$$

σ_l^{inc} increases the blocking time on P_{l+1} by $\sigma_l^{inc}\tau$.

From Eq(6.23) we have,

$$\sigma'_{p_{(l+1)}} \geq \frac{d_m - A_b - \sum_{k=1}^{l} \sigma'_{p_k}\tau}{\tau + \chi} \tag{6.32}$$

Combining Eq(6.32) with with Eq(6.31), we have,

$$\begin{aligned} \sigma'_{p_{(l+1)}} &\geq \frac{d_m - A_b - (\sum_{k=1}^{l} \sigma_{p_k} + \sigma_l^{inc})\tau}{\tau + \chi} \\ &= \frac{d_m - A_b - \sum_{k=1}^{l} \sigma_{p_k}\tau}{\tau + \chi} - \frac{\sigma_l^{inc}\tau}{\tau + \chi} \end{aligned}$$

$$\text{That is} \qquad \sigma'_{p_{(l+1)}} \geq \sigma_{p_{(l+1)}} - \frac{\sigma_l^{inc}\tau}{\tau + \chi} \tag{6.33}$$

For the first $(l+1)$ nodes:

$$\sum_{k=1}^{l+1} \sigma'_{p_k} = \sum_{k=1}^{l} \sigma'_{p_k} + \sigma'_{p_{(l+1)}}$$

Replace $\sum_{k=1}^{l} \sigma'_{p_k}$ with Eq(6.31), we have,

$$\sum_{k=1}^{l+1} \sigma'_{p_k} = \sum_{k=1}^{l} \sigma_{p_k} + \sigma_l^{inc} + \sigma'_{p_{(l+1)}} \tag{6.34}$$

Replace $\sigma'_{p_{l+1}}$ with Eq(6.33), we get:

$$\begin{aligned}
\sum_{k=1}^{l+1} \sigma'_{p_k} &\geq \sum_{k=1}^{l} \sigma_{p_k} + \sigma_l^{inc} + \sigma_{p_{(l+1)}} - \sigma_l^{inc} \frac{\tau}{\tau + \chi} \\
&= \sum_{k=1}^{l+1} \sigma_{p_k} + \sigma_l^{inc} - \sigma_l^{inc} \frac{\tau}{\tau + \chi} \\
&= \sum_{k=1}^{l+1} \sigma_{p_k} + \sigma_l^{inc} (1 - \frac{\tau}{\tau + \chi}) \\
&= \sum_{k=1}^{l+1} \sigma_{p_k} + \sigma_l^{inc} \beta \\
&\geq \sum_{k=1}^{l+1} \sigma_{p_k} \qquad (6.35)
\end{aligned}$$

That is:

$$\sum_{k=1}^{l+1} \sigma'_{p_k} \geq \sum_{k=1}^{l+1} \sigma_{p_k} \tag{6.36}$$

Eq(6.36) shows that the actual accumulated workload is no less than the estimated workload. Thus, $\forall\ l \in [0, N]$ we have,

$$\begin{aligned}
& \sum_{k=1}^{l} \sigma'_{p_k} \geq \sum_{k=1}^{l} \sigma_{p_k} \qquad (6.37) \\
\Rightarrow \quad & \hat{\sigma}^{MN} \geq \hat{\sigma}^{AN} \qquad (6.38)
\end{aligned}$$

We proved that if computing nodes have priorities, the workload that is dispatched in $[A_b, d_m]$ is no less than the estimated workload. In next step, we relax the node priority constraint. Without priority, workloads can be dispatched to any available node, such that high index node can block low index nodes. As an example shown in Figure 6.9-(A), data transmission $\sigma_1\tau$ on P_1 blocks P_2, denoted as B_2'. The dispatcher starts dispatching σ_{21} to P_2 immediately after σ_1's data transmission. When P_1 completes processing σ_1, it is blocked by P_2 until σ_{21}'s data transmission completes. This blocking is denoted as B_1'. For this case, it is difficult to derive the workload processed by each node. because a node can be blocked by any other nodes. But we can show that no-priority, mixed blocking case can be reduced to a case, where the priority is enforced.

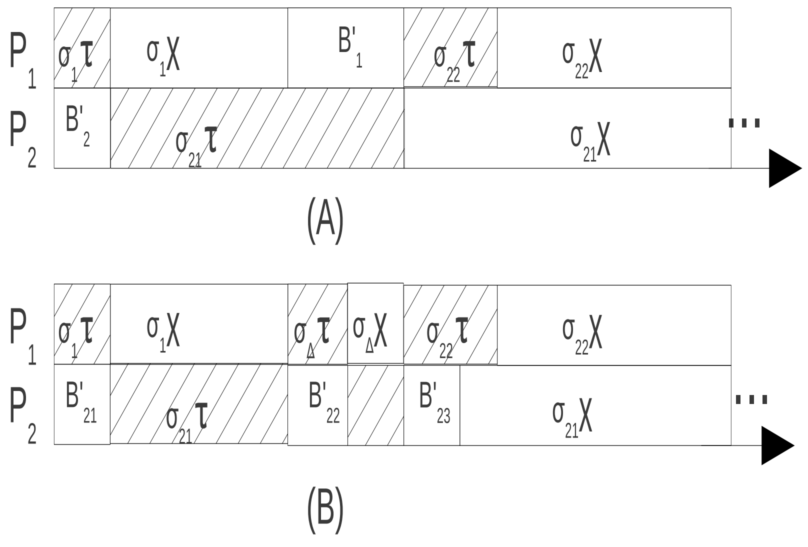


Figure 6.9: Another MNA Scenario.

Assume a low index node can be blocked by a high index node. Without loss of generality, we assume node P_1 is blocked by node P_2 in B_1'. If we remove $\sigma_\Delta = \frac{B_1'}{\tau+\chi}$ workload from P_2 and assume that the workload were assigned to P_1, as shown in Figure 6.9-(B). This workload can be processed in B_1' time. The σ_Δ on P_1 increases the blocking time on P_2 by $\sigma_\Delta\tau$, denoted as B_{22}', and reduces the computation time on P_2 by $\sigma_\Delta\chi$, denoted as B_{23}', which corresponds to the removed workload. Therefore, $B_1' = B_{22}' + B_{23}'$. This way as shown in Figure 6.9-(B), we can reverse the blocking time order without changing the blocking amount. Next, we justify the existence of the blocking time B'_{23}, showing that after the conversion, the blocking time on node P_2 is still no more that the sum of data transmission time on node P_1. In Figure 6.9-(A),

$$\begin{aligned} B'_1 &= \sigma_\Delta(\tau + \chi) \leq \sigma_{21}\tau && (6.39) \\ \sigma_{21}\chi &= \sigma_{22}(\tau + \chi) && (6.40) \end{aligned}$$

Multiply both sides of Eq (6.40) by τ/χ, we have,

$$\sigma_{21}\tau = \sigma_{22}(\tau + \chi)\frac{\tau}{\chi} \tag{6.41}$$

From Eq(6.39) and Eq(6.41), we have,

$$\sigma_\Delta(\tau + \chi) \leq \sigma_{22}(\tau + \chi)\frac{\tau}{\chi} \tag{6.42}$$

Multiply both side of Eq(6.42) by $\chi/(\tau + \chi)$, we have,

$$\begin{aligned} \sigma_\Delta\chi &\leq \sigma_{22}\tau && (6.43) \\ \text{i.e.,} \qquad B'_{23} &\leq \sigma_{22}\tau && (6.44) \end{aligned}$$

In the converted case, $B'_{21} + B'_{22} + B'_{23} \leq (\sigma_1 + \sigma_\Delta + \sigma_{22})\tau$. That is the total blocking time on P_2 is no more than the sum of data transmission time on P_1. This conforms to a scenario in the priority enforced case. Same method can be applied to the multiple node scenario, where the mixed blocking time can be reversed among two nodes in each step until the we reach the previous case.

Therefore, the no priority case can be reduced to the priority enforced case. Thus for both cases, we can conclude:

$$\sum_{j=1}^{N} \sigma'_{p_j} \geq \sum_{j=1}^{N} \sigma_{p_j} \tag{6.45}$$

$$\hat{\sigma}^{MN} \geq \hat{\sigma}^{AN} \tag{6.46}$$

With Equations (6.46), (6.7), and (6.8), we conclude that $\sigma^{MN} \geq \sigma^{AN}$ is true, which contradicts Eq (6.6). Therefore, the original assumption does not hold and no task misses its deadline.

6.3 Hybrid Approach

In Section 6.2.4, we have showed that although ANA admission controller is fast, it is a little pessimistic in comparison to the exact MNA admission controller. In this section, we introduce a hybrid approach that combines the best of both strategies. To avoid unnecessary task rejections, when the waiting queue length is small and the overhead is little, the hybrid admission controller applies the exact MNA policy. If the waiting queue length is large and the scheduling overhead becomes non-negligible, it switches to the fast ANA admission controller. By dynamically switching between MNA and ANA-based strategies, the hybrid approach can provide both efficiency and better real-time performance.

Next, we describe the hybrid admission control algorithm (Algorithm 9) in detail. According to a pre-specified queue length threshold, the hybrid approach switches between the fast admission control algorithm AC-FAST (Algorithm 6) and the exact MNA-based admission controller as adopted by EDF-IIT [17, 45] (line 1). When switching to AC-FAST, since the currently-running tasks' remaining workload will delay the start time of waiting tasks, this delay must be calculated (lines 2-3). We then compute the expected execution time, completion time and slack time of all waiting tasks (line 4). This computation is only needed at the switch point (lines 5-6). In addition, because AC-FAST is only applied when the TWQ is large and thus the cluster is busy, there is no need to call the "adjust" function (Algorithm 8) in this hybrid algorithm (lines 7-8). To switch from AC-FAST to EDF-IIT, the schedule of all waiting tasks must be calculated based on the current resource availability (line 9-11). Since this schedule recalculation occurs only if the queue length is small, the resultant scheduling overhead is tolerable.

Algorithm 9 HybridAC(τ,TWQ)

```
1: if (Queue_Length >= Switch_Threshold) then
2:    if (FLAG) then
3:       r = RunningTaskExeTime()
4:       AdjANACompTime(r)
5:       FLAG = false
6:    end if
7:    AC-FAST($\tau$, TWQ)
8: else
9:    EDF-IIT($\tau$, TWQ)
10:   FLAG=true
11: end if
```

Algorithm 10 RunningTaskExeTime()

```
1: r = 0 //current time to node end time
2: for ($n_i \in$ cluster) do
3:    if $n_i.EndTime > curTime$ then
4:       r += $n_i.EndTime - curTime$
5:    end if
6: end for
7: $\sigma_r = \frac{r}{\chi}$ //running workload
8: $\mathcal{E}_r = \mathcal{E}(\sigma_r, N)$
9: return $\mathcal{E}_r$
```

Algorithm 11 AdjANACompTime(delayTime)

```
1: C = curTime + delayTime
2: //update tasks' execution and completion time
3: for ($\tau_i \in$ TWQ) do
4:     $\tau_i.\mathcal{E} = \mathcal{E}(\tau_i.\sigma, N)$
5:     $\tau_i.C = C + \tau_i.\mathcal{E}$
6:     $C = \tau_i.C$
7: end for
8: //compute slack time of all tasks
9: for ($\tau_i \in$ TWQ) do
10:    $\tau_i.slack = \tau_i.(A + D) - \tau_i.C$
11: end for
```

6.4 Evaluation

In Section 6.2, we have presented an efficient divisible load scheduling algorithm. Since the algorithm is based on EDF scheduling and it eliminates IITs, we use FAST-EDF-IIT to denote it. The EDF-based algorithm proposed in [46] is represented by EDF-IIT-1 and that in [17] by EDF-IIT-2. We use HYBRID to denote the hybrid algorithm introduced in Section 6.3. This section compares their performance.

We have developed a discrete simulator, called DLSim, to simulate real-time divisible load scheduling in clusters. This simulator, implemented in Java, is a component-based tool, where the main components include a workload generator, a cluster configuration component, a real-time scheduler, and a logging component. For every simulation, three parameters, N, τ and χ are specified for a cluster.

In Sections 6.4.1 and 6.4.2, we evaluate the performance of FAST-EDF-IIT. In Section 6.4.3, we evaluate the performance of the hybrid algorithm HYBRID.

6.4.1 FAST-EDF-IIT: Real-Time Performance

We first evaluate the algorithm's real-time performance. The workload is generated following the same approach as described in [46, 45] and due to the space limitation, we choose not to repeat the details here. Similar to the work by Lee et al. [43], we adopt a metric $SystemLoad = \mathcal{E}(Avg\sigma, 1)\frac{\lambda}{N}$ to represent how loaded a cluster is for a simulation, where $Avg\sigma$ is the average task data size, $\mathcal{E}(Avg\sigma, 1)$ is the execution time of running an average size task on a single node (see Eq (7.1) for $\mathcal{E}$'s calculation), and $\frac{\lambda}{N}$ is the average task arrival rate per node. To evaluate the real-time performance, we use two metrics — *Task Reject Ratio* and *System Utilization*. Task reject ratio is the ratio of the number of task rejections to the number of task arrivals. The smaller the ratio, the better the performance. In contrast, the greater the system utilization, the better the performance.

For simulations in this subsection, we assume that the cluster is lightly loaded and thus we can ignore the scheduling overheads. In these simulations, we observe that all admitted tasks complete successfully by their deadlines. Figure 6.10 illustrates the algorithm's *Task Reject Ratio* and *System Utilization*. As we can see, among the three algorithms, EDF-IIT-2 provides the best real-time performance, achieving the least *Task Reject Ratio* and the highest *System Utilization*, while FAST-EDF-IIT performs better than EDF-IIT-1. The reason that FAST-EDF-IIT does not have the best real-time performance is due to its admission controller's slightly pessimistic estimates of the data transmission blocking time (Section 6.2). Focusing on reducing the scheduling overhead, FAST-EDF-IIT trades real-time performance for algorithm efficiency. In the next subsection, we use experimental data to demonstrate that in busy clusters with long task waiting queues, scheduling overheads become significant and inefficient algorithms like EDF-IIT-1 and EDF-IIT-2 can no longer be applied, while FAST-EDF-IIT wins for its huge advantages in scheduling efficiency.

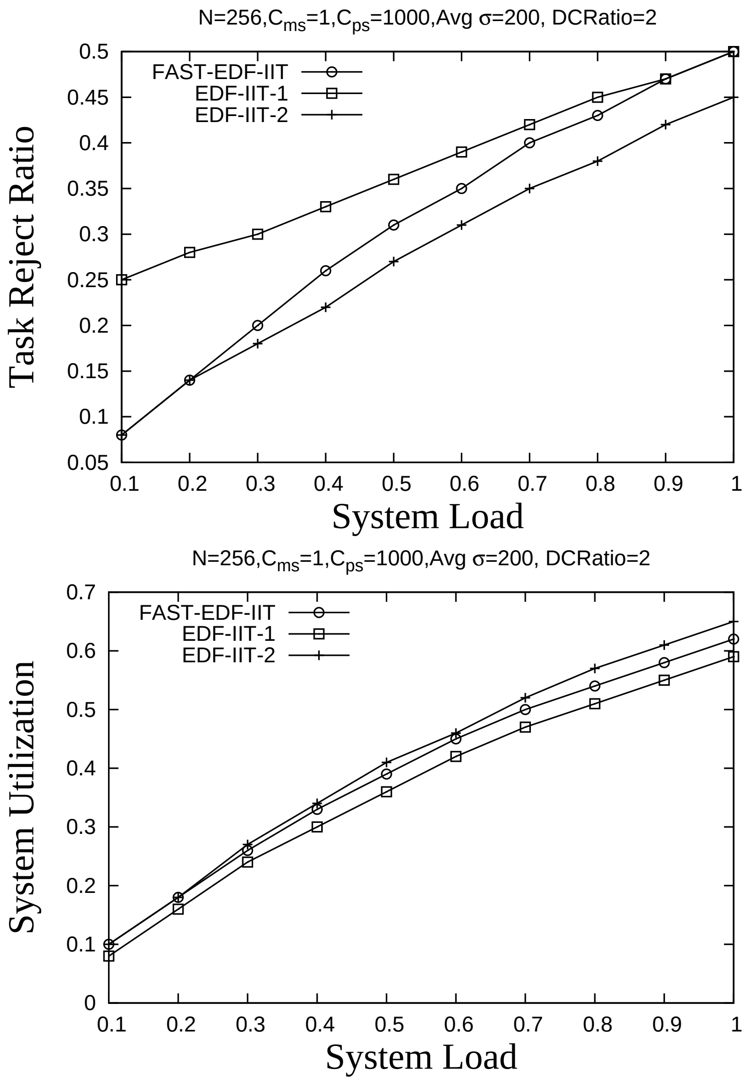
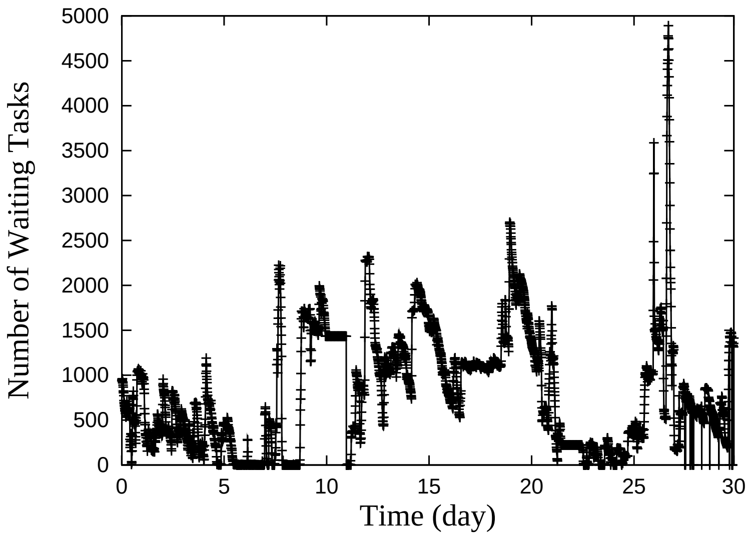


Figure 6.10: Algorithm's Real-Time Performance.

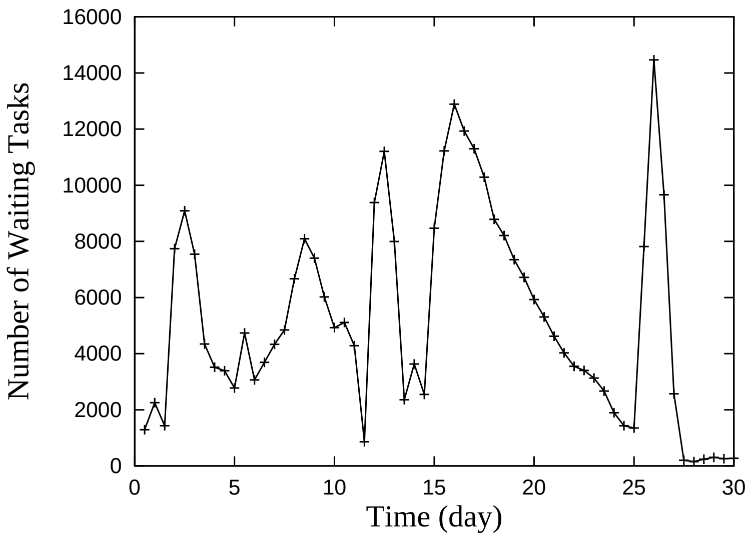
6.4.2 FAST-EDF-IIT: Scheduling Overhead

A second group of simulations are carried out to evaluate the overhead of the scheduling algorithms. Before discussing the simulations, we first present some typical cluster workloads, which lay out the rationale for our simulations.

In Figure 6.2, we have shown the TWQ status of a cluster at University of California, San Diego. From the curves, we observe that 1) waiting tasks could increase from $3,000$ to $17,000$ in one hour (Figure 6.2a) and increase from $15,000$ to $25,000$ in about three hours (Figure 6.2b) and 2) during busy hours, there could be on average more than $5,000$ and a maximum of $37,000$ tasks waiting in a cluster. Similarly busy and bursty workloads have also been observed in other clusters (Figure 6.11) and are quite common phenomena.[1] Based on these typical workload patterns, we design our simulations and evaluate the algorithm's scheduling overhead.



(a) Red Cluster at Univ. of Nebraska - Lincoln



(b) GLOW Cluster at Univ. of Wisconsin

Figure 6.11: Typical Cluster Status.

[1]To illustrate the intensity and commonness of the phenomena, Figures 6.2 and 6.11 show the TWQ statistics on an hourly and a daily basis respectively.

In this group of simulations, the following parameters are set for the cluster: N=512 or 1024, τ=1 and χ=1000. have thousands of CPUs. We choose to simulate modest-size clusters (i.e., those with 512 or 1024 nodes). According to our analysis, the time complexities of algorithms FAST-EDF-IIT, EDF-IIT-1 and EDF-IIT-2 are respectively $O(\max{(N, n)})$, $O(nN^3)$ and $O(nNlog(N))$. Therefore, if we show by simulation data that in modest-size clusters of N=512 or 1024 nodes FAST-EDF-IIT leads to much less overheads, then we know for sure that it will be even more *advantageous* if we apply it in larger clusters like those listed in Table 6.1.

To create cases where we have a large number of tasks in TWQ, we first submit a huge task to the cluster. Since it takes the cluster a long time to finish processing this one task, we can submit thousands of other tasks and get them queued up in TWQ. As new tasks arrive, the TWQ length is built up. In order to control the number of waiting tasks and create the same TWQ lengths for the three scheduling algorithms, tasks are assigned long deadlines so that they will all be admitted and put into TWQ. That is, in this group of simulations, we force task reject ratios to be 0 for all three algorithms so that the measured scheduling overheads of the three are comparable.

We first measure the average scheduling time of the first n tasks, where n is in the range [100, 3000]. The simulation results for the 512-node cluster are shown in Table 6.2 and Figure 6.12. From the data, we can see that for the first $3,000$ tasks, FAST-EDF-IIT spends an average of 48.87ms to admit a task, while EDF-IIT-1 and EDF-IIT-2 average respectively 6206.91ms and 1494.91ms, 127 and 30 times longer than FAST-EDF-IIT.

Table 6.2: 512-Node Cluster: First n Tasks' Average Scheduling Time (ms).

n	FAST-EDF-IIT	EDF-IIT-1	EDF-IIT-2
300	0.96	410.44	151.32
1000	4.84	1321.08	494.07
2000	20.46	3119.76	988.95
3000	48.87	6206.91	1494.91

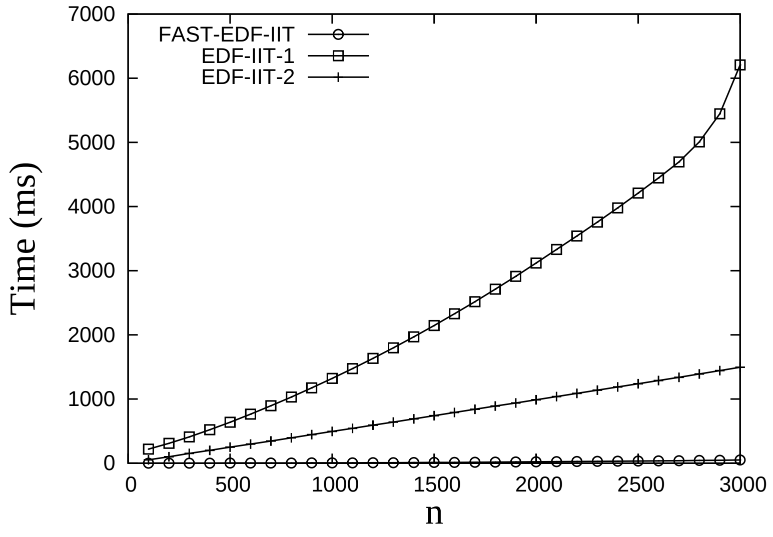


Figure 6.12: 512-Node Cluster: Algorithm's Real-Time Scheduling Overhead: First n Tasks' Average Scheduling Time.

Because the scheduling overhead increases with the number of tasks in TWQ, we then measure the task scheduling time *after* n tasks are queued up in TWQ. Table 6.3 shows the average scheduling time of 100 new tasks after there are already n tasks in TWQ of the 512-node cluster. The corresponding curves are in Figure 6.13. As shown, when there are $3,000$ waiting tasks, FAST-EDF-IIT takes 157ms to admit a task, while EDF-IIT-1 and EDF-IIT-2 respectively spend about 31 and 3 seconds to make an admission control decision.

Table 6.3: 512-Node Cluster: Average Task Scheduling Time (ms) after n Tasks in TWQ.

n	FAST-EDF-IIT	EDF-IIT-1	EDF-IIT-2
300	1.71	850.01	349.22
1000	16.25	3006.01	1034.21
2000	67.24	7536.32	2030.48
3000	157	31173.86	3050.86

Now, let us examine the simulation results and analyze their implication for real-world clusters. It is shown in Figure 6.2a that the length of TWQ in a cluster could increase from $3,000$ to $17,000$ in an hour. users before their submissions, we

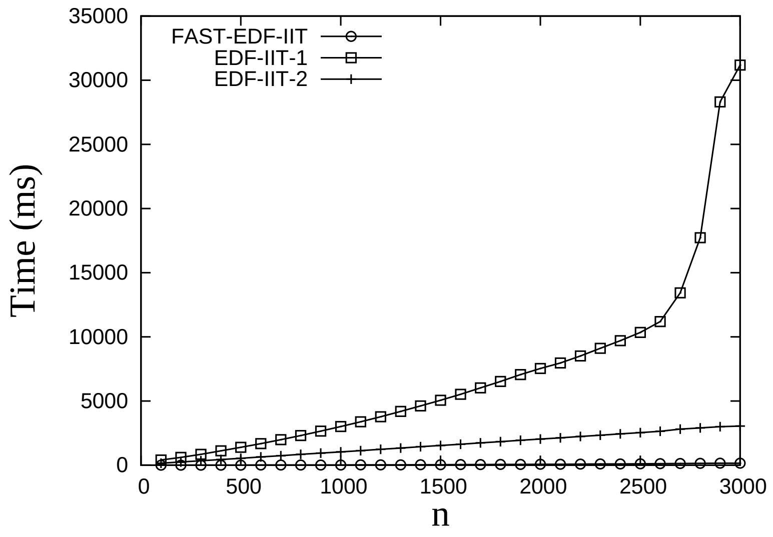


Figure 6.13: 512-Node Cluster: Algorithm's Real-Time Scheduling Overhead: Average Scheduling Time after n Tasks in TWQ.

actually have a smaller number of "whole" tasks waiting in the queue. By studying a 16-month-long log of the RED cluster at University of Nebraska-Lincoln [2], we find that the average number of tasks submitted by a user consecutively is 7. Based on this data, we believe it is reasonable to assume that a "whole" task comprises of 10 subtasks. Therefore, a TWQ length growing from $3,000$ to $17,000$ in an hour means that the number of submitted "whole" tasks increases from 300 to $1,700$ in an hour. From Table 6.3, we know that for EDF-IIT-1 and EDF-IIT-2, it takes respectively more than 31 and 3 seconds to admit a task when the TWQ length is over 3,000. Therefore, to schedule the $14,000$ new tasks arrived in that hour, it takes more than 7,000 and 700 minutes respectively. Even if we assume that the last one of the $14,000$ tasks has arrived in the last minute of the hour, its user has to wait for at least 700-60=640 minutes to know if the task is admitted or not. On the other hand, if FAST-EDF-IIT is applied, it takes a total of 37 minutes to make admission control decisions on the $14,000$ tasks. This example demonstrates that our new algorithm is much more efficient than existing approaches and is the only algorithm that can

[2]Red is a 215 node/1140 core production-mode LINUX cluster.

be applied in busy clusters. If we analyze the algorithms using data in Figure 6.2b where waiting tasks increase from $15,000$ to $25,000$, the difference in scheduling time will be even more striking.

Table 6.4: First n Tasks' Average Scheduling Time (ms).

n	FAST-EDF-IIT		EDF-IIT-2	
	N=1024	N=512	N=1024	N=512
300	1.01	0.96	363.29	151.32
1000	4.90	4.84	1545.51	494.07
2000	21.1	20.46	3089.6	988.95
3000	50	48.87	4923.91	1494.91

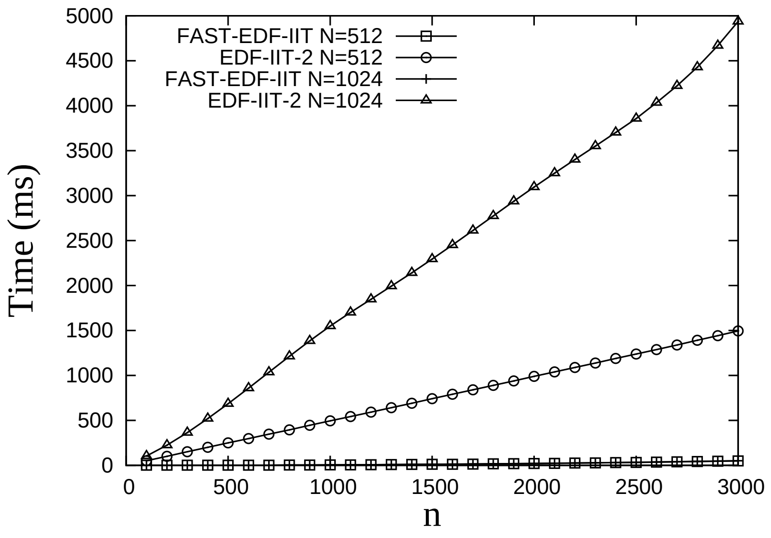


Figure 6.14: Algorithm's Real-Time Scheduling Overhead: First n Tasks' Average Scheduling Time.

The simulation results for the 1024-node cluster are reported in Table 6.4 and Figure 6.14. Due to EDF-IIT-1's huge overhead and cubic complexity with respect to the number of nodes in the cluster, a simulation for a busy cluster with a thousand nodes would take weeks — with no new knowledge to be learned from the experiment. Therefore, on the 1024-node cluster, we only simulate EDF-IIT-2 and FAST-EDF-IIT. For easy comparison, Table 6.4 and Figure 6.14 include not only data for the 1024-node cluster but also those for the 512-node cluster. As shown by the simulation results, when the cluster size increases from 512 to 1024 nodes, the scheduling overhead of FAST-EDF-IIT only increases slightly. FAST-EDF-IIT has a time complexity of $O(\max(N, n))$. Therefore, for busy clusters with thousands of tasks in TWQ (i.e., n in the range [3000, 17000]), the cluster size increase does not lead to a big increase of FAST-EDF-IIT's overhead. In contrast, EDF-IIT-2, with a time complexity of $O(nNlog(N))$, has a much larger scheduling overhead on the 1024-node cluster than that on the 512-node cluster.

6.4.3 Evaluation of Hybrid Approach

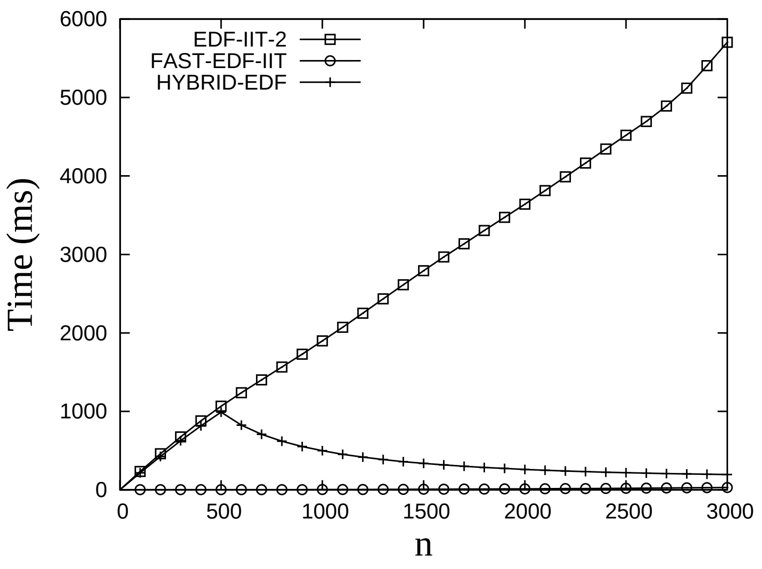


Figure 6.15: 1024-Node Cluster: Hybrid Algorithm's Overhead: First n tasks' Average Scheduling Time.

Scheduling Overhead. In the following group of simulations, we analyze the scheduling overhead of the Hybrid algorithm and compare it with FAST-EDF-IIT and EDT-IIT-2. The simulation results for the 1024-node cluster are shown in Figures 6.15 and 6.16. For this set of simulations, the switch point of the Hybrid approach is set at 500 tasks. The cluster admission controller is based on EDF-IIT-2 when the queue length is smaller than 500, and switches to AC-FAST (Algorithm 6) when the

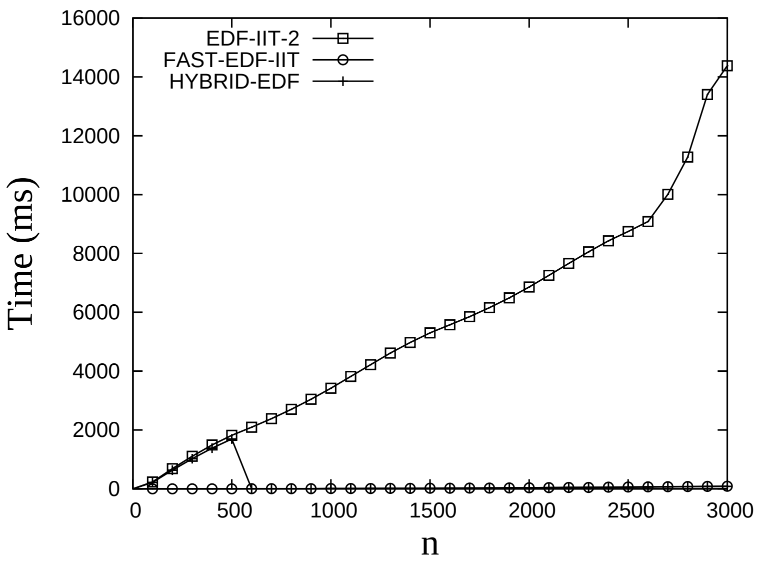
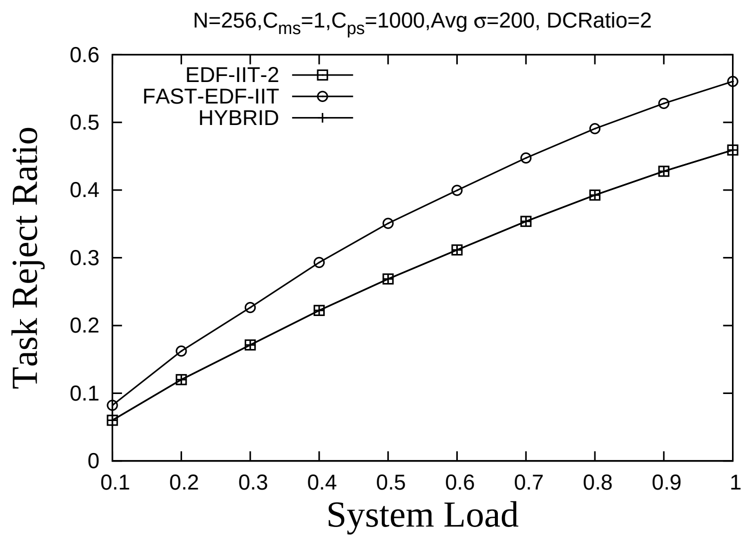


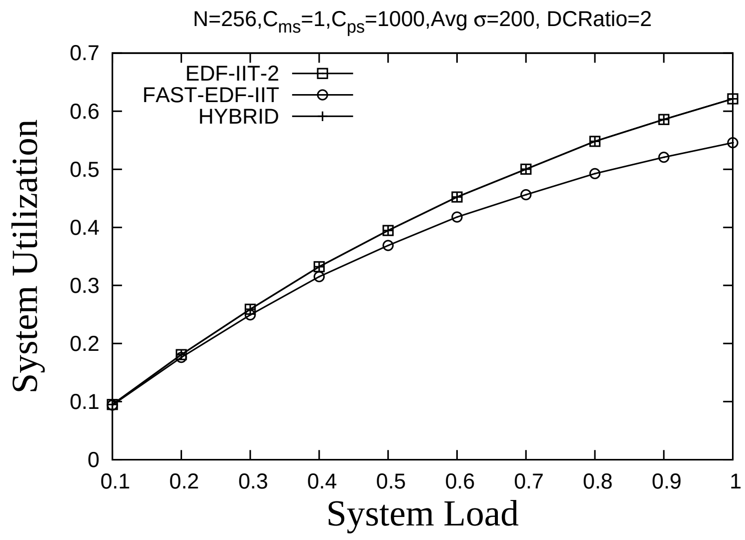
Figure 6.16: 1024-Node Cluster: Hybrid Algorithm's Overhead: Average Scheduling Time after n Tasks in TWQ.

queue length goes beyond 500. We measure both the average scheduling time of first n tasks and the scheduling time after n tasks in TWQ. As shown in Figure 6.15, as the number of tasks increases in TWQ, the average scheduling time of EDF-IIT-2 gets bigger while that of FAST-EDF-IIT remains small. Before the switch point is reached, the HYBRID algorithm's overhead increases with the number of waiting tasks; but afterwards, the HYBRID algorithm switches to applying AC-FAST and its overhead slowly converges to be the same as that of FAST-EDF-IIT. Figure 6.16 shows the scheduling time after n tasks in TWQ. The scheduling overhead of the HYBRID approach decreases immediately after it switches to AC-FAST.

Real-Time Performance. We next examine the HYBRID algorithm's real-time performance. The simulation configuration is the same as that in Section 6.4.1. We first compare the performance of EDF-IIT-2, FAST-EDF-IIT and HYBRID with a workload similar to that adopted in Section 6.4.1. Focusing on testing the real-time performance, this workload does not generate a large TWQ. As shown in Figure 6.17, the HYBRID approach performs the same as EDF-IIT-2, since with this workload



(a)

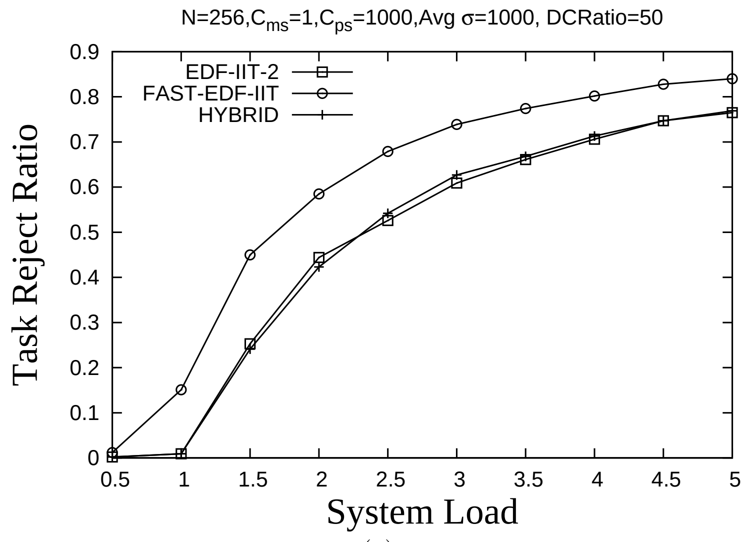


(b)

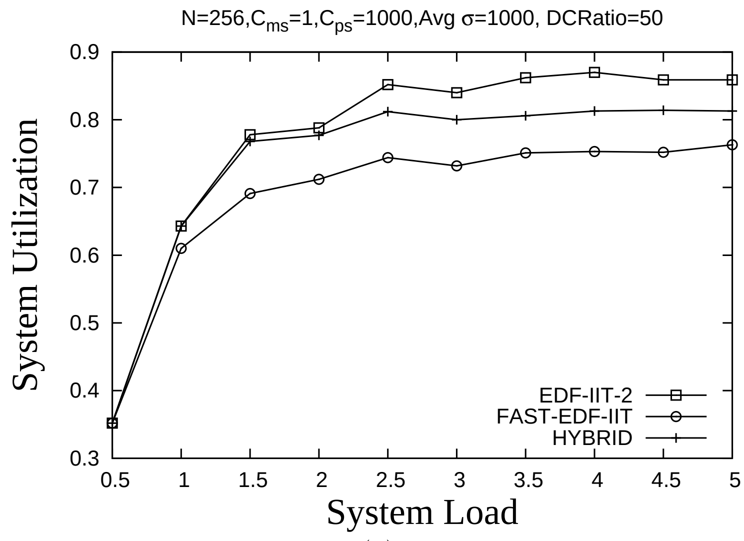
Figure 6.17: Hybrid Algorithm's Real-Time Performance with Small TWQ.

the TWQ is very small and as a result the HYBRID admission controller is essentially based on EDF-IIT-2.

In the next group of simulations, we analyze the HYBRID algorithm's performance with a longer TWQ. To generate a longer TWQ, we made two changes to the configuration. First, we increased DCRatio to allow tasks to wait longer in TWQ without missing their deadlines. Second, we increased the system workload five times



(a)



(b)

Figure 6.18: Hybrid Algorithm's Real-Time Performance.

to force tasks to wait longer in TWQ. For these simulations, we set the HYBRID algorithm's switch point at 50 waiting tasks. Since the overhead is not the focus of this evaluation, for this group of simulations we still assume that the scheduling overhead is negligible and not big enough to affect the algorithms' real-time performance. The simulation results are shown in Figure 6.18.

As we can see from Figure 6.18a, both HYBRID and EDF-IIT-2 algorithms achieve better task reject ratios than FAST-EDF-IIT while the two algorithms perform similarly to each other. This is because HYBRID algorithm is based on EDF-IIT-2 when TWQ is small and only after TWQ size increases and reaches to the switch point does the HYBRID algorithm switch to applying the slightly pessimistic AC-FAST.

For the heavy system workload, TWQ is never empty. It implies that the cluster could always be busy or 100% utilized. However, when computing nodes are waiting for data or blocked by other nodes' data transmission, they are not counted as utilized. Therefore, system utilization cannot really reach 100% but saturates around some value. As we can see from Figure 6.18b, as the system workload goes beyond 1, the system utilization starts to saturate.

In this subsection, we have used simulations to prove the feasibility of the hybrid approach that combines advantages of FAST-END-IIT and EDF-IIT-2. To use the hybrid algorithm in a real-world cluster, an administrator could set the algorithm's switch point at a proper level according to the user-tolerable overhead.

6.5 Summary

This chapter presents a novel algorithm for scheduling real-time divisible loads in clusters. The algorithm assumes a different scheduling rule in the admission controller than that adopted by the dispatcher. Since the admission controller no longer generates an exact schedule, the scheduling overhead is reduced significantly. Unlike the previous approaches, where time complexities are $O(nN^3)$ [46] and $O(nNlog(N))$ [17], our new algorithm has a linear time complexity, i.e. $O(\max(N, n))$. We prove that the proposed algorithm is correct, provides admitted tasks real-time guarantees, and utilizes cluster resources efficiently. We also propose a hybrid admission control algorithm that combines advantages of both the proposed fast admission control algorithm and the previous approaches proposed in [17, 45]. We experimentally compare our algorithm with existing approaches. Simulation results demonstrate that the proposed algorithm scales well and can schedule large numbers of tasks efficiently. With growing cluster sizes and number of taks submitted, we expect our algorithm to be even more advantageous.

Chapter 7

Feedback-Control Based Real-Time Divisible Load Scheduling

7.1 Introduction

Existing real-time divisible load scheduling approaches are based on worst-case parameters or assuming that task execution times can be accurately known or derived in advance. In addition, the cluster is assumed to be 100% reliable. In practice, node failures occur frequently and the failure rate increases with the cluster size. For existing "open-loop" scheduling algorithms, schedules once created, are not adjusted according to the current system status. While they function well in predictable environments, their performance in open and dynamic environments could be unacceptably poor. We, therefore, need a feedback-control based approach to dynamically handle workload and system variances and maintain desired performance.

In this chapter, we propose a novel algorithm, which integrates feedback control and real-time divisible load scheduling. Next, in Section 7.2, we discuss the real-time scheduling algorithm and Section 7.3 evaluates the algorithm performance.

7.2 Algorithm

In this section, we present our feedback-control based algorithm on real-time divisible load scheduling. We adopt the regular task model in Section 3.1.1, system model and notations in Section 3.2. The objective of the algorithm is to provide a guarantee on low deadline miss ratio while maintaining a high system utilization. The deadline miss ratio is defined as the percentage of accepted tasks that miss their deadlines. The system utilization is the ratio of processors' busy time to processors' available time. A processor is considered busy when it is transmitting or computing a task. Previous work uses pessimistic approaches, which avoid deadline misses by extremely pessimistic estimation of task execution time, resulting in low system utilization. Our approach is optimistic, where we allow non-zero but low deadline miss ratio to trade for high system utilization.

In our control structure, the deadline miss ratio $MR(k)$ is the *controlled variable*. We choose a positive value (for instance, 5%) as the miss ratio *set point* M_s, i.e., the targeted miss ratio. The goal is to guarantee $MR(k) = M_s$. Note that 0% cannot be chosen as the *set point* because to reach 0% miss ratio, the controller could reject all tasks and make the system completely idle, which is a correct but undesirable state. Therefore, the *set point* is chosen to be greater than zero so that the scheduler can slightly overload the system to ensure high utilization. In the control structure, the system utilization is the *manipulated variable*. Following the control input (i.e., the controller output), the *actuator* sets a utilization bound for the task scheduler, which then regulates the system utilization accordingly. Our task scheduler includes two major components: the admission controller and the dispatcher.

Figure 7.1 shows the feedback-control architecture, which has a PI controller, a deadline miss ratio monitor, an admission controller and a dispatcher. The control loop is invoked once in every T_s sampling period, where 1) the monitor sends the

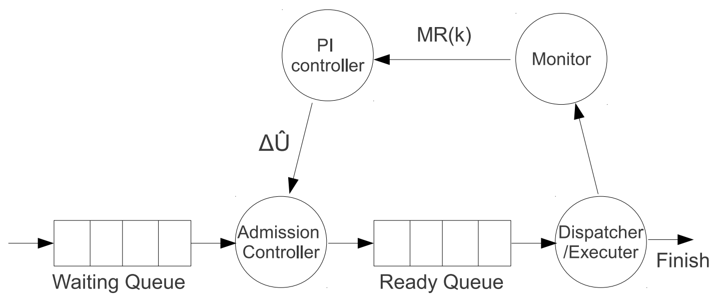


Figure 7.1: Feedback-Control Architecture.

measured deadline miss ratio $MR(k)$ to the controller; 2) based on the error between $MR(k)$ and its set point M_s, the controller computes the change $\Delta\hat{U}$ of the utilization bound; and 3) as a result, a new utilization bound $\hat{U}(k+1) = \hat{U}(k) + \Delta\hat{U}$ is generated and passed to the admission controller, which accordingly controls task admission and system utilization in the next period. The dispatcher dispatches and executes admitted tasks.

Next, we present the details of the admission controller, the dispatcher, and the design of the PI controller on the utilization bound.

7.2.1 Admission Controller

When a new task arrives, the admission controller determines if it is feasible to schedule the new task without compromising the guarantees for previously admitted tasks. In this chapter, it is to provide the $x\%$, where $x > 0$, deadline miss ratio guarantee. Unlike existing approaches which make the decision based on worst-case estimate of task parameters, our admission control algorithm follows the guide of the feedback controller, i.e., the system utilization bound $\hat{U}(k)$.

Algorithm 12 provides the admission control pseudo code. We use $\mathcal{E}$ and C to respectively denote the estimated task execution time and completion time. Here, estimates are *not* based on the worst-case. Instead, the estimated task parameters

Algorithm 12 AdmissionTest($\tau(A, \sigma, D)$, TWQ)

```
1:  TWQ.insert($\tau$) // following EDF order
2:  $\tau_p$ = getLastDispatchedTask()
3:  **if** ($\tau_p$ != null) **then**
4:      $S = \max(\tau_p.C, \tau.A)$
5:  **else**
6:      $S = \tau.A$
7:  **end if**
8:  **for** ($\tau_i \in$ TWQ) **do**
9:      $\tau_i.\mathcal{E} = \mathcal{E}(\tau_i.\sigma, N)$
10:     U $= \tau_i.\mathcal{E}/(\tau_i.A + \tau_i.D - S)$
11:     **if** ($U > \hat{U}(k)$) **then**
12:         **return false**
13:     **end if**
14:     S=S+$\tau_i.\mathcal{E}$
15: **end for**
16: **return true**
```

can be smaller, equal or greater than the actual values. In the admission controller, for easy estimation, it is assumed that an accepted task will execute in parallel on all N processing nodes of the cluster. A task is partitioned following the divisible load theory, which states that the optimal execution time is obtained when all nodes allocated to a task complete their computation at the same time [79]. Thus, we get the estimated execution time of running a task $\tau(A, \sigma, D)$ on N processing nodes as [45],

$$\mathcal{E}(\sigma, N) = \frac{1-\beta}{1-\beta^N}\sigma(\tau' + \chi') \tag{7.1}$$

$$where \quad \beta = \frac{\chi'}{\tau' + \chi'} \tag{7.2}$$

and τ' and χ' are the estimated task parameters. When a new task τ arrives, it is tentatively inserted into the task waiting queue (TWQ) following an earliest deadline first (EDF) order (line 1). Among currently running tasks, we then identify the last dispatched one (line 2). S denotes the start time of the next task. If there are tasks currently running in the cluster, the admission controller estimates its completion time and assumes that the next task cannot start until then (line 4). Otherwise, the waiting task can start immediately (line 6). Next, the admission controller analyzes each waiting task, compares its resultant momentary system utilization with the utilization bound $\hat{U}(k)$. If all waiting tasks successfully pass the test, the new task is accepted. Otherwise it is rejected (lines 8-16).

7.2.2 Dispatcher

The dispatcher partitions a divisible task into subtasks and dispatches them to execute on the allocated processors. When a processing node becomes available, the dispatcher takes the first task $\tau(A, \sigma, D)$ in TWQ, partitions the task and sends a subtask of size $\hat{\sigma}$ to the node, where $\hat{\sigma} = \min\left(\frac{A+D-CurrentTime}{m(\tau'+\chi')}, \sigma\right)$. The remaining portion of the task $\tau(A, \sigma - \hat{\sigma}, D)$ is left in TWQ. As we can see, the dispatcher chooses a proper size $\hat{\sigma}$ to guarantee that the dispatched subtask completes no later than the task's absolute deadline $A + D$, assuming the subtask's actual execution time is no more than m times of the estimated execution time. If the actual execution time is less than the estimate, a processor may complete the subtask earlier and receive multiple rounds of subtasks from the same task. This dispatching algorithm allocates fewer processors to a task with a longer deadline, while for a task with a shorter deadline, it spreads the task to more processors to finish it faster and before its deadline.

7.2.3 PI Controller Design

We apply a control-theoretic methodology to design a PI controller on the utilization bound. By manipulating the utilization bound $\hat{U}(k)$, the PI controller keeps the deadline miss ratio $MR(k)$ around its set point M_s. We first specify the performance requirements for the controller, and then use system identification techniques to establish a dynamic model for the cluster. Based on the dynamic model, we use the Root Locus method to design the PI controller that meets the performance specifications.

A key benefit of our control-theoretic approach is that it enables us to perform rigorous analysis on critical system properties such as stability, overshoot, and settling time. A dynamic system is *stable* if it converges to the equilibrium point for any initial condition [33]. In our real-time cluster, the equilibrium point is the deadline miss ratio set point M_s. Hence, a stable system guarantees the convergence to M_s. The *overshoot* represents the maximum amount by which the set point M_s is exceeded and the *settling time* denotes how fast the system converges. Through rigorous theoretical analysis, we can prove that a cluster controlled by our approach meets the performance specifications despite significant workload and system variations.

For control theory based design and analysis, we need a dynamic model (e.g., difference equations) as the foundation, which characterizes the relationship between control inputs and controlled variables of the system. In contrast to mechanical and electrical systems whose dynamics are usually well understood, the lack of existing dynamic models for open real-time systems has been a key hurdle in applying control-theoretic approaches to such systems. Since the task waiting queue (TWQ) is an integrator of flow (which gives rise to difference equations), the controlled real-time cluster can be modeled as a difference equation with unknown parameters. That is,

$$MR(k) = \sum_{j=1}^{n} a_j MR(k-j) + \sum_{j=1}^{n} b_j \hat{U}(k-j) \tag{7.3}$$

We use system identification [5] to estimate unknown parameter values (i.e., the values of a_j and b_j, where $j = 1, \cdots, n$). In an n^{th} order model, there are $2n$ parameters that need to be decided by the least-squares estimator. Based on this dynamic model, we then apply the Root Locus method [33] to design the PI controller to meet the performance specifications. Our approach is similar to that of our previous work [52].

7.3 Evaluation

In the previous section, we have presented the feedback-control based divisible load scheduling algorithm. This section evaluates its performance.

We used the discrete simulator DLSim, described in Section 4.5. The workload is generated following the same approach as described in Section 4.5. To simulate the uncertainty of task transmission and computation costs, two values F_{τ_i} and F_{χ_i} are randomly picked in the interval $[0.1, 2]$ as task T_i's transmission and computation cost factors so that T_i's actual costs are $\tau = \tau' F_{\tau\_i}$ and $\chi = \chi' F_{\chi\_i}$. Similar to the work by Lee et al. [43], we adopt a metric $SystemLoad = \mathcal{E}(Avg\sigma, 1)\frac{\lambda}{N}$ to represent how loaded a cluster is for a simulation, where $Avg\sigma$ is the average task data size, $\mathcal{E}(Avg\sigma, 1)$ is the execution time of running an average size task on a single node (see Eq (7.1) for $\mathcal{E}$'s calculation), and $\frac{\lambda}{N}$ is the average task arrival rate per node. In this group of simulation, we choose the following parameters: N= 16, $\tau' = 1$, χ'=100, $Avg\sigma$=200. In addition, the sampling period is T_s = 100,000 time unit and the miss ratio set point is $M_s = 5\%$.

7.3.1 Effects of Unpredictable Workload

We first evaluate the algorithm under a varied and unpredictable workload. Three metrics — *Deadline Miss Ratio*, *Task Reject Ratio* and *System Utilization* are adopted. The task reject ratio is the ratio of the number of task rejections to the number of task arrivals. The greater the system utilization and the smaller the deadline miss ratio and the task reject ratio, the better the performance.

For simulations in this subsection, we start with a light cluster workload and then let it gradually increase. Figure 7.2 illustrates the resultant *Deadline Miss Ratio*. As

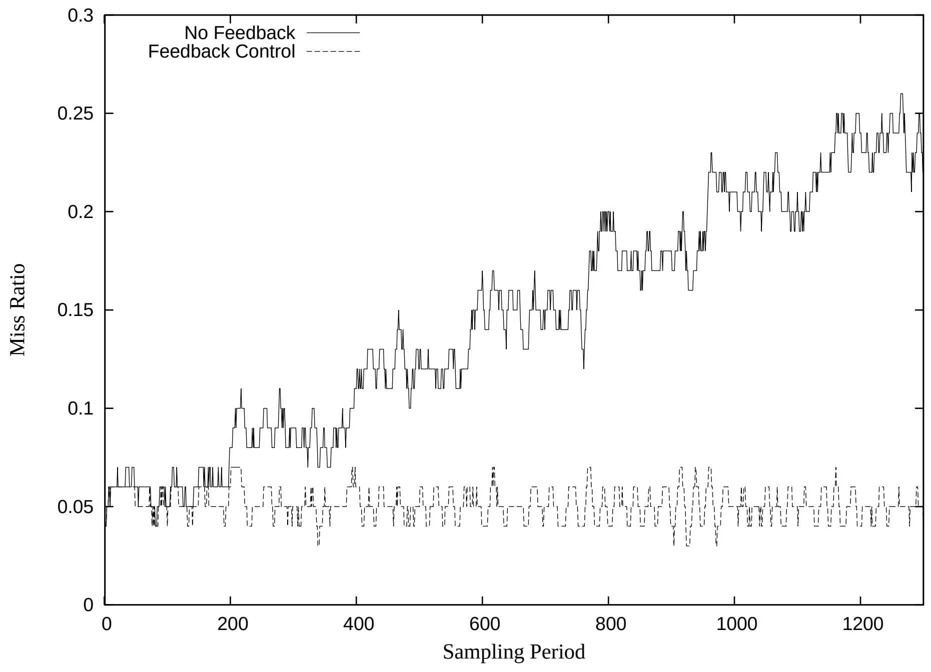


Figure 7.2: Deadline Miss Ratio upon Changing Workload.

we can see from the figure, if no feedback control is applied (that is, tasks are always accepted when the system utilization is below 1), the deadline miss ratio increases with the workload. When the workload is heavy, 25% of the accepted tasks miss their deadlines. This is not an acceptable performance for real-time applications. On the contrary, when the feedback control is applied, the scheduler dynamically adjusts the system utilization bound so that the admission controller accepts fewer tasks when the system is overloaded. Therefore, the deadline miss ratio can be maintained at the target 5%.

Next, we compare the task reject ratio and the system utilization of the feedback-control based scheduler with two baseline strategies: 1) the ideal case where the scheduler knows task execution parameters accurately; and 2) the worst case where the scheduler assumes the worst case transmission and computation costs. Figure 7.3

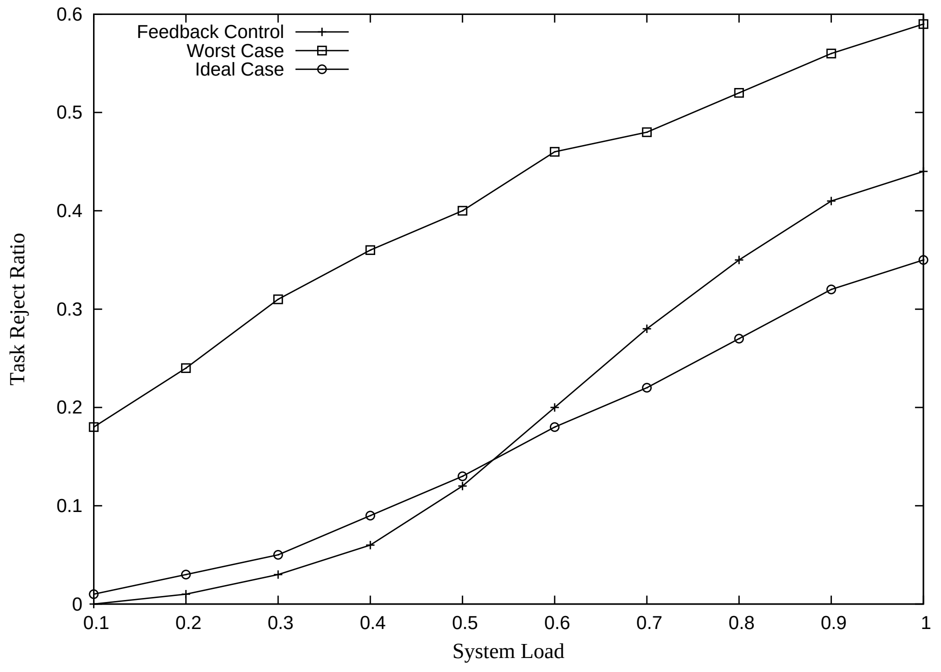


Figure 7.3: Task Reject Ratio upon Changing Workload.

and Figure 7.4 show the resultant *Task Reject Ratio* and *System Utilization*. As we can see, the ideal case provides the best performance: it rejects a small number of tasks to ensure no deadline misses, while utilizing the system well. However, in practice, it is impossible to have accurate knowledge of task execution times in advance. When the workload is light, our feedback-control based scheduler performs similarly to the ideal case and much better than the worst case; by slightly overloading the system and allowing a small number of deadline misses, the feedback-control based scheduler achieves an even better task reject ratio than the best case. Under the heavy workload, due to the lack of accurate task knowledge, our scheduler achieves less than ideal utilization. However, it keeps the deadline miss ratio around 5% and the utilization around 75%. In contrast, for an algorithm that assumes the worst case task parameters (like any of the existing real-time divisible load scheduling algorithms), it performs much worse than the feedback-control based algorithm. Note, in this simulation, the worst case task execution cost is assumed to be no more than twice of

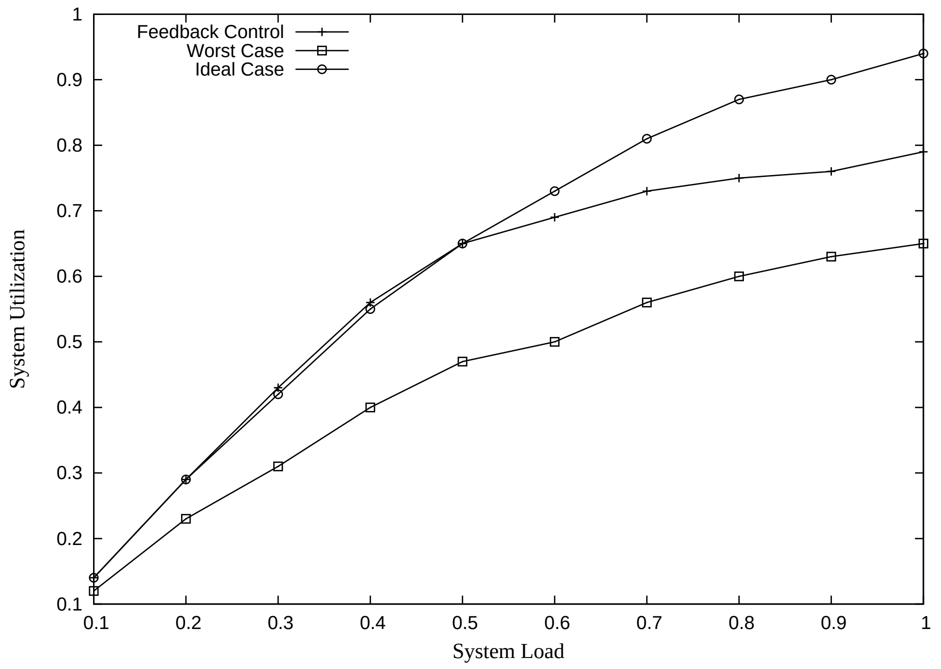


Figure 7.4: System Utilization upon Changing Workload.

the actual cost. In practice, the estimate could be much worse and so is the resultant task reject ratio and system utilization of a worst-case based algorithm.

7.3.2 Effects of Node Failures

The previous subsection shows that the feedback-control based scheduler provides stable and predictable performance in spite of a changing and unpredictable workload. This subsection evaluates how our algorithm performs upon node failures. Node failure is a common problem in clusters. The scheduling algorithms in previous work [35, 43, 17, 46, 45, 44, 58] simply assume node failures never occur, thus do not tolerate them. As a result, upon a node failure, existing schedulers cannot detect it. The dispatcher may be able to flag off the failed node, since running a task on the failed node never returns the completion signal. However, it is difficult to differentiate between a node failure and the case where the actual execution time of a task is much longer than the estimate. Therefore, existing approaches do not have an effective mechanism to handle node failures. In contrast, our feedback-control based scheduling algorithm can dynamically reduce the system utilization when the cluster capacity decreases as a result of node failures.

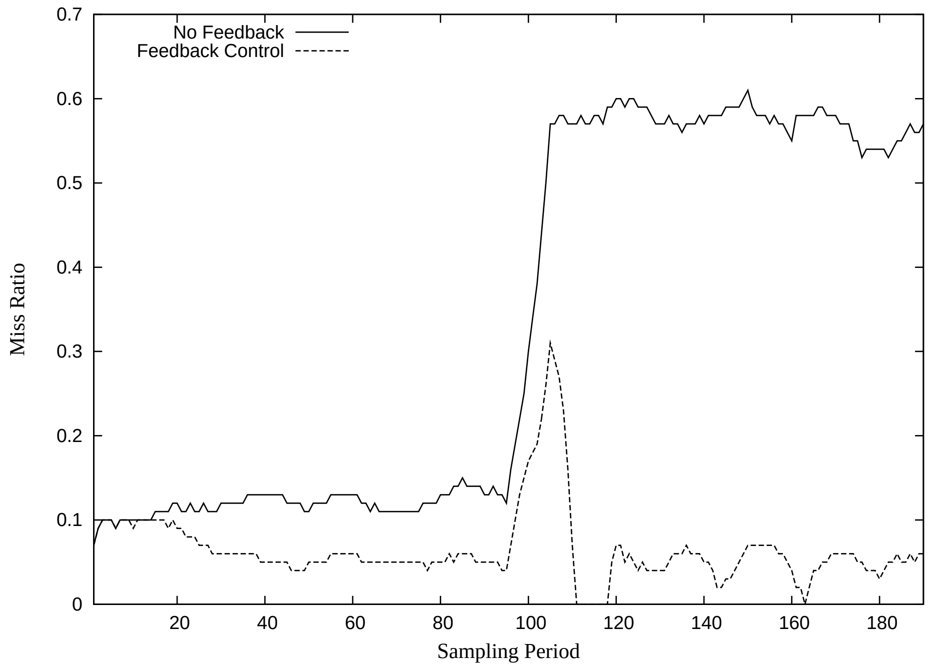


Figure 7.5: Deadline Miss Ratio upon Node Failures.

In this simulation, system workload is kept at a constant level. To evaluate the effects of node failures, we turn off 40% of the cluster nodes in the middle of the simulation. As we can see from Figure 7.5, when 40% of the nodes fail around the 90^{th} sampling period, because the no-feedback scheduler cannot detect this capacity change, its deadline miss ratio increases and keeps at 60%. For the feedback-control based scheduler, the deadline miss ratio slightly increases, because some waiting tasks miss their deadlines due to node failures. However, thanks to the feedback-control mechanism, our scheduler can quickly react and bring the deadline miss ratio back to its set point 5%.

7.4 Summary

In this chapter, we have investigated the problem of providing real-time QoS guarantees for arbitrarily divisible applications in unpredictable environments. The proposed feedback-control based real-time divisible load scheduler can dynamically control the system utilization bound to maintain low deadline miss ratio and high system utilization. Simulation results have demonstrated that the proposed algorithm can provide stable performance despite unpredictable workload and node failures.

Chapter 8

Conclusions

This dissertation addresses the challenging problem of providing determinitic QoS for cluster computing. We have integrated Divisible Load Theory into real-time scheduling theory, and developed real-time divisible load scheduling algorithms for cluster computing. This research has contributed significantly to the area of real-time divisible load scheduling. We made the following contributions.

1. *Real-Time Divisible Load Scheduling with Setup Costs*: For an arbitrarily divisible load, the required number of processing nodes to meet a task deadline is not fixed and is adjustable in accordance to the available resources. Our goal is to exploit the arbitrarily divisible property to improve system performance. Building on our team's previous work, we developed a novel algorithmic approach integrating divisible load theory (DLT) and earliest deadline first (EDF) scheduling. By integrating these together, we developed a new real-time scheduling approach for arbitrarily divisible loads. We first identified three important and necessary design decisions for cluster-based real-time divisible load scheduling, i.e., (1) workload partitioning, (2) node assignment, and (3) task execution order. We proposed a scheduling framework that can configure differ-

ent policies for each of the three design decisions and use it to generate various algorithms. In particular, we investigated the scenarios where communication and computation setup costs are not negligible. In this dissertation, we systematically studied these algorithms and identified scenarios where the choices of design parameters have significant effects. Most significantly, we proved that an established claim on real-time divisible load scheduling is not valid. Prior to our work, researchers believed that the minimum node assignment always leads to a better real-time performance than the maximum node assignment. We used experimental data to prove that this claim does not always hold.

2. *Real-Time Divisible Load Scheduling with Advance Resource Reservations*: A grid scheduler, unlike a central scheduler, has neither immediate access to all system information nor full control of grid resources and grid tasks. With the emergence of grid applications that require simultaneous access to multi-site resources, supporting advance reservations in a cluster has become increasingly important. We combined real-time cluster scheduling of arbitrarily divisible loads with resource reservation protocols to provide real-time scheduling theory and tools for grid computing. This research presents the first real-time divisible load scheduling algorithm that can support advance reservations in a cluster. The approach we proposed not only enforces the real-time agreement but also addresses the under-utilization concerns raised by advance reservations. We systematically studied the impact of advance reservations on the performance of a cluster's scheduler and showed that, with our proposed algorithm and appropriate advance reservations, the system performance could be maintained at the same level as with the no reservation case.

3. *Efficient Algorithm for Real-Time Divisible Load Scheduling*: As the cluster size and the number of tasks in the waiting queue increase, existing arbitrarily divisible load scheduling approaches become inefficient and do not scale well. We designed and developed an efficient algorithm for real-time divisible load scheduling, time complexity of which is linear in the number of tasks and number of nodes in a cluster. Unlike existing approaches, the new algorithm relaxes the tight coupling between the task admission controller and the task dispatcher. By eliminating the need to generate exact schedules in the admission controller, the algorithm avoids the large scheduling overhead. We showed that to make admission control decision on a typical cluster workload, the previously best-known algorithm would take 11 hours while our new algorithm only needs 37 minutes. The algorithm reduces the scheduling overhead significantly and is useful for large and busy clusters.

4. *Feedback-control based Real-Time Divisible Load Scheduling*: Current arbitrarily divisible load scheduling approaches are based on "open-loop" scheduling. These approaches perform well in predictable environments, but their performance in open and dynamic environments may be unacceptable. "Open-loop" real-time schedulers are often designed based on worst-case workload parameters to ensure task deadlines. This could result in a highly underutilized system based on the pessimistic estimation of the workloads. In an open environment like a general-purpose cluster, where workloads are unknown and may vary at run-time, we need adaptive solutions that can maintain desired performance by handling system variations dynamically. We developed a real-time divisible load scheduler based on the feedback-control paradigm. It can dynamically control the system utilization bound to maintain low deadline miss ratio and high system utilization. We showed that the proposed algorithm can provide stable

performance despite unpredictable workload and node failures. By handling system and workload variations dynamically, our algorithm is able to provide QoS guarantees and fault tolerance to soft real-time applications.

Bibliography

[1] T. F. Abdelzaher and V. Sharma. A synthetic utilization bound for aperiodic tasks with resource requirements. In *Proc. of 15th Euromicro Conference on Real-Time Systems*, pages 141–150, Porto, Portugal, July 2003.

[2] A. Amin, R. Ammar, and A. E. Dessouly. Scheduling real time parallel structure on cluster computing with possible processor failures. In *Proc of 9th IEEE International Symposium on Computers and Communications*, pages 62–67, July 2004.

[3] R. A. Ammar and A. Alhamdan. Scheduling real time parallel structure on cluster computing. In *Proc. of 7th IEEE International Symposium on Computers and Communications*, pages 69–74, Taormina, Italy, July 2002.

[4] J. Anderson and A. Srinivasan. Pfair scheduling: Beyond periodic task systems. In *Proc. of the 7th International Conference on Real-Time Systems and Applications*, pages 297–306, Cheju Island, South Korea, December 2000.

[5] K. J. Astrom and B. Wittenmark. *Adaptive Control (2nd Ed.)*. Addison-Wesley, 1995.

[6] C. E. L. f. P. P. ATLAS (A Toroidal LHC Apparatus) Experiment. Atlas web page. http://atlas.ch/.

[7] D. Babbar and P. Krueger. On-line hard real-time scheduling of parallel tasks on partitionable multiprocessors. In *ICPP*, pages 29–38, 1994.

[8] L. Barsanti and A. C. Sodan. Adaptive job scheduling via predictive job resource allocation. In E. Frachtenberg and U. Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 115–140. Springer Verlag, 2006. Lect. Notes Comput. Sci. vol. 4376.

[9] V. Bharadwaj, T. G. Robertazzi, and D. Ghose. *Scheduling Divisible Loads in Parallel and Distributed Systems.* IEEE Computer Society Press, Los Alamitos, CA, 1996.

[10] A. Block, B. Brandenburg, J. H. Anderson, and S. Quint. An adaptive framework for multiprocessor real-time system. In *ECRTS '08: Proceedings of the 20th Euromicro Conference on Real-Time Systems*, pages 23–33, July 2008.

[11] S. A. Brandt, S. Banachowski, C. Lin, and T. Bisson. Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, pages 396–409, December 2003. Cancun, Mexico.

[12] J. Cao and F. Zimmermann. Queue scheduling and advance reservations with cosy. In *Parallel and Distributed Processing Symposium, 2004*, page 63a, April 2004.

[13] H. Cheng and S. Goddard. Vre-net: A qos-supported network subsystem for multimedia applications. In *Proceedings of the IEEE 20th International Conference on Advanced Information Networking and Applications (AINA 2006)*, pages 113–118, April 2006. Vienna, Austria.

[14] N. Christin, J. Liebeherr, and T. F. Abdelzaher. A quantitative assured forwarding service. In *In IEEE Infocom*, pages 864–873, 2002.

[15] H.-H. Chu and K. Nahrstedt. Cpu service classes for multimedia applications. In *ICMCS, Vol. 1*, pages 296–301, 1999.

[16] G. Chun, H. Dail, H. Casanova, and A. Snavely. Benchmark probes for grid assessment. In *IPDPS*, 2004.

[17] S. Chuprat and S. Baruah. Scheduling divisible real-time loads on clusters with varying processor start times. In *14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '08)*, pages 15–24, Aug 2008.

[18] S. Chuprat, S. Salleh, and S. Baruah. Evaluation of a linear programming approach towards scheduling divisible real-time loads. In *International Symposium on Information Technology*, pages 1–8, Aug 2008.

[19] W. Cirne and F. Berman. Using moldability to improve the performance of supercomputer jobs. *Journal of Parallel and Distributed Computing*, 62(10):1571–1601, 2002.

[20] K. Czajkowski, I. Foster, C. Kesselman, V. Sander, and S. Tuecke. Snap: A protocol for negotiating service level agreements and coordinating resource management in distributed systems, 2002.

[21] K. Czajkowski, I. T. Foster, N. T. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. In *IPPS/SPDP '98: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82, London, UK, 1998. Springer-Verlag.

[22] M. Degermark, T. Kohler, S. Pink, and O. Schelen. Advance reservations for predictive service in the internet. *Multimedia Systems*, 5(3):177–186, 1997.

[23] M. L. Dertouzos and A. K. Mok. Multiprocessor online scheduling of hard-real-time tasks. *IEEE Trans. Softw. Eng.*, 15(12):1497–1506, 1989.

[24] Y. Diao, J. L. Hellerstein, A. Storm, M. Surendra, S. Lightstone, S. Parekh, and C. Garcia-Arellano. Using mimo linear control for load balancing in computing systems. In *American Control Conference (ACC)*, pages 2045–2050, July 2004.

[25] Y. Diao, C. W. Wu, J. L. Hellerstein, A. J. Storm, M. Surendra, S. Lightstone, S. Parekh, C. Garcia-Arellano, M. Carroll, L. Chu, and J. Colaco. Comparative studies of load balancing with control and optimization techniques. In *American Control Conference (ACC)*, June 2005.

[26] M. Drozdowski and P. Wolniewicz. Experiments with scheduling divisible tasks in clusters of workstations. In *6th International Euro-Par Conference on Parallel Processing*, pages 311–319, August 2000.

[27] M. Eltayeb, A. Dogan, and F. Özgüner. A data scheduling algorithm for autonomous distributed real-time applications in grid computing. In *Proc. of 33rd International Conference on Parallel Processing*, pages 388–395, Montreal, Canada, August 2004.

[28] D. Ferrari, A. Gupta, and G. Ventre. Distributed advance reservation of real-time connections. *Multimedia System*, 5(3):187–198, 1997.

[29] C. M. S. C. E. for the Large Hadron Collider at CERN (European Lab for Particle Physics). Cms web page. http://cmsinfo.cern.ch/Welcome.html/.

[30] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy. A distributed resource management architecture that supports advance reservations and co-allocation. In *Proceedings of the International Workshop on Quality of Service*, pages 27–36, 1999.

[31] I. Foster, A. Roy, and V. Sander. A quality of service architecture that combines resource reservation and application adaptation, 2000.

[32] I. T. Foster, M. Fidler, A. Roy, V. Sander, and L. Winkler. End-to-end quality of service for high-end applications. *Computer Communications*, 27(14):1375–1388, 2004.

[33] G. F. Franklin, D. J. Powell, and A. Emami-Naeini. *Feedback Control of Dynamic Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[34] S. Funk and S. Baruah. Task assignment on uniform heterogeneous multiprocessors. In *Proc of 17th Euromicro Conference on Real-Time Systems*, pages 219–226, July 2005.

[35] L. He, S. Jarvis, D. Spooner, X. Chen, and G. Nudd. Hybrid performance-oriented scheduling of moldable jobs with qos demands in multiclusters and grids. In *Proc. of the 3rd International Conference on Grid and Cooperative Computing*, pages 217–224, Wuhan, China, October 2004.

[36] C. V. Hollot, V. Misra, D. F. Towsley, and W. Gong. A control theoretic analysis of red. In *INFOCOM'01*, pages 1510–1519, 2001.

[37] Z. Huang and Y. Qiu. A bidding strategy for advance resource reservation in sequential ascending auctions. In *Autonomous Decentralized Systems, 2005. ISADS*, pages 284 – 288, April 2005.

[38] D. Isovic and G. Fohler. Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints. In *Proc. of 21st IEEE Real-Time Systems Symposium*, pages 207–216, Orlando, FL, November 2000.

[39] C. Jin, D. X. Wei, and S. H. Low. Fast tcp: Motivation, architecture, algorithms, performance, 2004.

[40] S. Keshav. A control-theoretic approach to flow control. pages 3–15, 1991.

[41] J.-K. Kim, S. Shivle, H. J. Siegel, A. A. Maciejewski, T. D. Braun, M. Schneider, S. Tideman, R. Chitta, R. B. Dilmaghani, R. Joshi, A. Kaul, A. Sharma, S. Sripada, P. Vangari, and S. S. Yellampalli. Dynamically mapping tasks with priorities and multiple deadlines in a heterogeneous environment. *Journal of parallel and distributed computing*, vol. 67(no.2):154–169, 2007.

[42] S. Kim and J. B. Weissman. A genetic algorithm based approach for scheduling decomposable data grid applications. In *Proc. of International Conference on Parallel Processing*, pages 406–413, Montreal, Canada, August 2004.

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

[44] X. Lin, Y. Lu, J. Deogun, and S. Goddard. Enhanced real-time divisible load scheduling with different processor available times. In *14th International Conference on High Performance Computing*, December 2007.

[45] X. Lin, Y. Lu, J. Deogun, and S. Goddard. Real-time divisible load scheduling for cluster computing. In *Proceedings of the 13th IEEE Real-Time and Embedded Technology and Application Symposium*, pages 303–314, Bellevue, WA, April 2007.

[46] X. Lin, Y. Lu, J. Deogun, and S. Goddard. Real-time divisible load scheduling with different processor available times. In *Proceedings of the 2007 International Conference on Parallel Processing (ICPP 2007)*, 2007.

[47] X. Lin, A. Mamat, Y. Lu, J. Deogun, and S. Goddard. Real-time scheduling of divisible loads in cluster computing environments. *Journal of Parallel and Distributed Computing (JPDC)*, 70:296–308, March 2010.

[48] C. Liu, L. Yang, I. Foster, and D. Angulo. Design and evaluation of a resource selection framework for grid applications. In *In Proceedings of the 11th IEEE Symposium on High-Performance Distributed Computing*, July 2002.

[49] X. Liu and S. Goddard. Supporting dynamic qos in linux. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 246–254, May 2004. Toronto, Canada.

[50] X. Liu, X. Zhu, P. Pradeep, Z. Wang, and S. Sharad. Optimal multivariate control for differentiated services on a shared hosting platform. In *IEEE Conference on Decision and Control*, pages 3792–3799, 2007.

[51] C. Lu, T. F. Abdelzaher, J. A. Stankovic, and S. H. Son. A feedback control approach for guaranteeing relative delays in web servers. In *In IEEE Real-Time Technology and Applications Symposium*, pages 51–62, 2001.

[52] C. Lu, Y. Lu, T. F. Abdelzaher, J. A. Stankovic, and S. H. Son. Feedback control architecture and design methodology for service delay guarantees in web servers. *IEEE Transactions on Parallel Distributed Systems (TPDS)*, 17(9):1014–1027, 2006.

[53] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. Design and evaluation of a feedback control EDF scheduling algorithm. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 56–67, 1999.

[54] C. Lu, X. Wang, and X. Koutsoukos. Feedback utilization control in distributed real-time systems with end-to-end tasks. *IEEE Transactions on Parallel and Distributed Systems*, 16:550–561, 2005.

[55] Y. Lu, T. Abdelzaher, C. Lu, and G. Tao. An adaptive control framework and its application to differentiated caching services. In *International Workshop on Quality of Service (IWQoS)*, Miami Beach, FL, May 2002.

[56] Y. Lu, T. F. Abdelzaher, and A. Saxena. Design, implementation, and evaluation of differentiated caching services. *IEEE Transactions on Parallel Distributed Systems (TPDS)*, 15(5):440–452, 2004.

[57] J. MacLaren. Advance reservations: State of the art. http: //www.fz-juelich.de /zam/RD /coop/ggf/graap/graap-wg.html.

[58] A. Mamat, Y. Lu, J. Deogun, and S. Goddard. Real-time divisible load scheduling with advance reservations. In *20th Euromicro Conference on Real-Time Systems*, pages 37–46, July 2008.

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

[60] M. W. Margo, K. Yoshimoto, P. Kovatch, and P. Andrews. Impact of reservations on production job scheduling. In *13th Workshop on Job Scheduling Strategies for Parallel Processing*, June 2007.

[61] P. Mohapatra. Dynamic real-time task scheduling on hypercubes. *J. of Parallel and Distributed Computing*, 46(1):91–100, 1997.

[62] M. A. S. Netto and R. Buyya. Offer-based scheduling of deadline-constrained bag-of-tasks applications for utility computing systems. In *Proceedings of the 18th International Heterogeneity in Computing Workshop, in conjunction with the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS'09)*, Roma, Italy, May 2009.

[63] S. Parekh, J. Hellerstein, T. S. Jayram, N. Gandhi, D. Tilbury, and J. Bigus. Using control theory to achieve service level objectives in performance management, 2001.

[64] P. Pop, P. Eles, Z. Peng, and V. Izosimov. Schedulability-driven partitioning and mapping for multi-cluster real-time systems. In *Proc. of 16th Euromicro Conference on Real-Time Systems*, pages 91–100, July 2004.

[65] X. Qin and H. Jiang. Dynamic, reliability-driven scheduling of parallel real-time jobs in heterogeneous systems. In *Proc. of 30th International Conference on Parallel Processing*, pages 113–122, Valencia, Spain, September 2001.

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

[67] K. Ramamritham, J. A. Stankovic, and W. Zhao. Distributed scheduling of tasks with deadlines and resource requirements. *IEEE Transactions on Computers*, 38(8):1110–1123, 1989.

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

[69] G. Sabin, M. Lang, and P. Sadayappan. Moldable parallel job scheduling using job efficiency: An iterative approach. In *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), in conjunction with ACM SIGMETRICS*, pages 94–114, Saint Malo, France, June 2006.

[70] M. Siddiqui, A. Villazón, and T. Fahringer. Grid allocation and reservation—grid capacity planning with negotiation-based advance reservation for optimized qos. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 103, New York, NY, USA, 2006. ACM Press.

[71] K. Skadron, T. Abdelzaher, and M. R. Stan. Control-theoretic techniques and thermal-rc modeling for accurate and localized dynamic thermal management. In *In International Symposium on High Performance Computer Architecture*, pages 17–28, 2002.

[72] W. Smith, I. Foster, and V. Taylor. Scheduling with advanced reservations. In *IPDPS '00: Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, pages 127–132, Washington, DC, USA, 2000. IEEE Computer Society.

[73] B. Sotomayor, K. Keahey, and I. Foster. Overhead matters: A model for virtual resource management. In *VTDC '06: Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing*, page 5, Washington, DC, USA, 2006. IEEE Computer Society.

[74] S. Srinivasan, S. Krishnamoorthy, and P. Sadayappan. A robust scheduling strategy for moldable scheduling of parallel jobs. In *2003 IEEE International Conference on Cluster Computing*, pages 92–99, Hong Kong, China, December 2003.

[75] J. A. Stankovic, T. He, T. Adbelzaher, M. Marley, G. Tao, S. Son, and C. Lu. Feedback control scheduling in distributed real-time systems*. In *In IEEE Real-Time Systems Symposium*, pages 59–72, 2001.

[76] D. Swanson. Personal communication. Director, UNL Research Computing Facility (RCF) and UNL CMS Tier-2 Site, August 2005.

[77] TERAGRID. http://www.teragrid.org/.

[78] K. van der Raadt, Y. Yang, and H. Casanova. Practical divisible load scheduling on grid platforms with apst-dv. In *Proc. of 19th International Parallel and Distributed Processing Symposium*, Denver, CA, April 2005.

[79] B. Veeravalli, D. Ghose, and T. G. Robertazzi. Divisible load theory: A new paradigm for load scheduling in distributed systems. *Cluster Computing*, 6(1):7–17, 2003.

[80] X. Wang, X. Fu, X. Liu, and Z. Gu. Power-aware cpu utilization control for distributed real-time systems. In *RTAS '09: Proceedings of the 2009 15th IEEE Symposium on Real-Time and Embedded Technology and Applications*, pages 233–242, 2009.

[81] J. T. Wen and M. Arcak. A unifying passivity framework for network flow control. In *IEEE Transactions on Automatic Control*, pages 162–174, 2002.

[82] L. C. Wolf and R. Steinmetz. Concepts for resource reservation in advance. *Multimedia Tools and Applications*, 4(3):255–278, 1997.

[83] W. Xu, X. Zhu, S. Singhal, and Z.Wang. Predictive control for dynamic resource allocation in enterprise data centers. In *IEEE/IFIP Network Operations & Management Symposium*, April 2006.

[84] D. Yu and T. G. Robertazzi. Divisible load scheduling for grid computing. In *Proc. of IASTED International Conference on Parallel and Distributed Computing and Systems*, Los Angeles, CA, November 2003.

[85] L. Zhang. Scheduling algorithm for real-time applications in grid environment. In *Proc. of IEEE International Conference on Systems, Man and Cybernetics*, volume 5, page 6 pp., Hammamet, Tunisia, October 2002.

[86] X. Zhu, Z. Wang, and S. Singhal. Utility driven workload management using nested control design. In *American Control Conference (ACC)*, June 2006.