University of Nebraska - Lincoln
DigitalCommons@University of Nebraska - Lincoln

CSE Technical reports | Computer Science and Engineering, Department of

5-22-2009

An Efficient Algorithm for Real-Time Divisible Load Scheduling

Anwar Mamat
University of Nebraska - Lincoln, anwar@cse.unl.edu

Ying Lu
University of Nebraska - Lincoln, ylu4@unl.edu

Jitender S. Deogun
University of Nebraska - Lincoln, jdeogun1@unl.edu

Steve Goddard
University of Nebraska – Lincoln, goddard@cse.unl.edu

Follow this and additional works at: http://digitalcommons.unl.edu/csetechreports

Part of the Computer Sciences Commons

Mamat, Anwar; Lu, Ying; Deogun, Jitender S.; and Goddard, Steve, "An Efficient Algorithm for Real-Time Divisible Load Scheduling" (2009). *CSE Technical reports.* Paper 74.
http://digitalcommons.unl.edu/csetechreports/74

This Article is brought to you for free and open access by the Computer Science and Engineering, Department of at DigitalCommons@University of Nebraska - Lincoln. It has been accepted for inclusion in CSE Technical reports by an authorized administrator of DigitalCommons@University of Nebraska - Lincoln.

University of Nebraska–Lincoln, Computer Science and Engineering
Technical Report TR-UNL-CSE-2009-0008
Issued May 22, 2009

An Efficient Algorithm for Real-Time Divisible Load Scheduling Technical Report 06/05/2009

Anwar Mamat, Ying Lu, Jitender Deogun, Steve Goddard
Department of Computer Science and Engineering
University of Nebraska - Lincoln
Lincoln, NE 68588
{*anwar, ylu, deogun, goddard*}*@cse.unl.edu*

Abstract

Providing QoS and performance guarantees to arbitrarily divisible loads has become a significant problem for many cluster-based research computing facilities. While progress is being made in scheduling arbitrarily divisible loads, existing approaches are not very efficient and cannot scale to large clusters. In this paper we propose an efficient algorithm for real-time divisible load scheduling, which has a time complexity linear to the number of tasks and the number of nodes in the cluster.

1 Introduction

Arbitrarily divisible or embarrassingly parallel workloads can be partitioned into an arbitrarily large number of load fractions, and are quite common in bioinformatics as well as high energy and particle physics. For example, the CMS (Compact Muon Solenoid) [10] and ATLAS (AToroidal LHC Apparatus) [6] projects, associated with LHC (Large Hadron Collider) at CERN (European Laboratory for Particle Physics), execute cluster-based applications with arbitrarily divisible loads. As such applications become a major type of cluster workloads [25], providing QoS to arbitrarily divisible loads becomes a significant problem for cluster-based research computing facilities like the U.S. CMS Tier-2 sites [26].

There has been extensive research on real-time divisible load scheduling [16, 18, 17, 19, 8, 9]. Focusing on providing real-time guarantees and better utilizing the cluster, existing approaches give little emphasis to scheduling efficiency. They assume that scheduling takes much less time than the execution of a task, and thus ignore the scheduling overhead.

Table 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

However, clusters are becoming increasingly bigger and busier. In Table 1, we list the sizes of some OSG (Open Science Grid) clusters. As we can see, these clusters all have more than one thousand CPUs, with the largest providing over 40 thousand CPUs. Figure 1 shows the number of waiting tasks in the OSG cluster at University of California, San Diego for two 20-hour periods, demonstrating that there could sometimes 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 5, 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 1a, the best-known real-time algorithm [8] takes more than 11 hours to make admission control decisions on the 14,000 tasks arrived in an hour, while our new algorithm needs only 37 minutes.

To address the deficiency of existing approaches, in this paper, we present an efficient algorithm for real-time divisible load scheduling. The time complexity of the proposed algorithm is linear to the number of

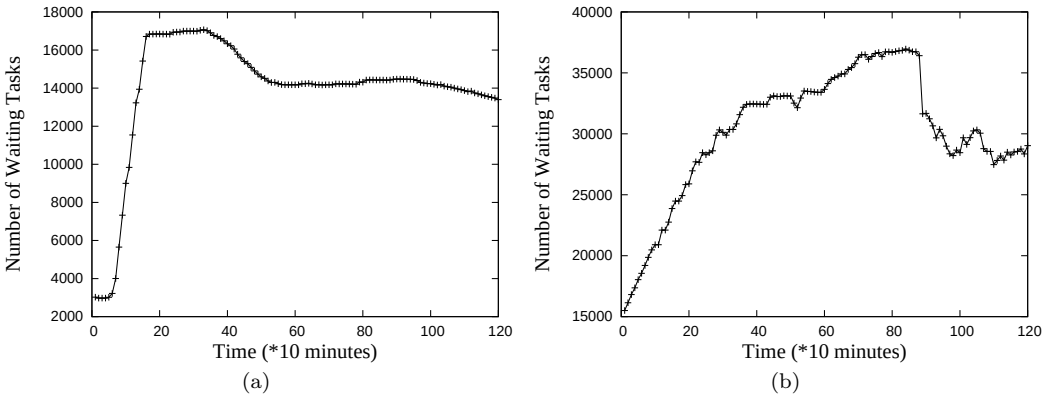


Figure 1: Status of a UCSD Cluster.

tasks in the waiting queue and the number of nodes in the cluster. In addition, the algorithm performs similar to previous algorithms in terms of providing real-time guarantees and utilizing the cluster.

The remainder of this paper is organized as follows. Related work is presented in Section 2. We describe both task and system models in Section 3. Section 4 discusses the real-time scheduling algorithm and Section 5 evaluates the algorithm performance. We conclude the paper in Section 6.

2 Related Work

Divisible load theory (DLT) has long been studied and applied in distributed systems scheduling [7, 25, 27]. It provides the foundation for optimally partitioning arbitrarily divisible loads to distributed resources. These workloads represent a broad variety of real-world applications in cluster and grid computing, such as BLAST (Basic Local Alignment Search Tool) [2], a bioinformatics application, and high energy and particle physics applications in ATLAS (AToroidal LHC Apparatus) [6] and CMS (Compact Muon Solenoid) [10] projects. Currently, large clusters usually use batch schedulers [13] to handle their workload. A Batch scheduler monitors the task execution and sends queued jobs to a cluster node when it becomes available. The goal is to optimize the system utilization. Satisfying QoS and task real-time constraints are, however, not a major objective of such schedulers.

The scheduling models investigated for distributed or multiprocessor systems most often (e.g., [1, 5, 14, 15, 21, 23, 24]) assume periodic or aperiodic sequential jobs that must be allocated to a single resource and executed by their deadlines. With the evolution of cluster computing, researchers have begun to investigate real-time scheduling of parallel applications on a cluster [3, 4, 12, 22, 28]. 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).

One closely related work is scheduling "scalable real-time tasks" in a multiprocessor system [16]. Similar to a divisible load, a scalable task can be executed on more than one processor and as more processors are allocated to it, its pure computation time decreases. If we use N to represent the number of processors and n to denote the number of waiting tasks in the system, the time complexity of the most efficient algorithms (i.e., MWF-FA and EDF-FA) proposed in that paper is $O(n^2+nN)$ [16].

There have been some existing work on cluster-based real-time divisible load scheduling [8, 9], including our own previous work [17, 18]. In [18], we have developed several scheduling algorithms for real-time divisible loads. Following those algorithms, a task must wait until a sufficient number of processors become available. This could cause a waste of processing power as some processors are idle when the system is waiting for enough processors to become available. This system inefficiency is referred to as the Inserted Idle Times (IITs) [17]. To reduce or completely eliminate IITs, several algorithms have been developed [17, 8, 9], which enable a task to utilize processors at different processor available times. Although those algorithms lead to better cluster utilizations, they have high scheduling overheads. The time complexity of algorithms proposed in [8, 9] is $O(nNlogN)$ and the algorithm in [17] has a time complexity of $O(nN^3)$.

In this paper, we propose an efficient algorithm for scheduling real-time divisible loads in clusters. Similar to algorithms in [17, 8, 9], our new algorithm elim-

inates IITs. Furthermore, with a time complexity of $O(\max(N, n))$, the algorithm is efficient and can scale to large clusters.

3 Task and System Models

In this paper, we adopt the same task and system models as our previous work [18]. For completeness, we briefly present these below.

Task Model. We assume a real-time aperiodic task model in which each aperiodic task τ_i consists of a single invocation specified by the tuple (A, σ, D), where A is the task arrival time, σ is the total data size of the task, and D is its relative deadline. The task absolute deadline is given by $A + D$.

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 processing nodes. Since different nodes process different data chunks, the head node sequentially sends every data chunk to its corresponding processing node via the switch. We assume that data transmission does not occur in parallel.[1] Therefore, only after the head node P_0 and the communication channel become available can a processing node get its data transmission to start a new task. Since for arbitrarily divisible loads, tasks and subtasks are independent, there is no need for processing nodes to communicate with each other.

According to divisible load theory, linear models are used to represent processing and transmission times [27]. In the simplest scenario, the computation time of a load σ is calculated by a cost function $Cp(\sigma) = \sigma C_{ps}$, where C_{ps} 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 $Cm(\sigma) = \sigma C_{ms}$, where C_{ms} is the time to transmit a unit of workload from the head node to a processing node.

4 Algorithm

This section presents our new algorithm for scheduling real-time divisible loads in clusters. Due to their special property, when scheduling arbitrarily divisible loads, the algorithm needs to make three important decisions. First, it determines the task execution order, which could be based on policies like EDF (Earliest Deadline First) or MWF (Maximum Workload derivative First) [16]. Second, it decides the number n of processing nodes that should be allocated to each task. Third, a strategy is chosen to partition the task among the allocated n nodes.

As is typical for dynamic real-time scheduling algorithms [11, 20, 23], 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 [16, 17, 18, 8, 9], in order to do 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 the large overhead 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 huge overhead.

To address the deficiency of existing approaches, we develop a new scheduling algorithm, which reduces 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

[1] It is straightforward to generalize our model and include the case where some pipelining of communication may occur.

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 communication channel become available, eliminating IITs.

Due to the superior property of EDF-based divisible load scheduling [18], our new algorithm schedules tasks in EDF order as well.[2] In the following, we describe in detail the two modules of the algorithm: admission controller (Section 4.1) and dispatcher (Section 4.2). 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 4.3). Section 4.4 proves the correctness of our algorithm.

4.1 Admission Controller

Upon task arrival, 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 [16, 18, 17, 19, 8, 9], the admission controller follows a brute-force 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 paper, two significant changes are made in our new admission control algorithm. First, schedulability of a new task can be determined by checking the information of the two neighboring 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 1 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 [27]. Applying this optimal partitioning, we get the execution time of running a task $\tau(A, \sigma, D)$ on N processing nodes as [18],

$$\mathcal{E}(\sigma, N) = \frac{1-\beta}{1-\beta^N}\sigma(C_{ms} + C_{ps}), \quad (1)$$

$$where \quad \beta = \frac{C_{ps}}{C_{ms} + C_{ps}}. \quad (2)$$

When a new task τ arrives, the algorithm first checks if the head node P_0 will be available for 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 neighboring 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 $slack_{min}$. We use S to denote the task start time. A task's slack is defined as,

$$slack = A + D - (S + \mathcal{E}), \quad (3)$$

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. $\tau_i.slack_{min}$ represents the *minimum* slack of all tasks scheduled after τ_{i-1}. That is,

$$\tau_{i \cdot slack_{min}} = \min(\tau_{i \cdot slack}, \tau_{i+1 \cdot slack}, \cdots, \tau_{n \cdot slack}). \quad (4)$$

If τ's execution time is less than its successor'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 always a task in TWQ, then the cluster is busy all the time. 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 4.4 for a proof). However, if TWQ becomes empty, the available resources could be put 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

[2] Although in this paper we describe the algorithm assuming EDF scheduling, the idea is applicable to other divisible load scheduling such as MWF-based scheduling algorithms [16].

will not exceed the cluster capacity. For a detailed discussion of this adjustment, please refer to Section 4.3. Once a new task τ is accepted, 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 of the two neighboring tasks. Since TWQ is sorted, locating τ's insertion point takes $O(\lg n)$ time and so do functions $getPredecessor(\tau)$ and $getSuccessor(\tau)$. Function adjust(τ) runs in $O(N)$ time (see Section 4.3) and it only occurs when TWQ is empty. The time complexity of function *updateSlacks* is $O(n)$. Therefore, algorithm AC-FAST's time complexity is $O(\max(N, n))$, linear to the number of nodes and tasks in the cluster.

Algorithm 1 AC-FAST($\tau(A, \sigma, D)$, TWQ)

1: //check head node's available time
2: **if** ($\tau.(A + D) \leq P_0$.AvailableTime) **then**
3: **return false**
4: **end if**
5: τ_p = getPredecessor(τ)
6: τ_s = getSuccessor(τ)
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** (τ_p = null) **then**
12: $\tau.S = \tau.A$
13: **else**
14: $\tau.S = \tau_p.C$
15: **if** (TWQ = $\emptyset$) **then**
16: adjust(τ)
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(τ)
26: updateSlacks(τ, 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 2 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**

4.2 Dispatcher

The dispatching algorithm is rather straightforward. When a processing node and the communication channel 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 $\hat{\sigma} = \min\left(\frac{A+D-CurrentTime}{C_{ms}+C_{ps}}, \sigma\right)$. 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, for a given task, all its subtasks 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.

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 2A); in reality, they are dispatched and executed as shown in Figure 2B, occupying the minimum numbers of nodes needed to meet their deadline requirements.

4.3 Admission Controller Adjustment

As discussed in previous sections, the admission controller assumes a different schedule than 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 accurate knowledge of the system. The admitted tasks are dispatched following the MNA policy and they 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

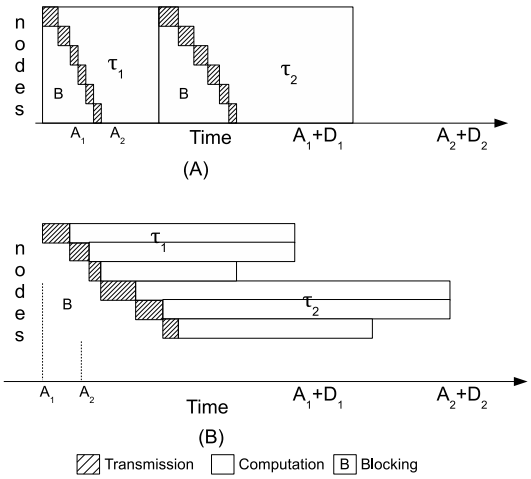


Figure 2: An Example Scenario (A) Admission Controller's View (B) Actual Task Execution.

it takes resource idleness into account to make correct admission control decisions.

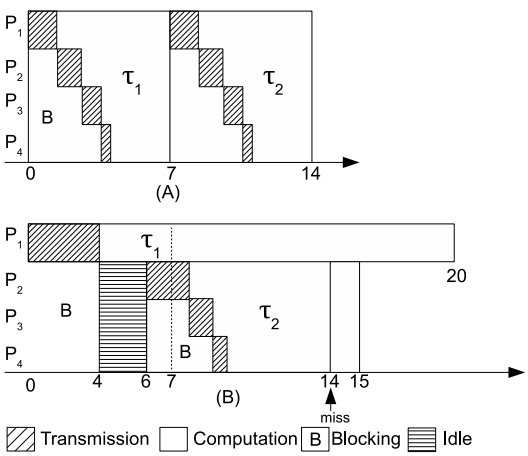


Figure 3: An Illustration of the Problem (A) Admission Controller's View (B) An Incorrect Task Execution.

We illustrate the problem in Figure 3. τ_1 arrives at time 0. The admission controller accepts it and estimates it to complete at time 7 (Figure 3A). 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 3B). 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 unused 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 $[6, 20]$, the dispatcher completes τ_2 at time 15 and misses its deadline.

To solve this problem, when a new task arrives at an empty TWQ, the admission controller invokes Algorithm 3 to compute the idle time and make a proper adjustment. The algorithm first computes the workload

Algorithm 3 adjust(τ)

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

(σ_{idle}) that could have been processed using the idled resources (lines 1-6). With all N nodes, it takes w time 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 postpone τ's start time by w (line 8).

4.4 Algorithm Correctness

In this section, we prove all tasks that have been admitted by the admission controller can be dispatched successfully by the dispatcher and be 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 executed before τ_m. Among these preceding tasks, let τ_b be the latest one 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.

σ^{AN} is defined as the total workload that has been admitted to execute in the time interval $[A_b, d_m]$. 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 conclude that the admission controller has assumed that all these tasks

can complete by τ_m's deadline d_m. That is,

$$\sigma^{AN} \geq \sum_{i=b}^{m} \sigma_i. \tag{5}$$

Since all dispatched subtasks are guaranteed to finish by their deadlines (Section 4.2), 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 to execute in the time interval $[A_b, d_m]$ must be less than $\sum_{i=b}^{m} \sigma_i$. With Eq(5), we have,

$$\sigma^{AN} > \sigma^{MN} \tag{6}$$

Next, we prove that Eq(6) cannot hold.

As mentioned, 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 4.3, 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 3), 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 to execute in the time interval $[A_b, d_m]$. Then, we have,

$$\hat{\sigma}^{AN} = \sigma^{AN} + \sum_{i=1}^{v} \bar{\sigma}_i \tag{7}$$

$$\hat{\sigma}^{MN} = \sigma^{MN} + \sum_{i=1}^{v} \bar{\sigma}_i \tag{8}$$

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 4.1 *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,*

$$\mathcal{E}(\sum_{i=1}^{h} \sigma_i, N) = \sum_{i=1}^{h} \mathcal{E}(\sigma_i, N). \tag{9}$$

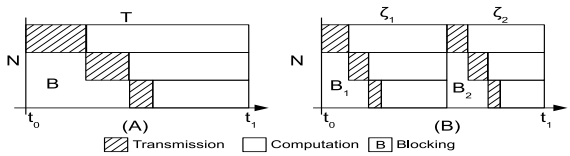


Figure 4: Merging Multiple Tasks into One Task.

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

$$\mathcal{E}(\sigma, N) = \frac{1-\beta}{1-\beta^N} \sigma (C_{ms} + C_{ps}) \tag{10}$$

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

$$\begin{aligned} \sum_{i=1}^{h} \mathcal{E}_i(\sigma_i, N) &= \sum_{i=1}^{h} (\frac{1-\beta}{1-\beta^N} \sigma_i (C_{ms} + C_{ps})) \\ &= \frac{1-\beta}{1-\beta^N} \sum_{i}^{h} \sigma_i (C_{ms} + C_{ps}) \end{aligned} \tag{11}$$

Therefore, we have,

$$\sum_{i=1}^{h} \mathcal{E}(\sigma_i, N) = \mathcal{E}(\sum_{i=1}^{h} \sigma_i, N). \tag{12}$$

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(10), we have

$$\hat{\sigma}^{AN} = \frac{d_m - A_b}{\frac{1-\beta}{1-\beta^N}(C_{ps} + C_{ms})}. \tag{13}$$

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:

$$\sigma_{p_1} = \frac{d_m - A_b}{C_{ms} + C_{ps}} \tag{14}$$

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:

$$\sigma_{p_2} = \frac{d_m - A_b - \sigma_{p_1} C_{ms}}{C_{ms} + C_{ps}} = \frac{d_m - A_b - B_{p_2}}{C_{ms} + C_{ps}} \quad (15)$$

In general, we have,

$$\sigma_{p_k} = \frac{d_m - A_b - \sum_{j=1}^{k-1} \sigma_{p_j} C_{ms}}{C_{ms} + C_{ps}} = \frac{d_m - A_b - B_{p_k}}{C_{ms} + C_{ps}} \quad (16)$$

Thus, as shown in Figure 5, we have,

$$\hat{\sigma}^{AN} = \sum_{k=1}^{N} \sigma_{p_k}. \quad (17)$$

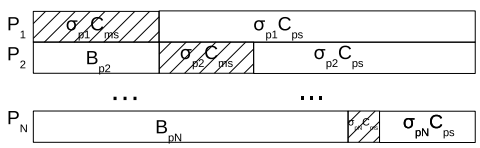


Figure 5: All Node Assignment Scenario.

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

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, 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 σ'_{p_k} to denote the sum of workloads that are processed by node P_k in the interval $[A_b, d_m]$. We have,

$$\hat{\sigma}^{MN} = \sum_{k=1}^{N} \sigma'_{p_k}. \quad (18)$$

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:

$$\sigma'_{p_1} = \frac{d_m - A_b}{C_{ms} + C_{ps}} \quad (19)$$

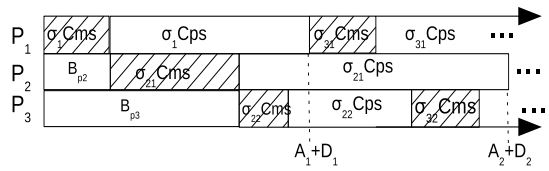


Figure 6: A Minimum Node Assignment Scenario.

As shown in Figure 6, 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, 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'_{p_2} is equal or less than the sum of data transmission time on P_1. That is:

$$B'_{p_2} \leq \sigma'_{p_1} C_{ms} \quad (20)$$

Therefore,

$$\sigma'_{p_2} = \frac{d_m - A_b - B'_{p_2}}{C_{ms} + C_{ps}} \geq \frac{d_m - A_b - \sigma'_{p_1} C_{ms}}{C_{ms} + C_{ps}} \quad (21)$$

In general,

$$B'_{p_k} \leq \sum_{j=1}^{k-1} \sigma'_{p_j} C_{ms} \qquad k = 2, 3, \cdots, N \quad (22)$$

$$\sigma'_{p_k} = \frac{d_m - A_b - B'_{p_k}}{C_{ms} + C_{ps}} \geq \frac{d_m - A_b - \sum_{j=1}^{k-1} \sigma'_{p_j} C_{ms}}{C_{ms} + C_{ps}} \quad (23)$$

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 (14),(15),(19), and (21), we have,

$$\sigma_{p_1}^{'} = \sigma_{p_1} \tag{24}$$

$$\text{and} \quad \sigma_{p_2}^{'} \geq \sigma_{p_2} \tag{25}$$

From Eq(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 $\sigma_{\Delta_i} C_{ms}$, as shown in Figure 7.

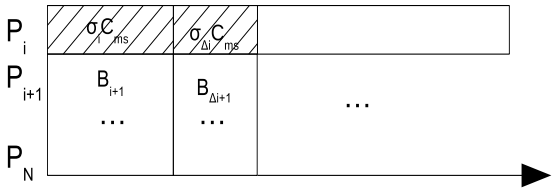


Figure 7: Increased Blocking Time.

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 $B_{\Delta_{i+1}}{=}\sigma_{\Delta_i} C_{ms}$ 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 time is,

$$B_{\Delta_{i+1}} = (\sigma_{p_i}^{'} - \sigma_{p_i}) C_{ms} \tag{26}$$

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

$$\sigma = \frac{t}{\frac{1-\beta}{1-\beta^N}(C_{ms} + C_{ps})} \tag{27}$$

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

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

That is,

$$\frac{B_{\Delta_{i+1}}}{\frac{1-\beta}{1-\beta^N}(C_{ms} + C_{ps})} \leq \sigma_{\Delta_i} \tag{28}$$

From Eq(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 $\sigma_{\Delta_2} C_{ms}$ 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 (24) and (25), we have,

$$\sum_{k=1}^{2} \sigma_{p_k}^{'} \geq \sum_{k=1}^{2} \sigma_{p_k} \tag{29}$$

We assume

$$\sum_{k=1}^{l} \sigma_{p_k}^{'} \geq \sum_{k=1}^{l} \sigma_{p_k} \tag{30}$$

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

$$\sigma_l^{inc} = \sum_{k=1}^{l} \sigma_{p_k}^{'} - \sum_{k=1}^{l} \sigma_{p_k} \tag{31}$$

σ_l^{inc} increases the blocking time on P_{l+1} by $\sigma_l^{inc} C_{ms}$. From Eq(23) we have,

$$\sigma_{p_{(l+1)}}^{'} \geq \frac{d_m - A_b - \sum_{k=1}^{l} \sigma_{p_k}^{'} C_{ms}}{C_{ms} + C_{ps}} \tag{32}$$

Combining Eq(32) with with Eq(31), we have,

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

$$\text{That is} \quad \sigma_{p_{(l+1)}}^{'} \geq \sigma_{p_{(l+1)}} - \frac{\sigma_l^{inc} C_{ms}}{C_{ms} + C_{ps}} \tag{33}$$

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

$$\sum_{k=1}^{l+1} \sigma_{p_k}^{'} = \sum_{k=1}^{l} \sigma_{p_k}^{'} + \sigma_{p_{(l+1)}}^{'} \tag{34}$$

Replace $\sum_{k=1}^{l} \sigma_{p_k}^{'}$ with Eq(31), we have,

$$\sum_{k=1}^{l+1} \sigma_{p_k}^{'} = \sum_{k=1}^{l} \sigma_{p_k} + \sigma_l^{inc} + \sigma_{p_{(l+1)}}^{'} \tag{35}$$

Replace $\sigma^{'}_{p_{l+1}}$ with Eq(33), we get,

$$\begin{aligned}
\sum_{k=1}^{l+1}\sigma^{'}_{p_k} &\geq \sum_{k=1}^{l}\sigma_{p_k}+\sigma_l^{inc}+\sigma_{p_{(l+1)}}-\sigma_l^{inc}\frac{C_{ms}}{C_{ms}+C_{ps}} \\
&= \sum_{k=1}^{l+1}\sigma_{p_k}+\sigma_l^{inc}-\sigma_l^{inc}\frac{C_{ms}}{C_{ms}+C_{ps}} \\
&= \sum_{k=1}^{l+1}\sigma_{p_k}+\sigma_l^{inc}(1-\frac{C_{ms}}{C_{ms}+C_{ps}}) \\
&= \sum_{k=1}^{l+1}\sigma_{p_k}+\sigma_l^{inc}\beta \\
&\geq \sum_{k=1}^{l+1}\sigma_{p_k} \qquad (36)
\end{aligned}$$

That is,

$$\sum_{k=1}^{l+1}\sigma^{'}_{p_k} \geq \sum_{k=1}^{l+1}\sigma_{p_k} \qquad (37)$$

Eq(37) shows that the actual accumulated workload is no less than the estimated workload. Thus, $\forall\ l \in [0, N]$ we have,

$$\sum_{k=1}^{l}\sigma^{'}_{p_k} \geq \sum_{k=1}^{l}\sigma_{p_k} \qquad (38)$$

$$\Rightarrow \quad \hat{\sigma}^{MN} \geq \hat{\sigma}^{AN} \qquad (39)$$

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 8A, data transmission $\sigma_1 C_{ms}$ 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.

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 $\sigma_\Delta = \frac{B^{'}_1}{C_{ms}+C_{ps}}$ workload from P_2 and assume that the workload were assigned to P_1, as shown in Figure 8B. This workload can be processed in $B^{'}_1$ time. The σ_Δ on P_1 increases the blocking time on P_2 by $\sigma_\Delta C_{ms}$, denoted as $B^{'}_{22}$, and reduces the computation time on P_2

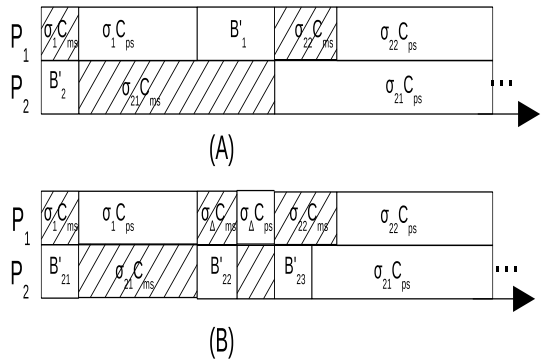


Figure 8: Another MNA Scenario.

by $\sigma_\Delta C_{ps}$, denoted as $B^{'}_{23}$, which corresponds to the removed workload. Therefore, $B^{'}_1 = B^{'}_{22} + B^{'}_{23}$. This way as shown in Figure 8B, 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 8A,

$$B^{'}_1 = \sigma_\Delta(C_{ms}+C_{ps}) \leq \sigma_{21}C_{ms} \qquad (40)$$

$$\sigma_{21}C_{ps} = \sigma_{22}(C_{ms}+C_{ps}) \qquad (41)$$

Multiply both sides of Eq (41) by C_{ms}/C_{ps}, we have,

$$\sigma_{21}C_{ms} = \sigma_{22}(C_{ms}+C_{ps})\frac{C_{ms}}{C_{ps}} \qquad (42)$$

From Eq(40) and Eq(42), we have,

$$\sigma_\Delta(C_{ms}+C_{ps}) \leq \sigma_{22}(C_{ms}+C_{ps})\frac{C_{ms}}{C_{ps}} \qquad (43)$$

Multiply both side of Eq(43) by $C_{ps}/(C_{ms}+C_{ps})$, we have,

$$\sigma_\Delta C_{ps} \leq \sigma_{22}C_{ms} \qquad (44)$$

$$\text{i.e.,} \quad B^{'}_{23} \leq \sigma_{22}C_{ms} \qquad (45)$$

In converted case, $B^{'}_{21} + B^{'}_{22} + B^{'}_{23} \leq (\sigma_1 + \sigma_\Delta + \sigma_{22})C_{ms}$. 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,

$$\sum_{j=1}^{N} \sigma'_{p_j} \geq \sum_{j=1}^{N} \sigma_{p_j} \quad (46)$$

$$\hat{\sigma}^{MN} \geq \hat{\sigma}^{AN} \quad (47)$$

With Equations (47), (7), and (8), we conclude that $\sigma^{MN} \geq \sigma^{AN}$ is true, which contradicts Eq(6). Therefore, the original assumption does not hold and no task misses its deadline.

5 Evaluation

In the previous section, we 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 [17] is represented by EDF-IIT-1 and that in [8] by EDF-IIT-2. This section evaluates their performance.

Cluster Configuration. We use a discrete simulator to simulate a range of clusters that are compliant with the system model presented in Section 3. For every simulation, three parameters, N, C_{ms} and C_{ps} are specified for a cluster.

5.1 Real-Time Performance

We first evaluate the algorithm's real-time performance. The workload is generated following the same approach as described in [18, 17] and due to the space limitation, we choose not to repeat the details here. We define a metric $SystemLoad = \mathcal{E}(Avg\sigma, 1)\frac{\lambda}{N}$ to analyze how loaded a cluster is for a simulation, where $\frac{\lambda}{N}$ is the average task arrival rate per node, $Avg\sigma$ is the average task data size, and $\mathcal{E}(Avg\sigma, 1)$ is the execution time of running a task of size $Avg\sigma$ on a single node (see Eq(10) for $\mathcal{E}$'s calculation). 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. Figure 9 illustrates the algorithm's real-time performance. 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 no worse 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 pessimistic estimates of the data transmission blocking time (Section 4). Focusing on reducing the scheduling overhead, FAST-EDF-IIT trades real-time performance for algorithm efficiency. In the next section, we use experimental data to demonstrate that FAST-EDF-IIT has huge advantages in scheduling efficiency.

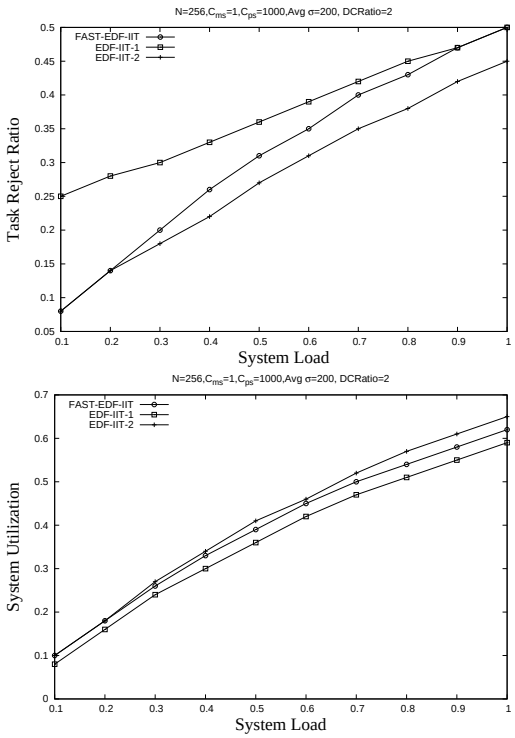
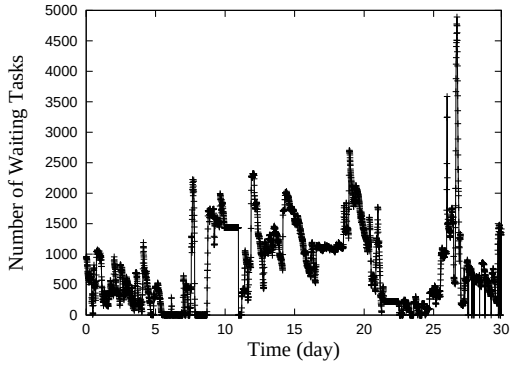


Figure 9: Algorithm's Real-Time Performance.

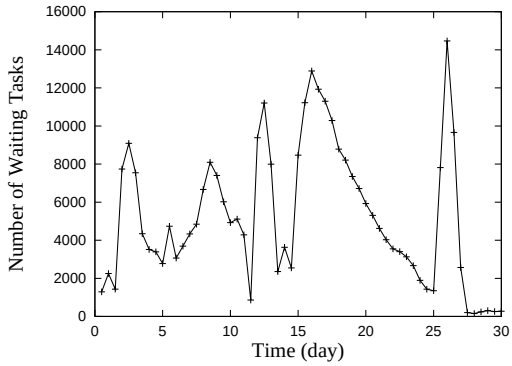
5.2 Scheduling Overhead

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 1, 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 1a) and increase from $15,000$ to $25,000$ in about three hours (Figure 1b) 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 (Figures 10) and are quite common phenomena. Based on these typical workload patterns, we design our simulations and evaluate the algorithm scheduling overhead.



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



(b) GLOW Cluster at Univ. of Wisconsin

Figure 10: Typical Cluster Status.

In this group of simulations, the following parameters are chosen for the cluster: $N = 512$, $C_{ms} = 1$ and C_{ps}=1000. In Table 1, we illustrate that it is common for a cluster to have thousands of CPUs. However, we simulate a cluster of a relatively small size, i.e., with only 512 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 $O(nNlog(N))$. So, if we show by simulation data that in a small cluster of $N = 512$ nodes FAST-EDF-IIT leads to a much less overhead, then we know for sure that it will be even more *advantageous* if we apply it in larger clusters.

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 make task reject ratios be 0 for all three algorithms so that the measured scheduling overheads of the three are comparable.

Table 2: First n Tasks' Average Scheduling Time (ms).

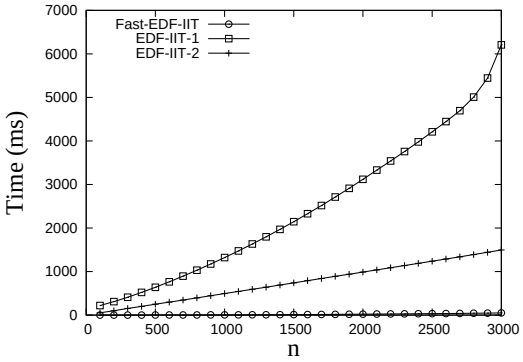
n	FAST-EDF-IIT	EDF-IIT-1	EDF-IIT-2
300	0.96	410.44	151.32
1000	4.84	1321.08	494.07
2000	20.46	3119.76	988.95
3000	48.87	6206.91	1494.91

We first measure the average scheduling time of the first n tasks, where n is in the range [100, 3000]. The simulation results are shown in Table 2 and Figure 11a. 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. 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 3 shows the average scheduling time of 10 new tasks after there are already n tasks in TWQ. The corresponding curves are in Figure 11b. 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 spend about 31 and 3 seconds to make an admission control decision.

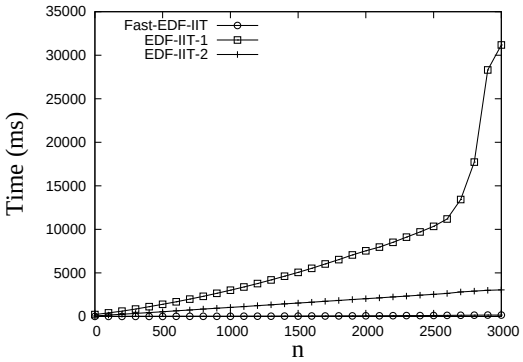
Now, let us take the simulation results and analyze what they imply for real-world clusters. It is shown in Figure 1a that the TWQ length of a cluster could increase from $3,000$ to $17,000$ in an hour. From Table 3, we know that for EDF-IIT-1 and EDF-IIT-2, it takes more than 31 and 3 seconds to admit 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 exam-

Table 3: 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



(a) Average Scheduling Time of the First n Tasks



(b) Average Scheduling Time after n Tasks Queued Up

Figure 11: Algorithm's Real-Time Scheduling Overhead.

ple has demonstrated that our new algorithm is much more efficient than existing approaches. If we analyze the algorithms using data in Figure 1b where waiting tasks increase from $15,000$ to $25,000$, the difference in scheduling time will be even more striking.

6 Conclusion

This paper 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, whose time complexities are $O(nN^3)$ [17] and $O(nNlog(N))$ [8], our new algorithm has a time complexity of $O(\max(N, n))$. We prove that the new algorithm is correct, which provides admitted tasks real-time guarantees, and it utilizes cluster resources well. We also compare our algorithm with existing approaches experimentally. Simulation results demonstrate that it scales well and can schedule large numbers of tasks efficiently. With growing sizes of clusters, we expect it to be even more advantageous.

References

[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] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, pages 403–410, 1990.

[3] 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.

[4] 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.

[5] J. Anderson and A. Srinivasan. Pfair scheduling: Beyond periodic task systems. In *7th International Conference on Real-Time Computing Systems and Applications*, Los Alamitos, CA, Dec 2000.

[6] ATLAS (AToroidal LHC Apparatus) Experiment, CERN (European Lab for Particle Physics). Atlas web page. http://atlas.ch/.

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

[8] 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.

[9] 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.

[10] Compact Muon Solenoid (CMS) Experiment for the Large Hadron Collider at CERN (European Lab for Particle Physics). Cms web page. http://cmsinfo.cern.ch/Welcome.html/.

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

[12] 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.

[13] Y. Etsion and D. Tsafrir. A short survey of commercial cluster batch schedulers. Technical Report 2005-13, School of Computer Science and Engineering, the Hebrew University, Jerusalem, Israel, May 2005.

[14] 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.

[15] 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*, Orlando, FL, November 2000.

[16] 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.

[17] 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)*.

[18] 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.

[19] 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*, July 2008.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

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

[27] 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.

[28] L. Zhang. Scheduling algorithm for real-time applications in grid environment. In *Proc. of IEEE International Conference on Systems, Man and Cybernetics*, October 2002.