Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 08-003

Exploring the Throughput-Fairness Tradeoff of Deadline Scheduling in Heterogeneous Computing Environments

Vasumathi Sundaram, Abhishek Chandra, and Jon Weissman

January 31, 2008

Exploring the Throughput-Fairness Tradeoff of Deadline Scheduling in Heterogeneous Computing Environments

Vasumathi Sundaram, Abhishek Chandra, and Jon Weissman
Department of Computer Science and Engineering
University of Minnesota, Twin Cities
{vassun,chandra,jon}@cs.umn.edu

Abstract

The scalability and computing power of large-scale computational platforms that harness processing cycles from distributed nodes has made them attractive for hosting compute-intensive time-critical applications. Many of these applications are composed of computational tasks that require specific deadlines to be met for successful completion. In scheduling such tasks, replication becomes necessary due to the heterogeneity and dynamism inherent in these computational platforms. In this paper, we show that combining redundant scheduling with deadline-based scheduling in these systems leads to a fundamental tradeoff between throughput and fairness. *We propose a new scheduling algorithm called* Limited Resource Earliest Deadline (LRED) *that couples redundant scheduling with deadline-driven scheduling in a flexible way by using a simple tunable parameter to exploit this tradeoff. Our evaluation of LRED using trace-driven and synthetic simulations shows that LRED provides a powerful mechanism to achieve desired throughput or fairness under high loads and low timeliness environments, where these tradeoffs are most critical.*

1 Introduction

Large-scale computational platforms such as Grid infrastructures [18] and cycle sharing systems [39, 1] have grown in popularity for running computationally intensive applications in areas spanning Bioinformatics [17], High Energy Physics [30], Climate prediction [15], etc. These systems provide scalability and enormous computational power by harnessing idle processing cycles from computing hosts distributed around the Internet. Their low deployment and operational cost in addition to their scalability has made these infrastructures attractive for hosting large-scale time-critical applications, for example, Biomedical applications such as medical image processing [8, 19], and Real-time MRI analysis [3].

Many of these applications are composed of computational tasks that require specific deadlines to be met for successful completion. These deadlines could result either from the time-constrained or real-time nature of the applications, or due to internal task dependencies, requiring some of the critical tasks to be finished in a timely manner. However, scheduling such time-constrained tasks in cycle sharing systems is challenging because of several reasons. First, the nodes in such a system are highly heterogeneous, with different CPU speeds, network connectivity, and load conditions. In addition, the capacity of individual nodes is also highly variable due to varying loads, fluctuating network bandwidth and churn. Such heterogeneity and dynamism makes it extremely difficult to select the right nodes to execute tasks in a timely manner. As a result, redundant scheduling [7, 21], where a task is assigned to multiple nodes to improve its chance of successful completion, is often employed in such systems.

However, the use of redundant scheduling creates a fundamental dilemma in choosing the right order of task scheduling. Giving preference to low deadline tasks, as is done by traditional deadline-based scheduling algorithms such as Earliest Deadline First (EDF) [27], results in consuming more resources,

since such tasks have more stringent deadlines and need more resources for their timely completion. On the other hand, ordering the tasks in decreasing order of their deadlines (we refer to this ordering as Latest Deadline First or LDF), while potentially providing better resource utilization, is likely to starve tighter deadline (and hence potentially more important) tasks. This dilemma can be understood as a *tradeoff between throughput and fairness* in the system: should a scheduler focus on successfully completing more tasks, or should it partition the available resources more equitably among tasks with different deadlines?

In this paper, we propose a new scheduling algorithm called *Limited Resource Earliest Deadline (LRED)* that is specifically designed to address this throughput-fairness tradeoff in such heterogeneous, dynamic computational environments. LRED couples redundant scheduling with deadline-driven scheduling in a flexible way to exploit this tradeoff. Intuitively, LRED works by limiting the number of resources consumed per task (thus improving throughput), while scheduling the selected tasks in earliest deadline order (thus improving fairness). An important feature of LRED is that it can achieve a desired throughput-fairness tradeoff using a simple tunable parameter.

The design of the LRED algorithm has resulted in the following key research contributions:

- We define a statistical notion of *timeliness* for a computational node which can incorporate both inter-node heterogeneity as well as intra-node dynamism, and provide a simple technique to estimate this timeliness based on the node's past execution history.
- LRED uses these timeliness values to couple redundant scheduling with deadline-driven scheduling in a seamless manner.
- LRED can achieve the desired throughput-fairness tradeoff in the system by using a tunable parameter to control the scheduling order of the tasks. LRED is a generalization of EDF and LDF, so that, by tuning this parameter, LRED reduces to EDF in one extreme, and to (a close variant of) LDF in another extreme.

We use trace-driven and synthetic simulations to evaluate the performance of LRED in a heterogeneous environment under different system conditions such as load and overall timeliness level of the system. Our results show that the load and the timeliness environment have a significant impact on the throughput-fairness tradeoff of task scheduling. We find that LRED provides a powerful mechanism to achieve desired throughput or fairness under high loads and low timeliness environments, where these tradeoffs are most critical.

2 System Model

In this section, we present our system model and define some of the concepts that will be used throughout the rest of the paper.

2.1 Task model

Our task model consists of a task pool with a set of homogeneous [1] tasks in terms of their computational requirements. These tasks are continuously created and submitted to the pool by an application. Each task T_i is associated with a *deadline* D_i that is defined as the time by which the task T_i must be completed[2]. The task is deemed as completed successfully, only if a result is computed within time D_i from the instant the task arrived in the system, otherwise, it is assumed to be failed. Different tasks can have different deadlines, either based on their time of arrival, or because of difference in importance or priorities (e.g., a task with large number of task dependencies may be assigned a shorter deadline).

2.2 Computational model

The computing environment consists of a set of heterogeneous worker nodes that provide their computational resources for executing the tasks. We assume a pull-based task scheduling model (commonly used in large-scale computational systems such as BOINC [1]), where each worker node requests work from a central scheduler and is assigned a task from

[1] Many of the applications we consider produce homogeneous tasks. We intend to explore heterogeneous tasks in the future.

[2] By task completion time, we mean the time at which the task result is returned back to the scheduler. We will use response time and task completion time interchangeably in the rest of the paper, unless otherwise noted.

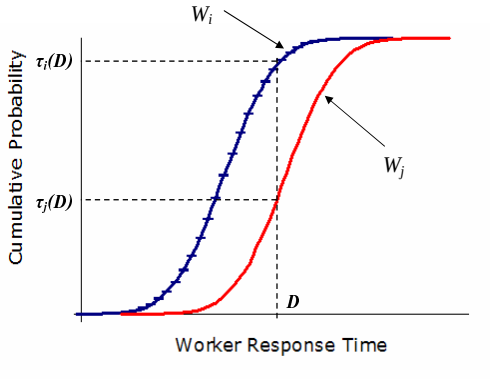


Figure 1. Computing the timeliness of two workers for a given deadline D.

the existing task pool. It then executes the task and return the results back to the scheduler. Different worker nodes can take different amounts of time to complete the same task due to differences in their computational capabilities (e.g., CPU speeds), network bandwidth, and load. Further, each worker may provide different response times for the same task during different periods, due to dynamic conditions such as varying loads and fluctuating network latency. Thus, we assume *heterogeneity across the worker nodes*, as well as *dynamism over the same node over time*, in terms of the completion time for a task.

2.3 Timeliness model

Based on the above worker and task model, we present a timeliness model that incorporates the inter-node heterogeneity as well as intra-node dynamism of the worker nodes for different task deadlines. We associate a response time distribution with each worker, which models the probability with which a worker is able to finish a task within a given amount of time. Such a distribution can be constructed based on a worker's past execution history. Using this distribution, we can estimate the likelihood that a worker will be able to meet a deadline.

Definition 1 Timeliness: *The timeliness $\tau_i(D)$ of a worker W_i for a task with deadline D is defined as the probability that the worker will be able to finish the task within time D.*

$$\tau_i(D) = CDF_i(D), \quad (1)$$

where CDF_i is the CDF of the worker's response time. Figure 1 illustrates this notion of timeliness.

Based on this definition, timeliness takes on a value between 0 and 1, where a timeliness value of 1 for a deadline D means that the worker node always returns within time D, while a value of 0 implies that the worker node will never return before D. Further, note that the timeliness of a worker depends on the deadline of a task, so that a worker will have a smaller timeliness value for more stringent deadlines. Finally, different workers will have different timeliness values for the same deadline based on their response time distributions. For example, in Figure 1, for the deadline D, the two workers W_i and W_j have different timeliness values $\tau_i(D)$ and $\tau_j(D)$ respectively.

2.4 Redundant Scheduling

Because of the worker heterogeneity and different ranges of task deadlines, it is possible that a task with a stringent deadline may have only a small probability of being successfully completed by any worker on its own. However, its success probability can be increased by redundantly allocating it to multiple workers. For instance, let us assume that a task T arrives with a deadline of 100 and there are two workers W1 and W2 with timeliness values of 0.8 and 0.6 respectively for this deadline. Now, if we want the task to be completed with a high probability, say 0.9, then the task cannot be scheduled to either of the workers individually and be expected to complete successfully with the desired probability. On the other hand, by assigning the task to both W1 and W2 and waiting for the first response, we increase the probability of successful task completion to 0.92, which meets our requirement.

We assume the use of such redundant scheduling in our system to achieve a desired *target success rate (TSR)* for each task. The target success rate can be defined as the desired probability with which each task must be completed within its deadline. TSR can be thought of as the overall task completion rate desired by an application. Note that a TSR of less than 1 may be acceptable for many applications in our target environment, largely because most of them use soft task deadlines with a possibility to re-execute a failed task in the worst case. Moreover, it may be infeasible to

achieve a TSR of 1 in an uncertain environment that we are considering.

The redundancy level required to satisfy a given TSR for a task is then determined by the task deadline and the timeliness of the available workers. In particular, it can be determined using the following notion of group timeliness of a group of workers for a task:

Definition 2 Group Timeliness: *The group timeliness $\tau_G(D)$ for a group G of workers $\{W_1, W_2, \ldots, W_n\}$ for a task with deadline D is defined as the probability of successful completion of the task within time D by at least one of the workers in G.*

$$\tau_G(D) = 1 - \prod_{i=1}^{n}(1 - \tau_i(D)) \quad (2)$$

Thus, to meet the TSR for a task with deadline D, a redundant scheduling algorithm would assign the task to a group of workers whose group timeliness value exceeds the TSR.

3 Combining Deadline-Driven and Redundant Scheduling

Having defined our system model and the notion of timeliness, we next examine the implications of using redundant scheduling for deadline-based tasks with respect to two key metrics of interest: *throughput* and *fairness*. In the context of deadline-based scheduling, we define throughput as the total number of tasks that are completed within their deadlines. Fairness can be defined as a measure of the share of the worker resources utilized for tasks with different deadlines. In other words, fairness can be thought as capturing the difference in the proportion of tasks completed for different deadlines. The smaller this difference is, the more fair is the scheduling algorithm. We measure the fairness of a schedule using Jain's fairness index [22][3].

Let us start by examining possible ways of combining deadline-driven scheduling with redundant scheduling. *Earliest Deadline First (EDF)* [27] is a classical scheduling algorithm used for deadline-driven scheduling. EDF always selects the task with the shortest deadline for execution. EDF has been shown to provide an optimal schedule for a uniprocessor environment as long as the tasks do not require more computing power than is available in the system. In other words, EDF can provide the highest throughput in such an environment. The following example illustrates how EDF will perform in our system model, in the presence of redundant scheduling.

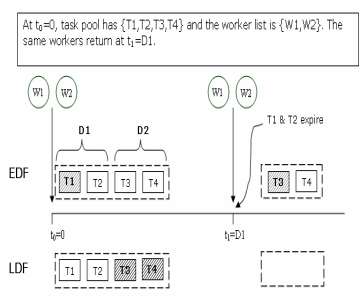


Figure 2. Comparison of LDF with EDF showing the throughput-fairness tradeoff

Example 1 *Consider a set of tasks T1, T2 and T3 in the task pool with successively higher deadlines, and a set of workers W1, W2, and W3. Let us assume that the deadlines of the tasks T2 and T3 are such that they can be successfully completed with a high probability (based on a given TSR) by any one of the workers. However, the deadline of task T1 is so stringent that it needs to be assigned to all three workers to have a high likelihood of timely completion. Since EDF selects the task with minimum deadline, it will schedule T1 to W1, W2, and W3, and ignore the other two tasks, resulting in a throughput of 1. Let us see what happens if we use a different scheduler, namely,* Latest Deadline First (LDF)*, that schedules tasks in decreasing order of their deadlines. LDF will assign tasks T2 and T3 to W2 and W3 respectively (assuming ties are broken arbitrarily), while the task T1 will be ignored, resulting in a throughput of 2.*

The above example illustrates the problem of naively coupling redundant scheduling with deadline-based scheduling. Redundant scheduling results in non-uniform resource requirements for different tasks. Since tasks with more stringent deadlines consume

[3]For ease of exposition, we defer the quantitative definition of Jain's fairness index to Section 5, and use fairness here in a more intuitive sense.

more resources, giving priority to such tasks will not always result in higher throughput for the system. The above example raises the question whether a scheduler like LDF is more desirable in heterogeneous and non-dedicated environment, as it is likely to achieve higher throughput. We examine this question using the following example.

Example 2 *Figure 2 shows an example scenario where at time 0, there are 4 tasks in the task pool: tasks T1 and T2 with deadline D1, and tasks T3 and T4 with deadline D2, where D2=2*D1. Further assume there are two workers W1 and W2 in the system at this time, each of which can finish a task with deadline D2 (with probability TSR), but both of them need to be grouped together to finish a task with deadline D1. With this setup, at time 0, EDF will use both workers to schedule task T1, while LDF will assign one worker each to tasks T3 and T4.*

Now, suppose both workers successfully finish their tasks and come back at time D1. At this point, the deadlines of T1 and T2 would have expired, while the effective deadlines of T3 and T4 would have reduced to D1 due to passage of time. In this case, EDF will use both workers to schedule task T3 (since T2 has already missed its deadline), while LDF will have no remaining task to schedule (as T1 and T2 have already missed their deadlines).

Thus, over the two scheduling instances, both EDF and LDF achieve the same throughput of 2. However, while EDF schedules one task for each deadline, LDF schedules both higher deadline tasks, while starving the lower deadline tasks. We can see that EDF provides more fairness for this case which can also be verified using Jain's fairness index [22] which gives a fairness value of 1 for EDF and 0.5 for LDF (higher is fairer).

This example shows that while LDF may have equal or better throughput than EDF in general, it suffers from higher unfairness because of its bias towards longer deadline tasks, at the expense of starving short deadline tasks. Note that even though EDF is biased towards shorter deadline tasks, it is also likely to schedule higher deadline tasks, as their deadlines become more stringent with the passage of time. On the other hand, by preferring more lax deadline tasks initially, LDF further decreases the possibility of eventually executing shorter deadline tasks in the future.

Based on the above examples, we see that in a heterogeneous, dynamic resource environment, there is a clear tradeoff between throughput and fairness, when coupling redundant scheduling with deadline-driven scheduling. We also see that EDF and LDF represent the opposite ends of the spectrum in this tradeoff, with one ensuring higher fairness while the other achieves higher throughput. We next use this insight to propose an algorithm that provides a flexible way to exploit this tradeoff.

4 Limited Resource Earliest Deadline Scheduling

In this section, we present *Limited Resource Earliest Deadline Scheduling (LRED)*: a general deadline-driven scheduler that explicitly incorporates redundant scheduling, and thus provides a flexible way to exploit the throughput-fairness tradeoff in a heterogeneous, dynamic computation system. This algorithm is based on the following observations from the last section:

- Shorter deadline tasks consume a larger number of resources in general than higher deadline tasks, and
- Shorter deadline tasks are more likely to fail with the passage of time, as compared to higher deadline tasks.

These observations lead to the following key insights:

- **Insight 1:** A task requiring more nodes for its successful completion will lead to a lower throughput.
- **Insight 2:** Scheduling a shorter deadline task before a higher deadline task is likely to achieve higher fairness.

LRED uses these insights to exploit the throughput-fairness tradeoff. We first present the high-level intuition behind the algorithm, followed by the key concepts used by this algorithm, and then describe the algorithm's working in detail.

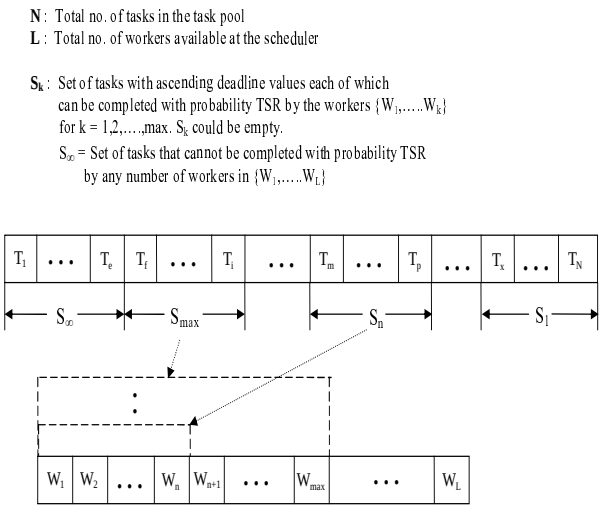


Figure 3. Partitioning of the task list by LRED

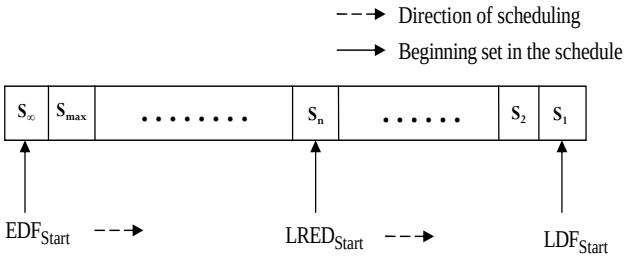


Figure 4. Scheduling by LDF, EDF and LRED

4.1 Intuitive Description of LRED

Intuitively, LRED works by limiting the number of resources consumed per task (thus improving throughput), while scheduling the selected tasks in earliest deadline order (thus improving fairness). To achieve this goal, LRED sorts the task pool in increasing order of deadlines, so that shorter deadline tasks require more resources compared to higher deadline tasks. Then, LRED schedules the tasks in earliest deadline first order *starting* from the first task that needs a (specified) limited number of resources. This limit is specified as a parameter to the algorithm, and allows LRED to control the throughput-fairness tradeoff. For instance, specifying a limit of 3 would result in LRED begin scheduling from the shortest deadline task that needs only 3 nodes, resulting in a higher throughput but lower fairness than specifying a limit of 7. Next we describe the key concepts used by the algorithm and its working in more detail.

4.2 Key Concepts

Consider a set of N workers and L tasks in the system. Let us assume that the task list is sorted in increasing order of task deadlines. Further, let us assume that the worker queue is sorted in decreasing order of the mean timeliness value τ of workers[4]. Then, we can define the following:

Definition 3 k-dependent task: *A task is said to be k-dependent if it needs exactly the k most timely workers in the worker queue to complete successfully with a high probability (based on a desired target success rate)*[5].

Definition 4 k-dependent task set (S_k): *The set of all k-dependent tasks in the task queue.*

Figure 3 illustrates the concept of k-dependent tasks and sets. As seen from the figure, tasks T_m to T_p require the top n workers from the worker queue for their successful execution, and thus are n-dependent tasks, and belong to the set S_n. Similarly, tasks T_f to T_i belong to the set S_{max}, while tasks T_x to T_N belong to the set S_1. Note that the set S_∞ represents the set of tasks that cannot be successfully completed with any number of workers from $\{W_1, \ldots, W_L\}$, and are thus infeasible with the current worker pool.

In this way, the task list is divided into disjoint k-dependent sets for $k = 1, 2, \ldots, max$, where max is the maximum number of workers required by any task in the task list. Note that the size of one of these sets S_k could be zero, which means that there may be no tasks that can be completed with exactly k most timely workers in the worker queue. Also, it can be shown that each set S_k consists of consecutive tasks from the sorted task list, and all tasks in a set S_n have lower deadlines than the tasks in a set S_m, for $n > m$. These

[4]Note that based on the shapes of the timeliness distributions, a worker with a higher mean timeliness value need not always have a higher timeliness for a given deadline value D, compared to a worker with a smaller mean timeliness value. We use this ordering mainly as a heuristic.

[5]From here on, we will assume successful completion to be dependent on a given TSR, and omit its mention unless required specifically.

properties are based on the following lemma, whose proof is given in the Appendix.

Lemma 1 *If T_i and T_j are two tasks such that $D_i \leq D_j$ and $T_i \in S_n$, $T_j \in S_m$, then $m \leq n$.*

Proof:

$T_i \in S_n \Rightarrow \exists G = \{W_1, \ldots, W_n\}$ such that $\tau_G(D_i) \geq TSR$, where $\{W_1, \ldots, W_n\}$ are the first n workers in the sorted worker queue.

Now, using Equation 2 in Section 2.4, the timeliness of G for T_j is given by:
$\tau_G(D_j) = 1 - \prod_{l=1}^{n}(1 - \tau_l(D_j))$.

From Equation 2,
$D_j \geq D_i \Rightarrow \tau_l(D_j) \geq \tau_l(D_i)$ for each worker $W_l \in G$

$$\Rightarrow (1 - \tau_l(D_j)) \leq (1 - \tau_l(D_i))$$
$$\Rightarrow \tau_G(D_j) \geq \tau_G(D_i) \geq TSR$$

Thus, T_j does not need more than n workers to satisfy TSR, and hence, $m \leq n$.

4.3 Algorithm Description

We now describe how the LRED algorithm works in practice. We begin by describing how EDF and LDF would schedule tasks based on the concepts of k-dependent tasks and k-dependent sets presented above.

EDF schedules tasks in the increasing order of their deadlines, so that it will start with the tasks in S_∞ as shown in Figure 4. However, since none of the tasks in S_∞ can be successfully completed by the available workers, EDF will skip these tasks. Thus, EDF will effectively start scheduling from the tasks in S_{max}. Since it tends to consume larger number of nodes per task, it is likely to achieve lower throughput due to Insight 1, but because of its ordering of tasks, it achieves higher fairness due to Insight 2.

LDF, on the other hand, schedules the tasks in the decreasing order of their deadlines, so it will begin by scheduling the longest deadline task in S_1, as shown in Figure 4. LDF will schedule tasks in S_2 only if S_1 is empty, and continue moving towards S_∞ as each successive set becomes empty. It, however, will not schedule any task in S_∞ since none of these tasks can be executed with the available workers. As we illustrated in Examples 1 and 2 in Section 3, this ordering is likely to result in higher throughput due to Insight 1, but in lower fairness due to Insight 2.

LRED generalizes the scheduling orders of EDF and LDF by introducing a new set pointer LRED$_{Start}$, which refers to the first k-dependent set S_k that the algorithm will schedule tasks from. By using S_k as the starting set pointer, we limit the number of workers to schedule a task to k, hence, we call it a *Limited Resource* algorithm. Once the tasks in the initial set S_k are exhausted, the algorithm moves on to the next k-dependent set with a smaller value of k. The values of the pointer LRED$_{Start}$ can be seen to be S_1 and S_∞ respectively for LDF and EDF, as shown in Figure 4. The fairness of an LRED schedule can be increased from that of LDF if LRED$_{Start}$ is set to S_2 instead of S_1, but throughput will reduce accordingly. Similarly, the throughput of LRED can be increased but fairness reduced from that of EDF by setting LRED$_{Start}$ to S_{max-1} instead of S_{max} (or S_∞). In this manner, the fairness and the throughput of the system can be adjusted by sliding the pointer LRED$_{Start}$ along the task queue.

Besides the choice of the starting k-dependent set S_k, the other question is the ordering of the tasks within S_k. Using Insights 1 and 2, we can achieve higher fairness without sacrificing throughput by traversing S_k in the increasing order of deadlines. LRED also uses this ordering, referring to the *Earliest Deadline* part of the name. Note that because of this ordering, setting LRED$_{Start}$ to S_1 will differ from a pure LDF algorithm (achieving higher fairness and throughput than LDF, as we will show in Section 5).

To summarize, LRED initializes the task set pointer LRED$_{Start}$ to start from a set S_n to obtain a particular level of throughput and fairness. The shortest deadline task from this set S_n is chosen and scheduled to the first group of n most timely workers from the worker list. S_n is traversed in the increasing order of task deadlines, and once S_n becomes empty, a task from the next non-empty set in the list $\{S_{n-1}, S_{n-2}, \ldots, S_1\}$ is chosen for scheduling. To make the algorithm work-conserving, once all tasks in the sets $\{S_n, S_{n-1}, \ldots, S_1\}$ are exhausted, the algorithm moves to tasks in S_{n+1}, and continues moving towards S_∞ as each successive set becomes empty.

Algorithm 1 shows the pseudocode for LRED. It takes a parameter n which corresponds to the set S_n to be used as the set pointer LRED$_{Start}$. The basic algorithm works by scheduling the group of the n most

Algorithm 1 LRED(n)

```
1: W ←Set of all available workers
2: Sort W in decreasing order of τ
3: Sort the task pool in increasing order of D
4: while W is non-empty do
5:    Organize the task pool into the list {S_1, S_2, ..., S_max} based on τ of workers in W
6:    V ← Set of all tasks in the list {S_n, ..., S_2, S_1}
7:    if V is non-empty then
8:       T ← First task from the first non-empty set S_k in V
9:       Schedule T to k most timely workers
10:      Update W by removing the k assigned workers
11:   else if n < max then
12:      LRED(n+1)
13:   else
14:      break
15:   end if
16: end while
```

timely workers among the available worker list to the shortest deadline task T in S_n. The value of n signifies which task among all the tasks in the task pool will be chosen to be scheduled first. When n = 1, the execution of LRED(1) corresponds to LDF (except for the ordering of tasks within S_1). When n = max (or ∞), it schedules tasks from S_{max} until either S_{max} becomes empty or all the capable workers are exhausted before moving on to S_{max-1}. This corresponds to an execution of EDF. Also, to make the algorithm work-conserving, once it exhausts all tasks in the sets S_k, for $k = 1, \ldots, n$, it recursively calls LRED(n+1).

	Task pool	Deadline	LRED(2)	LRED(1)
S_2	T_1	D_1	$T_1 \rightarrow \{W_1, W_2\}$	
	T_2	D_1		
	T_3	D_1		
S_1	T_4	D_2	$T_4 \rightarrow \{W_3\}$	$T_4 \rightarrow \{W_1\}$
	T_5	D_2		$T_5 \rightarrow \{W_2\}$
	T_6	D_2		$T_6 \rightarrow \{W_3\}$

Figure 5. An example schedule by LRED(n) for values of n = 1, 2. L = 6, N = 3

Figure 5 gives an illustration of the schedule created by the LRED(n) algorithm for values of n = 1, 2. The figure shows that a higher value of n=2 produces a lower throughput but completes more short deadline tasks, whereas with a smaller value of n=1, short deadline tasks starve while increasing the net throughput. We next provide detailed quantitative evaluation of this algorithm using a simulation study.

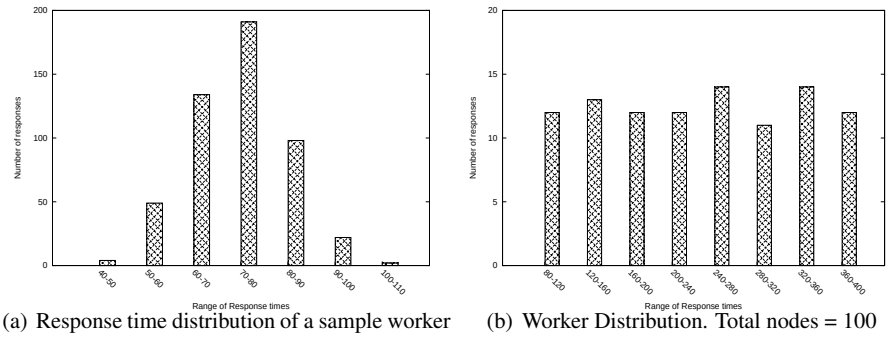
5 Evaluation

We now evaluate the performance of the LRED algorithm through simulation. We begin by describing our simulation methodology, followed by a definition of the metrics used for evaluation and the results obtained from the simulations.

5.1 Simulation Methodology

The simulator consists of a central task scheduler and a set of workers that arrive at the scheduler requesting work. Each worker is associated with an underlying response time distribution, which is sampled to generate a response time for each task it is assigned. The scheduler maintains a database of timeliness behavior information for all the individual workers by observing the past history of response times of each worker. The timeliness distribution of each worker is then represented by a histogram of its observed response times for the tasks assigned to it in the past. A sample worker histogram is shown in Figure 6(a). This is used to calculate the timeliness of the worker with respect to a given task deadline D, while the group timeliness of a group of workers is computed as per Equation 2 presented in Section 2.4. The length of the maintained history is bounded to accommodate recent changes in the worker's timeliness behavior into the estimated timeliness. The histogram of each worker is updated by the scheduler whenever a worker returns a result for an assigned task. All the workers are initially assumed to be available at the scheduler, after which the workers return to the scheduler at time intervals based on task response times sampled from their underlying response time distributions.

To emulate the timeliness behavior of individual nodes, we have used a synthetic trace as well as a PlanetLab trace. The synthetic trace consists of the worker response time distributions emulated by normal distributions, with equal standard deviation and with the distribution means uniformly distributed over a specific deadline range. Figure 6(b) shows the distribution of mean worker timeliness values ranging from



(a) Response time distribution of a sample worker with mean = 73

(b) Worker Distribution. Total nodes = 100

Figure 6. Representing a single worker and all workers in the environment

80-400 time units. We have also used a trace from PlanetLab that contains the response times of 90 PlanetLab nodes running for 5 hours executing same size tasks, collected during a live execution of BLAST over BOINC [1]. BLAST is a bio-informatics application that performs genetic matching of an input DNA sequence against a gene sequence database. The PlanetLab trace and results based on this trace are presented in detail in Section 5.7.

In our simulations, fixed-size tasks are generated with deadlines uniformly distributed in a given deadline range. We vary the task deadline range to emulate different overall timeliness levels of the computational environment for a given worker distribution. For instance, we use a task deadline range towards the lower end of the response time range of the workers, to represent a low timeliness environment (LowTE). Similarly a high (HighTE) or moderately timely environment (ModTE) is simulated by moving the task deadline range over the worker timeliness range accordingly. The inter-arrival times for new tasks arriving into the system follow an exponential distribution. The mean of this inter-arrival time distribution is varied to vary the load level in the system, so that a low mean value corresponds to a high volume of tasks arriving in the system.

For all our experiments, we use two additional parameters: *MaxWkrs* and *Waiting time(w)*. $MaxWkrs$ is defined as the maximum number of workers in any group G a task can be replicated to run on. This parameter is used to avoid over-consumption of resources for executing a single task. *Waiting time* is defined as the maximum amount of time the scheduler waits between its scheduling decisions. If w is 0, it means that a scheduling decision is made every time a worker arrives at the scheduler requesting work. By setting w greater than 0, it is possible to have a larger number of workers available to the scheduler to enable better scheduling decisions. We discuss the significance and the impact of the waiting times in more detail later.

In all of our experiments, we have set the parameters $MaxWkrs$ = 7 (i.e., the maximum group size that can used to schedule a task) and TSR = 0.9. We executed the experiments for LRED(n) for values of n = 1, 4, 7 for each timeliness environment under different load conditions. We execute the EDF and LDF algorithm under the same conditions for comparison. In addition, we also simulated a Random algorithm (*Rand*), that selects tasks randomly in arbitrary order from the task pool, while keeping the worker list sorted.

5.2 Metrics

- *Throughput* : The total number of tasks that are completed within their deadlines.

- *Fairness* : We measure the fairness of a schedule using Jain's fairness index [22] as follows. Suppose the deadline range of the tasks is divided into m bins s.t. C_i is the number of tasks in bin i and X_i is the number of tasks successfully completed

in bin i, then the fairness index FI is given by:

$$FI = \frac{\left[\sum_{i=1}^{m} x_i\right]^2}{m \sum_{i=1}^{m} {x_i}^2} \qquad \text{where} \quad x_i = \frac{X_i}{C_i} \qquad (3)$$

The value of the fairness index ranges from 0 to 1, with a value of 1 representing absolute fairness in the system while 0 indicates absolute unfairness.

5.3 Throughput-Fairness Tradeoff

We start by presenting results for the synthetic worker timeliness distribution trace. Figures 7(a) and 7(b) show the tradeoff between fairness and throughput for a low timeliness environment (LowTE). In this environment, the task deadlines lie in the range 80-150, which makes only about 25% of the workers timely for these tasks. The load is kept high with a mean task arrival time of 5. The waiting time w is fixed at 2, which means that the scheduler will allocate tasks to workers every 2 time units. This waiting time gives a sufficient number of workers at each scheduling point.

Figures 7(a) and 7(b) plot the fairness index FI and throughput respectively for the different scheduling algorithms. As expected, the fairness of LRED increases as n increases, while throughput decreases as shown in the figures. EDF shows the highest fairness and lowest throughput. LDF has the lowest fairness, however, its throughput is lower than that of LRED(1), which demonstrates the benefit of scheduling tasks in the increasing order of deadlines within a k-dependent set (S_1 in this case). Rand shows slightly higher fairness and lower throughput than LDF because it happens to schedule a greater number of lower deadline tasks than LDF due to the randomness in choosing tasks. EDF does not show any dramatic improvement over LRED(7), because the majority of the tasks that could be finished with unlimited number of workers needed only a size of 7 at maximum.

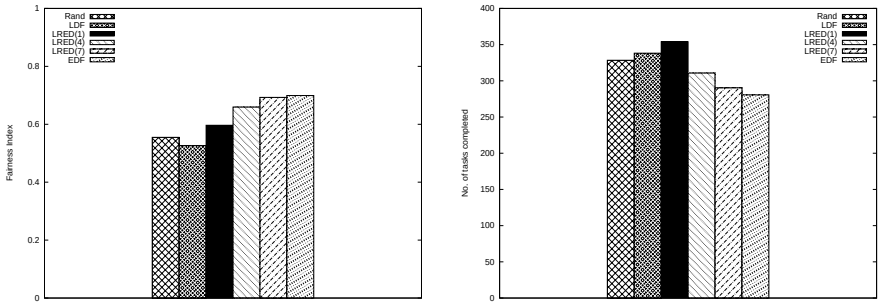
To understand these results better, Figure 8(a) shows the ratio of tasks completed in each deadline bin by the different algorithms. The fairness level of an algorithm is indicated by how flat its curve is. As seen from the figure, LRED(7) could finish more low deadline tasks than that of LRED(1). In addition, if there are no timely workers to schedule the shortest deadline tasks, it finishes some tasks from higher deadline bins requiring smaller group sizes than 7. This gives LRED(7) a flatter curve and consequently higher fairness.

This result can be verified by Figure 8(b) which shows the ratio of completed tasks grouped by the k-dependent sets S_k to which they belonged at scheduling time. As seen from the figure, LRED(1) shows a heavy bias towards tasks from S_1 (high deadline tasks), while the tasks scheduled by LRED(7) are more equally distributed across the various sets. For instance, if we consider the sets S_1 and S_3, LRED(1) finishes 67% of the tasks in S_1 as opposed to only 4% of the tasks in S_3. On the other hand, LRED(7) finishes 30% of the tasks in S_1 and 18% in S_3, showing a smaller deviation than LRED(1). This explains the high fairness for LRED(7) compared to LRED(1). Yet, the throughput is lower than LRED(1) since it effectively finishes less number of tasks from all the sets. Also, we see that none of the algorithms could do many tasks from $\{S_5, S_6, S_7\}$ because of the lack of timely workers in the LowTE.

5.4 Impact of Load

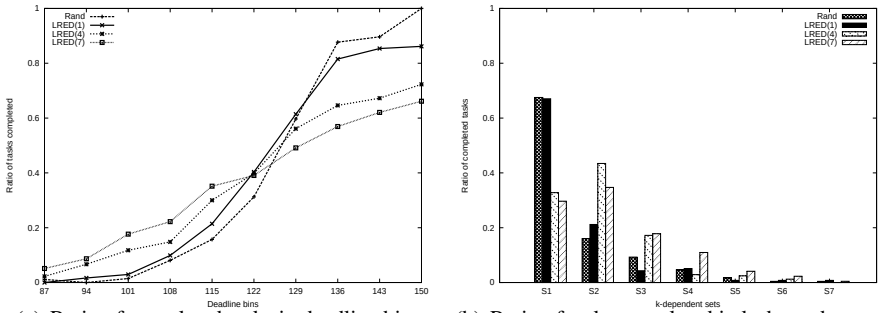
We next explore the effect of system load on the performance of LRED for different n values for LowTE. We varied the task arrival times and observed the resulting fairness and throughput for each of n=1,4,7, and for Rand in Figures 9(a) and 9(b). A high mean task arrival time represents a light load condition while a low mean value creates a heavy load condition.

LRED(1) has the lowest fairness and highest throughput when the system load is high, because tasks arrive at a faster rate and keep LRED(1) busy scheduling along the task list with lower k-dependent sets. So, most tasks use smaller group sizes and consequently the throughput is higher. However, LRED(1) has fewer chances to move to higher k-dependent sets due to the higher task arrival rate and consequently exhibits less fairness compared to LRED(7) and LRED(4). On the other hand, as the mean task arrival rate increases, reducing load on the system, the fairness and throughput of LRED for all values of n converges, because now the number of waiting tasks is small enough that most tasks can be picked up by



(a) Fairness (b) Throughput

Figure 7. Comparing FI and Throughput of LDF and Rand with LRED for n=1,4,7,∞ in a LowTE



(a) Ratio of completed tasks in deadline bins (b) Ratio of tasks completed in k-dependent sets

Figure 8. (a) LRED performing in each deadline bin. LRED(1) completes more tasks in higher bins while LRED(7) performs better in lower bins (b) LRED performing in each k-dependent set. LRED(1) spends more on S_1 while LRED(7) is more evenly spread across majority of the sets

the workers irrespective of their deadlines. We also see that the total throughput decreases with decreasing load due to fewer tasks arriving in the system.

5.5 Impact of Timeliness Environment

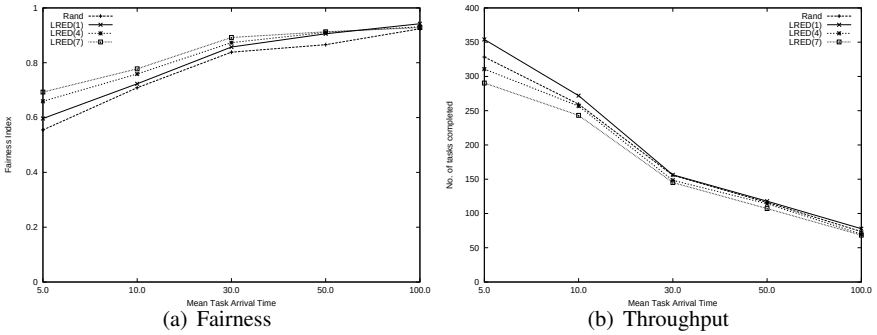
We plot the impact of the underlying timeliness environment on the performance of LRED, Rand, LDF and EDF in Figures 10(a) and 10(b). The timeliness of the environment is increased by moving the task deadline ranges towards the higher range of the worker timeliness in the distribution Figure 6(b). LowTE, ModTE and HighTE correspond to the task deadline ranges of 80-150 ($<25\%$ workers are timely), 120-200 (approximately 40% workers are timely) and 200-350 ($>75\%$ are timely) respectively.

We make two main observations from these figures regarding the behavior of LRED. First, the figures show that as the overall timeliness level of the environment increases, the fairness as well as the throughput values converge for all values of n. Secondly, we also see that as the timeliness level gets higher, both fairness and throughput increase for the same value of n.

These results occur because as the timeliness level increases, smaller groups of workers are required to successfully complete most of the tasks. Evidence for this is seen from Table 1 which shows the maximum group size required for the shortest deadline task for

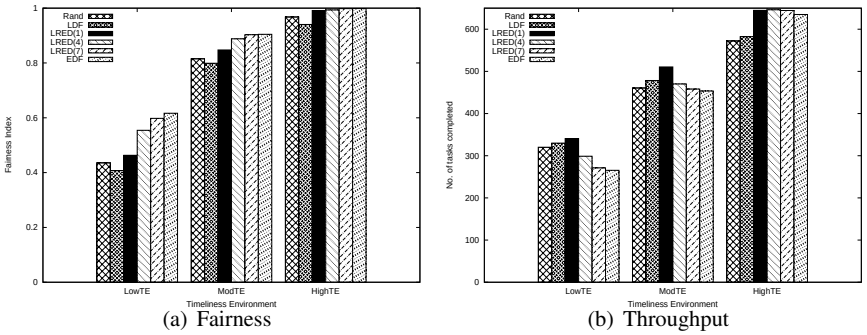
TE	n
LowTE	5.2
ModTE	3.1
HighTE	1.1

Table 1. Maximum n needed for the shortest deadline task scheduled



(a) Fairness (b) Throughput

Figure 9. Impact of load on fairness and throughput of LRED in a LowTE



(a) Fairness (b) Throughput

Figure 10. Fairness-Throughput of LRED for different Timeliness environments with a fixed load

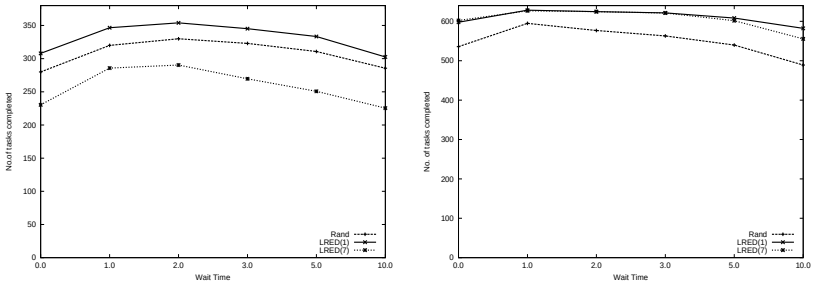
each timeliness environment. As seen from the table, this value decreases from 5.2 to 1.1 as we go from LowTE to HighTE. This implies that while most tasks are spread between S_1 to S_6 for LowTE, most of the tasks are concentrated in S_1 only for HighTE. As a result, for HighTE, there is less differentiation between the different values of n for LRED(n) resulting in similar throughput and fairness. Moreover, since most tasks require only 1 worker for successful execution in HighTE, the overall throughput and fairness also increase.

Rand and LDF, however, show poor fairness as well as throughput as compared to LRED even in HighTE, due to the poor choices they make for each task. They both might assign highly timely workers to very high deadline tasks leaving the short deadline tasks to be assigned to larger sized groups, reducing the overall throughput. It affects fairness due to the starvation of the short deadline tasks for the same reason.

5.6 Significance of waiting time

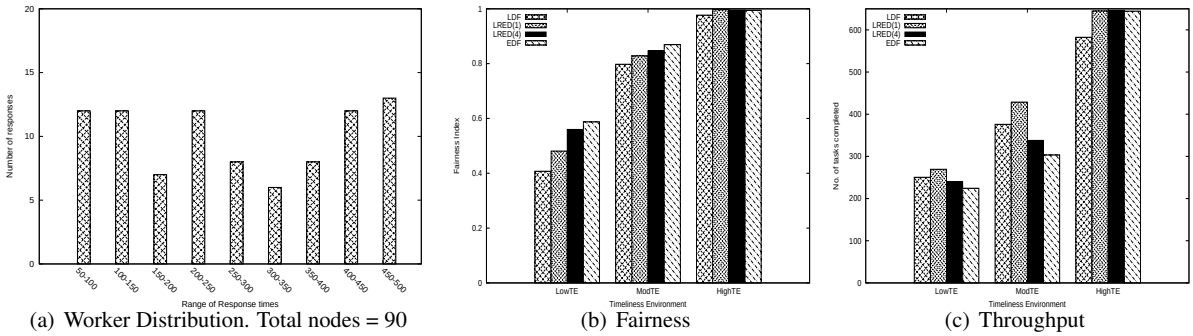
Since the scheduler needs to wait for workers to arrive before making scheduling decisions, we conducted experiments to see how long we can wait without affecting throughput. We used a high load setting in a LowTE and HighTE, and varied the waiting times. The results are shown in Figure 11.

Figure 11(a) shows the impact of waiting time on the throughput in a LowTE. As seen from the figure, the throughput initially increases as we increase the waiting time, but then it starts decreasing after a point. This is because, with no waiting, we may not have enough timely workers to form good groups, since most of the workers in this environment have poor timeliness. This is clearly shown by the low throughput at w=0. If we wait for some amount of time (in this case until w=2), we get a few more timely workers to form better groups so that more tasks can be finished. However, if we wait too long, the deadlines of the pooled tasks drop too much to be successfully



(a) LowTE (b) HighTE

Figure 11. Impact of waiting time on the throughput of LRED and Rand



(a) Worker Distribution. Total nodes = 90 (b) Fairness (c) Throughput

Figure 12. LRED for different TEs for PlanetLab traces

completed in time.

Figure 11(b) shows that the drop in throughput happens at a much lower waiting time value (with only a slight increase from w=0 to w=1). This is because, with HighTE, since the workers already have good timeliness without waiting, we can achieve a good throughput. Increasing the waiting time only results in reducing the deadlines of waiting tasks without much benefit in getting better workers.

Overall, waiting for some time may enable the arrival of better workers to result in better scheduling decisions, however, waiting too long could expire task deadlines. Therefore, these two effects have to be balanced based on the environment.

5.7 Results with PlanetLab Trace

In addition to the synthetic worker distributions, we conducted some experiments with a worker trace from PlanetLab. The trace contains the response times of 90 PlanetLab nodes running equal-sized tasks, recorded over a duration of 5 hours to capture sufficient timeliness data about all the workers. The worker distribution with their mean response times is shown in Figure 12(a). We used these traces to initialize the workers and ran experiments with LRED and LDF for LowTE, ModTE and HighTE with the task deadline ranges of 60-160 (approximately 20% of workers are highly timely), 130-230 (approximately 40% of workers are timely) and 200-280 ($>70\%$ are timely) respectively. The mean task arrival time was fixed at 5 representing high load and the waiting time w set to 2. The results are summarized in Figures 12(b) and 12(c). The observations are similar to that for synthetic worker distributions. LRED(1) leads in fairness while LRED(7) leads in throughput for LowTE and ModTE. However, all LRED(n) show the same performance for HighTE while LDF still lags behind.

5.8 Summary of Results

We summarize the major results below.

- Reducing the value of n with LRED provides better throughput because it uses fewer workers per

task.

- Using LRED with higher values of n produces higher fairness, as the workers are now more spread out across different deadline tasks.
- When the load on the system is high, the throughput-fairness tradeoff is more visible than at low loads.
- The throughput-fairness tradeoff also becomes more prominent as the overall timeliness level of the environment decreases.

Some of the key conclusions we can draw from these results are as follows. When a system is heavily loaded in terms of task arrivals or the timeliness level is low (so that there are more stringent tasks or less timely workers in the system), there is a greater opportunity to exploit the tradeoff between throughput and fairness to suit the system requirements. LRED provides a way to tune this tradeoff by simply increasing or decreasing its group size parameter. When the system load reduces or the timeliness level increases, however, simple EDF can be used because a good throughput-fairness combination can be obtained irrespective of any limit on the group size. Note that the ability of LRED to achieve flexibility under high loads and low timeliness makes it powerful as these system conditions are more likely to impact an application's performance.

6 Related Work

Deadline-based scheduling: There is a large body of work on deadline-based scheduling [14]. Earliest-deadline-first (EDF) [27] is a commonly used deadline-based online scheduling algorithm that has been shown to be optimal for uniprocessor systems as long as the demand does not exceed the system capacity. Many ideas have been proposed to ensure the optimality of deadline scheduling under overloads [6, 24], while variations of EDF such as least slack first [26] have also been proposed over the years. In this paper, we have looked at the problem of combining deadline-based scheduling with redundant scheduling in a heterogeneous environment, and our main focus is on providing probabilistic guarantees for soft deadline tasks.

Fairness in scheduling: Fairness has been emphasized by many scheduling algorithms in allocating processor bandwidth to applications in both uniprocessor as well as multiprocessor systems. Many systems use proportional share allocation based on Generalized Processing Sharing [29], which is an ideal algorithm in which each application is allocated bandwidth based on the weight assigned to the application. Many other algorithms [16, 20, 34, 13, 10] have been proposed that approximate GPS in different domains. Fairness has also been widely investigated in queuing systems [2, 32, 31, 4]. Adam Wierman et al. in [37, 36] provide a classification of scheduling policies with respect to fairness. Our focus is on exploring the tradeoff between fairness and throughput in the context of deadline-driven scheduling. Also, we focus on a heterogeneous system with dynamic resource availability.

In the context of scheduling tasks with constraints (e.g. deadlines), [5] introduced the concept of Pfairness. Under Pfair scheduling, tasks are scheduled according to a fixed-size allocation quantum so that deviation from an ideal allocation is strictly bounded. We focus on a heterogeneous multi-resource environment with probabilistic requirements.

Deadline scheduling in large-scale systems: One of the earlier works that proposed an on-line deadline scheduling algorithm for client-server Grid systems is [35]. Caron et al. [11] improved this work by associating a priority with each task. In [23], the authors propose a fair scheduling algorithm for wireless networks that ensures packets are dropped fairly among users in case of missed deadlines. In this paper, our mechanism provides a way to control the fairness among the task deadlines.

Redundant scheduling: Redundancy has been used in several contexts such as Byzantine fault tolerant systems [25, 12]. Redundancy has also been widely employed by data storage systems to ensure high availability, performance and fault tolerance [7, 21]. Here, we use redundancy for achieving robust scheduling guarantees.

Reputation-based scheduling: There are a number of papers that have proposed the use of reputation to store the trust and reliability values of nodes for use in scheduling [38, 33, 9]. However, many of these systems do not consider task deadlines which creates a second dimension in the reputation computation,

as shown by our use of node timeliness distribution, rather than each node having a fixed reputation across all tasks.

Scheduling in heterogeneous environments: Maheswaran et al. [28] study heuristics similar to EDF for scheduling independent tasks in heterogeneous computing systems, with the objective of maximizing throughput. Our work is on scheduling independent tasks with deadline constraints, with additional requirements about fairness.

7 Conclusion

In this paper, we examined the problem of deadline-driven task scheduling in a heterogeneous and dynamic computational environment. We showed that combining deadline-based scheduling with redundant scheduling typically used in such systems leads to a fundamental tradeoff between throughput and fairness. In particular, we showed that earliest deadline first (EDF) results in lower throughput because the tasks with stringent deadlines consume more resources, while at the same time ensuring a higher fairness. However, while latest deadline first (LDF) provides better resource utilization, it is likely to starve stringent deadline tasks.

To exploit this tradeoff in such heterogeneous and dynamic environments, we proposed a new scheduling algorithm called Limited Resource Earliest Deadline (LRED) that couples redundant scheduling with deadline-driven scheduling in a flexible way by using a simple tunable parameter. This algorithm uses a statistical notion of timeliness for each computational node that captures both inter-node heterogeneity as well as intra-node dynamism in the system, and can be estimated based on the node's past execution history. Further, we showed that LRED is a generalization of EDF and LDF, so that, by tuning its parameter, LRED reduces to EDF in one extreme, and to (a close variant of) LDF in another extreme.

We used trace-driven and synthetic simulations to evaluate the performance of LRED. Our results show that load and the timeliness level of the underlying environment have a significant impact on the throughput-fairness tradeoff of task scheduling. We find that LRED provides a powerful mechanism to achieve desired throughput or fairness under high loads and low timeliness environments, where these tradeoffs are most critical.

In the future, we intend to implement this algorithm on a live system such as PlanetLab to observe the tradeoff when the nodes change in behavior abruptly during the course of the experiments. In addition, we plan to extend our algorithm to incorporate heterogeneous tasks as well.

References

[1] D. Anderson. BOINC: A System for Public-Resource Computing and Storage. *IEEE/ACM GRID*, 2004.

[2] B. Avi-Itzhak and H. Levy. On measuring fairness in queues. *Advances of Applied Probability*, 2004.

[3] E. Bagarinao, L. F. G. Sarmenta, Y. Tanaka, K. Matsuo, and T. Nakai. Real-Time Functional MRI Analysis Using Grid Computing. *High Performance Computing and Grid*, 2004.

[4] N. Bansal and M. Harchol-Balter. Analysis of SRPT Scheduling: Investigating Unfairness. *ACM SIGMETRICS*, 2001.

[5] S. Baruah, N. Cohen, C. Plaxton, and D. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996.

[6] S. Baruah, G. Koren, B. Mishra, A. Raghunathan, L. Rosier, and D. Shasha. Online Scheduling in the Presence of Overload. *Foundations of Computer Science*, 1991.

[7] R. Bhagwan, K. Tati, Y.-C. Cheng, S. Savage, and G. Voelker. Total Recall: System Support for automated availability management. *NSDI*, 2004.

[8] P. Bonetto, M. Guarracino, and F. Inguglia. Integrating Medical Imaging into a Grid Based Computing Infrastructure. *Computational Science and Its Applications - ICCSA*, 3044:505–514, Apr 2004.

[9] K. Budati, J. Sonnek, A. Chandra, and J. Weissman. RIDGE: Combining Reliability and Performance in Open Grid Platforms. *HPDC*, Jun 2007.

[10] B. Caprita, W. Chan, J. Nieh, C. Stein, and H. Zheng. Group Ratio Round-Robin: O(1) Proportional Share Scheduling for Uniprocessor and Multiprocessor Systems. *USENIX Annual Technical Conference*, Apr 2005.

[11] E. Caron, P. K. Chouhan, and F. Desprez. Deadline Scheduling with Priority for Client-Server Systems on the Grid. *ACM GRID*, Nov 2004.

[12] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. *OSDI*, Feb 1999.

[13] A. Chandra, M. Adler, P. Goyal, and P. Shenoy. Surplus Fair Scheduling: A Proportional-Share CPU Scheduling Algorithm for Symmetric Multiprocessors. *USENIX OSDI*, 2000.

[14] S.-C. Cheng, J.-A. Stankovic, and K. Ramamritham. Scheduling Algorithms for Hard Real-Time Systems: A Brief Survey. *Tutorial: Hard Real Time Systems*, 1989.

[15] Climate Prediction Network. `http://www.climateprediction.net/`.

[16] A. Demers, S. Keshav, and S. Shenkar. Analysis and Simulation of a Fair Queuing Algorithm. *Journal of Internetworking Research and Experience*, 1990.

[17] Folding@home distributing computing project. `http://folding.stanford.edu`.

[18] I. Foster and C. Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kauffman, CA, USA, 2004.

[19] C. Germain, V. Breton, P. Clarysse, Y. Gaudeau, T. Glatard, E. Jeannot, Y. Legre, C. Loomis, J. Montagnat, J.-M. Moureaux, A. Osorio, X. Pennec, and R. Texier. Grid-enabling medical image analysis. *CCGrid*, May 2005.

[20] P. Goyal, X. Guo, and H. M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. *OSDI*, 1996.

[21] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly Durable, Decentralized Storage despite Massive Correlated Failures. *NSDI*, 2005.

[22] R. Jain, D. Chiu, and W. Hawe. A Quantitative Measure of Fairness and Discrimination for Resource Allocation in Shared Systems. *DEC Research Report TR-301*, Sep 1984.

[23] A. K. F. Khattab and K. M. F. Elsayed. Channel-Quality Dependent Earliest Deadline Due Fair Scheduling Schemes for Wireless Multimedia Networks. *ACM MSWiM*, 2004.

[24] T. Lam and K. To. Performance guarantee for Online Deadline Scheduling in the Presence of Overload. *ACM SODA*, 2001.

[25] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, Jul 1982.

[26] J. Leung. A New Algorithm for Scheduling Periodic Real-Time Tasks. *Algorithmica*, 1989.

[27] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the Association for Computing Machinery 20*, 1:46–61, Jan 1973.

[28] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund. Dynamic Matching and Scheduling of a Class of Independent Tasks onto Heterogeneous Computing Systems. *Heterogeneous Computing Workshop*, Apr 1999.

[29] A. K. Parekh and R. G. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks-The Single Node Case. *ACM/IEEE Transactions on Networking*, 1991.

[30] PPDG: Particle Physics Data Grid. `http://www.ppdg.net`.

[31] D. Raz, B. Avi-Itzhak, and H. Levy. Fair Operation of Multi-Server and Multi-Queue Systems. *ACM SIGMETRICS*, 2005.

[32] D. Raz, H. Levy, and B. Avi-Itzhak. A Resource Allocation Queuing Fairness Measure. *ACM SIGMETRICS*, 15:600–625, 2004.

[33] J. Sonnek, M. Nathan, A. Chandra, and J. Weissman. Reputation-Based Scheduling on Unreliable Distributed Infrastructures. *ICDCS*, Jul 2006.

[34] I. Stoica, H. Abdel-Wahab, and K. Jeffay. A Proportional Share Resource Allocation Algorithm for Real-Time Time-Shared Systems. *Real Time Systems Symposium*, 1996.

[35] A. Takefusa, S. Matsuoka, H. Casanova, and F. Berman. A Study of Deadline Scheduling for Client-Server Systems on the Computational Grid. *HPDC*, 2001.

[36] A. Wierman. Fairness and classifications. *ACM SIGMETRICS Performance Evaluation Review*, 2007.

[37] A. Wierman and M. Harchol-Balter. Classifying Scheduling Policies with respect to Unfairness in an M/G/1. *ACM SIGMETRICS*, 2003.

[38] S. Zhao, V. Lo, and C. GauthierDickey. Result Verification and Trust-based Scheduling in Peer-to-Peer Grids. *P2P*, 2005.

[39] D. Zhou and V. Lo. Cluster Computing on the Fly: Resource Discovery in a Cycle Sharing Peer-to-Peer System. *CCGrid*, 2004.