Efficient Online Scheduling for Deadline-Sensitive Jobs

[Extended Abstract]

Brendan Lucier
Microsoft Research
Cambridge, MA, USA
brlucier@microsoft.com

Ishai Menache
Microsoft Research
Redmond, WA, USA
ishai@microsoft.com

Joseph (Seffi) Naor[*]
CS Department, Technion
Haifa, Israel
naor@cs.technion.ac.il

Jonathan Yaniv
CS Department, Technion
Haifa, Israel
jyaniv@cs.technion.ac.il

ABSTRACT

We consider mechanisms for online deadline-aware scheduling in large computing clusters. Batch jobs that run on such clusters often require guarantees on their completion time (i.e., deadlines). However, most existing scheduling systems implement fair-share resource allocation between users, an approach that ignores heterogeneity in job requirements and may cause deadlines to be missed.

In our framework, jobs arrive dynamically and are characterized by their value and total resource demand (or estimation thereof), along with their reported deadlines. The scheduler's objective is to maximize the aggregate value of jobs completed by their deadlines. We circumvent known lower bounds for this problem by assuming that the input has *slack*, meaning that any job could be delayed and still finish by its deadline. Under the slackness assumption, we design a preemptive scheduler with a constant-factor worst-case performance guarantee. Along the way, we pay close attention to practical aspects, such as runtime efficiency, data locality and demand uncertainty. We evaluate the algorithm via simulations over real job traces taken from a large production cluster, and show that its actual performance is significantly better than other heuristics used in practice.

We then extend our framework to handle provider commitments: the requirement that jobs admitted to service must be executed until completion. We prove that no algorithm can obtain worst-case guarantees when enforcing the commitment decision to the job arrival time. Nevertheless, we design efficient heuristics that commit on job admission, in the spirit of our basic algorithm. We show empirically that these heuristics perform just as well as (or better than) the original algorithm. Finally, we discuss how our scheduling framework can be used to design *truthful* scheduling mechanisms, motivated by applications to commercial public cloud offerings.

[*]Work supported in part by the Technion-Microsoft Electronic Commerce Research Center, and by ISF grant 954/11.

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*sequencing and scheduling*; K.6.2 [**Management of Computing and Information Systems**]: Installation Management—*pricing and resource allocation*

General Terms

Algorithms

Keywords

Online Scheduling, Resource Allocation, Scheduling Algorithms, Truthful Mechanisms

1. INTRODUCTION

1.1 Background and Motivation

Batch processing constitutes a significant portion of the computing load across both large internal clusters and public clouds. Examples include data processing jobs (e.g., MapReduce, DryadLINQ, SCOPE), web search index updates, eScience applications, monte carlo simulations, and data analytics. Such jobs are often business-critical and time-sensitive, mandating strict service level agreements on completion time. Moreover, these jobs are not homogeneous in their timing requirements or value. For example, delays in updating website content must be minimized as they can lead to a significant loss in revenue, and financial trading firms must deliver the output of their analytics before the next trading day commences, but many simulation and rendering tasks are less urgent and can be completed at any time between two business days.

The promise of batch computing is that by centralizing the execution of diverse tasks, one can make efficient use of computing resources. For example, one could delay low-priority and time-insensitive tasks when usage peaks, responding dynamically as new jobs arrive in an online fashion. Unfortunately, resource allocation schemes currently used in practice do not live up to this promise. A common approach is to simply divide computing resources in some fair manner between applications (e.g., [7]), neglecting deadline awareness. Another approach is to give strict priority to deadline-sensitive jobs, but such heavy-handed schemes risk terminating low-priority jobs unnecessarily, lowering overall throughput; see [3] for an overview. Finally, external (i.e. paid) clouds generally eschew scheduling concerns, offloading the task of allocating sufficient resources for completing a job by its deadline to the customer.

The mismatch between current approaches and the evident need for *deadline-aware* scheduling mechanisms is due, in part, to the algorithmic difficulties of online scheduling. From a worst-case perspective, the problem of scheduling deadline-sensitive, online-arriving jobs with the goal of maximizing the value of completed jobs is inherently difficult. In its most general form, the problem admits a polylogarithmic lower bound on the competitive ratio of any randomized algorithm [2]. Previous works [12, 13, 2] have constructed algorithms with competitive ratios depending on the ratio κ between the maximal and minimal value, with the best one providing bound polylogarithmic in κ; however, as κ can be arbitrarily high, these bounds are unrealistic in practice. Constant competitive ratios are only known for special cases, e.g., identical job sizes [6], or job values which are proportional to their sizes [1]; yet both cases do not encompass realistic settings. A natural goal, then, is to develop constant-factor approximations under assumptions that can reasonably be assumed to hold for realistic input profiles in practice.

Any reasonable solution, in addition to overcoming the theoretical difficulty, must cope with practical constraints, such as inaccurate estimation of resource requirements, the need to complete admitted jobs (provider commitments), and resuming preempted jobs at the same physical location (to avoid large data transfers). This has lead the community to use heuristic methods, which do not have explicit worst-case guarantees but work well empirically, despite the known lower bounds. What aspects of practical input enable such heuristics to perform well? As it turns out, the lower bound demonstrated in [2] has an extreme property of requiring a job to start executing immediately upon arrival to meet its deadline. In "natural" inputs for which deadlines are not unreasonably tight, one might expect natural heuristics to perform reasonably well. The lack of completely tight deadlines is generally referred to as "slackness" in the input.

Our contribution is based on the following idea: if the existence of slackness in deadline constraints provides an empirical means of escape from worst-case lower bounds, then one can also revisit the theoretical problem under this assumption of slackness. Specifically, we assume that admissible jobs have lax time constraints, i.e., no job extremely pressures the system by requiring immediate and continuous execution in order to meet its deadline. This is a natural and justifiable assumption in practice. With this assumption, we find that a natural algorithmic approach provides strong theoretical guarantees, circumventing the prior polylogarithmic lower bound examples [2]. Moreover, our algorithmic approach performs well empirically, and is simple and robust enough for practical use. We therefore believe that this work provides an important step towards an efficient deadline-aware "ecosystem", which may capture complicated job models, as well as economic considerations such as user truthfulness.

We note that previous work by Garay et al. [4] considers online scheduling with input slackness, but under the assumption that the value of a job equals its size (for which constant factor approximations exist without slackness). Our work generalizes the scheduling model to incorporate arbitrary job values and demands (sizes), where it is necessary to circumvent the known barrier to constant factor approximation.

1.2 Our Results

We consider the following scheduling model. A service provider (scheduler), in charge of a computer cluster with multiple servers, receives processing requests of batch jobs that arrive online (over time); jobs are known to the system only upon arrival. Each job is characterized by a *value* for completion, the total resource *demand* (or estimation thereof) and a *deadline*. The goal of the scheduler is to maximize the throughput of the system, i.e., the total value of fully-completed jobs. The main body of the paper focuses on the simplified case of a *single server* that has to be shared between multiple users. We do so to ease the exposition of the algorithm and proofs. The extension to multiple servers is elaborated on in Section 3.3.

The performance guarantees of our online algorithms depend on the *slackness* of the input jobs, denoted s, which is the minimum allowed ratio between a job's availability time (interval between arrival and deadline) and its minimal execution time. Our contribution can be classified into two domains, differing by the level of commitment required from the scheduler. A *committed* scheduler must finish executing any job that it begins to process, whereas a *non-committed* scheduler is not required to do so.

Non-Committed Scheduling. The main theoretical contribution we provide is a worst-case performance guarantee on the competitive ratio of our algorithm for online preemptive scheduling. Specifically, we design in Section 3 an algorithm $\mathcal{A}$ with a competitive ratio bounded by:

$$cr(\mathcal{A}) \quad \leq \quad \begin{cases} 3 + O\left(\frac{1}{(s-1)^2}\right) & 1 < s < 2 \\ 2 + O\left(\frac{1}{\sqrt[3]{s}}\right) & s \geq 2, \end{cases}$$

where we recall that s is the slack. We emphasize that algorithms obtaining constant competitive ratios for online preemptive scheduling under slackness assumptions with general job specifications have not been previously known, and that our work closes a large gap open for nearly a decade between positive and negative results related to this fundamental online scheduling problem.

To obtain the main result, we rely on a proof methodology we developed in our previous work [10] for offline scheduling with identical arrival times problem under slackness assumptions. In this work the scheduling problem is formulated as a linear program with strengthened constraints which are somewhat reminiscent of knapsack constraints. Using insights from [10], we approach the more challenging online scheduling problem and develop a novel algorithm for it. We note that although the online problem is completely different in its algorithmic nature from the offline version considered in [10], our proof techniques do share some common concepts. We provide performance guarantees for the online problem using the dual fitting technique together with sophisticated charging arguments tailored to this specific context.

From a practical viewpoint, our algorithm incorporates important design principles, which could be easily tracked and implemented in real systems. First, the job's value density (the ratio between the job value and demand) is the major factor which determines its precedence, rather than its deadline. Accordingly, an executing job j is preempted only if a newly arriving job has a value density which is at least γ times higher than the value density of j, where γ is a tunable *threshold* parameter of the algorithm. Further, due to input slackness, the scheduler need not decide whether to schedule a job right upon arrival, and may take a pre-defined lag for its decision. This principle is incorporated into our algorithm, by starting job execution only if the remaining time until its deadline is sufficiently large with respect to a *gap* parameter μ.

Committed Scheduling. In Section 4 we consider two variants of committed scheduling, in which the scheduler commits to jobs it decides to process: (1) *commitment on job arrival*, in which the scheduler decides upon job arrival whether it commits to the job or rejects it; for this model, we prove that no algorithm can provide any worst-case guarantee, even under a slackness assumption. (2) *commitment on job admission*, in which the scheduler guarantees job completion only once it begins processing it; that is, once a job is *admitted* (which need not happen immediately upon arrival), the scheduler must meet its deadline. Unfortunately, the theoretical guarantees we obtained for non-committed scheduling do not apply in this setting. Hence, we use insights gleaned from our theoretical result in order to design a heuristic solution. Specifically, we apply our original algorithm with a small change: we do not admit a new job whose execution would prevent the cluster from completing jobs to which it already committed to. While this heuristic does not have worst-case guarantees, we find that it performs very well in practice. We evaluate our solutions (both for non-committed and committed scheduling) through comprehensive simulations on empirical traces (extracted from a Microsoft cluster). Appealingly, our algorithms outperform other plausible heuristics, typically by $10 - 50$x.

Finally, we extend our heuristic for committed scheduling to accommodate economic considerations emerging from paid cloud service applications. We design a *truthful* scheduling mechanism, in which participants are incentivized not to manipulate the system for personal interest by misreporting their true job values and parameters (demands and deadlines). We show experimentally that our modified solution comes without utility loss for realistic input profiles.

1.3 Related Work

We provide a brief overview of recent related work in the context of datacenter resource allocation. Resource allocation is becoming a vital and central problem in today's large clusters. Quincy [8] is an algorithmic framework for assigning resources to batch jobs based on locality and fairness constraints. However, this work does not cover deadline considerations. Similarly, [5] the multi-resource allocation problem has been studied in the context of datacenters with fairness being the main performance criterion. Jockey [3] is a system that aims at finishing data-processing jobs (SCOPE) by their deadlines using dynamic allocation of CPU resources, based on offline and online profiling of jobs. However, Jockey focuses on the single job case, and does not explicitly address the scheduling of multiple jobs. Bazaar [11] considers the assignment of both bandwidth and CPU resources for meeting deadlines of multiple batch jobs. The basic idea is to profile jobs in advance and form an estimate of job completion time as a function of (bandwidth, CPU), then heuristically allocate these resources to maximize the number of jobs that complete by their deadlines. Unlike our model, all jobs are assumed to have equal value, and consequently resource allocation is kept static and job preemption is not required.

2. PRELIMINARIES

2.1 Problem Description

Job requests are submitted to a cluster consisting of C identical servers (resources), denoted $1, 2, \ldots, C$. All servers are fully available throughout time and each server can process at most one job at any given time. The cluster is managed by a service provider (scheduler), which also determines the resource allocation. The input is a finite set[1] of batch jobs, denoted $\mathcal{J}$. These jobs arrive to the system online, over the (continuous) time interval $\mathbb{R}^+ = [0, \infty)$. Every job $j \in \mathcal{J}$ is revealed to the system only upon its arrival time a_j. Upon arrival, each job specifies its deadline, demand and value. The deadline d_j indicates the latest acceptable completion time for job j. The interval $W_j = [a_j, d_j]$ is called the *availability window* of job j.

The size D_j of job j, also referred to as the *demand* of the job, is the total resource amount required to complete the job (e.g., in CPU hours). In the bulk of the paper, we assume that D_j is deterministic. The case of demand uncertainty is treated separately in Appendix 3.4. A *value* of v_j is gained by the system if and only if job j is fully executed by its deadline (i.e., allocated D_j units of resource by time d_j). We emphasize that partial execution does not result in partial value. For any set of jobs $S \subseteq \mathcal{J}$, we denote by $v(S) = \sum_{j \in S} v_j$ its aggregate value. We denote $\rho_j \triangleq v_j / D_j$ as the *value-density* of job j (i.e., the ratio between its value and its demand). Value-densities will play a significant role in the design of our algorithms.

The goal of the scheduler is to maximize the *throughput*: total value of jobs fully completed by their deadlines. The scheduler is not required to complete all jobs. Specifically, if a job reaches its deadline without being completed, there is no benefit to allocating additional servers to it. We assume that at most k servers can be allocated to a single job at any given time. This parameter may stand for a common parallelism bound across jobs[2], or represent a management constraint such as a virtual cluster. For example, $k = 1$ means that every job can be processed on at most one server at any time. At any given time, the scheduler may allocate any number of servers between 0 and k to any job, subject to the capacity constraint C. In particular, jobs may be preempted. Execution of preempted jobs may be resumed from the point at which they were preempted (assuming proper checkpointing of intermediate states).

The performance guarantees of our online algorithms depend on a parameter $s \geq 1$ called the *slackness* of the input. We say that the input has slackness s if for each job j, $d_j - a_j \geq s \cdot (D_j / k)$. The slackness parameter s limits the tightness of a job's deadline with respect to its minimal execution time D_j / k. From a practical perspective, we can think of slackness either as a feature of the input or as a constraint imposed by the system (by declaring s). As we shall see, the performance of our algorithms improves as s increases.

2.2 Definitions

The following definitions refer to the execution of an online allocation algorithm $\mathcal{A}$ over an input set $\mathcal{J}$ of jobs. We drop $\mathcal{A}$ and $\mathcal{J}$ from notation when they are clear from context.

Competitive Ratio. For an online algorithm $\mathcal{A}$ and an input sequence of jobs $\mathcal{J}$, denote by $\mathcal{A}(\mathcal{J})$ the set of jobs that are fully completed by $\mathcal{A}$ over an online sequence of arriving jobs $\mathcal{J}$. The throughput gained by $\mathcal{A}$ is $v(\mathcal{A}(\mathcal{J}))$. Let $OPT(\mathcal{J})$ denote the set of jobs completed by an optimal offline allocation, i.e., one that has full knowledge of $\mathcal{J}$ in advance. We are interested in the worst-case performance guarantees of online algorithms, namely their competitive ratios:

$$\text{cr}(\mathcal{A}) \quad = \quad \max_{\mathcal{J}} \left\{ \frac{v(OPT(\mathcal{J}))}{v(\mathcal{A}(\mathcal{J}))} \right\}.$$

The competitive ratio is a standard measurement of the performance of online algorithms. Note that $\text{cr}(\mathcal{A}) \geq 1$, and that a smaller competitive ratio implies better performance guarantees.

[1] Algorithms described in this paper are well-defined for infinite job sequences; we assume finiteness for notational convenience.

[2] E.g., if jobs have different parallelism bounds, then k is the minimum thereof. We note that more involved parallelism models, such as Amdahl's law, are beyond the scope of our paper. Nevertheless, we believe that the insights and design principles obtained here may carry over to such models.

Job Allocations. Denote by $j^i_{\mathcal{A}}(t)$ the job running on server i at time t and by $\rho^i_{\mathcal{A}}(t)$ its value-density. We use $y^i_j(t)$ as a binary[3] variable indicating whether job j is running on server i at time t, i.e., whether $j = j^i_{\mathcal{A}}(t)$ or not. We often refer to the function y^i_j as the *allocation* of job j on server i. Define $y_j(t) = \sum_{i=1}^{C} y^i_j(t)$ to be the total number of servers allocated to j at time t, and define Δ_j to be the overall amount of resources allocated to job j. The *starting point* $st(y^i_j) = \min\left\{\{t \mid y^i_j(t) = 1\} \cup \{\infty\}\right\}$ of job j on server i is the first time at which j is allocated to server i. If no such t exists, $st(y^i_j) = \infty$.

Job Availability and Status. For a job j, write $W_j^{-\mu}$ for the time interval $[a_j, d_j - \mu \cdot (D_j/k)]$. Note $W_j^0 = W_j$ is the availability window of j. Correspondingly, $A^{-\mu}(t) = \left\{j \in \mathcal{J} \mid t \in W_j^{-\mu}\right\}$ is defined as the set of jobs j at time t whose remaining availability time is at least μ times their minimal execution time D_j/k. The algorithms we design in this paper limit the starting time of jobs by selecting jobs to be processed only from the set $A^{-\mu}(t)$.

We divide the job set $\mathcal{J}$ into three sets, depending on the jobs' final execution status: (1) *completed (fully processed)* jobs $\mathcal{J}^F$, which have been completed by their corresponding job deadlines; (2) *partially processed* jobs $\mathcal{J}^P$, which have begun their execution but were not completed on time; and (3) *unprocessed (empty)* jobs $\mathcal{J}^E$, which have not been processed at all. We say that a job has been *admitted (allocated)* if it has begun execution, i.e., it is in $\mathcal{J} \setminus \mathcal{J}^E$. We denote by $\mathcal{J}^P_i$ the set of jobs that have been partially processed on server i.

2.3 The Dual Fitting Technique

A core element of our analysis is a dual fitting argument. Dual fitting is a common technique for bounding the performance of algorithms (the competitive ratio in our case). The technique uses weak duality, originating from optimization theory, to obtain an upper bound on the value gained by the optimal solution $OPT(\mathcal{J})$. In the field of algorithmic design, the dual fitting technique is typically applied over linear programming relaxations to combinatorial optimization problems. In our context, the relaxed formulation we use takes a slightly different form compared to standard linear programs, as a result of the non-discreteness of time. In the sequel we describe the dual fitting technique through its specific application to the scheduling model considered in the paper.

In the first step of this technique, we describe an optimization problem over linear constraints, called the *primal program*. The primal program is a *relaxed* formulation of the scheduling problem, i.e., every possible schedule of jobs in our scheduling problem is also a feasible solution to the primal program. The primal program may allow additional feasible solutions. As a consequence, the optimal solution to the primal program may have higher value than the optimal schedule. Formally, we denote by $OPT^*(\mathcal{J})$ the optimal solution to the primal program. This solution is super-optimal, meaning that $v(OPT(\mathcal{J})) \leq v(OPT^*(\mathcal{J}))$. We now describe the primal program used in context of our scheduling model.

Primal Program. The primal program is presented in equations (1)-(5). Notice that some of the constraints of the original problem were relaxed and replaced by a linear constraint. For example, the primal program does not constrain the variable $y_j(t)$ to be binary. Instead, the variable $y_j(t)$ may receive any number from the range $[0, 1]$. The objective function (1) represents the value gained from scheduled jobs. Notice that as a consequence of relaxing the variables $y_j(t)$, the objective function represents the total partial value gained from jobs. In other words, according to the primal program, the scheduler may gain partial value over partially completed jobs.

$$\max \quad \sum_{j \in \mathcal{J}} \rho_j \Delta_j \tag{1}$$

$$\Delta_j \;=\; \int_{a_j}^{d_j} \sum_{i=1}^{C} y^i_j(t)dt \;\leq\; D_j \qquad \forall j \tag{2}$$

$$\sum_{j: t \in W_j} y^i_j(t) \;\leq\; 1 \qquad \forall i, t \in \mathbb{R}^+ \tag{3}$$

$$\sum_{i=1}^{C} y^i_j(t) - k \cdot \frac{\Delta_j}{D_j} \;\leq\; 0 \qquad \forall j, t \in W_j \tag{4}$$

$$y^i_j(t) \geq 0 \qquad \forall j, i, t \in W_j \tag{5}$$

Recall that Δ_j represents the total amount of resources allocated to job j, and that the gain from a completed job is exactly $\rho_j D_j = v_j$. The primal program maximizes the throughput (1) under the following constraints: demand constraints (2), capacity constraints (3) and parallel execution constraints (4). Note that (3) implicitly requires that $y^i_j(t) \leq 1$. The last set of constraints, suggested by [9], are strengthened parallelism constraints. That is, instead of naturally bounding the amount of resources a job j may receive per time by k, we scale down the amount of servers the job may receive by Δ_j/D_j. Notice that a feasible solution to the relaxed formulation does not necessarily require that an executed job will necessarily be fully completed. Intuitively, according to these constraints, a job that is 50% completed may be allocated at most $0.5k$ servers at every moment. These set of strengthened constraints allow us to obtain better results, as we will see.

Dual Program. The dual optimization problem is associated with the primal program. By weak duality, every feasible solution to the dual program defines an upper bound to $v(OPT^*(\mathcal{J}))$.

$$\min \quad \sum_{j \in \mathcal{J}} D_j \alpha_j + \sum_{i=1}^{C} \int_0^\infty \beta_i(t)dt \tag{6}$$

$$\text{s.t.} \quad \alpha_j + \beta_i(t) + \pi_j(t) - \frac{k}{D_j} \int_{a_j}^{d_j} \pi_j(\tau)d\tau \;\geq\; \rho_j \qquad \forall j, i, t \in W_j \tag{7}$$

$$\alpha_j, \beta_i(t), \pi_j(t) \;\geq\; 0 \qquad \forall j, i, t \in W_j \tag{8}$$

For a solution (α, β, π) to the dual program, we refer to the value of the objective function induced by the solution as the *dual cost* of the solution. For every job j, the dual program defines a set of covering constraints (7) over its availability window W_j. We say that a constraint is *covered* by a solution if the constraint is satisfied.

There are three kinds of dual variables. Every job j has a variable α_j that appears in all of its covering constraints. Notice that by setting $\alpha_j = \rho_j$, all of the constraints of job j can be satisfied at a dual cost of $D_j \alpha_j = D_j \rho_j = v_j$. The second set of dual variables $\beta_i(t)$ are used to cover the remaining constraints, and can be thought of as a set of continuous functions $\beta_i : \mathbb{R}^+ \to \mathbb{R}^+$, one per server. The last set of variables $\pi_j(t)$ are a result of the gap decreasing constraints (4). These dual variables are used in the analysis for the case of multiple servers; in the analysis for a single server they will be set to 0.

[3]In Section 2.3 we extend the range of values $y^i_j(t)$ may receive. However, we will always treat it as an allocation indicator.

In Section 3 we present online algorithms for preemptive scheduling without commitments. To bound the competitive ratios of each algorithm, we will first consider the total value gained by the algorithm, $v(\mathcal{J}^F)$. We will then use the structure of $\mathcal{J}^F$ to construct a feasible solution to the dual program and evaluate its dual cost (the value of the solution according to the dual objective function). We will show that the constructed dual solution has cost $r \cdot v(\mathcal{J}^F)$ for some constant $r > 1$, which then implies (by the duality principle) that $v(OPT(\mathcal{J})) \leq r \cdot v(\mathcal{J}^F)$. This will allow us to conclude that the competitive ratio of the algorithm is at most r. We summarize our discussion in the following standard theorem.

THEOREM 2.1. *Let $\mathcal{A}$ be an online scheduling algorithm. If for every job input set $\mathcal{J}$ there exists a feasible solution (α, β, π) to the dual program with a dual cost of $r \cdot v(\mathcal{A}(\mathcal{J}))$, then $cr(\mathcal{A}) \leq r$.*

3. NON-COMMITTED SCHEDULING

In this section we present our main theoretical results: online algorithms for non-committed scheduling with guaranteed competitive ratio. We first present an algorithm for the single server case (Section 3.1) and analyze its performance Section 3.2. We then consider the extension to multiple servers in Section 3.3, and finally handle demand uncertainties in Section 3.4.

3.1 Single Server Algorithm

Throughout this subsection we assume there is a single server, so we drop the server index i from our notation. We define two parameters, each representing a simple principle that our scheduling algorithm will follow. The first principle incorporates the conditions for preempting a running job, and it is characterized by a *threshold parameter* $\gamma > 1$.

PRINCIPLE 3.1. *A pending job j' can preempt a running job j only if $\rho_{j'} > \gamma\rho_j$.*

Roughly, jobs are prioritized according to value-densities. This may seem counter-intuitive at first, since a small job can preempt a large job with much higher value. Presumably, such a scenario might lead to a great loss of value. However, we take advantage of the slackness assumption to compensate for the lost value. To do so, we incorporate a second principle, which restricts the starting time of jobs and is parameterized by a *gap parameter* μ ($1 \leq \mu \leq s$).

PRINCIPLE 3.2. *A job j cannot begin its execution after time $d_j - \mu D_j$.*

We provide brief intuition for the selection of these two principles. First, for a job j to be unprocessed, any job executed during $[a_j, d_j - \mu D_j]$ must have a value-density of at least ρ_j/γ. Second, for a job j to be partially processed (yet incomplete), any other job executed during $[d_j - \mu D_j, d_j]$ must have a value-density of at least $\gamma\rho_j$. This intuition will become more clear once we analyze the performance of the algorithm.

The algorithm $\mathcal{A}$ presented here for online preemptive single server scheduling (Algorithm 1) follows these two principles. The decision points of the algorithm occur at one of two events: either upon the arrival of a new job, or at the completion of the processed job. The algorithm handles both events similarly. When a new job arrives, the algorithm invokes a *threshold preemption rule*, which decides whether or not to preempt the currently running job. The preemption rule selects the pending job $j^* \in A^{-\mu}(t)$ of maximal value-density, and replaces the currently running job j with j^* only if $\rho_{j^*} > \gamma\rho_j$; ties broken arbitrarily. Note that it is always possible to complete any $j^* \in A^{-\mu}(t)$ before its deadline, since $\mu \geq 1$.

Algorithm 1: Single Server Algorithm $\mathcal{A}$

```
Event: On arrival of job $j$ at time $t = a_j$:
  1. Call the threshold preemption rule.

Event: On job completion at time $t$:
  1. Resume execution of the preempted job with highest
value-density.
  2. Call the threshold preemption rule.

Threshold Preemption Rule ($t$):
  1. $j \leftarrow$ job currently being processed.
  2. $j^* \leftarrow \arg\max\{\rho_{j^*} \mid j^* \in A^{-\mu}(t)\}$.
  3. if $(\rho_{j^*} > \gamma \cdot \rho_j)$
        3.1. Preempt $j$ and run $j^*$.
```

When a job is completed, the second type of event, the preemption rule is also called. Here, j is selected as the preempted job with the highest value-density among the partially processed jobs. Before we proceed to analyze the competitive factor of $\mathcal{A}$, we summarize some of the properties of the algorithm in the following claim.

CLAIM 3.3. *The following properties of $\mathcal{A}$ hold:*

1. *Any allocated job $j \notin \mathcal{J}^E$ satisfies $st(y_j) \leq d_j - \mu D_j$.*
2. *For any t, let j be the job running at time t and let $j' \in A^{-\mu}(t)$ be a job that has not been completed by time t. Then $\rho_{j'} \leq \gamma\rho_j$.*
3. *Let $j' \in \mathcal{J}^P$ be a job partially processed by $\mathcal{A}$. Any job j running at some time t such that $st(y_{j'}) \leq t \leq d_{j'}$ satisfies $st(y_{j'}) < st(y_j)$.*

PROOF. Claim 1 follows directly from the threshold preemption rule. For the algorithm to start a job j at time t, job j must satisfy $j \in A^{-\mu}(t)$, which implies that $t \leq d_j - \mu D_j$.

To prove Claim 2, assume toward contradiction that $\rho_{j'} > \gamma\rho_j$. Consider the first time in the interval $[a_{j'}, t]$ that j is being processed. There must be such a time, since j is being processed at time t. This first time cannot be $a_{j'}$, since j' would preempt j at time $a_{j'}$ if j were being processed at that time. It is also impossible that j started its execution after $a_{j'}$, since j' is not complete by time t and j' would be preferred over j by the threshold preemption rule. Therefore, j must have been preempted before $a_{j'}$. This yields a contradiction, since j would not have resumed its execution as long as j' is present in the system.

For Claim 3, suppose for contradiction that $st(y_{j'}) \geq st(y_j)$. Since j continues executing at $t > st(y_{j'})$, this implies that j' was chosen for execution over j at time $st(y_{j'})$, and hence $\rho_{j'} > \rho_j$. However, since $j' \in \mathcal{J}^P$, it must be that j was chosen for execution over j' at time t, and hence $\rho_j \geq \rho_{j'}$, a contradiction. $\square$

3.2 Analysis of Single Server Algorithm

3.2.1 Competitive Ratio

Our competitive ratio analysis of the single server algorithm $\mathcal{A}$ relies on the dual fitting technique described in Section 2.3. Our analysis is post factum, that is, we retrospectively analyze the performance of the online algorithm after it has finished admitting jobs. Recall that $\rho_{\mathcal{A}}(t)$ represents the value-density of the job that was processed by $\mathcal{A}$ at time t. The analysis proceeds in two parts. In the first part of our analysis we show how to construct a feasible solution to the dual program, and bound its dual cost in terms of

$\int_0^\infty \rho_\mathcal{A}(t)dt$. Notice that time intervals in which incomplete jobs were processed do not contribute to the total value gained by the algorithm. Hence, the expression $\int_0^\infty \rho_\mathcal{A}(t)dt$ does not represent the throughput of the algorithm. Therefore, in the second part of our analysis, we bound the ratio between $\int_0^\infty \rho_\mathcal{A}(t)dt$ and the value $v(\mathcal{A}(\mathcal{J}))$ gained by the algorithm; the later can be simply written as $v(\mathcal{J}^F)$. The first part of our analysis is summarized in the following theorem.

THEOREM 3.4. *Consider an execution of $\mathcal{A}$ over an input set of arriving jobs $\mathcal{J}$. Let $\rho_\mathcal{A} : \mathbb{R} \to \mathbb{R}$ be a function representing the value-density of the job that was executed by $\mathcal{A}$ at every time t. Then, there exists a solution $\big(\alpha, \beta, \pi\big)$ to the dual program with a dual cost of at most:*

$$v(\mathcal{J}^F) + \gamma \cdot \frac{s}{s-\mu} \cdot \int_0^\infty \rho_\mathcal{A}(t)dt.$$

Before proving Theorem 3.4 we need to make several preliminary observations and develop additional machinery. The $\pi_j(t)$ variables are not necessary for the single server analysis of our algorithm, so for the remainder of this section we set them to 0. The dual constraints (7) then reduce to the following form:

$$\alpha_j + \beta(t) \;\geq\; \rho_j \qquad \forall j, t \in W_j \tag{9}$$

Our goal is to construct a feasible solution to the dual program, that is, a solution that covers (satisfies) all of the dual constraints (9). Notice that by setting $\alpha_j = \rho_j$ for each completed job $j \in \mathcal{J}^F$ we cover all of the dual constraints corresponding to j. This step increases the dual cost by exactly $\sum_{j\in\mathcal{J}^F} D_j\alpha_j = \sum_{j\in\mathcal{J}^F} D_j\rho_j = v(\mathcal{J}^F)$. To cover the remaining dual constraints of incomplete jobs, we use the β function. Notice that the variable $\beta(t)$ appears in all of the dual constraints (9) corresponding to time t. The β function allows us to cover the dual constraints of incomplete jobs $j \notin \mathcal{J}^F$ without having to pay for them separately using their corresponding α_j variables, as we did for completed jobs. To obtain a feasible solution to the dual program, we require that β satisfies for every time $t \in \mathbb{R}^+$: $\beta(t) \;\geq\; \max\big\{\rho_j \mid j \in A(t) \wedge j \notin \mathcal{J}^F\big\}$. Consider the following function $\beta^{-\mu} : \mathbb{R}^+ \to \mathbb{R}^+$, defined by:

$$\beta^{-\mu}(t) \;=\; \max\Big\{\rho_j \mid j \in A^{-\mu}(t) \wedge j \notin \mathcal{J}^F\Big\}. \tag{10}$$

The function $\beta^{-\mu}$ satisfies two useful properties. First, by Claim 3.3-2 we have $\beta^{-\mu}(t) \leq \gamma \cdot \rho_\mathcal{A}(t)$ for every time $t \in \mathbb{R}^+$. Second, the function $\beta^{-\mu}$ covers the dual constraints of every incomplete job $j \notin \mathcal{J}^F$ and time $t \in W_j^{-\mu} = [a_j, d_j - \mu D_j]$. This nearly completes the analysis, since we can cover most of the dual constraints (9) using the $\beta^{-\mu}$ function, and bound its cost in terms of $\rho_\mathcal{A}$. To cover the remaining constraints, we "stretch" the function $\beta^{-\mu}$ by using the following lemma.

LEMMA 3.5 (STRETCHING LEMMA). *Let $\beta^{-\mu} : \mathbb{R}^+ \to \mathbb{R}^+$ be a function satisfying: $\beta^{-\mu}(t) \;\geq\; \rho_j$ for every $j \notin \mathcal{J}^F$ and $t \in W_j^{-\mu}$. Then, there exists a function $\beta : \mathbb{R}^+ \to \mathbb{R}^+$ satisfying: $\beta(t) \geq \rho_j$ for every $j \notin \mathcal{J}^F$ and $t \in W_j$, such that*

$$\int_0^\infty \beta(t)dt \;\leq\; \frac{s}{s-\mu} \cdot \int_0^\infty \beta^{-\mu}(t)dt.$$

By combining the above observations with the stretching lemma, we construct a feasible solution to the dual program. The remaining details are given in the following proof.

PROOF OF THEOREM 3.4: Set $\alpha_j = \rho_j$ for every completed job $j \in \mathcal{J}^F$, and $\alpha_j = 0$ otherwise. To cover the remaining dual constraints, we apply the stretching lemma on the function $\beta^{-\mu}$ defined in (10) and obtain the function β. The $\pi_j(t)$ variables are all set to 0. The total cost of the dual solution (α, β, π) is bounded by:

$$\begin{aligned}\sum_{j\in\mathcal{J}} D_j\alpha_j + \int_0^\infty \beta(t)dt \quad &\leq \quad v(\mathcal{J}^F) + \frac{s}{s-\mu}\int_0^\infty \beta^{-\mu}(t)dt \\ &\leq \quad v(\mathcal{J}^F) + \gamma \cdot \frac{s}{s-\mu}\int_0^\infty \rho_\mathcal{A}(t)dt.\end{aligned}$$

The last inequality follows since $\beta^{-\mu}(t) \leq \gamma \cdot \rho_\mathcal{A}(t)$ for every $t \in \mathbb{R}^+$. $\square$

This completes the first part of the analysis. In the second part of the analysis, we bound the total cost of $\int_0^\infty \rho_\mathcal{A}(t)dt$ by applying a charging argument motivated by principles 3.1 and 3.2. This leads to the following theorem.

THEOREM 3.6. *Let $\mathcal{J}^F$ be the set of jobs completed by $\mathcal{A}$ with input $\mathcal{J}$, and let $v(\mathcal{J}^F)$ denote the total value gained by the algorithm. Let $\rho_\mathcal{A}(t)$ represent the value density of the job executed by $\mathcal{A}$ at time t. Then:*

$$\int_0^\infty \rho_\mathcal{A}(t)dt \quad\leq\quad v(\mathcal{J}^F) \cdot \left[\frac{(\gamma-1)(\mu-1)}{(\gamma-1)(\mu-1)-1}.\right]$$

Our goal is to bound the expression $\int_0^\infty \rho_\mathcal{A}(t)dt$. We divide the timeline into two sets: $\mathcal{T}^F$, times during which the algorithm processed jobs that were eventually completed; and $\mathcal{T}^P$.

$$\mathcal{T}^F = \Big\{t \in \mathbb{R}^+ \mid j_\mathcal{A}(t) \in \mathcal{J}^F\Big\} \quad ; \quad \mathcal{T}^P = \mathcal{T} \setminus \mathcal{T}^F$$

We can break the integral $\int_0^\infty \rho_\mathcal{A}(t)dt$ into two, according to $\mathcal{T}^F$ and $\mathcal{T}^P$. Notice that integrating $\rho_\mathcal{A}(t)$ over $\mathcal{T}^F$ gives us exactly $v(\mathcal{J}^F)$. Hence, it remains to bound $\int_{\mathcal{T}^P} \rho_\mathcal{A}(t)dt$. This expression represents the partial value that was lost over incomplete jobs, or formally: $\sum_{j\in\mathcal{J}^P} \rho_j\Delta_j$. Consider a partially processed job $j \in \mathcal{J}^P$. Define the *admission window* $Ad_j = [st(y_j), d_j]$ of job j as the interval between the job admission time (i.e., execution starting time) and its deadline. By Claim 3.3-1, the size of the admission window is at least μD_j. Let $I_j \subseteq Ad_j$ represent the times during which job j has been processed. Since job j has not been completed, its total execution time is at most D_j. Hence, the total time in Ad_j during which $\mathcal{A}$ processed jobs different than j is at least $(\mu-1)D_j$; denote this set of times by O_j. According to Claim 3.3-3, each of the jobs executed during O_j has a value-density at least γ times larger than ρ_j. Integrating $\rho_\mathcal{A}(t)$ over O_j gives us a total value of at least $\gamma(\mu-1)v_j$. Intuitively, the value gained during O_j can used to "pay" over the partial value $\rho_j\Delta_j \leq v_j$ that the algorithm lost by not completing job j. However, there is a flaw in the argument, since jobs that were processed during O_j have not necessarily been completed. To succeed, a more rigorous analysis is required.

We bound $\int_0^\infty \rho_\mathcal{A}(t)dt$ using a charging argument, which is motivated by the last paragraph. Initially, we charge every job running at some time t a value of $\rho_\mathcal{A}(t)$. We then apply a charging procedure that iteratively transfers charges away from incomplete jobs, until finally only completed jobs are charged. Finally, we bound the total amount each completed job is charged for (Lemma 3.7).

The Charging Procedure. Let $ch : \mathbb{R}^+ \to \mathbb{R}^+$ be a charging function, representing an amount charged from the job that has been processed per time t. Initially, we set $ch(t) = \rho_{\mathcal{A}}(t)$ for every time t. We describe a procedure that shifts values of $ch(t)$ towards completed jobs, that is, time slots in $\mathcal{T}^F$. After initializing ch, we sort the partially executed jobs in $\mathcal{J}^P$ according to their starting time $st(y_j)$. For each job $j \in \mathcal{J}^P$ in this order, we do the following:

1. Define: $I_j = \{t \in \mathbb{R}^+ \mid j = j_{\mathcal{A}}(t)\}$.
2. Define: $O_j = Ad_j \setminus I_j$.
3. Let $\psi_j : I_j \to O_j$ be some bijection from I_j to O_j.
4. For every $t \in I_j$, increase $ch(\psi(t))$ by $(|I_j| \,/\, |O_j|)\, ch(t)$ and set $ch(t)$ to 0.

Let $ch'(t)$ denote the value of $ch(t)$ at the end of the procedure. At each iteration of the charging procedure, some incomplete job $j \in \mathcal{J}^P$ transfers all of the charges associated with it towards jobs that execute during O_j. Claim 3.3 implies that jobs processed in O_j have either been completed or have started executing after j. Since we sorted jobs by start times, this implies that after we transfer charges away from a job $j \in \mathcal{J}^P$, we never subsequently transfer charges back to j. This will imply that, at the end of the procedure, only jobs in $\mathcal{J}^F$ are charged. Our goal now is to obtain a good bound on $ch'(t)$ as a function of $ch(t)$. This goal is complicated by the fact that charges can be transferred multiple times, and along multiple paths, before reaching jobs from $\mathcal{J}^F$. The following is our main technical lemma, which analyzes the structure of the charging procedure in order to bound $ch'(t)$.

LEMMA 3.7. *At the end of the charging procedure:*

1. $\int_0^\infty \rho_{\mathcal{A}}(t)dt = \int_0^\infty ch'(t)dt$.

2. For every $t \in \mathcal{T}^F$, $ch'(t) \le \rho_{\mathcal{A}}(t) \cdot \frac{(\gamma-1)(\mu-1)}{(\gamma-1)(\mu-1)-1}$.

3. For every $t \notin \mathcal{T}^F$, $ch'(t) = 0$.

PROOF. Claim 1 holds since every iteration of the charging procedure (lines 1-4) does not change the value of $\int_0^\infty ch(t)dt$. We now prove Claim 3. Recall that the charging procedure sorts jobs in $\mathcal{J}^P$ according to their starting times. Every job $j \in \mathcal{J}^P$ transfers all of its charges towards jobs in Ad_j, which are either completed jobs or other jobs in $\mathcal{J}^P$. Claim 3.3-3 states that a job in $\mathcal{J}^P$ processed in Ad_j must have started after job j. This guarantees that at the end of the charging procedure, $ch'(t) = 0$ for every $t \notin \mathcal{T}^F$.

The rest of the proof is dedicated to proving Claim 2. Our goal is to bound the ratio between $ch'(t)$ and $\rho_{\mathcal{A}}(t)$ for every time $t \in \mathcal{T}^F$. Up until now we treated entries of the function $\rho_{\mathcal{A}}$ as value that has or has not been gained by the algorithm. However, the notion of value may be confusing in the current context of analyzing a charging argument. To avoid confusion, in this proof we refer to entries of $\rho_{\mathcal{A}}$ as *costs* that need to be paid for. Specifically, we are interested in costs that eventually affect $ch'(t)$.

Consider some time $t_i \in \mathcal{T}^P$. The role of i will be explained later on, and at this point can be ignored. Let j_i be the job that has been processed at time t_i. Consider the cost $\rho_{\mathcal{A}}(t_i)$. Initially, $ch(t_i)$ is charged for the cost of $\rho_{\mathcal{A}}(t_i)$. When the iterative loop of the charging procedure reaches job j_i, the cost $\rho_{\mathcal{A}}(t_i)$ is transferred to a different time $t_{i-1} = \psi_{j_i}(t_i)$ and scaled down by a factor of $1/(\mu-1)$, at least[4]. Let i represent how many times the cost $\rho_{\mathcal{A}}(t_i)$ has been transferred, and let t_i^F represent the final time to which the cost is transferred. By Claim 3, the job bring processed at t_i^F must have been completed by the algorithm.

[4]We note that the transferred value, $ch(t_i)$, may be larger than $\rho_{\mathcal{A}}(t_i)$, because of other costs transferred to t_i at an earlier stage. However, at this point in the analysis we are only interested in the portion of the transferred value corresponding to $\rho_{\mathcal{A}}(t_i)$.

Now consider an incomplete job $j' \in \mathcal{J}^P$, and some time t. We say that job j' *charges time* t *after* i *transfers* if there some time t_i for which $t_i^F = t$. We would like to understand how much of the final charge at t, $ch'(t)$, was transferred from job j'. A complication is that j' can charge time t in multiple ways. For example, it may be that $t \in O_{j'}$, so that the charge at t increases when j' is handled by the charging procedure. However, there may be another job incomplete j'' that was also being processed in the interval $O_{j'}$, which receives part of the charge of j'; when j'' is handled by the charging procedure, it might also transfer some of its charge – which includes charge received from j' – to time t. In general, charge may transfer from j' to time t via multiple paths of varying lengths; we will bound this transfer over all possible paths.

Let k be the number of incomplete jobs that have started between $st(y_{j'})$ and t (not including j'). We are interested in bounding the number of times that job j' charges t after i transfers. We claim this number is at most $\binom{k}{i-1}$. To see this, consider a cost $\rho_{\mathcal{A}}(t_i)$. Let $t_i \to t_{i-1} \to \cdots \to t_0 = t_i^F$ denote the path through which the cost $\rho_{\mathcal{A}}(t_i)$ is transferred, and by $j_i, j_{i-1}, \ldots, j_0$ the corresponding jobs processed during those times. Notice that the set $j_i, j_{i-1}, \ldots, j_0$ is unique for each such t_i, since every ψ is a bijection. Moreover, notice that the jobs j_i, j_{i-1}, j_1 must be sorted in ascending order of starting time, since by Claim 3.3-3 an incomplete job only charges jobs that have started after it. Hence, the number of options for such a t_i is the number of unique paths from j_i to j_0, which is the number of options to choose $j_{i-1}, \ldots, j_1$ out of k jobs. Notice that i can range between 1 and $k+1$.

The last step of the proof is to bound $\rho_{j'}$. Without loss of generality, we assume that j' actually charges t after some amount of transfers, otherwise j' is irrelevant for the discussion. Consider the k incomplete jobs that started between $st(y_{j'})$ and t in ascending order of their starting times. Each job in this order must be contained in the admission window of its predecessor. By Claim 3.3-2, we get that $\rho_{j'} \le \rho_{\mathcal{A}}(t)/\gamma^{k+1}(\mu-1)^i$. Since each job j' is uniquely identified by the number k of jobs that start between time t and the start of j', and each path to such a j' from t has length at most $k+1$, this gives us the following:

$$\begin{aligned}
ch'(t) &\le \rho_{\mathcal{A}}(t) + \sum_{k=0}^{\infty}\sum_{i=1}^{k+1}\binom{k}{i-1}\frac{\rho_{\mathcal{A}}(t)}{\gamma^{k+1}(\mu-1)^i} \\
&= \rho_{\mathcal{A}}(t) + \sum_{k=0}^{\infty}\frac{\rho_{\mathcal{A}}(t)}{\gamma^{k+1}(\mu-1)}\sum_{i=1}^{k+1}\binom{k}{i-1}\frac{1}{(\mu-1)^{i-1}} \\
&= \rho_{\mathcal{A}}(t) + \sum_{k=0}^{\infty}\frac{\rho_{\mathcal{A}}(t)}{\gamma^{k+1}(\mu-1)}\left(1+\frac{1}{\mu-1}\right)^k \\
&= \rho_{\mathcal{A}}(t)\left[1+\frac{1}{\gamma(\mu-1)}\sum_{k=0}^{\infty}\left(\frac{\mu}{\gamma(\mu-1)}\right)^k\right] \\
&= \rho_{\mathcal{A}}(t)\left[1+\frac{1}{\gamma(\mu-1)-\mu}\right] \\
&= \rho_{\mathcal{A}}(t)\left[\frac{(\gamma-1)(\mu-1)}{(\gamma-1)(\mu-1)-1}\right],
\end{aligned}$$

which is exactly what was required in claim 2, thus completing the lemma. □

Theorem 3.6 follows by simply integrating over t and applying Lemma 3.7. Combining Theorems 3.4 and 3.6 leads to our main result.

COROLLARY 3.8. *The competitive ratio of the single server algorithm $\mathcal{A}$ for online scheduling is at most:*

$$cr(\mathcal{A}) \leq 1 + \gamma \cdot \frac{s}{s-\mu} \cdot \left[\frac{(\gamma-1)(\mu-1)}{(\gamma-1)(\mu-1)-1}\right]. \quad (11)$$

Optimizing the bound. The bound on $\mathrm{cr}(\mathcal{A})$ can be further optimized. A straightforward calculation shows that for any value of μ, the bound (11) is minimized for a unique optimal value of $\gamma^*(\mu) = \frac{\sqrt{\mu}}{\sqrt{\mu}-1}$; at this value, the bound becomes:

$$\mathrm{cr}(\mathcal{A}) \leq 1 + \frac{s}{s-\mu} \cdot \frac{\sqrt{\mu}+1}{\sqrt{\mu}-1}.$$

There are two ways to interpret the above result. One may think of μ as a constraint set by the service provider. For example, by setting $\mu = s/2$, the service provider can limit the starting time of jobs to the first half of their availability window; as a result, the bound becomes $3 + O\left(1/\sqrt{s}\right)$ for $s > 2$. On the other hand, the bound can be further optimized. By choosing $\mu \approx s^{2/3}$ for $s > 2$, or $\mu = (s+1)/2$ for $1 < s < 2$, we obtain the bounds stated in the introduction:

$$\mathrm{cr}(\mathcal{A}) \leq \begin{cases} 3 + O\left(\frac{1}{(s-1)^2}\right) & 1 < s < 2 \\ 2 + O\left(\frac{1}{\sqrt[3]{s}}\right) & s \geq 2 \end{cases}$$

We note that one can optimize over μ and obtain more explicit bounds on the competitive ratio. We omit the details for brevity.

3.2.2 *Bounding the Loss over Incomplete Jobs*

In general, non-committed scheduling algorithms may begin processing jobs without having to complete them. This may lead to an undesired result, where the loss over incomplete jobs is relatively large compared to the gained value. To limit this loss, the service provider can incorporate the loss directly into the objective function. Let f be a penalty factor set by the service provider. Consider the following alternative objective:

$$\text{maximize } v(\mathcal{J}^F) - f \cdot v(\mathcal{J}^P). \quad (12)$$

We show that our algorithm maintains low loss compared to the value it gains. Proof details can be found in the full paper.

THEOREM 3.9. *For the objective* (12)*, the competitive ratio of the non-committed algorithm $\mathcal{A}$ is at most:*

$$\left(1 + \gamma \cdot \frac{s}{s-\mu} \cdot \left[\frac{(\gamma-1)(\mu-1)}{(\gamma-1)(\mu-1)-1}\right]\right) \cdot$$
$$\cdot \left[1 - \frac{f}{(\gamma-1)(\mu-1)-1}\right]^{-1}. \quad (13)$$

3.3 Multiple Servers

The focus of this section is the 0-k Resource Allocation model for multiple identical servers, presented in Section 2. Here, each job may be allocated, at any point, any number of servers between 0 and a bound k set by the service provider. In this section, we construct a new algorithm called the NONCOMMITTED algorithm, which obtains the same competitive ratio as its single server counterpart, regardless of the values of k and C.

Our construction is incremental. We first consider the case of $k = 1$, in which each job may be allocated at most a single server at any time. Then, we generalize our solution to any value of k without incurring any loss in performance or runtime.

0-1 Resource Allocation. The algorithm for multiple servers with $k = 1$, which we denote by $\mathcal{M}$, is based on its single server counterpart $\mathcal{A}$. From an overall perspective, the algorithm $\mathcal{M}$ runs a local copy of the single server algorithm $\mathcal{A}$ on each of the C servers, under a restriction called the *job locality* restriction. According to the job locality restriction, a job preempted from server i may only resume its execution on server i; in other words, the algorithm prevents migration of preempted jobs between servers. The algorithm, given fully in Algorithm 2, executes this general approach in an efficient manner. When job j arrival at time t, we only invoke the threshold preemption rule on server $i_{\min}(t)$, which is the server running the job with lowest value-density (unused servers run idle jobs of value-density 0). Notice that it suffices to invoke the threshold preemption rule of server $i_{\min}(t)$: if job j is rejected, it would be rejected by the threshold preemption rule of any other server. When job j completes on server i, we first load the job with maximal value-density out of the jobs preempted from server i, and then invoke the threshold preemption rule.

Algorithm 2: Multiple Servers with $k = 1$

```
Event: On arrival of job $j$ at time $t = a_j$:
  1. Call Threshold Preemption Rule ($i_{\min}(t), t$), with:
       $i_{\min}(t) = \arg\min \{\rho^i_{\mathcal{M}}(t) \mid 1 \leq i \leq C\}$.

Event: On job completion on server $i$ at time $t$:
  1. Resume execution of the preempted job with highest
     value-density among jobs preempted from server $i$.
  2. Call Threshold Preemption Rule($i, t$).

Threshold Preemption Rule ($i, t$):
  1. $j \leftarrow$ job currently processed on server $i$.
  2. $j^* \leftarrow \arg\max \{\rho_{j^*} \mid j^* \in A^{-\mu}(t)\}$.
  3. if ($\rho_{j^*} > \gamma \cdot \rho_j$)
       3.1. Preempt $j$ and run $j^*$ on server $i$.
```

Theorem 3.10 summarizes our analysis of the multiple server algorithm $\mathcal{M}$. The arguments used in the single server analysis can be extended to the multiple server case, without incurring any loss of guaranteed performance. Since the analysis is conceptually similar to its single server counterpart, we omit the full details from the proceedings version and only provide a high-level proof sketch.

THEOREM 3.10. *The algorithm $\mathcal{M}$ for multiple servers with no parallelized scheduling obtains a competitive ratio of at most:*

$$\mathrm{cr}(\mathcal{M}) \leq 1 + \gamma \cdot \frac{s}{s-\mu} \cdot \left[\frac{(\gamma-1)(\mu-1)}{(\gamma-1)(\mu-1)-1}\right].$$

PROOF SKETCH. The analysis is relatively similar to its single server equivalent, apart from several adjustments. As before, the dual fitting analysis will provide a bound on the competitive ratio of $\mathcal{M}$. In the multiple server case, this bound depends on the expression $\sum_{i=1}^{C} \int_0^\infty \rho^i_{\mathcal{M}}(t)dt$, which represents the total value gained by the algorithm if it were to gain a partial value of $\rho_j \Delta_j$ for each partially processed job j. We can apply the proof of Lemma 3.7 on each server individually and sum across all servers, and as a result bound this expression by $v(\mathcal{J}^F) \cdot \left[\frac{(\gamma-1)(\mu-1)}{(\gamma-1)(\mu-1)-1}\right]$.

What complicates the proof is the dual fitting argument. Notice that now there is a dual constraint for each tuple (j, i, t), and all of them should be covered. To understand how to overcome the difficulties, we first explain why the single server analysis does not work in its current form. Consider a partially processed job j and consider some time t in which the job was processed. Let i be the server that processed job j at time t. Recall that we set $\alpha_j = 0$ for j. Since we did not use the π variables so far, the dual constraints corresponding to job j at time t can only be covered by the $\beta_i(t)$ variables. In the original analysis, the $\beta_i(t)$ variable has been set to the highest value-density of an unprocessed job at time t. However, setting $\beta_i(t)$ so does not necessarily cover the dual constraint of (j, i', t) for any server i'. Even by setting $\beta_i(t) = \gamma \cdot \rho^i_{\mathcal{M}}(t)$, we cannot guarantee that a constraint (j, i', t) for $i' \neq i$ will be covered. To overcome this problem, we use the $\pi_j(t)$ variables. The usage of these variables, and the necessary adjustments needed in order to correct the analysis, are both highly non-trivial and technical. We refer the reader to the full version of the paper for the complete analysis. $\square$

0-k Resource Allocation. Up to now we have restricted the execution of every job to at most one server at any time. In the following, we consider a more general model, in which each job may be processed simultaneously on any number of servers, up to a parallelism bound k set by the service provider. The service provider may flexibly change the resource usage of each job at any point.

Our solution reduces the problem to the 0-1 resource allocation case. We divide the C servers into C/k equal-sized "virtual clusters" (VCs). We assume k divides C for ease of exposition. In our solution, jobs are allocated to VCs rather than to individual servers. When a job is allocated to a VC, it runs on all of its k servers in parallel. Each VC runs a copy of the single server algorithm $\mathcal{A}$, under the job locality restriction, just as in Algorithm 2.

Algorithm 3: Multiple Servers

```
1. Divide the $C$ servers into $C/k$ equal-sized clusters.
2. Run Algorithm 2 under the following modifications:
    - Capacity:  $C/k$.
    - Demand:  $D_j/k$ for each job $j$.
```

We prove that Algorithm 3 guarantees the same competitive ratio $\text{cr}(\mathcal{A})$ as for the single server case. The proof directly follows from our dual fitting analysis, therefore we leave it to the full version. We note that the algorithm we provide overcomes some concerns that may arise in practical settings. For example, the dynamic allocation of resources might in principle incur high network costs due to large data transfers. However, the job locality feature of our algorithm prevents jobs from migrating across VCs, thereby minimizing communication overheads. We emphasize that imposing this feature does not affect performance, as we guarantee the same competitive ratio of the single server algorithm. In other words, while our algorithm does not migrate jobs across VCs, and only ever allocates 0 or k resources to a job at any given time, our performance bounds are with respect to an optimal schedule without any such restrictions.

3.4 Demand Uncertainty

Up until now we have made a simplifying assumption that job resource requirements can be precisely specified by the job owner. Often times, however, the resource requirements may only be estimated, due to various reasons (unexpected data-processing overhead, outliers, etc.). We modify our algorithms to handle demand distributions with relatively low tail probability. Let α be the allowed deviation of the job resource requirement from the initial estimation (i.e., the expected demand). Formally, the resource demand may exceeds its expectation by a multiplicative factor of $(1 + \alpha)$, with probability at most β. We adjust our solution to accommodate this uncertainty model, and show that the resulting algorithm exhibits low degradation of performance compared to the deterministic case.

THEOREM 3.11. *The algorithm for online preempted scheduling on multiple servers under uncertainty, where each job's true demand exceeds the reported demand by a factor of* $(1 + \alpha)$ *with probability at most* β*, assuming* $\mu + \alpha + 1 < s$*, obtains a competitive ratio of at most:*

$$\frac{1}{1-\beta}\left(1 + \gamma \cdot \frac{s}{s-\alpha} \cdot \frac{s}{s-\mu} \cdot \left[\frac{(\gamma-1)(\mu-1)}{(\gamma-1)(\mu-1)-1}\right]\right).$$

4. COMMITTED SCHEDULING

In the previous section we focused on non-committed preemptive scheduling, in which the scheduler is not required to complete jobs that had been admitted to the system. While this setting may be plausible in some applications, for example when all of the jobs belong to the same user, some applications do require a guarantee for completion. We therefore consider the design of *committed* online scheduling algorithms, whereby the algorithm guarantees that admitted jobs are completed by their deadlines. We distinguish between two models of commitment, differing by the timing of the commitment.

1. *Commitment on arrival*: jobs are notified upon arrival whether they will be completed or rejected.
2. *Commitment on admission*: a less restrictive model, in which the scheduling algorithm may delay the decision whether to commit to a job. However, once the job is admitted (begins execution), the scheduler is guaranteed to complete it by its deadline.

Due to space limitations, we omit our full analysis and empirical evaluation from this section. We refer the reader for the extended version of the paper for more details.

4.1 Commitment on Arrival

The strict requirement of making the scheduling decision upon arrival leads to the following negative result

THEOREM 4.1. *Any online algorithm that commits to jobs on arrival has an unbounded competitive ratio for any slackness parameter* s.

4.2 Commitment on Admission

We design a heuristic solution, called COMMITTED, for this less restrictive commitment model. Our heuristic is based on the algorithm developed in Section 3 for non-committed scheduling. Specifically, to ensure that all committed jobs are completed, we modify the original threshold preemption rule as follows. An unallocated job j^* that passes the threshold choice rule of a server i will be allocated (i.e. will preempt the currently running job) only if no commitments are violated as a result. That is, we simulate the future schedule on server i that would occur if all partially executed jobs on that server had their completion times pushed back by D_{j^*}/k. If one of the completion times is pushed beyond the deadline of the corresponding job, we reject job j^* from server i. It is an open question whether this heuristic (or any other online algorithm) admits a satisfactory competitive ratio. Consequently, we evaluate the performance of our solution through extensive simulations, see Section 4.4.

We note that the gap parameter μ provides a method for the service provider to interpolate between the extremes of commitment on arrival and commitment on admission. For example, if $\mu = s/2$ then the service provider will necessarily commit to accepting or rejecting each job by the halfway point of its admission window at the latest. Larger values of μ result in even earlier notification times (at the expense of throughput), up to the extreme of $\mu = s$ which corresponds to commitment on arrival.

4.3 Economic-Driven Mechanisms with Commitment on Allocation

The design of scheduling mechanisms for external applications, such as paid cloud services, involves additional complications derived from the selfish nature of participants. Such mechanisms must be sensitive to potential manipulation by customers, as users striving to maximize their personal gain may attempt to do so by reporting false values or job parameters. Our previous work considers economic aspects of deadline-sensitive scheduling in the offline setting [10]; however, we must extend the insights from those works to handle online job arrivals. Moreover, the algorithmic solutions on which these mechanisms rely must satisfy economic-driven properties, such as monotonicity.

We suggest a scheduling mechanism for online scheduling called TRUTHFULCOMMITTED, which satisfies desirable economic-driven properties. The algorithmic core of the mechanism is a simplification of the COMMITTED algorithm, obtained by setting $\mu = 1$. We prove that the general framework described in [10] can be applied in this context, which leads to the design of a mechanism that is *truthful in dominant strategies*, meaning that it is always in the best interest of a customer to report its real job value and parameters (demand, deadline). The proof of truthfulness can be found in the full paper; the evaluation is briefly discussed in Section 4.4.

4.4 Empirical Evaluation

We evaluate the performance of our heuristic solutions, by comparing them to the non-committed algorithm, as well as to straw man mechanisms that are used in practice. To that end, we test all the scheduling mechanisms on both synthetic and empirical traces (extracted from Microsoft's production cluster). The main goal of the experiments is to compare the obtained throughput across different solutions, and also examine the value of incomplete jobs under the non-committed scheduler. The main highlights of our empirical study are the following. We show that for reasonable values of the input slackness ($s > 2$), COMMITTED and TRUTHFULCOMMITTED achieve nearly identical performance to the non-committed algorithm, for which we proved worst-case performance guarantees. Moreover, our algorithms outperforms straw man mechanisms by an order of magnitude (typically 10-50x).

5. CONCLUDING REMARKS

This paper introduces novel solutions for deadline-aware scheduling in large computing clusters. Our solution methodology is built upon two plausible assumptions: that the input exhibits deadline slackness, and that the provider has some leeway on its required responsiveness. Using our methodology, we design a simple online algorithm and prove constant-factor approximation guarantees for its performance. Based on this algorithm, we design additional heuristics that address important practical concerns such as provider commitments, demand uncertainties, and economic constraints. Our experiments on both synthetic and real job traces demonstrate the dominance of our scheduling framework over other potential heuristics. We also discuss how our framework can be used to design a truthful mechanism for online scheduling.

Our results motivate future study of more sophisticated models and scenarios. Specifically, we plan to consider data-processing job models (e.g, Hadoop, COSMOS), in which the system could benefit from uneven and time-varying allocation of resources across jobs (e.g., allocate resources differently for different "phases" of the job). Another challenging direction is to extend our framework to multi-dimensional resource allocation problems, e.g. allocating both CPU and bandwidth; see e.g., [5, 11]. We believe that this paper provides guidelines for designing mechanisms for the above cases and beyond.

6. REFERENCES

[1] A. Bar-Noy, R. Canetti, S. Kutten, Y. Mansour, and B. Schieber. Bandwidth allocation with preemption. *SIAM J. Comput.*, 28(5):1806–1828, 1999.

[2] R. Canetti and S. Irani. Bounding the power of preemption in randomized scheduling. *SIAM J. Comput.*, 27(4):993–1015, 1998.

[3] A. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 99–112. ACM, 2012.

[4] J. A. Garay, J. Naor, B. Yener, and P. Zhao. On-line admission control and packet scheduling with interleaving. In *INFOCOM*, 2002.

[5] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica. Multi-resource fair queueing for packet processing. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 1–12. ACM, 2012.

[6] M. T. Hajiaghayi, R. Kleinberg, M. Mahdian, and D. C. Parkes. Online auctions with re-usable goods. pages 165–174, 2005.

[7] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 18. ACM, 2011.

[8] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 261–276. ACM, 2009.

[9] N. Jain, I. Menache, J. Naor, and J. Yaniv. A truthful mechanism for value-based scheduling in cloud computing. *Algorithmic Game Theory*, pages 178–189, 2011.

[10] N. Jain, I. Menache, J. Naor, and J. Yaniv. Near-optimal scheduling mechanisms for deadline-sensitive jobs in large computing clusters. In *SPAA*, pages 255–266, 2012.

[11] V. Jalaparti, H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Bridging the tenant-provider gap in cloud services. In *ACM Symposium on Cloud Computing*. ACM, 2012.

[12] G. Koren and D. Shasha. D$^{\text{over}}$; an optimal on-line scheduling algorithm for overloaded real-time systems. In *RTSS*, pages 290–299. IEEE Computer Society, 1992.

[13] G. Koren and D. Shasha. Moca: A multiprocessor on-line competitive algorithm for real-time system scheduling. *Theor. Comput. Sci.*, 128(1&2):75–97, 1994.