Secondary Job Scheduling in the Cloud with Deadlines

Shiyao Chen† Ting He† Ho Yin Starsky Wong† Kang-Won Lee† Lang Tong†

Abstract—The highly dynamic nature of the cloud environment leads to a time-varying resource utilization and the cloud provider can potentially accommodate secondary jobs with the remaining resource. To better implement the idea of resource reutilization in the cloud environment, the problem of secondary job scheduling with deadlines under time-varying resource capacity is considered in this paper. A transformation is proposed to reduce the offline problem with time-varying processor capacity to that with constant capacity. For online scheduling of underloaded system, it is shown that the earliest deadline first (EDF) scheduling algorithm achieves competitive ratio 1. For the overloaded system, an online scheduling algorithm V-Dover is proposed with asymptotically optimal competitive ratio when a certain admissibility condition holds. It is further shown that, in the absence of the admissibility condition, no online scheduling algorithm exists with a positive competitive ratio. Simulation results are presented to illustrate the performance advantage of the proposed V-Dover algorithm.

Keywords—Cloud computing; deadline scheduling; competitive ratio analysis; time-varying processor capacity.

I. INTRODUCTION

The cloud computing environment is highly dynamic; the system load and the computing resource utilization exhibit a rapidly changing characteristic over time. Therefore the cloud service provider normally over-provision the computing resources to accommodate the peak load and the computing resources are typically left under-utilized at non-peak times.

The often significantly under-utilized cloud resources led to the potential of exploiting surplus resources for secondary jobs in cloud systems. Amazon EC2 Spot Instances [1] is a commercial implementation of such an idea of opportunistic resource reutilization. It allows customers to bid on unused Amazon EC2 capacity and acquire a VM instance for as long as their bids exceed the current spot price, which fluctuates with the supply and demand. EC2 Spot Instances is complementary to EC2 On-Demand Instances and Reserved Instances, providing a more economic option for obtaining the computing capacity.

While Amazon EC2 Spot Instances provides a new resource reutilization model for the cloud, the burden of resource management to complete a job is essentially with the customer. In this paper, we investigate the challenging problem of guaranteeing an important QoS metric for elastic compute jobs, namely job completion time for opportunistic cloud users in the framework of job scheduling with deadlines. In particular, we formulate the problem of scheduling low priority jobs, also referred to as secondary jobs, on the time-varying surplus cloud resources left by the execution of the high priority jobs or primary jobs. The proposed scheduling policy is describe in the context of a single server where the secondary jobs are virtual machines (VMs) for low-priority applications that can be dynamically sized to fit the remaining server resource (e.g., CPU capacity) in this paper. But the same policy can be applied to the cloud-wise scheduling of secondary user (e.g., Spot Market user) demands on unsold cloud instances with extensions.

In order to model this environment we associate with each secondary job a release time, certain deadline, a processing load requirement and a value (e.g., monetary income). The essence of such scheduling is to maximally pack secondary jobs onto shared resources without affecting the primary jobs’ execution. In particular, we assume the execution of a secondary job must be suspended to accommodate primary jobs upon conflict, and can also be preempted by other secondary jobs with greater potential values. The aim of the secondary job scheduler is to maximize the value obtained from the secondary jobs that meet their deadlines, under the remaining resource capacity left by the execution of the primary jobs. Because of the dynamic nature of primary job activities, the amount of available resource for secondary jobs will be time-varying. We model the resources available to the secondary jobs as a time-varying function observable to the scheduler in either online or offline manner.

A. Related Work

There are two main approaches to resource reutilization for jobs of different priorities which can potentially enable hosting secondary jobs in data centers. The first approach comes from real-time scheduling theory [2]. Solutions such as slack stealing, virtualization, or bandwidth reclaiming, are centralized techniques that schedule both primary and secondary jobs jointly. In this paper, we assume that secondary jobs cannot affect the scheduling of primary jobs, and only the leftover capacity can be exploited. As such, our approach is non-intrusive to the contracted cloud users.

The second approach exploits the idea of non-intrusive resource reutilization and proposes the notion of cycle stealing [3]. Systems such as Condor [4] represent practical implementations of this idea. These solutions enable fine-
grained resource reutilization thus provide practical mechanism for resource sharing between primary and secondary jobs. Based on these techniques, we model the resources consumed by primary jobs as a fraction of the actual computing resources that continuously fluctuates over time and characterize analytically the performance of non-intrusive secondary job scheduling algorithms.

Our work builds upon the past research in the job scheduling theory, both offline and online scheduling. Traditional job scheduling problem has assumed that the available processor capacity is constant (instead of time-varying), e.g., Dertouzos and Mok [5], Liu and Layland [6], Dertouzos [7], and Mok [8]. The offline scheduling problem with constant capacity has been known to be NP-hard [5], [9]. The online scheduling problem with constant capacity is first considered for the underloaded system. Specifically, the scheduling system is underloaded under an input set (consists of different input instances), if for any input instance (consists of a number of jobs to be scheduled) in the input set there exists an offline schedule such that all the jobs are finished by their deadlines. It has been shown that, for underloaded systems, simple online algorithms such as earliest deadline first (EDF) [6], [7] and least laxity first (LLF) [8] achieves the same performance as the optimal offline algorithm. The assumption of an underloaded system, however, is too restrictive and unverifiable in practice. Locke showed in [10] that both EDF and LLF can perform poorly when the system is overloaded.

There were efforts to develop an online scheduling algorithm with performance guarantee in terms of competitive ratio (a metric that measures how well an online algorithm performs against the optimal offline algorithm, to be defined in Definition 1), even when the system is overloaded. An online scheduling algorithm named Dover is proposed by Koren and Shasha in [11] for the constant capacity problem and shown to have the optimal competitive ratio of \( \frac{1}{(1+\sqrt{k})^2} \) where \( k \) is the upper bound of the importance ratio (to be defined in Definition 3).

When the processor capacity is time-varying, job scheduling with deadlines becomes much more challenging. The only existing work to our knowledge that tackles varying processor capacity is [12], which assumes the jobs and the capacity are known a priori, thus provides an offline solution. Also assumed is that scheduled jobs cannot be preempted. Such assumptions do not fit in the cloud computing setting since newly released primary jobs may take away available resources, making preempting secondary jobs necessary.

B. Summary of Results

We present new theoretical results for the offline and online job scheduling problem in a system with time-varying processor capacity. For the offline version, a transformation of the processor capacity and the job parameters is proposed to reduce the offline problem with time-varying capacity to that with constant capacity. For the online version, scheduling algorithms and competitive ratio analysis are given for both underloaded and overloaded systems. For underloaded systems, we show that EDF is still an optimal algorithm that achieves competitive ratio 1 under time-varying capacity. For overloaded systems, we propose an online scheduling algorithm V-Dover with asymptotically optimal competitive ratio when a certain admissibility condition holds. We also show that no online schedule exists with a positive competitive ratio when the admissibility condition does not hold. While the competitive ratio result guarantees the worst-case performance of V-Dover algorithm, we also evaluate its average performance by simulations, which show that V-Dover algorithm outperforms the state of art in the literature in all simulated cases.

II. PROBLEM FORMULATION AND KNOWN RESULTS

A. Problem Formulation

In our scheduling problem an input instance \( I \) includes an integer \( n \) and \( n \) jobs \( T_1, \ldots, T_n \) released with their workload \( p_i \), release time \( r_i \), deadline \( d_i \) and value \( v_i \). The jobs are scheduled on a single processor with time-varying capacity. The input instance also includes the processor capacity, which is given by a function \( c(t) \) over time \( t \) for the duration of the jobs, i.e., \( t \in [\max_j d_j, \min_i r_i] \).

In the process of scheduling an online scheduler knows the job parameters at the release time and the processor capacity \( c(t) \) up to the current time (at time \( t \), capacity function \( c(\tau) \) is known for \( \tau \leq t \)). Deadlines are firm, i.e., completing a job after its deadline yields zero value. The aim of the scheduler is to maximize the total value the system obtains from the jobs which are completed by their deadlines.

In contrast to the online scheduler, the offline scheduler is clairvoyant and knows the entire input instance, including the set of jobs released and the capacity function \( c(t) \) a priori. We assume the input capacity function has the following structure

\[
C(\underline{c}, \overline{c}) = \{c(t) \mid c(t) \text{ integrable, } \underline{c} \leq c(t) \leq \overline{c}\}.
\]

\( \delta = \overline{c}/\underline{c} \) indicates maximum variation of \( c(t) \). In the system the workload that can be finished in time interval \( [t_1, t_2] \) is given by \( \int_{t_1}^{t_2} c(\tau)d\tau \), as the name processor capacity indicates. Preemption is allowed at no cost in scheduling (i.e., a preempted job can be restarted from the point of preemption at a later time). The problem is to design an offline algorithm that maximizes the system value and an online algorithm with worst-case performance guarantee even in the presence of overload. The performance guarantee is given in terms of competitive ratio defined below.

Definition 1. Competitive ratio: An online algorithm \( A \) is \( \alpha \)-competitive for an input set \( I \) if \( \frac{\text{opt}(I)}{\text{opt}(I)} \geq \alpha \) where \( \text{opt}(I) \) varies over all possible input instances in \( I \), and \( A(I) \) and
opt(I) are the values the online algorithm A and the optimal offline algorithm obtain on input instance I, respectively.

That is, an \( \alpha \)-competitive online algorithm is guaranteed to achieve at least \( \alpha \) fraction of the optimal offline value under any input instance \( I \) in the input set \( \mathcal{I} \).

**B. Known Results for Constant Capacity Scenario**

In the constant capacity case \( (c = \tau = e) \), workload reduces to processing time with \( t_r(T_i) = p_r(T_i)/c \) where \( t_r(T_i) \) and \( p_r(T_i) \) are the remaining processing time and workload of job \( T_i \), respectively. The offline version of the constant capacity problem has been known to be NP-hard [5], [9]. For the online version with constant capacity we summarize the known competitive ratio results for under-loaded and overloaded systems separately. Before stating the results, we need several terminologies.

**Definition 2. Laxity** of job \( T_i \) at time \( t \) is \( d_i - t - t_r(T_i) \) where \( d_i \) and \( t_r(T_i) \) are the deadline and the remaining processing time of \( T_i \) at \( t \).

The laxity of job \( T_i \) indicates the maximum time the processor can stay idle or execute other jobs before \( d_i \), in order to guarantee the completion of \( T_i \) by its deadline \( d_i \). The laxity is a measure of the urgency of the jobs at the current time (a job with small laxity would have to be scheduled soon in order to meet the deadline).

**Definition 3. The value density** of job \( T_i \) is defined to be \( v_i/p_i \), i.e., its value divided by its workload.

The importance ratio of a collection of jobs \( I = \{T_1, \ldots, T_n\} \) is the ratio of the greatest value density to the least value density, i.e., \( k_I \equiv \max_{1 \leq i \leq n} v_i/p_i / \min_{1 \leq i \leq n} v_i/p_i \).

For convenience, we will always normalize the smallest value density to 1 (\( \min_{1 \leq i \leq n} v_i/p_i = 1 \)).

**Theorem 1.**

1) For underloaded systems EDF [6], [7] and LLF [8] both achieve competitive ratio 1.
2) For overloaded systems \( \frac{1}{(1+\sqrt{k})^2} \) is the optimal competitive ratio for input set \( \mathcal{I} = \{I \mid k_I \leq k\} \) [11] where \( k_I \) is the importance ratio of the input instance \( I \in \mathcal{I} \).

Dover in [11] achieves the competitive ratio \( \frac{1}{(1+\sqrt{k})^2} \), and \( 1/(1+\sqrt{k})^2 \) is meanwhile established as an upper bound of the competitive ratio for \( \mathcal{I} = \{I \mid k_I \leq k\} \) via an adversary argument.

**III. MAIN RESULTS: VARYING CAPACITY SCENARIO**

**A. Offline scheduling**

The main insight in the offline case is that the varying capacity problem can be reduced to the constant capacity problem via a time-scale stretch transformation of the processor capacity and the job parameters as described in the following.

Define the stretch function, which maps the original time \( t \) to the stretched time \( t' \) by \( t' = T(t; e) = \frac{1}{e} \int_0^t c(\tau) d\tau \). We suppress the argument \( e \) in later presentation. Define the stretched capacity to be \( c'(t') \equiv c \). Thus after the stretch transformation the processor capacity becomes constant. The transformation preserves the workload in between any two time epochs \( s \) and \( t \), i.e., \( \int_s^t c'(\tau) d\tau = \int_s^t c(\tau) d\tau \).

The transformation maps the original jobs \( T'_i \) to a set of stretched jobs \( T'_i \) which have the parameters as follows: release time \( r' \equiv T(r) \), workload \( p' \equiv p \), deadline \( d' = T(d) \) and value \( v' \equiv v \). The problem is then reduced to scheduling the set of stretched jobs \( T'_i \) in offline manner on the processor with constant capacity \( c' \equiv c \) after the transformation.

Specifically, given an offline schedule for the original system under which job \( T_i \) is executed during time intervals \( I_{1i}, \ldots, I_{ni} \), an offline schedule for the transformed system can be obtained by executing job \( T_i \) during time intervals \( T(I_{1i}), \ldots, T(I_{ni}) \), where the start and end points of \( T(I_i) \) are the image of the start and end points of \( I_i \) under the transformation. Similarly, we can obtain an offline schedule for the original system given an offline schedule for the transformed system. Therefore we have defined an offline schedule pair for the original and transformed systems. Since the transformation preserves the workload in between any two time epochs, job \( T_i \) completes by its deadline \( d_i \) under the schedule for the original system if and only if job \( T'_i \) completes by its deadline \( d'_i \) under the schedule for the transformed system.

Therefore we have established a bijection from the solutions (i.e., offline schedules) of the original system to the solutions of the transformed system which preserves the value obtained by the scheduler. Therefore the problem is reduced to scheduling the set of stretched jobs \( T'_i \) in offline manner on the processor with constant capacity \( c' \equiv c \). To this end the approximation algorithms for offline job scheduling can be readily applied to the varying capacity problem.

**B. Online scheduling: underloaded systems**

For online scheduling in underloaded systems with varying capacity, our main result is that EDF is still optimal and achieves competitive ratio 1 (Theorem 2). We note that it is difficult to generalize LLF for our problem because the remain processing time (or laxity) is not known in the time-varying capacity system.

**Theorem 2.** EDF achieves competitive ratio 1 for underloaded systems.

**Proof:** See the technical report [13].

Theorem 2 indicates that even though the capacity is time-varying, the restriction to the underloaded input set \( \mathcal{I} \) guarantees the optimality of EDF.
C. Online scheduling: overloaded systems

In the overloaded case the workload of input set \( I \) is greater than that of the system capacity. We find a certain condition (namely individual admissibility in Definition 4) has to be met to guarantee the performance of online scheduling algorithms.

**Definition 4.** We say job \( T_i \) is **individually admissible** if it can always be completed before its deadline \( d_i \), regardless of the capacity variation, had it been the only (secondary) job being scheduled, i.e., \( d_i - r_i \geq p_i/c \).

The notion of individual admissibility is introduced to guarantee that online scheduling algorithms with positive competitive ratio exist (see Theorem 3 (3)). In the constant capacity case (\( \sigma = \zeta = c \)), the condition reduces to \( d_i - r_i \geq p_i/c \), and it is not crucial for the constant capacity case since if for job \( T_i \), \( d_i - r_i < p_i/c \), neither the offline nor the online scheduler can complete \( T_i \) by \( d_i \). Therefore \( T_i \) can be effectively deleted from the input instance without affecting the competitive ratio. However, when the capacity input set has \( \sigma > \zeta \). Theorem 3 (3) shows that the presence of a job not individually admissible makes positive competitive ratio impossible. Also worth noting is that the statement that all jobs in the input set \( I \) are individually admissible and the statement that the system is underloaded under the input set \( I \) do not imply each other. Individually admissibility condition sets up the restriction on the input set such that each job has reasonable deadline with respect to the workload. On the other hand, underloaded condition restricts the input set such that all input instances (including the job input and capacity input pair) are guaranteed to be schedulable. The competitive ratio result for the overloaded system is provided in the following theorem.

**Theorem 3.** For overloaded systems, fix the input job set \( I = \{ I \mid k_I \leq k \} \) and the input capacity set \( C(\zeta, \sigma) \) where \( k_I \) is the importance ratio of input instance \( I \) and \( \delta = \sigma/\zeta > 1 \).

1) The competitive ratio \( \left( \sqrt{\frac{\sigma}{\zeta}} + \frac{1}{\sqrt{f(k, \delta)^2 + 1}} \right) \) upper bounds the competitive ratio.
2) The competitive ratio \( \frac{\sqrt{f(k, \delta)}}{\sqrt{f(k, \delta)^2 + 1}} \) is achievable, given that all jobs in \( I \) are individually admissible, where \( f(k, \delta) = 2\delta + 2 + \log(\delta k)/\log(\frac{\delta}{\delta - 1}) \).
3) The optimal competitive ratio is \( 0 \), if the input job set can contain jobs not individually admissible.

The result in Theorem 3 (2) is asymptotically optimal since as \( k \) tends to infinity the ratio between the achievability result in Theorem 3 (2) and the upper bound in Theorem 3 (1) approaches 1.

D. V-Dover Scheduling Algorithm

To prove Theorem 3 we propose V-Dover, a variant of Dover algorithm and analyze its competitive ratio. The proposed variant V-Dover is a combination of EDF and LLF. In the description and analysis of the algorithm, we assume that the individual admissibility condition holds.

Under time varying capacity the dynamics of the system with V-Dover scheduler can be described as follows: initially the system is underloaded and the jobs are finished in an EDF manner; from a certain moment, the job arrival gets heavier and accumulates to an overload; after some period of time, the overload is detected by the scheduler and resolved by selecting the jobs according to their value; later at a certain point in time, the system may be underloaded again and some of the jobs not selected previously may get scheduled at this point in time, provided they have not passed their deadlines yet.

We now describe the details of the algorithm. There are three types of interrupts, each associated with an interrupt handler. The scheduler waits for interrupts in a loop and calls the interrupt handlers upon interrupts to schedule the current ready jobs. The set of ready jobs at a certain moment is the set of jobs that are already released with deadlines not yet passed, excluding the currently executing job. We first define the notion of conservative laxity used in the scheduling algorithm.

**Definition 5.** Conservative laxity of job \( T_i \) at time \( t \) is \( claxity(T) = d_i - t - p_r(T)/c \), i.e., the laxity of \( T_i \) at \( t \) if the future capacity is always \( c \).

We observe that conservative laxity employs the most conservative estimation of the future capacity. The three types of interrupts in the algorithm are: 1) job completion or failure interrupt, raised by successful termination or reaching the deadline of a currently running job; 2) job release interrupt, raised by the arrival of a newly released job; 3) zero conservative laxity interrupt, raised by a job the current conservative laxity of which reaches zero. In this situation the job is considered “urgent” by the conservative estimation and the corresponding interrupt handler decides whether to schedule the “urgent” job.

The skeleton of the algorithm is given below. At time 0 the algorithm initializes the variables (line A.1) and job queues (line A.2) that the scheduler maintains and then starts the infinite loop in which the interrupt handlers are called upon interrupts.

**A: V-Dover Scheduling Algorithm**

<table>
<thead>
<tr>
<th>Procedure A: V-Dover Scheduling Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: ( cSlack = \infty ), ( flag = 0 )</td>
</tr>
<tr>
<td>2: ( QDF, QOther, QSupp = \emptyset )</td>
</tr>
<tr>
<td>3: ( \text{loop} )</td>
</tr>
<tr>
<td>4: Wait until an interrupt is raised;</td>
</tr>
<tr>
<td>5: Call the interrupt handler according to the interrupt type</td>
</tr>
</tbody>
</table>

At any given moment, the set of ready jobs is partitioned into two disjoint sets, the regular jobs and the supplement jobs. The supplement jobs are the jobs which raised zero conservative laxity interrupt at a previous time and did not
get scheduled by the corresponding handler (i.e., it had a negative conservative laxity the moment after the interrupt). In contrast, the regular jobs are those which have never had a negative conservative laxity up to the current time. Each newly released job is initially regular because of individual admissibility, and labeled supplement at the time it raises a zero conservative laxity interrupt but does not get scheduled. The set of regular jobs is again partitioned into two disjoint sets, recently EDF-scheduled regular jobs and other regular jobs, stored in job queues Qedf and Qother, respectively. The supplement jobs are stored in job queue Qsupp. The job queues are ordered by different parameters: Qedf and Qother ordered by deadline with the earliest deadline first, while Qsupp ordered by deadline with the latest deadline first. The queue operations used are: Dequeue operation outputs and deletes the first job in the queue (according to the order), FirstInQueue outputs the first job without deletion, and Insert operation inserts a certain job into the queue according to its required order.

The two variables cSlack and flag are maintained by the scheduler (line A.1). cSlack is the maximum slack time that can be devoted to new jobs without causing the currently executing regular job or the recently EDF-scheduled regular jobs (Qedf) to miss their deadlines, with the conservative estimation \(C\) of the future capacity. The processor status under the online scheduling algorithm is maintained by the variable flag: the value reg stands for executing a regular job, supp for executing a supplement job, and idle for being idle.

After the initialization at time 0, when the first job is released into the system, the job release handler is called and the procedure is given below. In the procedure now() returns the current system time and \(t_c(T, C)\) returns the conservative estimation of the remaining processing time of job \(T\).

The elements stored in Qother and Qsupp are the jobs while the elements in Qedf are tuples \((T, t_{insert}, cSlack_{insert})\) where \(T\) is the job that is previously preempted (according to EDF rule) at time \(t_{insert}\) and \(cSlack_{insert}\) is the value of cSlack at time \(t_{insert}\). The two additional quantities \(t_{insert}\) and \(cSlack_{insert}\) are maintained to update the value of cSlack.

### B: Job Release Handler procedure

1. if (flag=idle) then
2. \(T_{curr} = T_{arr}\), schedule \(T_{curr}\);
3. cSlack = claxity\((T_{curr})\);
4. flag = reg;
5. else if (flag=reg) then
6. if \(d_{arr} < d_{curr}\) and \(cSlack \geq t_c(T_{arr}, C)\) then
7. Insert \((T_{curr}, now, cSlack)\) into Qedf;
8. else if (Qedf is empty) then
9. \(T_{curr} = T_{arr}\), schedule \(T_{curr}\);
10. else
11. Insert \(T_{arr}\) into Qother;
12. flag = reg;
13. Insert \(T_{curr}\) into Qsupp;
14. Schedule \(T_{curr}\);

The newly released job \(T_{arr}\) is regular due to individual admissibility. \(T_{arr}\) gets scheduled to execution if the processor is idle (line B.2), and gets compared with the currently running job \(T_{curr}\) otherwise. If \(T_{curr}\) is regular, the deadlines of \(T_{arr}\) and \(T_{curr}\) are compared: if \(T_{curr}\) has earlier deadline and the workload of \(T_{arr}\) is no greater than \(c\cdot\text{cSlack}\) (i.e., EDF rule does not lead to overload, line B.6), then \(T_{curr}\) is preempted by \(T_{arr}\) (according to EDF rule) and becomes a recently EDF-scheduled job (line B.7 to B.9, note that the insertion of Qedf is done for the tuple on line B.7). Otherwise, if \(T_{arr}\) has later deadline or \(c\cdot\text{cSlack}\) is not big enough, \(T_{arr}\) is labeled as other regular job (line B.11). On the other hand, if \(T_{curr}\) is supplement, \(T_{arr}\) preempted \(T_{curr}\) immediately (line B.13 to B.15), i.e., the supplement jobs have lower priority than the regular jobs. This is due to the fact that the supplement jobs are considered not as promising as the regular jobs since they had a negative conservative laxity at the moment after the corresponding zero conservative laxity interrupt.

When a currently running job completes execution by its deadline or reaches its deadline with a failure, the job completion or failure handler is called, the procedure of which is given below.

### C: Job Completion or Failure Handler procedure

1. if (both Qedf and Qother are not empty) then
2. \((T_{Qedf}, t_{prev}, cSlack_{prev}) = \text{FirstInQueue(Qedf)}\);
3. cSlack = \(c\cdot\text{Slack}_{prev} - (\text{now}() - t_{prev});
4. \(T_{Qother} = \text{FirstInQueue(Qother)}\);
5. if \(d_{Qother} < d_{Qedf}\) and \(c\cdot\text{Slack} \geq t_c(T_{Qother}, C)\) then
6. Dequeue(Qother), schedule \(T_{Qother}\);
7. \(c\cdot\text{Slack} = \min(c\cdot\text{Slack} - t_c(T_{Qother}, C), \text{claxity}(T_{Qother}))\);
8. else
9. Dequeue(Qedf), schedule \(T_{Qedf}\);
10. if (Qother is not empty) then
11. \(T_{curr} = \text{Dequeue(Qother)}, \text{schedule Tcurr}\);
12. \(c\cdot\text{Slack} = \text{claxity(Tcurr)}\);
13. else if (Qedf is not empty) then
14. \((T_{curr}, t_{prev}, c\cdot\text{Slack}_{prev}) = \text{Dequeue(Qedf)}\), schedule \(T_{curr}\);
15. \(c\cdot\text{Slack} = c\cdot\text{Slack}_{prev} - (\text{now}() - t_{prev});
16. else
17. \(c\cdot\text{Slack} = \infty\);
18. if Qsupp is not empty then
19. \(T_{Qsupp} = \text{Dequeue(Qsupp)}, \text{schedule TQsupp}\);
20. flag = supp;
21. else
22. flag = idle;

Upon the successful termination or failure if there are regular jobs in the system, the regular job with the earliest deadline will be scheduled according to EDF rule (line C.6, C.11 and C.14) unless it is a job of Qother and overload is detected when scheduling the job together with Qedf.
jobs (line C.9, overload detection with \texttt{cSslack} on line C.5). Otherwise, if there are no regular jobs, the supplement job with the latest deadline gets scheduled (line C.19, working on the less promising supplement jobs).

Upon a zero conservative laxity interrupt raised by a job \( T_{0cl} \) the current conservative laxity of which reaches zero, the zero conservative laxity handler is called, the procedure of which is given below.

---

**D: Zero Conservative Laxity Handler** procedure

1. if \( \nu_{0cl} > \beta \cdot (\text{value}(T_{curr}) + \sum_{T \in \text{Qedf}} \text{value}(T)) \) then
2. Insert \( T_{curr} \) into \( \text{Qother} \);
3. Remove all jobs from \( \text{Qedf} \) and insert them into \( \text{Qother} \);
4. \( \text{cSlack} = 0 \);
5. Schedule \( T_{0cl} \);
6. else
7. Label \( T_{0cl} \) as supplement and insert \( T_{0cl} \) into \( \text{Qsupp} \);

---

The zero conservative laxity handler decides whether to schedule the “urgent” job by comparing the value of the “urgent” job with that of the jobs in \( \text{Qedf} \) and the currently executing regular job. Specifically, if the value of the “urgent” job is greater than \( \beta \) times the sum of \( T_{curr} \) and value in \( \text{Qedf} \) (line D.1), \( T_{curr} \) and all recently EDF-scheduled regular jobs are inserted into \( \text{Qother} \) and the “urgent” job gets scheduled (line D.2 to D.5). Otherwise, if the value of the “urgent” job is not big enough, it is labeled supplement (line D.7). The threshold \( \beta \) will be optimized after we derive the competitive ratio as a function of \( \beta \).

After deciding to schedule the “urgent” job, \( \text{cSlack} \) is set to 0 (line D.4) because scheduling the “urgent” job leaves no slack time for a newly released job under the conservative estimation of the future capacity.

We state the differences of the V-Dover algorithm from Dover: (i) a conservative estimation of the future capacity to handle the time-varying issue (in Dover no estimation is needed since the future capacity is known); and (ii) introducing the supplement jobs to compensate for the conservative estimation of the capacity variation. (The supplement jobs are labeled as supplement because their values are not big enough. However, the supplement jobs still have chance to complete, since the capacity varies over time in \( \xi \) and \( \tau \)).

---

**E. Analyzing the Structure of V-Dover Algorithm**

We term a regular job 0cl-scheduled if it is scheduled by the zero conservative laxity handler. With the schedules made by the three types of interrupt handlers, the V-Dover algorithm produces regular intervals defined as follows.

**Definition 6.** A regular interval is a continuous period of time from the first time (after time zero or the end of the last regular interval) that a regular job is scheduled when \( \text{Qedf} \) is empty to the first time afterwards that a regular job completes while \( \text{Qedf} \) is empty.

For example, the first regular interval begins when the execution of the first job begins, and ends either when the first time all jobs (including newly added jobs) in \( \text{Qedf} \) complete, if no job is 0cl-scheduled in this regular interval (in this case, the schedule in this regular interval coincides with EDF), or at the time of the first completion of an 0cl-scheduled job, if any. Two regular intervals may overlap only at their endpoints. Denote a regular interval by \( I_R = [\xi, \tau] \).

Denote by \( \text{regval}(I_R) \) and \( \text{clval}(I_R) \) the total value obtained from all jobs and that from 0cl-scheduled jobs during \( I_R \), respectively. We denote the union of all regular intervals by \( \text{REG} \), and refer to its total and 0cl-scheduled value by \( \text{regval} \) and \( \text{clval} \), respectively. We denote by \( \text{suppval} \) the total value obtained from supplement jobs. Lemma 1 (the varying capacity counterpart of Lemma 5.3 in [11]) upper bounds the total workload in a regular interval \( I_R \) with \( \text{regval}(I_R) \) and \( \text{clval}(I_R) \).

**Lemma 1.** The total workload of \( I_R = [\xi, \tau] \) satisfies

\[
\int_{\xi}^{\tau} c(t) dt \leq \text{regval}(I_R) + \frac{1}{\beta - 1} \text{clval}(I_R).
\]

**Proof:** See the technical report [13].

---

**F. Upper Bounding Optimal Offline Value**

Given a collection of jobs \( I \), denote the optimal value that an offline algorithm can obtain from scheduling the set of jobs \( I \) under the varying capacity \( c(t) \) by \( C(I, c) \). Later we suppress the argument \( c \). We derive an upper bound of \( C(I) \) for \( I \) being the set of released jobs. We partition the collection of jobs \( I = S^r \cup S^s \cup F \) where \( S^r \) denotes the successful regular jobs, \( S^s \) the successful supplement jobs and \( F \) the failed jobs under V-Dover algorithm.

Since \( C(F \cup S^r \cup S^s) \leq C(S^r \cup S^s) + C(F) \), we upper bound \( C(S^r \cup S^s) \) and \( C(F) \) separately. For the first term we use the equation \( \text{regval} - \text{suppval} = C(S^r \cup S^s) \). We upper bound \( C(F) \) by considering the optimal offline algorithm for \( F \) under a workload-granted-value setting. (The granted-value setting is first used in [11] to treat the constant capacity case, where certain value per \( \text{time} \) is granted to the offline scheduler.) Specifically, the offline scheduler receives an additional granted value besides the value obtained from \( F \). The amount of granted value depends on the offline schedule: a value density of \( k \) will be granted for the workload in \( \text{REG} \) that is not used for executing jobs in \( F \).

Under the granted-value setting the optimal offline scheduler must consider that scheduling a job might reduce the granted value (since workload in \( \text{REG} \) is used). Executing a job \( T_i \) results in a gain of \( v_i \) and a loss of the granted value for the workload of \( T_i \) that is executed in \( \text{REG} \).

One offline schedule under the granted-value setting is to schedule no jobs in \( F \) (therefore leaving the entire \( \text{REG} \) period untouched) and get only the (whole) granted
value. This scheduling-nothing schedule obtains a value of \( k \cdot \text{workload in } REG \). Since Lemma 1 upper bounds the total workload in a regular interval, we can upper bound the total workload in \( REG \), and thus the value obtained by the scheduling-nothing schedule.

However, the optimal offline schedule under the granted-value setting may use some workload of \( REG \) to schedule certain jobs in \( F \) to obtain more value than the scheduling-nothing schedule. To upper bound \( C(F) \) under the granted-value setting, we take the value of the scheduling-nothing schedule as a benchmark and turn to upper bounding the net gain the optimal schedule can have over the scheduling-nothing schedule by completing some jobs in \( F \).

Lemma 2 upper bounds the net gain of scheduling the jobs in \( F \) labeled supplement in \( I_R \). Lemma 2 is the key to the asymptotically optimal competitive ratio result and requires finer analysis than its counterpart in constant capacity scenario.

**Lemma 2.** Under the granted-value setting the total net gain obtained by the offline algorithm from scheduling the jobs in \( J_R = \{ T \in F \mid T \text{ is labeled supplement in } I_R = [t, t+I]\} \) is no greater than \( f(k, \delta) \beta \cdot \text{regval}(I_R) \) where \( f(k, \delta) = 2\delta + 2 + \log(\delta k) / \log(\delta k - 1) \).

**Proof:** See the technical report [13].

**G. Proving Theorem 3**

We prove Theorem 3 after bounding the net gain of scheduling the jobs in \( F \) labeled supplement in \( I_R \).

**Proof:** Claim 1): Since \( C(\xi, \xi) \) (constant capacity case) is a subset of \( C(\xi, \xi) \) where \( \delta = \tau / \xi > 1 \), and the competitive ratio decreases when the input set enlarges, Theorem 1 (2) implies that \( \frac{1}{(1+\sqrt{e})^2} \) serves as an upper bound of the competitive ratio for \( I \) and \( C(\xi, \xi) \).

Claim 2): Lemma 2 upper bounds the net gain for each regular interval. Summing over all regular intervals we can upper bound the total net gain of scheduling the jobs in \( F \) by \( f(k, \delta) \beta \cdot \text{regval}. \) Lemma 1 summed over all regular intervals yields workload in \( REG \leq \text{regval} + \text{clval} / (\beta - 1) \). Combining the upper bounds of the total net gain and the value of the scheduling-nothing benchmark, i.e., \( k \) multiplied by the workload in \( REG \), yields

\[
C(F) \leq f(k, \delta) \beta \cdot \text{regval} + k \{ \text{regval} + \text{clval} / (\beta - 1) \}
\]

Optimizing over \( \beta \) yields \( \beta = 1 + \sqrt{k / f(k, \delta)} \) and

\[
C(I) \leq C(F) + C(S^r \cup S^r) \leq \left( \sqrt{k} + \sqrt{f(k, \delta)} \right)^2 \cdot \text{regval} + C(S^r \cup S^r) \leq \left( 1 + \sqrt{k + \sqrt{f(k, \delta)}} \right)^2 \cdot \{ \text{suppval} + \text{regval} \}
\]

where \( I = F \cup S^r \cup S^r \) is the set of released jobs.

Since the value of the optimal offline and V-Dover algorithms are \( C(I) \) and \( \text{suppval} + \text{regval} \), respectively, the competitive ratio \( \frac{1}{(\sqrt{k + \sqrt{f(k, \delta)}})^2 + 1} \) is shown to be achievable, given individual admissibility.

Claim 3): An adversary argument establishes the pessimistic result of the competitive ratio when the admissibility condition does not hold true. Specifically, we construct for a fixed integer \( n \) an input instance \( I_n \) (including the job input and the capacity input), the job input of which contains one job not individually admissible, such that the competitive ratio for the singleton set \( \{ I_n \} \) is disproportional with \( n \). Taking the input set to be \( I = \{ I_n \mid n \geq 1 \} \), the best competitive ratio can be proved to be 0. See the technical report [13] for detailed proof.

**IV. SIMULATION**

We present the simulation results for the average performance of the online scheduling algorithm and compare with Dover. The simulation setup is as follows. The jobs are released according to a Poisson process with rate \( \lambda = [4.0, 5.0, 6.0, 7.0, 8.0, 10.0, 12.0] \) and the workload generated by exponential distribution with \( \mu = 1.0 \). The relative deadline (the time span from release time to deadline) is taken to be the workload divided by \( \xi \), i.e., all jobs have zero conservative laxity. The value density is uniformly distributed in \([1.0, 7.0] \). Thus \( 7.0 \) is the value of \( k \) in the simulation, i.e., the upper bound of the importance ratio. The simulation horizon is taken to be \( H = 2000.0 / \lambda \) (with expected job number 2000). The capacity \( c(t) \) follows a continuous time Markov process with two states: \( c(t) = 1.0 \) and \( c(t) = 35.0 \), and the sojourn time in each state is exponentially distributed with mean \( H/4 \). Thus in the simulation \( \xi = 1.0 \) and \( \tau = 35.0 \). Since the optimal offline value is hard to compute, we normalize the online value with the value of all jobs generated.

As an illustrative example, Fig. 1 depicts the plots of the values obtained by V-Dover and Dover versus time for \( \lambda = 6.0 \). Dover is originally proposed for the constant capacity case, and the laxity in Dover is computed with \( c = 1.0 \). However, since the capacity is time-varying in our situation, we associate with Dover a quantity \( \hat{c} \) with which Dover computes the laxity (can be thought of as Dover’s estimation of the future capacity). We test for four values of \( \hat{c} = [1.0, 10.5, 24.5, 35.0] \). The proposed V-Dover algorithm is compared with Dover with the best \( \hat{c} \) for each \( \lambda \). In Fig. 1 we first observe that V-Dover algorithm performs no worse than Dover in all tested cases. The line segments with the larger and smaller slope in Fig. 1 correspond to \( \hat{c} = 35.0 \) and \( \hat{c} = 1.0 \), respectively (more jobs can be finished under larger capacity, thus multiple value accrued). We observe in Fig. 1(a) that when \( \hat{c} = 1.0 \) V-Dover and Dover perform the same for the time when \( c(t) = 1.0 \) as one would expect, since V-Dover reduces to Dover under constant capacity. However, the performance of V-Dover gets better than Dover for the time when \( c(t) = 35.0 \), since Dover underestimates the actual capacity while V-Dover algorithm can leverage...
the increased capacity to complete supplement jobs. In Fig. 1(b)–1(d) when \( \hat{c} \) takes the other three values (10.5, 24.5 and 35.0) V-Dover and Dover perform roughly the same for the time when \( c(t) = 35.0 \). This is because (i) the capacity value \( c(t) = 35.0 \) is well above the arrival workload \( \lambda = 6.0 \); (ii) on the Dover side, \( \hat{c} \) used by Dover is also well above \( \lambda \); and (iii) on the V-Dover side, although some of the jobs are labeled supplement (since the jobs initially have zero conservative laxity), the supplement jobs are finished due to the capacity \( c(t) = 35.0 \). However, for the time when \( c(t) = 1.0 \) V-Dover outperforms Dover, since Dover overestimates the capacity \( c(t) \).

We then present the percentage of the value the online algorithms obtain out of the total value of the jobs averaged over 800 Monte Carlo runs in Table I. We compare the best values of Dover (bold numbers) with those of V-Dover and give the relative gain. We first observe that V-Dover performs no worse than Dover in all cases. For certain load (\( \lambda \in [5, 8] \)), the improvement is over 10%. As the job arrival rate \( \lambda \) gets larger or smaller, the gain decreases. Intuitively, the gain is not significant for small \( \lambda \) since there are not many supplement jobs scheduled by V-Dover; for large \( \lambda \), the gain is not significant either, since only a small portion of the supplement jobs scheduled by V-Dover is finished.

### Table I

<table>
<thead>
<tr>
<th>( \lambda )</th>
<th>( \tau = 1 )</th>
<th>( \tau = 10.5 )</th>
<th>( \tau = 24.5 )</th>
<th>( \tau = 35 )</th>
<th>Proposed</th>
<th>Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>55.0492</td>
<td>50.2227</td>
<td>50.1148</td>
<td>50.0898</td>
<td>59.8632</td>
<td>8.74</td>
</tr>
<tr>
<td>5</td>
<td>52.5579</td>
<td>49.6695</td>
<td>49.5731</td>
<td>49.542</td>
<td>58.1324</td>
<td>10.61</td>
</tr>
<tr>
<td>6</td>
<td>50.2227</td>
<td>49.4085</td>
<td>49.3237</td>
<td>49.3096</td>
<td>56.7624</td>
<td>13.02</td>
</tr>
<tr>
<td>7</td>
<td>48.3454</td>
<td>49.3596</td>
<td>49.2868</td>
<td>49.2871</td>
<td>55.7718</td>
<td>12.99</td>
</tr>
<tr>
<td>8</td>
<td>46.7045</td>
<td>49.3663</td>
<td>49.2996</td>
<td>49.2942</td>
<td>55.0123</td>
<td>11.44</td>
</tr>
<tr>
<td>9</td>
<td>44.0609</td>
<td>49.3973</td>
<td>49.3766</td>
<td>49.4071</td>
<td>54.0155</td>
<td>9.33</td>
</tr>
<tr>
<td>10</td>
<td>41.9286</td>
<td>49.5746</td>
<td>49.5874</td>
<td>49.5879</td>
<td>53.4026</td>
<td>7.69</td>
</tr>
</tbody>
</table>

Figure 1. Value versus time.

We then present the percentage of the value the online algorithms obtain out of the total value of the jobs averaged over 800 Monte Carlo runs in Table I. We compare the best values of Dover (bold numbers) with those of V-Dover and give the relative gain. We first observe that V-Dover performs no worse than Dover in all cases. For certain load (\( \lambda \in [5, 8] \)), the improvement is over 10%. As the job arrival rate \( \lambda \) gets larger or smaller, the gain decreases. Intuitively, the gain is not significant for small \( \lambda \) since there are not many supplement jobs scheduled by V-Dover; for large \( \lambda \), the gain is not significant either, since only a small portion of the supplement jobs scheduled by V-Dover is finished.

### V. CONCLUSION

We consider the problem of secondary job scheduling in the cloud with deadlines and time-varying processor capacity. We propose a transformation of the job parameters and the capacity function which reduces the offline varying capacity problem to the constant capacity problem. We study the competitive ratio of online scheduling of both underloaded and overloaded systems. For underloaded system, we show that the earliest deadline first (EDF) scheduling algorithm achieves competitive ratio 1; for overloaded system, an online scheduling algorithm V-Dover with asymptotically optimal competitive ratio is proposed under the individual admissibility condition. We also show that no online algorithm exists with positive competitive ratio if the admissibility condition is relaxed. Our simulation evaluation shows that the proposed V-Dover algorithm outperforms the best known algorithm in all cases.

### REFERENCES


