CRED: Cloud Right-sizing to Meet Execution Deadlines and Data Locality

Sultan Alamro, Maotong Xu, Tian Lan, and Suresh Subramaniam
Department of Electrical and Computer Engineering
The George Washington University, USA
{alamro, htfy8927, tlan, suresh}@gwu.edu

***Abstract*—As demands for cloud-based data processing continue to grow, cloud providers seek effective techniques that deliver value to the business without violating Service Level Agreements (SLAs). Cloud right-sizing has emerged as a very promising technique for making cloud services more cost-effective. In this paper,[1] we present CRED, a novel framework for cloud right-sizing with execution deadlines and data locality constraints. CRED jointly optimizes data placement and task scheduling in data centers with the aim of minimizing the number of nodes needed while meeting users' SLA requirements. We formulate CRED as an integer optimization problem and present a heuristic algorithm with provable performance guarantees to solve the problem. Competitive ratios of the proposed algorithm are quantified in closed form for arbitrary task parameters and cloud configurations. Simulation results using Google trace show that our proposed algorithm significantly outperforms existing heuristics such as first-fit by reducing up to 47% of required active servers, and achieves nearly-optimal performance in terms of cloud-right sizing.**

I. Introduction

With an increasing number of cloud-based solutions such as enterprise IT, social networks, financial services and scientific research, an explosive amount of data is being created, processed, and consumed online. Analytics over such data on the cloud are becoming more cost-sensitive, and cloud right-sizing has quickly emerged as a very promising technique for making clouds more cost-effective by dynamically adapting the number of active servers to match the target workload. Cloud right-sizing enables significant cost savings and power savings by auto-tuning the amount of active resources to handle the current workload [1, 2].

Existing work on cloud right-sizing mainly focuses on reducing energy consumption by dynamically allocating resources for given workloads [2, 3]. There is much less study on cloud right-sizing under both execution deadline and data locality constraints. Indeed, processing and analyzing data within certain deadlines have become more and more important, particularly due to the introduction of differentiated-QoS classes and time-dependent pricing mechanisms [4–6]. To improve data access efficiency and task throughput, data locality is often maximized by assigning tasks only to nodes that contain their input data [7–11]. However, pursuing these two objectives together could give rise to a conflict between "meeting deadlines" and "achieving locality" - for instance, a node with sufficient computing resources to complete a task on time may not possess the desired input data, and vice versa. The nature of cloud applications is becoming increasingly mission-critical and deadline-sensitive, e.g., traffic simulation and real-time web indexing. These applications are evolving in the direction of demanding hard completion times [5], and are likely to play crucial roles in the national infrastructure in the not too distant future. The cloud right-sizing problem is of interest to cloud providers in both private and public cloud settings.

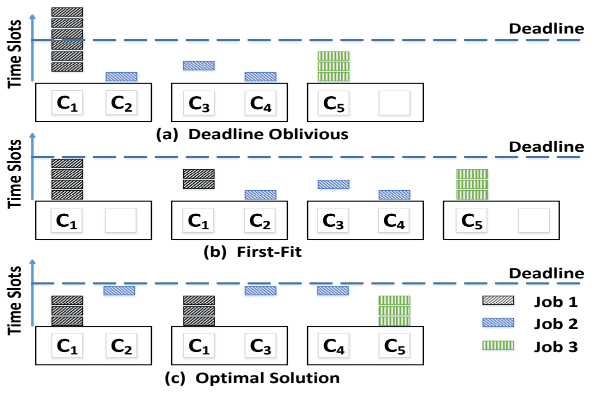


Fig. 1: An illustrative example of joint job scheduling and chunk placement for a cloud processing 3 jobs.

The need to solve cloud right-sizing under both execution deadline and data locality constraints can be illustrated by a very simple example, as shown in Figure 1. Consider a set of 3 jobs, j_1, j_2, and j_3, to be executed on a cloud for processing 5 data chunks $C_1, \ldots, C_5$. The jobs' resource requirements are heterogeneous – job j_1 accesses a single chunk C_1 and needs 6 time slots to process it, job j_3 accesses C_5 and needs 3 time slots, and job j_2 accesses three chunks, C_2, C_3, C_4, each requiring only one time slot to process. Our goal is to place the chunks in the nodes and schedule the jobs so as to minimize the number of active nodes needed to finish all three jobs before a deadline $d = 4$. Suppose each node has only one virtual machine (VM) (i.e., only one job can be processed by

[1]This work is supported by NSF grant CSR-1320226.

a node at each time slot), and is able to host 2 data chunks. The *deadline-oblivious* solution in Figure 1(a) considers only data locality constraint, i.e., assigns jobs to nodes that have the input data. It sequentially fills 3 nodes with the data chunks and assigns each job to nodes hosting its input chunks. While this solution minimizes the number of active nodes, it results in job j_1 failing to meet its deadline. The *first-fit* solution in Figure 1(b) finds the first node with both available time slots and storage space to accommodate a new job. A fraction of the job is assigned to the node until it either has no more time slots left before the deadline or cannot host any more chunks. This solution is able to meet all three jobs' deadlines, but increases the number of necessary nodes to 4 (i.e., over-sizing). Finally, the optimal solution in this example that uses only 3 nodes to meet the job deadlines is shown in Figure 1(c). The key insight is that we need to optimize the cloud over both chunk placement *and* job scheduling in order to achieve optimal right-sizing.

While adding new nodes can always improve cloud performance and increase its ability to meet deadlines [7], such a provisioning strategy is not cost-effective since servers and networks in datacenters contribute about 60% of the total expenses [12, 13]. It also may not always contribute to performance enhancements due to data locality [9, 10, 14]. In this paper, we introduce an optimization framework called *CRED* (cloud right-sizing with execution deadlines and data locality). To the best of our knowledge, this is the first work to consider cloud right-sizing under both deadline and locality constraints. Using a time-slotted system model, we present an algorithm for joint task scheduling and data placement. Then, we analyze the performance of the proposed algorithm and quantify its comparative ratio through closed-form bounds. In particular, we show that the proposed algorithm has a worst-case competitive ratio of 1.5 and is able to achieve the optimal solution under certain conditions. Extensive simulation results are presented, including a trace-driven simulation using 110 hours of Google Trace [15]. It is shown that our proposed algorithm outperforms heuristics such as first-fit by up to 44% node reduction. This saving indeed comes from the fact that our algorithm can significantly improve utilization of both computational resources and storage space by up to 25% and 35%, respectively.

II. System Model and Problem Formulation

Consider a set of J jobs that need to be processed by a cloud consisting of N physical machines (i.e., nodes) which are homogeneous [16, 17]. Note that the homogeneity assumption is only a technical condition required to quantify performance bounds in closed-form; all algorithms proposed in this paper work with heterogeneous nodes. Each job j has a deadline d_j and is required to access a data object that is split into a set $\mathcal{C}_j$ of equal-sized chunks. The chunks are stored in a distributed file system on the cloud. Each node is capable of hosting up to B data chunks and is equipped with S VMs. We consider a cloud framework similar to MapReduce, where jobs are partitioned into small tasks that are processed in parallel by different VMs. Thus, each node is able to simultaneously process S jobs. In this paper, we consider heterogeneous jobs with different processing times. In particular, the time for each job j to process a required data chunk, denoted by T_j, can vary from job to job. Note that T_j in our framework is known *a priori*. This follows from the model used in [14, 18, 19], which shows that 40% of the jobs are recurring and their characteristics, e.g., input data size, can be predicted with a small error of 6.5% on average, and the completion time's coefficient of variation is low. A job is completed once all required chunks are processed and will then exit the system.

Our goal is to minimize the total number of active nodes needed to complete the jobs satisfying a deadline constraint d_j for each job j, the data locality constraint and physical resource constraints on each node, i.e., B and S. We consider a time-slotted model where jobs are scheduled to execute in fixed-length time slots. Since each node is equipped with S VMs, it has S slots available at each time t. Our control knobs in the optimization include data chunk placement, job scheduling, and cloud sizing. We will first formulate this cloud right-sizing problem as an integer optimization.

Completing a job j before deadline d_j is equivalent to processing all the required chunks $c \in \mathcal{C}_j$ before the deadline.[2] When a chunk is accessed by multiple jobs, we need to guarantee that the chunk receives sufficient processing time (i.e., time slots) before each target deadline in order to support all the jobs. Therefore, we can formulate the job scheduling problem in terms of required processing time for each chunk. Denote $\mathcal{D}^{\uparrow}=\cup_j \{d_j\}$ to be the set of D distinct deadlines. Without loss of generality, we assume the deadlines in $\mathcal{D}^{\uparrow}$ are ordered, so that $d_i^{\uparrow} < d_s^{\uparrow}$ for all $i < s$ and $d_i^{\uparrow}, d_s^{\uparrow} \in \mathcal{D}^{\uparrow}$. We now formulate the job scheduling problem with respect to variables $f_{c,n,i}$, which is defined as the number of time slots on node n that are scheduled to process chunk c before the ith smallest deadline $d_i^{\uparrow}$. More precisely, the total time slots received by chunk c from all nodes (i.e., $\sum_n f_{c,n,i}$) before $d_i^{\uparrow}$ must satisfy:

$$\sum_{n=1}^{N} f_{c,n,i} \geq \sum_{j: d_j \leq d_i^{\uparrow}, c \in \mathcal{C}_j} T_j \triangleq F_{c,i}, \ \forall c, d_i^{\uparrow} \quad (1)$$

[2]In this paper, we ignore the time to set up a new machine including data transfer time from central storage, and assume that this time can be absorbed into job deadlines in online setting.

where a job j requires to access the chunk for T_j time-slots before time $d_i^{\uparrow}$ if we have $c \in \mathcal{C}_j$ and $d_j \leq d_i^{\uparrow}$. We define $F_{c,i}$ as the minimum number of required time slots for chunk c before deadline $d_i^{\uparrow}$. Equation (1) introduces a *deadline constraint* for the cloud right-sizing problem.

Let $p_{c,n}$ be a binary chunk placement variable that is 1 if a chunk c is hosted by node n and 0 otherwise. Similarly, we use $u_n = 1$ to denote that node n is active and $u_n = 0$ otherwise. Due to our *data locality constraint*, job i can be scheduled on node n only if the node is active and its required data chunks are available locally, i.e.,

$$f_{c,n,i} = 0 \text{ if } (p_{c,n} = 0 \text{ or } u_n = 0), \quad \forall c, n, d_i^{\uparrow}. \qquad (2)$$

Let $\mathcal{C} = \cup_j \mathcal{C}_j$ be the set of all data chunks. There are two types of physical resource constraints: (i) a *space constraint* that requires no more than B chunks to be placed on any active node, i.e.,

$$\sum_{c \in \mathcal{C}} p_{c,n} \leq B \cdot u_n, \ \forall n \qquad (3)$$

and (ii) a *computational-resource constraint* that limits the number of time slots available:

$$\sum_{c:p_{c,n}>0} f_{c,n,i} \leq d_i^{\uparrow} \cdot S \cdot u_n, \forall n, d_i^{\uparrow} \qquad (4)$$

where $\sum_{c:p_{c,n}>0} f_{c,n,i}$ is the total number of time slots assigned to different chunks before $d_i^{\uparrow}$. On the other hand, there are $d_i^{\uparrow}$ time slots available for each VM on node n that is equipped with S VMs.

Our proposed optimization problem aims to minimize the total number of active nodes to process all jobs, under the above constraints. It can be formulated as an integer optimization over the decision variables $\{f_{c,n,i}, p_{c,n}, u_n\}$:

$$\begin{aligned} \text{minimize} \quad & N = \sum_{n=1}^{N} u_n, \qquad (5) \\ \text{s.t.} \quad & (1), \ (2), \ (3), \text{ and } (4). \end{aligned}$$

III. Our Solution to CRED Problem

The key idea from our illustrative example in Fig. 1 is that solving the CRED problem requires a joint optimization of job scheduling and chunk placement that addresses both execution deadline and data locality constraints in a collaborative fashion. In this section, we propose a novel algorithm that harnesses workload-aware chunk placement to partition data chunks based on their workload and schedules jobs to efficiently utilize both space and computing resources on active nodes, thus minimizing the number of nodes required to process all jobs. To illustrate our key solution concept, we will first focus on a special case where all jobs require equal execution deadlines. Next, we extend it to solve CRED problem for the arbitrary number of deadlines. Performance of proposed algorithms is quantified through analytical upper and lower bounds.

We first introduce some notations. Consider the chunk set $\mathcal{C}_i$ for $d_i^{\uparrow}$ with size C_i. We sort all chunks in descending order based on the number of required time slots for $d_i^{\uparrow}$ and record the order in an array, i.e., $R_{d_i^{\uparrow}}$. The chunk recorded in the head of $R_{d_i^{\uparrow}}$ has the largest number of required time slots for $d_i^{\uparrow}$. In following discussion, each algorithm has multiple steps and each step needs multiple iterations. So, we denote $\mathcal{H}_{B,d_i^{\uparrow}}^{(r)}$ as the first set of B chunks, from the tail of the array, with the total number of required time slots larger than or equal to $Sd_i^{\uparrow}$, at the beginning of rth iteration. We denote $\sum_{c \in \mathcal{H}_b} F_{c,i}^{(r)}$ and $\sum_{c \in \mathcal{L}_b} F_{c,i}^{(r)}$ as the number of required time slots of the b chunks at the head of $R_{d_i^{\uparrow}}$ and at the tail of $R_{d_i^{\uparrow}}$, at the beginning of rth iteration, respectively. We denote $\mathcal{C}_{b,i}$ as a set of b chunks from $\mathcal{C}_i$.

A. Solving CRED with equal deadlines

Consider the case where all jobs require the same execution deadline, i.e., $d_j = d_1^{\uparrow}$ $\forall j$ and chunks need time slots $F_{c,1}$ $\forall c$. The algorithm, as shown in Algorithm 2 is comprised of 2 steps. As mentioned above, each step of the algorithm consists of multiple iterations and we use r to denote the rth iteration.

When $\sum_{c \in \mathcal{H}_B} F_{c,1}^{(r)} > Sd_1^{\uparrow}$, we place $\mathcal{H}_{B,d_1^{\uparrow}}^{(r)}$ into a node and call Algorithm 1 for time-slots' scheduling. The condition $\sum_{c \in \mathcal{H}_B} F_{c,1}^{(r)} > Sd_1^{\uparrow}$ guarantees that $\mathcal{H}_{B,d_i^{\uparrow}}^{(r)}$ exists. By choosing $\mathcal{H}_{B,d_1^{\uparrow}}^{(r)}$, we can schedule $Sd_1^{\uparrow}$ time slots in each node. Choosing $\mathcal{H}_{B,d_1^{\uparrow}}^{(r)}$ and calling Algorithm 1 guarantees that we can remove as many number of chunks as possible while scheduling $Sd_i^{\uparrow}$ time slots in each iteration. If the remaining number of required time slots of chunk c is 0, we can remove the chunk c from the chunk set $\mathcal{C}_1$ and reduce the size of the chunk set $\mathcal{C}_1$. When $\sum_{c \in \mathcal{H}_B} F_{c,1}^{(r)} \leq Sd_1^{\uparrow}$, we can place any B chunks, $\mathcal{C}_{B,1}$, into one node and remove all of them. The pseudocode is shown in Algorithm 2.

The basic idea of Algorithm 1 is that by scheduling time slots from the chunks with the smallest number of required time slots, we can remove more chunks. The inputs to Algorithm 1 are a set of chunks and the number of time slots needed for $d_1^{\uparrow}$. The number of time slots needed is the maximum number of time slots we want to schedule in the node. We denote the number of time slots needed in rth iteration as $NTS_{d_1^{\uparrow}}^{(r)}$. If the remaining number of required time slots of chunk c is less than or equal to $NTS_{d_1^{\uparrow}}^{(r)}$, we schedule the remaining number of required time slots of the chunk c in the node. We deduct the remaining number of required time slots of the chunk c from $NTS^{(r)}_{d^{\uparrow}_1}$. We mark the remaining number of required time slots of the chunk c as 0 and then remove the chunk c. If the remaining number of required time slots of the chunk c is larger than $NTS^{(r)}_{d^{\uparrow}_1}$, we schedule the $NTS^{(r)}_{d^{\uparrow}_1}$ from chunk c in the node. We deduct $NTS^{(r)}_{d^{\uparrow}_1}$ from the remaining number of required time slots of chunk c and mark $NTS^{(r)}_{d^{\uparrow}_1}$ as 0.

Algorithm 1: schedule($\mathcal{C}_1$, $NTS^{(r)}_{d^{\uparrow}_1}$)

```
sort $\mathcal{C}_1$ based on the remaining number of required time slots for $d^{\uparrow}_1$ in ascending order
**for** $c$=1:$C_1$ **do**
  **if** $F_{c,1}$-$NTS_{d^{\uparrow}_1}$>0 **then**
    $F_{c,1}$=$F_{c,1}$-$NTS_{d^{\uparrow}_i}$
    $NTS_{d^{\uparrow}_1}$=0
    break
  **else**
    $NTS_{d^{\uparrow}_1}$=$NTS_{d^{\uparrow}_1}$-$F_{c,1}$
    $F_{c,1}$=0
    remove the chunk $c$
  **end if**
**end for**
```

Algorithm 2: CRED-S

```
Input: $C_1$, $\sum_{i=1}^{C_1} f_i^{d^{\uparrow}_1}$, $B$, $d^{\uparrow}_1$
Output: # of nodes
$C^{(r)}$=$C_1$
**while** $C^{(r)}$>0 **do**
  sort chunks based on the number of required time slots
  **if** $\sum_{c\in\mathcal{H}_B} F^{(r)}_{c,1}$>$Sd^{\uparrow}_1$ **then**
    place $\mathcal{H}^{(r)}_{B,d^{\uparrow}_1}$ into one node
    schedule($\mathcal{H}^{(r)}_{B,d^{\uparrow}_1}$,$Sd^{\uparrow}_1$)
  **else**
    break
  **end if**
**end while**
**while** $C^{(r)}$>0 **do**
  place $\mathcal{C}_{B,1}$ into one node
  schedule($\mathcal{C}_{B,1}$,$Sd^{\uparrow}_1$)
**end while**
```

It is easy to see that Algorithm 2 will keep adding new nodes until all chunks get their required time slots $\sum_c F_{c,1}$ scheduled. Processing chunk c is only permitted on a node where chunk c is placed. This is to improve data access efficiency and task throughput. Thus, the algorithm is guaranteed to generate a feasible solution to the CRED problem. To analyze the performance, we derive an upper bound to quantify the maximum number of active nodes needed by a solution obtained from Algorithm 2. The upper bound is compared to a theoretical lower bound that establishes the minimum number of active nodes necessary for any feasible solution to CRED problem. We define K_i as the minimum number of nodes necessary to provide enough time slots for all jobs whose deadlines are equal to $d^{\uparrow}_i$. We define $k^{(r)}_i$ as the minimum number of nodes necessary to provide enough time slots for job remaining at the beginning of rth iteration, whose deadlines are equal to $d^{\uparrow}_i$. Therefore, K_i=$k^{(0)}_i$.

Next, we will analyze each step in Algorithm 2 to derive upper and lower bounds on the number of nodes needed, denoted by $\hat{N}$. The basic idea of deriving lower bound is only considering time slots or block constraint in each node. The basic idea of deriving the upper bound is fixing the number of removable chunks in each iteration of each step.

Theorem 1. *When* K_1>$\lfloor\frac{2C_1}{B}\rfloor$*, the bounds are given by* K_1≤$\hat{N}$≤K_1+1. *When* $\lfloor\frac{2C_1}{B}\rfloor$≥$K_1$≥$\lfloor\frac{C_1}{B-1}\rfloor$*, the bounds are given by* max($\lceil\frac{C_1}{B}\rceil$,$K_1$)≤$\hat{N}$≤$\frac{K_1}{2}$+$\frac{C_1}{B}$+1. *When* K_1<$\lfloor\frac{C_1}{B-1}\rfloor$*, the bounds are given by* $\lceil\frac{C_1}{B}\rceil$≤$\hat{N}$≤$\frac{K_1}{B}$+$\frac{C_1}{B}$+1.

Proof: In each node, we can place at most B chunks or schedule $Sd^{\uparrow}_1$ time slots. To place C_1 chunks, we need at least $\lceil\frac{C_1}{B}\rceil$ nodes. To schedule $\sum_j T_j$ required time slots, we need at least K_1 nodes. To place C_1 chunks and schedule $\sum jT_j$ required time slots, we need at least max($\lceil\frac{C}{B}\rceil$,K_1) nodes. As the result, the lower bound is max($\lceil\frac{C}{B}\rceil$,K_1).

We now derive the upper bound for Algorithm 2. Instead of removing as many chunks as possible, we only remove the assigned number of chunks in each iteration of each step. In following, we call it as the *simplified version* of Algorithm 2. If assigned number of chunks has been removed, even though the remaining number of required time slots of other chunks equal 0, we still consider those chunks in the following cases. For the simplified version, in step 1, when $\lfloor\frac{C^{(r)}_1}{B-1}\rfloor$≤$k^{(r)}_1$≤$\lfloor\frac{2C^{(r)}_1}{B}\rfloor$, we remove $\frac{B}{2}$ chunks in each iteration. When $k^{(r)}_1$≤$\lfloor\frac{C^{(r)}_1}{B-1}\rfloor$ and $\sum_{c\in\mathcal{H}_B} F^{(r)}_{c,1}$>$Sd^{\uparrow}_1$, we remove $B-1$ chunks in each iteration. In step 2, the simplified version also removes any B chunks in each iteration.

In the following, we first derive upper bounds for the simplified version. Secondly, we show the number of nodes needed by simplified version is no less than for Algorithm 2. Therefore, the upper bounds of the simplified version are also the upper bounds of Algorithm 2.

We introduce *Lemma 1* to verify that for simplified version, once $\lfloor\frac{C^{(r)}_1}{B-1}\rfloor$≤$k^{(r)}_1$≤$\lfloor\frac{2C^{(r)}_1}{B}\rfloor$, in each iteration, we can remove at least $\frac{B}{2}$ chunks. Also, we introduce

Lemma 2 to verify that once $k_1^{(r)}\le\lfloor\frac{C_1^{(r)}}{B-1}\rfloor$, in each iteration, we can remove at least $B-1$ chunks.

Lemma 1. *For simplified version, when $k_1^{(r)}\le\lfloor\frac{2C^{(r)}}{B}\rfloor$, where $C^{(r)}$=C_1-$r\cdot\frac{B}{2}$, we have the $\sum_{c\in\mathcal{L}_{B/2}}F_{c,1}^{(r)}\le Sd_1^{\uparrow}$. Here C_1 means the size of remaining chunk set when $r=0$.*

We denote the B-1 chunks with the smallest number of required time slots among $\mathcal{H}^{(r)}_{B,d_i^{\uparrow}}$ as $\mathcal{LH}^{(r)}_{B-1,d_i^{\uparrow}}$.

Lemma 2. *For simplified version, when $k_1^{(r)}\le\lfloor\frac{C^{(r)}}{B-1}\rfloor$ and $\sum_{c\in\mathcal{H}_B}F_{c,i}^{(r)}>Sd_1^{\uparrow}$, where $C^{(r)}$=C_1-$r\cdot$(B-1), we have the total required time slots of $\mathcal{LH}_{B-1,d_1^{\uparrow}}$ is less than $Sd_1^{\uparrow}$. Here C_1 means the size of remaining chunk set when r=0.*

The basic idea of following discussion is to consider the value of K_1 in three cases. By introducing the three cases, we can get tighter upper bounds.

Case 1: $K_1>\lfloor\frac{2C_1}{B}\rfloor$. Assume we need r_1^1 iterations of step 1 to make $k_1^{(r_1^1)}\le\lfloor\frac{2C_1^{(r_1^1)}}{B}\rfloor$. Assume we need another r_2^1 iterations of step 1 to make $k_1^{(r_1^1+r_2^1)}\le\lfloor\frac{C_1^{(r_1^1+r_2^1)}}{B-1}\rfloor$. Assume we need another r_3^1 iterations of step 1 to make $\sum_{c\in\mathcal{H}_B}F_{c,1}^{(r_1^1+r_2^1+r_3^1)}\le Sd_1^{\uparrow}$. So, the total number of nodes for step 1 and step 2 is

$$r_1^1+r_2^1+r_3^1+\left\lceil\frac{C_1-r_2^1B/2-r_3^1(B-1)}{B}\right\rceil. \tag{6}$$

We know that $B{\ge}2$ and r_1^1+r_2^1+$r_3^1\le K_1$. Since we do not remove any chunks within r_1^1 iterations, so r_1^1 equals $K_1-\lfloor\frac{2C_1}{B}\rfloor$. Thus, (6) is less than or equal to K_1+1.

Case 2: $\lfloor\frac{2C_1}{B}\rfloor\ge K_1\ge\lfloor\frac{C_1}{B-1}\rfloor$. Assume we need r_1^2 iterations of step 1 to make $k_1^{r_1^2}\le\lfloor\frac{C_1^{r_1^2}}{B-1}\rfloor$. Assume we need another r_2^2 iterations of step 1 to make $\sum_{c\in\mathcal{H}_B}F_{c,1}^{(r_1^2+r_2^2)}\le Sd_1^{\uparrow}$. So, the total number of nodes for step 1 and step 2 is

$$r_1^2+r_2^2+\left\lceil\frac{C_1-r_1^2B/2-r_2^2(B-1)}{B}\right\rceil. \tag{7}$$

We know that $B{\ge}2$ and r_1^2+$r_2^2\le K_1$. So, (7) is less than or equal to $\frac{K_1}{2}$+$\frac{C_1}{B}$+1.

Case 3: $K_1<\lfloor\frac{C_1}{B-1}\rfloor$. Assume we need another r_1^3 iterations of step 1 to make $\sum_{c\in\mathcal{H}_B}F_{c,1}^{r_1^3}\le Sd_1^{\uparrow}$. So, the total number of nodes for step 1 and step 2 is

$$r_1^3+\left\lceil\frac{C_1-r_1^3(B-1)}{B}\right\rceil. \tag{8}$$

We know that $r_1^3\le K_1$. So, (8) is less than or equal to $\frac{K_1}{B}$+$\frac{C_1}{B}$+1.

Lemma 3. *We show that in each iteration of step 1, for each chunk, the number of time slots scheduled by Algorithm* 2 *and simplified version are exactly the same.*

After finishing step 1, the size of remaining chunk set of simplified version is no less than Algorithm 2's. For step 2, we can remove B chunks in each iteration. Thus, the number of nodes needed of simplified version is larger than or equal to the number of nodes needed of Algorithm 2. ■

Remark. *As $K_1{\to}\infty$, the upper bound K_1+1 is achievable and competitive ratio equals* 1. *As $K_1{\to}0$, the upper bound $\frac{K_1}{2}$+$\frac{C_1}{B}$+1 and $\frac{K_1}{B}$+$\frac{C_1}{B}$+1 are achievable and competitive ratio equals to* 1. *For general case, the competitive ratio varies in the interval* $[1,1.5]$

B. Solving CRED with multiple deadlines

We propose a heuristic algorithm to solve CRED with multiple, arbitrary deadlines. Our idea is to iteratively apply Algorithm 2 to incrementally find chunk placement and time-slot scheduling to meet each deadline one-by-one. More precisely, after finding a solution for placing chunks $c\in\mathcal{C}_1,\ldots,\mathcal{C}_i$ to meet deadlines $d_1^{\uparrow},\ldots,d_i^{\uparrow}$, we reuse the already placed chunks on existing nodes (if there are remaining computation resources available) and optimize for the next deadline $d_{i+1}^{\uparrow}$ and minimize the number of new nodes we need to add in order to support $F_{c,i+1}$ for all chunks $c\in\mathcal{C}_{i+1}$. This process continues until all the deadlines are considered. The algorithm is summarized in Algorithm 3. Its performance is characterized in Theorem 2. We assume there are D distinct $d_i^{\uparrow}$. We consider chunk placement and time slots scheduling of distinct deadlines one-by-one, from $d_1^{\uparrow}$ to $d_D^{\uparrow}$. For deadline $d_i^{\uparrow}$, the Algorithm 3 first calls Algorithm 2 for $d_i^{\uparrow}$ and then schedules time slots for deadlines from $d_i^{\uparrow}$ to $d_D^{\uparrow}$.

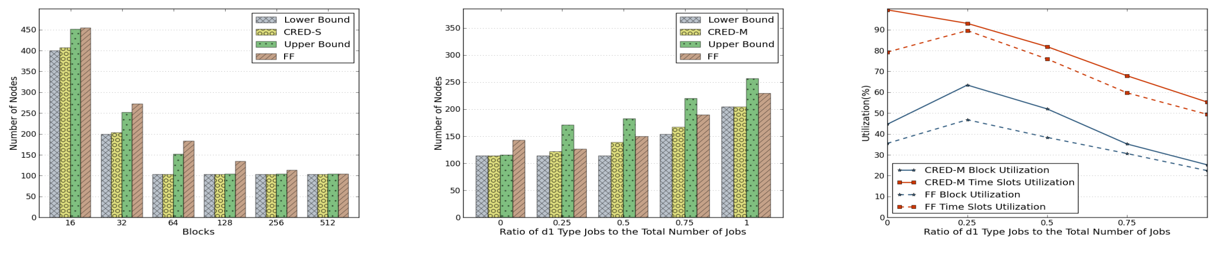
Algorithm 3: CRED-M

```
for i=1:D do
    # of nodes= Algorithm 2 ($C_i$, $\sum_{j:d_j=d_i^{\uparrow}}T_j$, $B$, $d_i^{\uparrow}$)
    for $n$=1:# of nodes do
        for $i_1$=i:D do
            schedule(the set of chunks in node $n$, $Sd_{i_1}^{\uparrow}$-#scheduled time slots in node $n$)
        end for
    end for
end for
```

Theorem 2. *The number of nodes needed is $\hat{N}$.* $\max\left(\max_i(K_i),\frac{\sum_{i=1}^{D}\sum_{j:d_j=d_i^{\uparrow}}T_j}{Sd_D^{\uparrow}},\frac{C}{B}\right)\le\hat{N}$ $\le\sum_{i=1}^{D}\max\left(K_i,\frac{K_i}{2}+\frac{C_i}{B}\right)$+$D$.

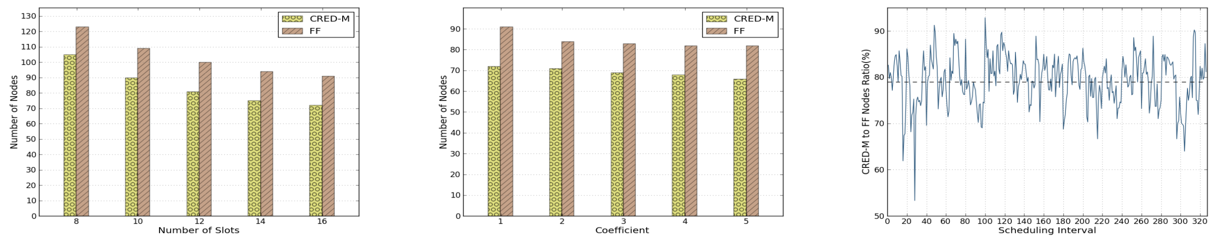
Proof: We first prove the lower bound and then prove the upper bound.

Based on Theorem 1, $\max(\max_i(K_i),\frac{C}{B})$ is obvious. Assume all jobs have the same deadlines and their deadlines are equal to $d_D^{\uparrow}$. Under this assumption, the



(a) (b) (c)

Fig. 2: Experiments with proposed algorithms CRED-S and CRED-M: (a) Effect of number of blocks to the number of nodes. (b) Effect of the ratio between d_1 and d_2 type of jobs to the number of nodes needed. (c) Effect of the ratio between d_1 and d_2 type of jobs to the time-slots and blocks utilization.



(a) (b) (c)

Fig. 3: Trace-driven comparison between CRED-M and First-Fit algorithm: (a) Effect of number of slots to the number of nodes. (b) Effect of the ratio between D_s and D_l type of jobs to the number of nodes needed. (c) Number of nodes needed in each scheduling interval.

minimal number of nodes is $\frac{\sum_{i=1}^{D}\sum_{j:d_j=d_i^{\uparrow}} T_j}{S d_D^{\uparrow}}$. Thus, the lower bound is $\max\left(\max_i(K_i), \frac{\sum_{i=1}^{D}\sum_{j:d_j=d_i^{\uparrow}} T_j}{S d_D^{\uparrow}}, \frac{C}{B}\right)$.

Next, we prove the upper bound. From Theorem 1, for single deadline $d_i^{\uparrow}$, the upper bound is $\max\left(K_i, \frac{K_i}{2}+\frac{C_i}{B}\right)$+1.

For D distinct deadlines, we consider deadlines iteratively one by one. Therefore, the upper bound is $\sum_{i=1}^{D}\max\left(K_i, \frac{K_i}{2}+\frac{C_i}{B}\right)+D$ ∎

Remark. *When D=1, the lower bound and upper bound are* $\max(K_1, \frac{C}{B})$ *and* $\max(K_1, \frac{K_1}{2}+\frac{C}{B})$+1, *respectively.*

IV. Evaluation

In this section, we perform various simulation experiments for equal and two deadlines to evaluate our algorithmic solution to the CRED problem. We focus on two aspects: first, we show that CRED is within the closed-form bounds we proved in Section III. Second, we compare CRED and First-Fit algorithm in terms of number of nodes and resource utilization.

To perform simulation experiments, we set the following global simulation parameters for all experiments, unless otherwise stated. Each node is set to have S = 4 slots (VMs) and B = 128 blocks. We create two classes of jobs: elephants and mice jobs. Elephants jobs are computation-intensive jobs; each job requires 300 time slots to process a chunk. Each mice job requires only one unit of time to process a chunk. Elephants and mice jobs form 5% and 95% of the total jobs (200 jobs in equal deadlines and 400 jobs in two deadlines), respectively. We make the number of files equal to 100, and each file contains 64 chunks. All jobs are assigned to files randomly. For equal deadline experiments, we fix the deadline to be equal to 500 for all experiments, and for two-deadline experiments, we fix d_1 and d_2 to be equal to 500 and 900, respectively. For all experiments, the figures show the average of 20 runs.

Figure 2(a) shows the effect of changing the number of blocks in each node to the number of nodes needed. It can be clearly seen that as we increase the number of blocks in each node, the problem moves from blocks constraint to time slots constraint. In addition, the figure shows, as expected, our simulation results outperform the first-fit algorithm. CRED is able to match the lower bound when $B \geq 64$ blocks and exceeds the lower bound by less than 2% on average when $B \leq 32$. Notice that when there is mixed of time-slots and blocks constraints, our algorithm outperforms FF by up to 44% in number of nodes.

Figure 2(b) shows that CRED-M is within the closed-form bounds we proved in Section III. In this experiment, each job is associated with either deadline d_1 or d_2. The *x-axis* shows the ratio of d_1 type of jobs to the total number of jobs. All files are accessed by both d_1 and d_2 type of jobs when $0 < ratio < 1$. As we increase the ratio, consequently, the number of nodes increases since $d_1 < d_2$.

Figure 2(c) compares CRED-M and FF in terms of time slots and blocks utilization defined as the ratio of number of processing time slots and blocks actually utilized to the total number available. In this figure, we use the same parameters as Figure 2(b). The figure shows for any ratio, CRED-M achieved higher utilization in both time-slots (up to 25%) and blocks (up to 35%). This is because, unlike FF, CRED always tries to fully utilize the time-slots and blocks in each node.

V. Trace-driven evaluation

In this section, we compare the performance of CRED-M with First-Fit algorithm through a trace-driven simulation using data from a publicly available Google trace [15]. The results show that CRED-M outperforms FF by up to 20% node (and cost) saving on average, and reduces the number of nodes required at peak utilization by 47%.

We select a subset data from Google trace with a length of 110 hours. Each job has a number of parameters, including job submission time, job ID, scheduling class, number of tasks and execution time of tasks. The scheduling class represents how latency-sensitive the job is. There exists 4 scheduling classes and jobs with level 3 are the most latency-sensitive. In this simulation, we consider jobs belonging to scheduling classes 1 and 2. For each class, we assign a deadline (D_l and D_s). We denote A_i as the average execution time of all jobs with scheduling class i in a scheduling interval. D_s equals A_2 and D_l equals to $\max(A_1,\ D_s)$. We set the scheduling interval to be 20 minutes. At the beginning of each scheduling interval, we schedule jobs arriving in the previous scheduling interval. At the end of each hour, we identify all nodes that have finished processing all the workload and release them to the system, in a way similar to Amazon's EC2 cloud (on-demand instances).

Figure 3 (a) compares the average number of active nodes (i.e., VM hours or cost) for the proposed CRED-M algorithm and FF, to meet the same job deadlines. Here, we set the number of blocks to be 16, while changing the number of slots from 8 to 16. The figure shows that CRED-M outperforms FF by up to 20% in average number of nodes that are required to meet all deadlines. It can be observed that as the number of slots increases, the number of nodes to meet the same deadlines decrease (since more tasks can be packed into a node), while our proposed algorithm achieves a consistent node reduction of 20%.

Figure 3 (b) compares the average number of active nodes for CRED-M algorithm and FF to meet various deadlines. Here, we set both the number of blocks and the number of slots to be 16. D_l equals to $\max(A_1,\ \alpha{\cdot}D_s)$ and we change α from 1 to 5. The figure shows that CRED-M algorithm outperforms FF by 17% in average number of nodes.

Figure 3 (c) shows the number of active nodes required by CRED-M algorithm for each scheduling interval during the entire simulation. Again, we set the number of blocks and the number of slots to be 16. The numbers are normalized by the number of nodes used by the FF algorithm, and thus any number smaller than 100% indicates node (or cost) saving. It is shown that CRED-M reduces the number of nodes required at peak utilization by 47%. The dashed line shows that CRED-M outperforms FF by 21% in average number of nodes.

VI. Conclusions

In this paper, we introduce an optimization framework, namely CRED, for cloud right-sizing under deadline and locality constraints. Algorithms are proposed to solve the CRED optimization, which minimizes the number of nodes needed by jointly optimizing task scheduling and data placement while the jobs' deadlines and data locality are met. We analyze the competitive ratio of the proposed algorithms in closed-form. The algorithms significantly outperform a first-fit heuristic in terms of cloud-size (i.e., number of active nodes needed) and node utilization. We compare the performance of CRED-M with First-Fit algorithm through a trace-driven simulation. The results show that CRED-M algorithm outperforms FF by up to 20% node (and cost) saving on average.

In our future work, we plan to consider heterogeneous nodes and multi-phase cloud systems, e.g., MapReduce [7], involving communication between tasks. We also intend to obtain a resilient solution, which allows successful recovery at run time from any node failure and is guaranteed to meet both deadline and locality constraints.

VII. Appendix

Lemma 1. *When* $k_1^{(r)}{\le}\lfloor\frac{2C^{(r)}}{B}\rfloor$, *where* $C^{(r)}{=}C_1\text{-}r\frac{B}{2}$, *we have* $\sum_{c\in\mathcal{L}_{B/2}}F_{c,1}^{(r)}{\le}Sd_1^{\uparrow}$

Proof: We use contradiction method to prove this lemma. Assume when $k_1^{(r)}{\le}\lfloor\frac{2C^{(r)}}{B}\rfloor$, where $C^{(r)}{=}C\text{-}r\frac{B}{2}$, we have $\sum_{c\in\mathcal{L}_{B/2}}F_{c,1}^{(r)}{>}Sd_1^{\uparrow}$. Suppose we partition $C^{(r)}$ chunks into $\lfloor\frac{2C^{(r)}}{B}\rfloor$ sets whose size is $\frac{B}{2}$. Since $\sum_{c\in\mathcal{L}_{B/2}}F_{c,1}^{(r)}{>}Sd_1^{\uparrow}$, so the number of required time slots of any set should be larger than $Sd_1^{\uparrow}$. So, the total number of required time slots should be larger than $\lfloor\frac{2C^{(r)}}{B}\rfloor{\cdot}Sd_1^{\uparrow}$. This is contrary to the condition that $k_1^{(r)}{\le}\lfloor\frac{2C^{(r)}}{B}\rfloor$ and the proof is complete. ■

We denote the B-1 chunks with the smallest number of required time slots among $\mathcal{H}_{B,d_i^{\uparrow}}^{(r)}$ as $\mathcal{LH}_{B-1,d_i^{\uparrow}}^{(r)}$.

Lemma 2. *When* $k_1^{(r)}{\le}\lfloor\frac{C^{(r)}}{B-1}\rfloor$ *and* $\sum_{c\in\mathcal{H}_B}F_{c,i}^{(r)}{>}Sd_1^{\uparrow}$, *where* $C^{(r)}{=}C_1$*-r(*B*-1), we have the total number of required time slots of* $\mathcal{LH}_{B-1,d_1^{\uparrow}}$ *is less than* $Sd_1^{\uparrow}$.

Proof: We use contradiction method to prove this lemma. Because $\sum_{c\in\mathcal{H}_B} F_{c,1}^{(r)} \geq Sd_1^{\uparrow}$, we can find B chunks whose total required time slots is larger than $Sd_1^{\uparrow}$. Based on similar proving method of *Lemma 1*, we know that $\sum_{c\in\mathcal{L}_{B-1}} F_{c,1}^{(r)} \leq Sd_1^{\uparrow}$. Assume the total required time slots of $\mathcal{LH}_{B-1,d_i^{\uparrow}}^{(r)}$ is larger than or equal to $Sd_1^{\uparrow}$, then we choose a chunk c with the smallest number of required time slots and combine the $\mathcal{LH}_{B-1,d_i^{\uparrow}}^{(r)}$ and the chunk c to be a new set of chunks $\mathcal{C}'$. At this time, the total number of required time slots of $\mathcal{C}'$ should be less than the total required time slots of $\mathcal{H}_{B,d_i^{\uparrow}}^{(r)}$ and the number of required time slots is larger than $Sd_1^{\uparrow}$. So, it is contrary to our condition that $\mathcal{H}_{B,d_1^{\uparrow}}^{(r)}$ is a set of B chunks with the smallest number of required time slots, while the total number of required time slots is larger than $Sd_i^{\uparrow}$. ■

Lemma 3. *We show that in each iteration of step 1, for each chunk, the number of time slots scheduled by Algorithm* 2 *and simplified version are exactly the same.*

Proof: In each iteration of step 1 of both Algorithm 2 and simplified version, we first sort chunks based on the remaining number of required time slots in descending order and choose $\mathcal{H}_{B,d_i^{\uparrow}}^{(r)}$. We use $\mathcal{C}_{min}^{(r)}$ to denote a subset of $\mathcal{H}_{B,d_i^{\uparrow}}^{(r)}$. $\mathcal{C}_{min}^{(r)}$ is chosen from the tail of the sorted $\mathcal{H}_{B,d_i^{\uparrow}}^{(r)}$. The number of required time slots of $\mathcal{C}_{min}^{(r)}$ is larger or equal to $Sd_1^{\uparrow}$. Also, the remaining number of required time slots of any chunk $c\in\mathcal{C}_{min}^{(r)}$ is non-zero. So, $\mathcal{C}_{min}^{(r)}$ is the chunk set used by Algorithm 1 for time slots scheduling. Because the total chunks set $\mathcal{C}$, before the first iteration, for both versions are the same. Thus, to prove *Lemma 3*, we only need to prove, in each iteration, $\mathcal{C}_{min}^{(r)}$ for both Algorithm 2 and simplified version are exactly the same.

When r=1, before calling Algorithm 1, the chunk set $\mathcal{H}_{B,d_i^{\uparrow}}^{(r)}$ for both Algorithm 2 and simplified version are the same. Thus, $\mathcal{C}_{min}^{(1)}$ should be the same.

When r=r', we assume $\mathcal{C}_{min}^{(r')}$ for both Algorithm 2 and simplified version are the same.

When r=r'+1, we need to prove the chunk sets $\mathcal{C}^{(r'+1)}$ are the same.

For any iteration r, after calling Algorithm 1, there is at most one chunk c, whose remaining number of required time slots is non-zero and has been scheduled by Algorithm 1. It means, for rth iteration, after calling Algorithm 1, the remaining chunk sets $\mathcal{C}^{(r)}$ at most have one chunk, whose the number of remaining required time slots is larger than 0, scheduled by Algorithm 1. Also, the chunk c is the chunk with the smallest number of required time slots for next iteration r+1. It means, for r+1th iteration, the chunk $c\in\mathcal{C}_{min}^{(r+1)}$.

$\mathcal{C}_{min}^{(r')}$ for both Algorithm 2 and simplified version are the same. Thus, in r'th iteration, $\mathcal{C}^{(r')}$ for both Algorithm 2 and simplified version are the same. Thus, the chunk sets $\mathcal{C}^{(r'+1)}$ are the same. Thus, in each iteration of step 1, for each chunk, the number of time slots scheduled by Algorithm 2 and simplified version are exactly the same. ■

References

[1] D. Jiang, B. C. Ooi, L. Shi, and S. Wu, "The performance of mapreduce: An in-depth study," *Proc. VLDB Endow.*, vol. 3, pp. 472–483, Sep. 2010.

[2] M. Lin, A. Wierman, L. Andrew, and E. Thereska, "Dynamic right-sizing for power-proportional data centers," in *INFOCOM*, 2011.

[3] Q. Zhang, M. Zhani, R. Boutaba, and J. Hellerstein, "Dynamic heterogeneity-aware resource provisioning in the cloud," *Cloud Computing, IEEE Transactions on*, vol. 2, pp. 14–28, Jan 2014.

[4] A. Verma, L. Cherkasova, V. Kumar, and R. Campbell, "Deadline-based workload management for mapreduce environments: Pieces of the performance puzzle," in *NOMS*, 2012.

[5] L. T. X. Phan, Z. Zhang, Q. Zheng, B. T. Loo, and I. Lee, "An empirical analysis of scheduling techniques for real-time cloud-based data processing," in *SOCA*, 2011.

[6] S. Shi, C. Wu, and Z. Li, "Cost-minimizing online vm purchasing for application service providers with arbitrary demands," in *CLOUD*, 2015.

[7] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, Jan. 2008.

[8] T. White, *Hadoop: The Definitive Guide*, 1st ed. O'Reilly Media, Inc., 2009.

[9] Z. Guo, G. Fox, and M. Zhou, "Investigation of data locality in mapreduce," in *CCGrid*, 2012.

[10] B. Palanisamy, A. Singh, L. Liu, and B. Jain, "Purlieus: Locality-aware resource allocation for mapreduce in a cloud," in *SC*, 2011.

[11] S. Tang, B. S. Lee, and B. He, "Dynamicmr: A dynamic slot allocation optimization framework for mapreduce clusters," *IEEE Transactions on Cloud Computing*, vol. 2, no. 3, pp. 333–347, July 2014.

[12] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The cost of a cloud: Research problems in data center networks," *SIGCOMM Comput. Commun. Rev.*, 2008.

[13] J. Hamilton, "Cost of power in large-scale data centers." [Online]. Available: http://goo.gl/FLa4CX.

[14] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar, "Network-aware scheduling for data-parallel jobs: Plan when you can," in *SIGCOMM*, 2015.

[15] "Google trace," https://github.com/google/cluster-data, 2011.

[16] B. Moseley, A. Dasgupta, R. Kumar, and T. Sarlós, "On scheduling in map-reduce and flow-shops," in *SPAA*, 2011.

[17] H. Xu and W. C. Lau, "Speculative execution for a single job in a mapreduce-like system," in *CLOUD*, 2014.

[18] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou, "Re-optimizing data-parallel computing," in *NSDI*, 2012.

[19] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca, "Jockey: Guaranteed job latency in data parallel clusters," in *EuroSys*, 2012.