IEEE TRANSACTIONS ON
PARALLEL AND
DISTRIBUTED SYSTEMS



# CRED: Cloud Right-sizing with Execution Deadlines and Data Locality

Maotong Xu, Sultan Alamro, Tian Lan, and Suresh Subramaniam
Department of Electrical and Computer Engineering
The George Washington University
{htfy8927, alamro, 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 businesses 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, 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. We also extend our work to obtain a resilient solution, which allows successful recovery at run time from any single node failure and is guaranteed to meet both deadline and locality constraints. Simulation results using Google trace show that our proposed algorithm significantly outperforms existing heuristics such as first-fit by reducing the number of required active servers by up to 47%, and achieves near-optimal performance. We also show that our algorithm can significantly improve utilization of both computational resources and storage space by up to 28% and 15%, respectively.

**Index Terms**—Cloud right-sizing, Scheduling, Data locality, Failure recovery

## 1 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 in 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/nodes to handle the current workload [2], [3].

Existing work on cloud right-sizing mainly focuses on reducing energy consumption by dynamically allocating resources for given workloads [3], [4]. 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 [5], [6], [7]. To improve data access efficiency and task throughput, data locality is often maximized by assigning tasks only to nodes that contain their input data [8], [9], [10], [11], [12]. 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 [6], 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.

The need to solve cloud right-sizing under both execution deadline and data locality constraints can be illustrated by a simple example, as shown in Figure 1. Consider a set of 3 jobs, $j_1$, $j_2$, and $j_3$, to be executed

*This paper is an extended version of [1].*

Digital Object Identifier 10.1109/TPDS.2017.2726071

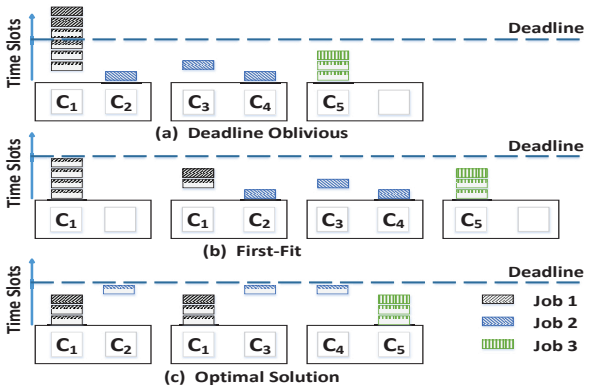



Fig. 1: An illustrative example of joint job scheduling and chunk placement for a cloud processing 3 jobs.

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 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 *deadline-aware 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 it 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 [8], such a provisioning strategy is not cost-effective since servers and networks in datacenters contribute about 60% of the total expenses [13], [14]. It also may not always contribute to performance enhancements due to data locality [10], [11], [15]. 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 competitive 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 [16]. It is shown that our proposed algorithm outperforms heuristics such as first-fit by up to 47% 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 28% and 15%, respectively.

The paper makes the following novel contributions:

- We formulate the CRED problem, which jointly optimizes job scheduling and data placement in cloud-based data processing in order to minimize the number of active nodes under task deadline and data locality constraints.
- We propose novel algorithmic solutions to the CRED problem and quantify their competitive ratio in closed form. Tight upper bounds for the proposed algorithms are derived and compared to lower bounds.
- The CRED problem and proposed algorithms are extended to obtain a resilient solution, which allows successful recovery at run time from any single node failure and is guaranteed to meet both deadline and locality constraints.
- Our work is validated through extensive simulations and compared with a first-fit heuristic. Significant cloud size reduction is verified.

The rest of this paper is organized as follows. In Section 3, we introduce our system model and problem formulation. Sections 4 and 5 present our solution to CRED in the cases of no failure and single node failure, respectively. We present the experimental evaluation of CRED in Section 6, and conclude the paper in Section 7.

## 2 Related Work

There has been intensive research on improving the performance of cloud-based frameworks. Data locality has a significant impact on system performance and is considered to be an important factor for scheduling

[10]. Achieving efficient data locality in a cloud cluster is critical for performance and reduces the cost of transferring data over the network. [17] designed and implemented a resource-aware scheduler to alleviate job starvation and avoid unfavorable data locality. ActCap [18], and MRA++ [19] proposed solutions for data placement to improve MapReduce performance on a heterogeneous cluster. [20] presented RCFile (Record Columnar File) as a fast and space-efficient data placement structure. [21] proposed scheduling techniques to enhance jobs' data locality. Similar to our work, Purlieus [11] proposed to couple data and VM placement in MapReduce cluster. However, unlike our work, they did not consider minimizing the number of nodes for a given cluster while meeting the execution deadlines for submitted jobs.

Deadline-aware schedulers for cloud-based frameworks have also been well studied. For example, [22] developed a deadline constraint scheduler and derived closed-form expressions for the minimum Map/Reduce tasks required to meet deadlines in the MapReduce framework. SAMES [23] proposed a scheduling algorithm for MapReduce jobs with deadlines. Its goal is to maximize the number of jobs that finish before their deadlines. AIRA [24] is proposed to allocate the appropriate amount of resources for a job to meet its required SLA, and [25] aims to improve resource utilization while observing deadlines. A framework designed to schedule Hadoop workflows with deadlines is proposed in [26] . It allows client nodes to locally generate scheduling plans of workflows, which a master node can use to prioritize jobs among multiple workflows.

While the above challenges have been extensively studied in the literature, to the best of our knowledge, no work considers optimizing data locality and task scheduling jointly to reduce cost of operation while meeting a given time constraint. We argue that the fundamental cause of violating users' requirement is neglecting optimizing data placement and scheduling jointly.

## 3 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) that are homogeneous [27], [28]. 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 assumed to be known *a priori*. This follows from the model used in [15], [29], [30], 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. 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 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.[1] 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 the variable $f_{c,n,i}$, which is defined as the number of time slots on node $n$ that are scheduled to process chunk $c$ before the $i$th smallest deadline $d_i^{\uparrow}$. More precisely, the total number of time slots received by chunk $c$ from all nodes (i.e., $\sum_n f_{c,n,i}$) before $d_i^{\uparrow}$ must satisfy:

$$\sum_{n=1}^{\hat{N}} f_{c,n,i} \geq \sum_{j:d_j \leq d_i^{\uparrow}} T_j \triangleq F_{c,i}, \ \forall c, d_i^{\uparrow} \tag{1}$$

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

where a job $j$ needs to access a chunk $c \in C_j$ for $T_j$ time-slots before time $d_i^{\uparrow}$ and $d_j \leq d_i^{\uparrow}$, and $\hat{N}$ is the number of active nodes needed. 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.,

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

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 *storage constraint* that requires no more than $B$ chunks to be placed on any active node, i.e.,

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

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

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

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 $\{f_{c,n,i}, p_{c,n}, u_n\}$:

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

## 4 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 the CRED problem for arbitrary number of deadlines. The performance of the 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, $R_{d_i^{\uparrow}}$. The chunk recorded in the head of $R_{d_i^{\uparrow}}$ has the largest number of required time slots for $d_i^{\uparrow}$. In the 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 $B$ contiguous chunks, from the tail of the array, with the total number of required time slots larger than or equal to $S \cdot d_i^{\uparrow}$, at the beginning of the $r$th 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 for the $b$ chunks at the head of $R_{d_i^{\uparrow}}$ and at the tail of $R_{d_i^{\uparrow}}$, respectively, at the beginning of the $r$th iteration. We denote $\mathcal{C}_{b,i}$ as a set of $b$ chunks from $\mathcal{C}_i$. We define $K_i$ as the minimum number of nodes necessary to provide enough time slots for all jobs whose deadlines are equal to $d_i^{\uparrow}$, which equals $\frac{\sum_c F_{c,i}}{S \cdot d_i^{\uparrow}}$. Also, we define $k_i^{(r)}$ as the minimum number of nodes necessary to provide enough time slots for jobs remaining at the beginning of the $r$th iteration, whose deadlines are equal to $d_i^{\uparrow}$. Therefore, $K_i = k_i^{(0)}$.

### 4.1 Solving CRED with equal deadlines

Consider the case where all jobs require the same execution deadline, i.e., $d_j = d_1^{\uparrow} = d \ \forall j$. In this case, we drop the subscript for simplicity and denote the variables as $\mathcal{C}$, $\mathcal{C}_b$, $K$, and $k^{(r)}$. Chunks need time slots $F_c \ \forall c$. The algorithm, shown in Algorithm 2 (CRED-S), is comprised of two steps. As mentioned above, each step of the algorithm consists of multiple iterations and we use $r$ to denote the $r$th iteration.

When $\sum_{c \in \mathcal{H}_B} F_c^{(r)} > S \cdot d$, we place $\mathcal{H}_{B,d}^{(r)}$ into a node and call SCHEDULE for time-slots' scheduling. The condition $\sum_{c \in \mathcal{H}_B} F_c^{(r)} > S \cdot d$ guarantees that $\mathcal{H}_{B,d}^{(r)}$ exists. By choosing $\mathcal{H}_{B,d}^{(r)}$, we can schedule $S \cdot d$ time slots in each node. Choosing $\mathcal{H}_{B,d}^{(r)}$ and calling SCHEDULE guarantees that we can take care of as many chunks as possible while scheduling $S \cdot d$ time slots in each iteration. If the remaining number of required time slots for chunk $c$ is 0, we can remove the chunk $c$ from the chunk set $\mathcal{C}$ and reduce the size of the chunk set $\mathcal{C}$. When $\sum_{c \in \mathcal{H}_B} F_c^{(r)} \leq S \cdot d$, we can

place any $B$ chunks into one node and remove all of them.

The basic idea of SCHEDULE 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 SCHEDULE are a set of chunks and the number of time slots needed for $d$. The number of time slots needed is the number of time slots waiting for scheduling in a node. We denote the number of time slots needed in the $r$th iteration as $NTS_d^{(r)}$. If the remaining number of required time slots of chunk $c$ is less than or equal to $NTS_d^{(r)}$, 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 $NTS_d^{(r)}$, 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 $NTS_d^{(r)}$, we schedule the $NTS_d^{(r)}$ from chunk $c$ in the node. We then deduct $NTS_d^{(r)}$ from the remaining number of required time slots of chunk $c$ and mark $NTS_d^{(r)}$ as 0.

**Algorithm 1:** SCHEDULE($\mathcal{C}$, $NTS_d^{(r)}$)

```
sort C based on the remaining number of required time slots for d in ascending order
for c = 1 : C do
    if F_c^(r) − NTS_d^(r) > 0 then
        F_c^(r) = F_c^(r) − NTS_d^(r)
        NTS_d^(r) = 0
        break
    else
        NTS_d^(r) = NTS_d^(r) − F_c^(r)
        F_c^(r) = 0
        remove the chunk c
    end if
end for
```

It is easy to see that CRED-S will keep adding new nodes until all chunks get their required time slots $\sum_c F_c$ scheduled. Processing chunk $c$ is only permitted on a node where chunk $c$ is placed. Thus, the algorithm is guaranteed to generate a feasible solution to the CRED problem. To analyze the performance, we first consider time complexity, and space complexity of CRED-S. Then, we derive an upper bound to quantify the maximum number of active nodes needed by CRED-S. The upper bound is compared to a theoretical lower bound that establishes the minimum number of active nodes necessary for any feasible solution to the CRED problem.

The time complexity of CRED-S is dominated by the sort, and the time complexity is $O(K{\cdot}C{\cdot}\lg(C))$, where $K$ is the maximum number of iterations, and $C$ is the maximum number of remaining chunks. We can use HashMap to store chunks, where keys are chunk indexes, and values are chunk time slots needed. So, the space complexity is $O(C)$. Next, we will analyze each step in CRED-S to derive upper and lower bounds on the number of nodes needed, denoted by $\hat{N}$. The basic idea of deriving the lower bound is to only consider time slots or block constraint in each node. The basic idea of deriving the upper bound is to fix the number of removable chunks in each iteration of each step.

**Algorithm 2:** CRED-S

```
Input: C, ∑_{i=1}^{C} f_i^d, B, d
Output: N̂
C^(r) = C
while C^(r) > 0 do
    sort chunks based on the number of required time slots
    if ∑_{c∈H_B} F_c^(r) > S·d then
        place H_{B,d}^(r) into one node
        SCHEDULE(H_{B,d}^(r), S·d)
    else
        break
    end if
end while
while C^(r) > 0 do
    place C_B into one node
    SCHEDULE(C_B, S·d)
end while
```

**Theorem 1.** *When $K > \lfloor\frac{2C}{B}\rfloor$, the bounds are given by $K \le \hat{N} \le K+1$. When $\lfloor\frac{2C}{B}\rfloor \ge K \ge \lfloor\frac{C}{B-1}\rfloor$, the bounds are given by $\max(\lceil\frac{C}{B}\rceil, K) \le \hat{N} \le \frac{K}{2} + \frac{C}{B} + 1$. When $K < \lfloor\frac{C}{B-1}\rfloor$, the bounds are given by $\lceil\frac{C}{B}\rceil \le \hat{N} \le \frac{K}{B} + \frac{C}{B} + 1$.*

*Proof.* In each node, we can place at most $B$ chunks or schedule $S{\cdot}d$ time slots. To place $C$ chunks, we need at least $\lceil\frac{C}{B}\rceil$ nodes. To schedule $\sum_j T_j$ required time slots, we need at least $K$ nodes. To place $C$ chunks and schedule $\sum jT_j$ required time slots, we need at least $\max(\lceil\frac{C}{B}\rceil, K)$ nodes. As a result, the lower bound is $\max(\lceil\frac{C}{B}\rceil, K)$.

We now derive the upper bound for CRED-S. Instead of removing as many chunks as possible, we only remove the assigned number of chunks in each iteration of each step. In the following, we call this as the *simplified version* of CRED-S (CRED-SS). If the assigned number of chunks has been removed, even though the remaining number of required time slots of

other chunks equals 0, we still consider those chunks in the following cases. For CRED-SS, in step 1, when $\lfloor \frac{C^{(r)}}{B-1} \rfloor \leq k^{(r)} \leq \lfloor \frac{2C^{(r)}}{B} \rfloor$, we remove $\frac{B}{2}$ chunks in each iteration. When $k^{(r)} \leq \lfloor \frac{C^{(r)}}{B-1} \rfloor$ and $\sum_{c \in \mathcal{H}_B} F_c^{(r)} > S \cdot d$, we remove $B-1$ chunks in each iteration. In step 2, CRED-SS also removes any $B$ chunks in each iteration.

In the following, we first derive upper bounds for CRED-SS. Then, we show that the number of nodes needed by CRED-SS is no fewer than the number needed by CRED-S. Therefore, the upper bounds for CRED-SS are also upper bounds for CRED-S.

We introduce *Lemma 1* to verify that for CRED-SS, once $\lfloor \frac{C^{(r)}}{B-1} \rfloor \leq k^{(r)} \leq \lfloor \frac{2C^{(r)}}{B} \rfloor$, in each iteration, we can remove at least $\frac{B}{2}$ chunks. Also, we introduce *Lemma 2* to verify that once $k^{(r)} \leq \lfloor \frac{C^{(r)}}{B-1} \rfloor$, in each iteration, we can remove at least $B-1$ chunks. Proofs of these lemmas, as well as *Lemma 3*, can be found in the *Appendix*.

**Lemma 1.** *For* CRED-SS, *when* $k^{(r)} \leq \lfloor \frac{2C^{(r)}}{B} \rfloor$, *where* $C^{(r)} = C - r \cdot \frac{B}{2}$, *we have* $\sum_{c \in \mathcal{L}_{B/2}} F_c^{(r)} \leq S \cdot d$. *Here* $C$ *is the size of the chunk set when* $r = 0$.

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

**Lemma 2.** *For* CRED-SS, *when* $k^{(r)} \leq \lfloor \frac{C^{(r)}}{B-1} \rfloor$ *and* $\sum_{c \in \mathcal{H}_B} F_c^{(r)} > S \cdot d$, *where* $C^{(r)} = C - r \cdot (B-1)$, *the total number of required time slots of* $\mathcal{LH}_{B-1,d}$ *is less than or equal to* $S \cdot d$. *Here* $C$ *is the size of the chunk set when* $r = 0$.

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

*Case 1:* $K > \lfloor \frac{2C}{B} \rfloor$. Assume we need $r_1^1$ iterations of step 1 to make $k^{(r_1^1)} \leq \lfloor \frac{2C^{(r_1^1)}}{B} \rfloor$. Assume we need another $r_2^1$ iterations of step 1 to make $k^{(r_1^1+r_2^1)} \leq \lfloor \frac{C^{(r_1^1+r_2^1)}}{B-1} \rfloor$. Assume we need another $r_3^1$ iterations of step 1 to make $\sum_{c \in \mathcal{H}_B} F_c^{(r_1^1+r_2^1+r_3^1)} \leq S \cdot d$. So, the total number of nodes for step 1 and step 2 is

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

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

*Case 2:* $\lfloor \frac{2C}{B} \rfloor \geq K \geq \lfloor \frac{C}{B-1} \rfloor$. Assume we need $r_1^2$ iterations of step 1 to make $k^{r_1^2} \leq \lfloor \frac{C^{r_1^2}}{B-1} \rfloor$. Assume we need another $r_2^2$ iterations of step 1 to make $\sum_{c \in \mathcal{H}_B} F_c^{(r_1^2+r_2^2)} \leq S \cdot d$. So, the total number of nodes for step 1 and step 2 is

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

We know that $B \geq 2$ and $r_1^2 + r_2^2 \leq K$. So, (7) is less than or equal to $\frac{K}{2} + \frac{C}{B} + 1$.

*Case 3:* $K < \lfloor \frac{C}{B-1} \rfloor$. Assume we need another $r_1^3$ iterations of step 1 to make $\sum_{c \in \mathcal{H}_B} F_c^{r_1^3} \leq S \cdot d$. So, the total number of nodes for step 1 and step 2 is

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

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

**Lemma 3.** *In each iteration of step 1, for each chunk, the number of time slots scheduled by* CRED-S *and* CRED-SS *are exactly the same.*

After finishing step 1, the size of the remaining chunk set of CRED-SS is no less than CRED-S's. For step 2, we can remove $B$ chunks in each iteration. Thus, the number of nodes needed by CRED-SS is larger than or equal to the number of nodes needed by CRED-S. □

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

It is easy to check the tightness of these upper bounds by simple examples.

### 4.2 Solving CRED with multiple deadlines

We propose a heuristic algorithm to solve CRED with multiple, arbitrary deadlines. Our idea is to iteratively apply CRED-S to incrementally find the chunk placement and time-slot schedules 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 CRED-M. $\hat{N}_i$ denotes the number of nodes needed for scheduling jobs with deadline $d_i^{\uparrow}$. CRED-M's performance is characterized in Theorem 2. Assume there are $D$ distinct deadlines. We consider chunk placement and time slots scheduling of distinct deadlines

one-by-one, from $d_1^{\uparrow}$ to $d_D^{\uparrow}$. For deadline $d_i^{\uparrow}$, CRED-M first calls CRED-S 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
    N̂_i = CRED-S (C_i, Σ_c F_{c,i}^{(r)}, B, d_i^↑)
    for n = 1 : N̂_i do
        for i_1 = i + 1 : D do
            SCHEDULE(the set of chunks in node n,
            S·d_{i_1}^↑ − #scheduled time slots in node n)
        end for
    end for
    N̂ += N̂_i
end for
```

**Theorem 2.** *Let the number of nodes needed be* $\hat{N}$. *Then,* $\max\left(\max_i(K_i), \frac{\sum_j |\mathcal{C}_j|\cdot T_j}{S\cdot d_D^{\uparrow}}, \frac{C}{B}\right) \leq \hat{N} \leq \sum_{i=1}^{D} \max\left(K_i, \frac{K_i}{2} + \frac{C_i}{B}\right) + D$.

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

Based on Theorem 1, the lower bound of $\max(\max_i(K_i), \frac{C}{B})$ is obvious. Another lower bound can be obtained by dividing the total number of time slots needed for all jobs ($\sum_j |\mathcal{C}_j|\cdot T_j$) by the maximum possible number of time slots that can be accommodated in each node ($S\cdot d_D^{\uparrow}$). Thus, the lower bound is $\max\left(\max_i(K_i), \frac{\sum_j |\mathcal{C}_j|\cdot T_j}{S\cdot d_D^{\uparrow}}, \frac{C}{B}\right)$.

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.*

## 5 Our solution to resilient CRED problem

In this section, we consider a resilient CRED problem (CRED-R) that aims to find the optimal cloud right-sizing while guaranteeing survivability under arbitrary single node failure. We assume that a failure could occur on any node and at any time during the execution of jobs. Once a failure happens, we assume that the node becomes unavailable from then on and its data chunks are lost. We consider an online recovery strategy, which updates the job schedules after failure in order to obtain a new solution that meets both execution deadline and data locality constraints. Due to low overhead for rescheduling jobs on the fly, such an online recovery strategy allows the cloud to perform efficient, on-demand recovery without activating new nodes or migrating data chunks.

To solve the resilient CRED problem under arbitrary single node failure, a solution must guarantee that (1) each chunk has replicas on different nodes, and (2) there exists a feasible job scheduling after the failure without changing chunk placement. The key steps in our solution are: (a) choose a group of chunks, (b) make copies of each group of chunks, and (c) place copies of each group on different nodes. The details are given in Section 5.1.

In the following, we denote the $a$th group of chunks as $\mathcal{G}_a$ and the $m$th copy of $\mathcal{G}_a$ as $\mathcal{G}_{a,m}$. The proposed solution can be illustrated using a directed graph $G(\mathcal{V}, \mathcal{E})$ in Figure 2. Each vertex represents a node/machine. Two vertices are connected by an arc if the corresponding time slots are redirected from the source node to the destination node. $W_{n,a,m}^{i}$ denotes the number of time slots needed by $\mathcal{G}_{a,m}$ in node $n$ for deadline $d_i^{\uparrow}$.

It is easy to see that cloud right-sizing now becomes the problem of minimizing the number of nodes $|\mathcal{V}|$ in $\mathcal{G}$. A chunk placement and job scheduling scheme represented by $\mathcal{G}$ satisfies the storage constraint in the CRED problem, because each node hosts exactly $B$ chunks. The deadline constraint now becomes

$$W_{n,a,m}^{i} \leq \frac{S\cdot d_i^{\uparrow}}{2},\ \forall n, a, m. \tag{9}$$

Let $W_{n,a,m,c}^{i}$ be the number of time slots needed for chunk $c$ by $\mathcal{G}_{a,m}$ in node $n$ for deadline $d_i^{\uparrow}$. The computing resource constraint is satisfied if

$$\sum_{n,a,m} W_{n,a,m,c}^{i} \geq F_{c,i},\ \forall c, i. \tag{10}$$

Also, the storage constraint becomes

$$|\mathcal{G}_{a,m}| \leq \frac{B}{2},\ \forall a, m \tag{11}$$

where $|\mathcal{G}_{a,m}|$ is the number of chunks in the group $\mathcal{G}_{a,m}$. Therefore, a solution represented by $\mathcal{G}$ satisfying (9), (10), and (11), is a feasible solution to the CRED problem. The solution can survive arbitrary single node failure if condition (12) is satisfied and Symmetric Recovery Method is used for placing groups in nodes. The details are given in Section 5.1.

$$\textstyle\sum_j |\mathcal{C}_j|\cdot T_j + S\cdot d_D^{\uparrow} \leq \hat{N}\cdot S\cdot d_D^{\uparrow} \tag{12}$$

Here, we give a toy example to explain how our resilient algorithm works. In an $S = 2$ and $B = 4$

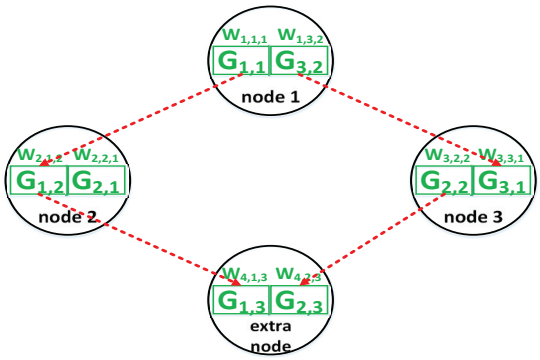



Fig. 2: Symmetric recovery graph.

system, we have 6 jobs and $T_j = 2, \forall\, j \in [1, 6]$. All jobs have equal deadlines, i.e., $d_j = 2$. We have 6 chunks and each chunk is accessed by one job. We make groups $\mathcal{G}_a$, $a \in [1, 3]$. Each group has two chunks, e.g., $\mathcal{G}_a$ contains the chunk $2a - 1$ and the chunk $2a$. We first make two copies of each group. The number of scheduling time slots of each copy is 2, which equals $\frac{S \cdot d_j}{2}$, and the number of scheduling time slots of each chunk in each copy is 1. We place $\mathcal{G}_{a,1}$ into node $a$ and place $\mathcal{G}_{a,2}$ into node $(a \mod 3) + 1$, where 3 is the total number of groups. Figure 2 shows the symmetric recovery graph for this example. We need one extra node $n_e$ for failure recovery. On the extra node, $\mathcal{G}_{1,3}$ and $\mathcal{G}_{2,3}$, which are two extra copies from randomly selected groups, are hosted and no processing slots are allocated at this time. Suppose node 1, hosting $\mathcal{G}_{1,1}$ and $\mathcal{G}_{3,2}$, fails. We redirect 2 scheduled time slots of $\mathcal{G}_{1,1}$ to $\mathcal{G}_{1,2}$ and redirect 2 scheduled time slots of $\mathcal{G}_{1,2}$ to $\mathcal{G}_{1,3}$ hosted in $n_e$. Also, we redirect 2 scheduled time slots of $\mathcal{G}_{3,2}$ to $\mathcal{G}_{3,1}$ and redirect 2 scheduled time slots of $\mathcal{G}_{2,2}$ to $\mathcal{G}_{2,3}$ hosted in $n_e$. Now, the number of scheduled time slots of each chunk is still 2. Thus, the failure has been recovered.

In Section 5.1, we first consider the single-deadline case and then use the result from Section 5.1 to derive the upper bound for the multiple-deadline case in Section 5.2.

### 5.1 Solving resilient CRED with equal deadlines

As before, we first consider the special case where all jobs have equal deadlines $d_i = d_1^{\uparrow} = d\ \forall i$, and propose an algorithm CRED-RS, which consists of two parts, i.e., GROUPCHUNKS and PLACINGGROUPS, to find a graph $\mathcal{G}$ that provides a solution to the resilient CRED problem. Also, we use the notations introduced in Section 4.1.

In GROUPCHUNKS, when $\sum_{c \in \mathcal{H}_{B/2}} F_c^{(r)} > S \cdot d$, in each iteration, we choose $\mathcal{H}_{B/2,d}^{(r)}$ as a group and make two copies of $\mathcal{H}_{B/2,d}^{(r)}$. After making copies, we use SCHEDULE for time slots scheduling for each copy and the maximum number of scheduling time slots of each copy is $\frac{S \cdot d}{2}$. The condition $\sum_{c \in \mathcal{H}_{B/2}} F_c^{(r)} > S \cdot d$ guarantees that we can always find $\mathcal{H}_{B/2,d}^{(r)}$. In each iteration, we satisfy two goals: scheduling more time slots and removing more chunks. When $\sum_{c \in \mathcal{H}_{B/2}} F_c^{(r)} \leq S \cdot d$, in each iteration, we choose $B/2$ as a group and make two copies of the group. After making copies, we use SCHEDULE for time slots scheduling for each copy and the number of time slots for scheduling of each copy is at most $\frac{S \cdot d}{2}$.

The main purpose of PLACINGGROUPS is to place copies from GROUPCHUNKS into nodes. The key idea is that we need to put copies from the same group into different nodes.

Suppose the total number of groups is $G$. $\mathcal{G}_{a,1}$ is placed on node $a$ and $\mathcal{G}_{a,2}$ is placed on node $(a \mod G)+1$ and we call this method for placing groups as SYMMETRIC RECOVERY METHOD in the following. For failure recovery, we need one extra node ($n_e$) hosting any two extra copies without any processing time slots scheduled. For simplicity, we suppose $\mathcal{G}_{G,3}$ and $\mathcal{G}_{G,4}$ are placed in the extra node $n_e$.

The time complexity of CRED-RS is dominated by the sort, and so the time complexity is $O(K \cdot C \cdot \lg(C))$, where $K$ is the maximum number of iterations, and $C$ is the maximum number of remaining chunks. We can use HashMap to store chunks, where keys are chunk indexes, and values are chunk time slots needed. So, the space complexity is $O(C)$. Next, we analyze CRED-RS and obtain bounds on the number of nodes needed. The basic idea of deriving the upper bound is to introduce a simplified version CRED-RSS of CRED-RS, derive the upper bound for CRED-RSS, and prove that the upper bound of CRED-RSS is also an upper bound for CRED-RS.

**Theorem 3.** *If* $K > \lfloor \frac{2C}{B} \rfloor$, *then* $K + 1 \leq \hat{N} \leq K + \frac{4C}{B^2} + 2$. *If* $K \leq \lfloor \frac{2C}{B} \rfloor$, *then* $\max(\frac{C}{B}, K) + 1 \leq \hat{N} \leq \frac{2K}{B} + \frac{2C}{B} + 2$.

*Proof.* Recall that any node can fail at any time. Suppose that either a node with $S \cdot d$ scheduled time slots or a node hosting $B$ chunks fails, then we need at least one extra node for failure recovery. Thus, the lower bound is $\max(\frac{C}{B}, K) + 1$ nodes.

From PLACINGGROUPS, we know the number of nodes needed equals the number of groups. Thus, we just need to derive the upper bound for the number of groups.

We modify step 1 of GROUPCHUNKS. Instead of removing as many chunks as possible in each iteration of step 1, CRED-RSS only removes specific number of chunks. For the CRED-RSS, in step 1, when

$k^{(r_1)} \leq \lfloor \frac{2C^{(r_1)}}{B} \rfloor$ and $\sum_{c \in \mathcal{H}_{B/2}} F_c^{(r_1)} > S \cdot d$, in each iteration, we remove $\frac{B}{2} - 1$ chunks. In step 2, CRED-RSS also removes any $B/2$ chunks in each iteration.

Based on Lemma 1, when $k^{(r_1)} \leq \lfloor \frac{2C^{(r_1)}}{B} \rfloor$, we can remove $\frac{B}{2} - 1$ chunks in each iteration and based on similar principles of Lemma 3, the number of scheduled nodes before the $r$th iteration of step 1 for GROUPCHUNKS and CRED-RSS should be the same. For step 2, the number of remaining chunks, including chunks with zero required time slots, of CRED-RSS is larger than the number of remaining chunks of GROUPCHUNKS. Also, we can remove $B/2$ chunks in each scheduling node of step 2. Thus, the number of nodes needed for CRED-RSS is larger than or equal to the number of nodes needed for GROUPCHUNKS.

In the following, we derive the upper bound of CRED-RSS for two cases.

*Case 1:* $K > \lfloor \frac{2C}{B} \rfloor$. Assume we need $r_1^1$ iterations of step 1 to make $K \leq \lfloor \frac{2C}{B} \rfloor$, and assume we need another $r_2^1$ iterations of step 1 to make $\sum_{c \in \mathcal{H}_{B/2}} F_c^{(r_1^1 + r_2^1)} \leq S \cdot d_1^{\uparrow}$. The total number of groups for both step 1 and step 2 is no more than

$$r_1^1 + r_2^1 + \left\lceil \frac{C - (\frac{B}{2} - 1) r_2^1}{\frac{B}{2}} \right\rceil. \tag{13}$$

We know that $r_1^1 = K - \lfloor \frac{2C}{B} \rfloor$ and $r \leq \lfloor \frac{2C}{B} \rfloor$. Thus, the total number of groups is no more than $K + \frac{4C}{B^2} + 1$. Also, in PLACINGGROUPS, we might need one extra node, thus the upper bound is $K + \frac{4C}{B^2} + 2$.

*Case 2:* $K \leq \lfloor \frac{2C}{B} \rfloor$. Assume we need another $r_1^2$ iterations of step 1 to make $\sum_{c \in \mathcal{H}_{B/2}} F_c^{(r_1^1 + r_2^1)} \leq S \cdot d$. The total number of groups is no more than

$$r_1^2 + \left\lceil \frac{C - (\frac{B}{2} - 1) r_1^2}{\frac{B}{2}} \right\rceil. \tag{14}$$

We have $r_1^2 \leq K$. Thus, the total number of groups is no more than $\frac{2K}{B} + \frac{2C}{B} + 1$. Also, in PLACINGGROUPS, we might need one extra node, thus the upper bound is $\frac{2K}{B} + \frac{2C}{B} + 2$. □

**Algorithm 4:** CRED-RS

1: Input: $C$, $\sum_{i=1}^{C_1} f_i^d$, $B$, $d$
2: Output: $\hat{N}$
3: GROUPCHUNKS($C$, $\sum_{i=1}^{C} f_i^d$, $B$, $d$)
4: PLACINGGROUPS: put copies of groups into nodes by using SYMMETRIC RECOVERY METHOD and add one extra node

The basic idea of failure recovery is that we can redirect scheduled time slots among copies of the same group and eventually redirect scheduled time slots from the failed node to healthy nodes. Suppose the node $n_1$ hosting $\mathcal{G}_{a_1,1}$ and $\mathcal{G}_{a_2,2}$ fails. We redirect processing time slots of $\mathcal{G}_{a_1,1}$ and $\mathcal{G}_{a_2,2}$ to copies of $\mathcal{G}_{a_1}$ and $\mathcal{G}_{a_2}$, respectively. Copies whose scheduled time slots have been redirected are marked, and will not be the target of any further redirection. If the total number of scheduled time slots in a node becomes more than $S \cdot d$, we redirect scheduled time slots of unmarked copy to another copy of the same group. If there is an extra copy in $n_e$ for any group, we first redirect time slots to that extra copy.

**Algorithm 5:** GROUPCHUNKS

1: Input: $C$, $\sum_{i=1}^{C} f_i^d$, $B$, $d$
2: Output: $groups$
3: $C^{(r)} = C$
4: **while** $C^{(r)} > 0$ **do**
5: sort $C^{(r)}$ chunks based on the remaining number of required time slots
6: **if** $\sum_{c \in \mathcal{H}_{B/2}} F_c^{(r)} > S \cdot d$ **then**
7: make two copies of $\mathcal{H}_{B/2,d}^{(r)}$
8: SCHEDULE($\mathcal{H}_{B/2,d}^{(r)}$, $\frac{S \cdot d}{2}$) for each copy
9: **else**
10: break
11: **end if**
12: **end while**
13: **while** $C^{(r)} > 0$ **do**
14: make two copies of $\mathcal{C}_{B/2}$
15: SCHEDULE($\mathcal{C}_{B/2}$, $\frac{S \cdot d}{2}$) for each copy
16: **end while**

**Theorem 4.** *Suppose the group set is $\mathcal{G}$ and the total number of nodes needed is $\hat{N}$. The sufficient conditions for successful recovery are:*

$$\sum_{n,a,m} W_{n,a,m,c} \geq F_c, \ \forall c \tag{15}$$

$$|\mathcal{G}_{a,m}| \leq \frac{B}{2}, \ \forall a, m \tag{16}$$

$$W_{n,a,m} \leq \frac{S \cdot d}{2}, \ \forall n, a, m \tag{17}$$

$$\sum_j |\mathcal{C}_j| \cdot T_j + S \cdot d \leq \hat{N} \cdot S \cdot d, \tag{18}$$

Using SYMMETRIC RECOVERY METHOD.

*Proof.* (15) means that the required time slots for chunk $c$ are scheduled before the deadline and it corresponds to (1) in Section 3. (16) means the number of chunks in each group is no more than $B/2$, and (17) means the number of scheduled time slots in each group is no more than $\frac{S \cdot d}{2}$. Also, PLACINGGROUPS guarantees that at most two groups are hosted in a node; so, the storage and computational constraints are satisfied. Thus, a solution that satisfies (15), (16), and (17) is a feasible solution to the CRED problem. Also, (17) guarantees that the number of scheduled

time slots in a node after redirection does not exceed $S{\cdot}d$. (18) ensures that the number of available time slots in all nodes is sufficient to accommodate the total number required time slots even after a single node failure.

To see this, suppose that the above online recovery algorithm can not recover from single node failure. This means that the redirection of time slots in the SYMMETRIC RECOVERY METHOD results in a solution where there are no available slots in the system. More precisely,

$$(N-1){\cdot}S{\cdot}d < \sum_j T_j. \tag{19}$$

But this is contrary to our constraint $\sum_j T_j + S{\cdot}d \geq \hat{N}{\cdot}S{\cdot}d$, so we have enough available time slots for scheduling time slots of all copies after one node failure. □

**Remark.** GROUPCHUNKS *guarantees (15), (16), and (17).* PLACINGGROUPS *guarantees (18), and ensures that the* SYMMETRIC RECOVERY METHOD *is feasible.*

### 5.2 Solving resilient CRED with multiple deadlines

Now we present our proposed algorithm to solve the resilient CRED problem with multiple deadlines. Our idea is to iteratively apply CRED-RS for deadline $d_i^{\uparrow}$, $\forall i$. The algorithm is summarized in CRED-RM. $\hat{G}_i$ denotes the number of groups needed for scheduling jobs with deadline $d_i^{\uparrow}$. CRED-RM's performance is characterized in Theorem 5. As before, suppose we have $D$ distinct deadlines.

**Theorem 5.** $\max\left(\max_i(K_i), \frac{\sum_{i=1}^{D}\sum_{j:d_j=d_i^{\uparrow}} T_j}{Sd_D^{\uparrow}}, \frac{\max_i C_i}{B}\right) + 1 \leq \hat{N} \leq \sum_{i=1}^{D} \max\left(K_i + \frac{4C_i}{B^2}, \frac{2K_i}{B} + \frac{2C_i}{B}\right) + D + 1.$

*Proof.* We first consider the lower bound. Consider storage constraint, computational constraint, and all jobs are for $d_D^{\uparrow}$. We can get the same lower bound as for CRED-M. Since any node can fail at any time, the lower bound is $\max\left(\max_i(K_i), \frac{\sum_{i=1}^{D}\sum_{j:d_j=d_i^{\uparrow}} T_j}{Sd_D^{\uparrow}}, \frac{\max_i C_i}{B}\right) + 1$. From Theorem 3, we know that for single deadline $d_i^{\uparrow}$, the upper bound is $\max\left(K_i + \frac{4C}{B^2}, \frac{2K_i}{B} + \frac{2C}{B}\right)+1$. We add the extra node at the end of the proof. For deriving the upper bound for multiple deadlines, we iteratively use the result of Theorem 3 for deadline $d_1^{\uparrow}, \ldots, d_D^{\uparrow}$. Thus, the upper bound for the total number of groups is

$$\sum_{i=1}^{D} \max\left(K_i + \frac{4C_i}{B^2}, \frac{2K_i}{B} + \frac{2C_i}{B}\right) + D. \tag{20}$$

We use PLACINGGROUPS to place groups into nodes and add one extra node without scheduling any time slot on that node. Thus, the upper bound for the total number of nodes is the same as

$$\sum_{i=1}^{D} \max\left(K_i + \frac{4C_i}{B^2}, \frac{2K_i}{B} + \frac{2C_i}{B}\right) + D + 1. \tag{21}$$

□

**Algorithm 6:** CRED-RM

```
for i = 1 : D do
    Ĝ_i=GROUPCHUNKS(C_i, Σ_c F_{c,i}^{(r)}, B, d_i^↑)
    for a = 1 : Ĝ_i do
        for m: copies in group a do
            for i_1 = i : D do
                SCHEDULE(G_{a,m}, S·d_{i_1}^↑/2 − # scheduled
                time slots in G_{a,m})
            end for
        end for
    end for
    G+ = Ĝ_i
end for
PLACINGGROUPS: put copies of groups into
nodes by using SYMMETRIC RECOVERY METHOD
and add an extra node
```

**Theorem 6.** *Suppose the group set is $\mathcal{G}$ and the total number of nodes is $\hat{N}$. The sufficient conditions for successful recovery are:*

$$\textstyle\sum_{n,a,m} W^i_{n,a,m,c} \geq F_{c,i},\ \forall c, i \tag{22}$$

$$|\mathcal{G}_{a,m}| \leq \tfrac{B}{2},\ \forall a, m \tag{23}$$

$$W^i_{n,a,m} \leq \tfrac{S{\cdot}d_i^{\uparrow}}{2}\ \forall i, n, a, m \tag{24}$$

$$\textstyle\sum_j |\mathcal{C}_j|{\cdot}T_j + S{\cdot}d_D^{\uparrow} \leq \hat{N}{\cdot}S{\cdot}d_D^{\uparrow} \tag{25}$$

Using SYMMETRIC RECOVERY METHOD

*Proof.* The failure recovery method for multiple deadlines is the same as the failure recovery method for equal deadlines case. We give a brief proof here. (22), (23), and (24) guarantee that computational, storage, and deadline constraints are satisfied. (25) makes the number of available time slots be no less than $S{\cdot}d_D^{\uparrow}$, and SYMMETRIC RECOVERY METHOD guarantees that the redirection can go through all nodes. To see this, suppose that the online recovery algorithm cannot recover from a single node failure. This means there are no available time slots in any node in the system. More precisely,

$$(N-1){\cdot}S{\cdot}d_D^{\uparrow} < \sum_j T_j, \tag{26}$$

but this is contrary to our constraint $\sum_j T_j + S{\cdot}d_D^{\uparrow} \geq \hat{N}{\cdot}S{\cdot}d_D^{\uparrow}$, so the assumption is incorrect. Thus, the system can recover from any single node failure. □

**Remark.** GROUPCHUNKS *guarantees (22), (23), and (24).* PLACINGGROUPS *guarantees (25), and ensures that the* SYMMETRIC RECOVERY METHOD *is feasible.*

## 6 EVALUATION

In this section, we perform various simulation experiments for the single- and two- deadline cases to evaluate our algorithmic solution to the CRED problem. We focus on two aspects: first, we show that CRED's performance is within the closed-form bounds we proved in Sections 4 and 5. Second, we compare CRED's performance and *deadline-aware first-fit* (FF) algorithm in terms of number of nodes and resource utilization.

### 6.1 Evaluation by simulation

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: elephant and mouse jobs. Elephant jobs are computation-intensive jobs; the number of time slots required to process a chunk is randomly chosen between 200 and 500. The number of time slots required to process a chunk of a mouse job is randomly chosen between 1 and 10. Elephants and mice jobs form 2% and 98% of the total jobs (100 jobs with equal deadlines and 200 jobs with two deadlines), respectively. The number of files is 100, and the number of chunks per file is randomly chosen between 16 and 64. All jobs are assigned to files randomly. For equal-deadline experiments, we fix the deadline to be equal to 600 (time slots) for all experiments; for two-deadline experiments, we fix $d_1$ and $d_2$ to be equal to 600 and 1200, respectively. For all experiments, the figures show the average over 20 trials. Error bars, where are shown in Figure 3(a), 3(b), and 4(a), represent $95\%$ confidence intervals.

Figure 3(a) shows the effect of changing the number of blocks in each node on 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 a mix of time-slots and blocks constraints, our algorithm outperforms FF by up to 18% in the number of nodes.

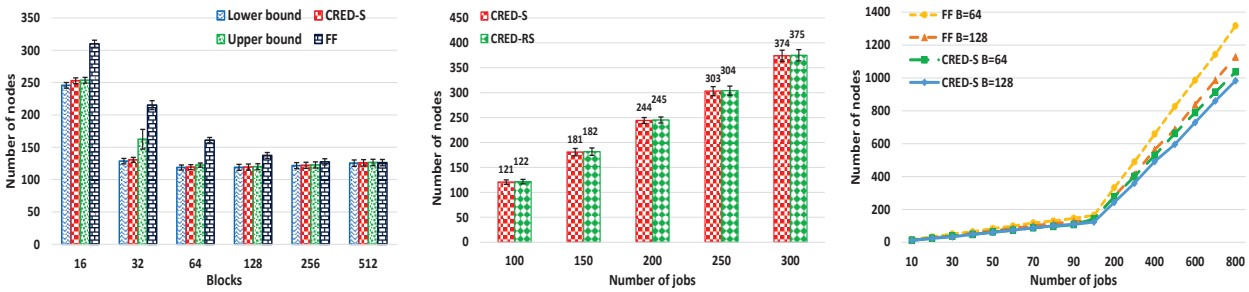
In Figure 3(b), we compare CRED-S and CRED-RS in terms of number of nodes needed. In this experiment, the results show that we always meet the lower bound for CRED-S and CRED-RS. The lower bound here for CRED-S and CRED-RS are $K$ and $K+1$, respectively. This validates our proof in Section 5.

In Figure 3(c), we compare CRED-S with the first-fit algorithm in terms of scalability. In this experiment, we have two classes of jobs, elephants and mice jobs. Elephants and mice jobs form 20% and 80% of the total jobs, respectively. We show two experiments, with $B = 64$ and $B = 128$. The figure shows that on average the first-fit algorithm requires about 23% and 18% more nodes than CRED-S when $B = 64$ and $B = 128$, respectively. These percentages stay the same for any number of jobs. This means that CRED-S is scalable and works even if the system is large. The figure also shows that when the number of jobs is greater than 100, the number of nodes increases due to time-slots constraint.

Figure 4(a) shows that CRED-M's performance is within the closed-form bounds we proved in Section 4. 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, the number of nodes increases since $d_1 < d_2$.

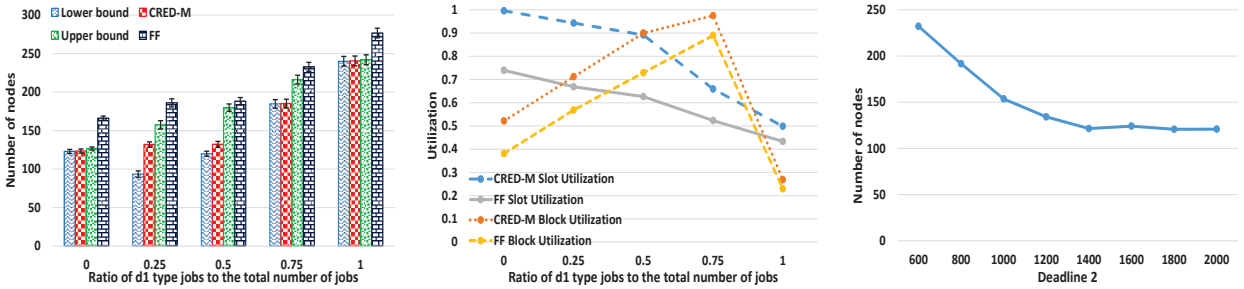
Figure 4(b) compares CRED-M and FF in terms of time slots and blocks utilization, defined as the ratio of the number of processing time slots and blocks actually utilized to the total number available. In this figure, we use the same parameters as Figure 4(a). The figure shows that for any ratio, CRED-M achieves higher utilization in both time-slots (up to 28%) and blocks (up to 15%). This is because, unlike FF, CRED always tries to fully utilize the time-slots and blocks in each node.

Figure 4(c) shows how CRED-M performs as we adjust the time difference between $d_1$ and $d_2$. In this experiment, we set $d_1$ = 600 and increase $d_2$ from 600 to 2000. The total number of jobs equals 200, and 100 jobs are associated with $d_1$. As we increase $d_2$, the number of nodes needed decreases dramatically at the beginning. But as $d_2$ increases further, we reach a point of diminishing returns where the number of nodes stays almost the same. In fact, this shows the benefit of having multiple deadlines for different jobs. By treating all jobs equally, $d_1$ = $d_2$, the number of nodes needed is about 232. If we increase some of the non-delay sensitive jobs' deadline by just 200, we can



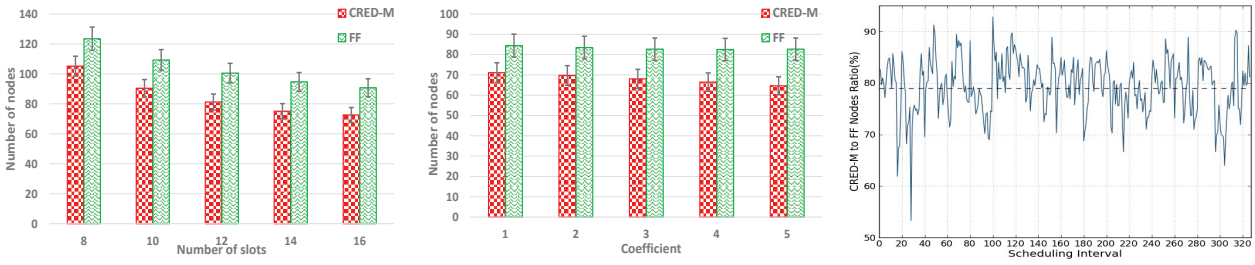

(a) (b) (c)

Fig. 3: Experiments with proposed algorithms CRED-S and CRED-RS: (a) Effect of number of blocks on the number of nodes. (b) Comparison between CRED-S and CRED-RS in terms of number of nodes needed. (c) Comparison between CRED-S and FF in terms of scalability.




(a) (b) (c)

Fig. 4: Experiments with proposed algorithm CRED-M: (a) Effect of the ratio of $d_1$- to $d_2$-type of jobs on the number of nodes needed. (b) Effect of the ratio of $d_1$- and $d_2$-type jobs on the time-slots and blocks utilization. (c) Effect of the time difference between $d_1$ and $d_2$ on the total number of nodes.




(a) (b) (c)

Fig. 5: Trace-driven comparison between CRED-M and First-Fit algorithm: (a) Effect of the number of slots on the number of nodes. (b) Effect of the ratio of $D_s$- and $D_l$-type jobs to the number of nodes needed. (c) Number of nodes needed in each scheduling interval.

reduce the number of nodes by 18%. Starting from $d_2$ = 1200, the blocks constraint becomes dominant, and the number of nodes stays more or less the same.

### 6.2 Trace-driven evaluation

In this section, we compare the performance of CRED-M with deadline-aware First-Fit algorithm through a trace-driven simulation using data from a publicly available Google trace [16]. The results show that CRED-M outperforms FF by up to 20% in nodes 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.

TABLE 1: Statistics of Google trace for trace-drive evaluation

| Class | # of jobs | avg # of tasks | Min task exec time | Avg task exec time | Max task exec time | First 5% | First 10% |
|---|---|---|---|---|---|---|---|
| 1 | 11084 | 261 | 320 sec | 1428 sec | 49462 sec | 10356 | 10774 |
| 2 | 1656 | 223 | 313 sec | 1511 sec | 50260 sec | 1504 | 1572 |

The scheduling class represents how latency-sensitive the job is. There exist 4 scheduling classes, with jobs at level 3 being the most latency-sensitive. In this simulation, we consider jobs belonging to scheduling classes 1 and 2. Summary statistics of the Google trace used for evaluation are presented in Table 1. The table shows show the number of jobs, average number of tasks, minimum execution time of a task, average execution time of a task, maximum execution time of a task, the number of first 5% of jobs with the smallest average execution time of tasks, and the number of first 10% of jobs with the smallest average execution time of tasks. For each class, a deadline $D_l$ or $D_s$ is assigned. 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 $\max(A_1, D_s)$. 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 5(a) compares the average number of active nodes (i.e., VM hours or cost) for the proposed CRED-M 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 5(b) compares the average number of active nodes for CRED-M 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 $\max(A_1, \alpha \cdot D_s)$ and we change $\alpha$ from 1 to 5. The figure shows that CRED-M outperforms FF by 17% in average number of nodes.

Figure 5(c) shows the number of active nodes required by CRED-M 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.

## 7 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 constraints are met. We analyze the competitive ratio of the proposed algorithms in closed-form and extend all results to solve a resilient CRED problem with arbitrary single node failure. The algorithms significantly outperform a first-fit heuristic in terms of cloud-size (i.e., number of active nodes needed) and node utilization.

In our future work, we will extend our work to heterogeneous cloud nodes equipped with different computing and storage resources. We plan to allocate different types of jobs to nodes equipped with different resources. Also, we plane to consider energy utilization and energy efficiency in the joint optimization problem. In addition, we plan to investigate deadline-aware scheduling algorithms for multi-phase cloud systems, e.g., MapReduce, which involve communication among tasks.

## 8 Acknowledge

This work is supported by NSF grant CSR-1320226.

## References

[1] S. Alamro, M. Xu, T. Lan, and S. Subramaniam, "Cred: Cloud right-sizing to meet execution deadlines and data locality," in *IEEE CLOUD*, 2016.

[2] 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.

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

[4] 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.

[5] A. Verma, L. Cherkasova, V. Kumar, and R. Campbell, "Deadline-based workload management for mapreduce environments: Pieces of the performance puzzle," in *NOMS*, 2012.
[6] 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.
[7] S. Shi, C. Wu, and Z. Li, "Cost-minimizing online vm purchasing for application service providers with arbitrary demands," in *CLOUD*, 2015.
[8] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, Jan. 2008.
[9] T. White, *Hadoop: The Definitive Guide*, 1st ed. O'Reilly Media, Inc., 2009.
[10] Z. Guo, G. Fox, and M. Zhou, "Investigation of data locality in mapreduce," in *CCGrid*, 2012.
[11] B. Palanisamy, A. Singh, L. Liu, and B. Jain, "Purlieus: Locality-aware resource allocation for mapreduce in a cloud," in *SC*, 2011.
[12] 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.
[13] 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.
[14] J. Hamilton, "Cost of power in large-scale data centers." [Online]. Available: http://goo.gl/FLa4CX.
[15] 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.
[16] "Google trace," https://github.com/google/cluster-data, 2011.
[17] J. Tan, X. Meng, and L. Zhang, "Coupling task progress for mapreduce resource-aware scheduling," in *INFOCOM*, 2013.
[18] J. J. Bo Wang and G. Yang, "Actcap: Accelerating mapreduce on heterogeneous clusters with capability-aware data," in *INFOCOM*, 2015.
[19] J. C. Anjos, I. Carrera, W. Kolberg, A. L. Tibola, L. B. Arantes, and C. R. Geyer, "Mra++: Scheduling and data placement on mapreduce for heterogeneous environments," *Future Generation Computer Systems*, vol. 42, pp. 22 – 35, 2015.
[20] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu, "Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems," in *ICDE*, 2011.
[21] C. He, Y. Lu, and D. Swanson, "Matchmaking: A new mapreduce scheduling technique," in *CLOUDCOM*, 2011.
[22] K. Kc and K. Anyanwu, "Scheduling hadoop jobs to meet deadlines," in *CloudCom*, 2010.
[23] X. Wang, D. Shen, M. Bai, T. Nie, Y. Kou, and G. Yu, "Sames: deadline-constraint scheduling in mapreduce," *Frontiers of Computer Science*, vol. 9, pp. 128–141, 2015.
[24] A. Verma, L. Cherkasova, and R. H. Campbell, "Aria: Automatic resource inference and allocation for mapreduce environments," in *ICAC*, 2011.
[25] J. Polo, Y. Becerra, D. Carrera, J. Torres, E. Ayguade, and M. Steinder, "Adaptive mapreduce scheduling in shared environments," in *CCGrid*, 2014.
[26] S. Li, S. Hu, S. Wang, L. Su, T. Abdelzaher, I. Gupta, and R. Pace, "Woha: Deadline-aware map-reduce workflow scheduling framework over hadoop clusters," in *ICDCS*, 2014.
[27] B. Moseley, A. Dasgupta, R. Kumar, and T. Sarlós, "On scheduling in map-reduce and flow-shops," in *SPAA*, 2011.
[28] H. Xu and W. C. Lau, "Speculative execution for a single job in a mapreduce-like system," in *CLOUD*, 2014.
[29] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou, "Re-optimizing data-parallel computing," in *NSDI*, 2012.
[30] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca, "Jockey: Guaranteed job latency in data parallel clusters," in *EuroSys*, 2012.

PLACE PHOTO HERE

Biography text here.

PLACE PHOTO HERE

Biography text here.

PLACE PHOTO HERE

Biography text here.

PLACE PHOTO HERE

Biography text here.