Shed+: Optimal Dynamic Speculation to Meet Application Deadlines in Cloud

Sultan Alamro, *Student Member, IEEE*, Maotong Xu, *Student Member, IEEE*, Tian Lan, *Member, IEEE*, and Suresh Subramaniam, *Fellow, IEEE*

***Abstract*—With the growing deadline-sensitivity of cloud applications, adherence to specific deadlines is becoming increasingly crucial, particularly in shared clusters. A few slow tasks called stragglers can potentially adversely affect job execution times. Equally, inadequate slotting of data analytics applications could result in inappropriate resource deployment, ultimately damaging system performance. Against this backdrop, one effective way of tackling stragglers is by making extra attempts (or clones)[1] for every single straggler after the submission of a job. This paper proposes Shed+, which is an optimization framework utilizing dynamic speculation that aims to maximize the jobs' PoCD (Probability of Completion before Deadline) by making full use of available resources. Notably, our work encompasses a new online scheduler that dynamically recomputes and reallocates resources during the course of a job's execution. According to our findings, Shed+ successfully leverages cloud resources and maximizes the percentage of jobs meeting their deadlines. In our experiments, we have seen this percentage for heavy load going up to 98% for Shed+ as opposed to nearly 68%, 40%, 35% and 37% for Shed, Dolly, Hopper and Hadoop with speculation enabled, respectively.**

***Index Terms*—Cloud, mapreduce, stragglers, scheduling, cloning, speculation, deadlines.**

I. Introduction

MODERN applications such as financial services, enterprise IT, big data analytics and social networks are increasingly relying on distributed cloud computing frameworks, such as Dryad and MapReduce [1], [2], in order to attain mission-critical performance objectives. Massive quantities of data are divided into blocks and then stored within an underlying file system so as to support simultaneous processing of computation jobs across nodes and clusters on the cloud. As an example, the open-source software framework Hadoop, which is used to process big data [3], makes use of the MapReduce programming model.

The performance of such frameworks is often negatively impacted by stragglers, which are slow running tasks that result in long job execution time, thus rendering them infeasible for latency-sensitive applications requiring time guarantees for job completion. According to previous research, stragglers can be as much as 8 times slower than the median task [4]–[6]. In other words, all it takes is a few stragglers to have a massive effect on the performance of job completion times and breach the SLAs (Service Level Agreements).

In contemporary online applications including the retail platforms of Amazon and Facebook, the long tail of latency has shown to be of particular concern, given that 99.9th percentile response times are orders of magnitude worse than the mean [7]. Several factors are known to cause stragglers in the cloud. To begin with, nodes are mostly comprised of commodity components and entail inconsistent degrees of heterogeneity. As a result, nodes end up processing tasks at varying speeds. Secondly, the large-scale nature of datacenters has been shown to cause errors in both hardware and software, thus resulting in interruptions of task execution and snags in machines. Thirdly, virtualization and resource sharing require co-scheduling of tasks, which can lead to stragglers and resource interference when done on the same machine [5], [8]. Previous research has also demonstrated that congested links across datacenter networks can actually last for extensive periods of time and are another contributor of stragglers [9].

Recently, researchers have suggested reactive as well as proactive approaches to mitigate the impact of stragglers [4]–[6], [10]–[14]. While reactive strategies are aimed at tracking stragglers upon their occurrence and then launching speculative or extra copies of slow running tasks [4], [5], proactive approaches trigger clones that are replicas of original tasks upon job submission [6]. They unveil speculative tasks proactively without waiting for the detection of stragglers. Another work proposes a statistical learning technique called Wrangler. It forecasts stragglers before they occur on the basis of past data [12].

One of the key requirements for mission-critical and latency sensitive applications is the ability to meet deadlines [15], [16]. Currently, neither reactive nor proactive approach is capable of providing formal performance guarantees with regard to meeting application deadlines. In this paper, we propose Shed+, an optimization framework that leverages *dynamic speculation* in

Manuscript received December 7, 2019; revised March 17, 2020; accepted April 4, 2020. This work was supported in part by NSF grant CSR-1320226. The associate editor coordinating the review of this article and approving it for publication was H. Lutfiyya. *(Corresponding author: Sultan Alamro.)*

Sultan Alamro is with the Department of Electrical and Computer Engineering, George Washington University, Washington, DC 20052 USA, and also with the Department of Electrical Engineering, College of Engineering, Qassim University, Buraidah 51452, Saudi Arabia (e-mail: alamro@gwu.edu).

Maotong Xu is with News Feed, Facebook, Menlo Park, CA 94025 USA (e-mail: maotongxu@gmail.com).

Tian Lan and Suresh Subramaniam are with the Department of Electrical and Computer Engineering, George Washington University, Washington, DC 20052 USA (e-mail: tlan@gwu.edu; suresh@gwu.edu).

Digital Object Identifier 10.1109/TNSM.2020.2986477

[1]Cloning or extra attempts means task duplication.

order to increase the probability of meeting job deadlines. Our dynamic speculation approach differentiates itself from these works by taking into consideration individual job deadlines, and optimizing the number of speculative copies assigned for each task in order to collectively maximize the probability of meeting job deadline and available cloud resources. Furthermore, since task executions are stochastic, while introducing speculative copies can mitigate corresponding straggled tasks, other tasks (either already in the system or arriving in the future) may become stragglers later due to uncertainty in their executions. Thus, we periodically check for stragglers every θ seconds and re-balance the replication factor *r* among all active jobs to maximize the PoCD. We also develop and roll out a novel speculative launching approach which preserves the work performed by the original task/straggler prior to speculation, thus facilitating seamless execution of tasks and progress transfer to task speculation. Through the proposed framework, we jointly maximize the probability of all jobs meeting their deadlines by leveraging underutilized cloud resources.

Shed+ leverages a metric called PoCD (Probability of Completion before Deadlines), which was introduced in [17], [18] to quantify the likelihood that a MapReduce job would meet the desired deadline. Through the analysis of PoCD premised on cloud processing models, we formulate an optimization problem so as to maximize the overall PoCD of all active jobs by identifying and outlining the optimal number of speculative (or extra) attempts for each straggler as regards cloud resource constraints and job progress. Unlike our work, Chronos [18] assumes unlimited resources, where VMs are provisioned as needed and subject to the overall cost. Thus, new arrivals always have VMs available and no resource constraint is introduced. In addition, Chronos optimizes once and does not assume that task execution time is stochastic, which are subject to discrepancies as they progress during the lifetime of a job. Shed+ is an enhanced version of Shed, which was published in our preliminary work in [17]. Shed+ differs from Shed in the sense that it does more fine-grained allocation of resources to jobs. Unlike Shed which maximizes the PoCD at the job level (all tasks receive the same number of extra copies), the PoCD in Shed+ is maximized at the task level. In particular, Shed assumes the same replication factor for all tasks within the same job. That is, when a straggler is identified, Shed finds a common replication factor *r* for all its tasks to maximize the job's PoCD. Clearly, this would lead to launching unnecessary replicas for some tasks that are not yet stragglers, while being inadequate for some other tasks. In contrast, Shed+ is to assign a different replication factor r_i to each individual task *i*. Thus, an optimal number of replicas are launched for each straggling task, in a way that is aligned with its execution conditions, impact on job deadlines, overall resource availability, etc. This fine-granularity control of individual tasks can efficiently reduce the required system resources needed in PoCD optimization, allowing significant speedup. Second, Shed only performs re-optimization upon new jobs arrivals. Thus, replicas are only optimized and launched until a new job arrives at the system. To overcome this shortcoming, Shed+ periodically checks for stragglers every θ seconds and re-optimizes resources (i.e., replication factors r_i for different tasks) accordingly. We note that this is a continuous re-optimization during job execution, offering fine-grained control over the temporal dimension. Finally, to make sure we have accurate straggler information for the optimization, we introduce another parameter ξ in Shed+, so Shed+ waits for a job to progress ξ% (when a more accurate estimate of completion time can be obtained) before launching extra copies only for identified stragglers. Shed+ is able to jointly make decisions in the coupled design space.

Unlike Dolly [6] and default Hadoop which rely on a fixed number of clones for each job upon submission and launch one speculative/extra attempt for each straggler, respectively, Shed+ is dynamically able to optimize the number of speculative copies/clones assigned for each straggler and enables the optimization of total PoCD of all active jobs on the cloud. After job arrivals, Shed+ waits for new jobs to progress $\xi\%$ then optimizes all active jobs with respect to the number of speculative/clones copies for each straggler, depending on current system load and the collective job progress with respect to deadlines. Moreover, Shed+ periodically checks for stragglers every θ seconds and readjusts the number of extra attempts for each straggler if it exists. Since this is a difficult integer programing problem, we tackle it effectively by using a simple, greedy heuristic which directs more cloud resources towards jobs that have a greater potential for utility improvement. Our proposed solution entails the development of an online algorithm that recomputes and re-allocates resources dynamically during the course of the task's execution to achieve PoCD maximization.

The solution is prototyped on Hadoop and assessed by utilizing a realistic workload in a local cluster and an Amazon EC2 testbed that consists of as many as 139 and 121 nodes, respectively. Shed+ is specifically implemented as a plug-in scheduler in Hadoop, and is evaluated with I/O- as well as CPU-bound benchmarks. Using extensive experiments, we compare the efficacy of our solution with a few popular straggler mitigation strategies – Shed [17], the default strategy of Hadoop with speculation [3], Dolly [6] and Hopper [19]. This work explicitly focuses on meeting application deadlines, while leaving other objectives such as energy for future consideration. The findings substantiate our assertion that the proposed dynamic speculation strategy leverages underutilized cloud resources to maximize the probability of jobs meeting their deadlines - marking a significant improvement. In our experiments, we have seen this improvement for light load going up to 100% for both Shed and Shed+ as opposed to nearly 85%, 42% and 43% for Dolly, Hopper and Hadoop with speculation enabled, respectively, while the percentage for heavy load going up to 98% for Shed+ as opposed to nearly 68%, 40%, 35% and 37% for Shed, Dolly, Hopper and Hadoop with speculation enabled, respectively.

The rest of this paper is organized as follows. Section II presents related work, and Section III presents the system model. The optimization framework is presented in Section IV, and the algorithm's implementation is described in Section V. Experimental results are presented in Section VI, and finally the paper is concluded in Section VII.

II. Background and System Model

In this work, we consider a parallel computation framework similar to MapReduce that splits analytical jobs into smaller tasks to be processed concurrently in multiple nodes. Each node is capable of executing one task at a time. The framework is comprised of map and reduce tasks; the map tasks' outputs get passed as input to the reduce tasks. This work considers a system with limited resource capacity of *m* VMs (Virtual Machines); and a fraction λ of this capacity is available for job allocation. Similar to [20]–[22], we narrow our focus to map-only jobs/tasks implemented in one wave in homogeneous nodes.

Let us consider a set of $\mathcal{J}$ jobs submitted to the MapReduce processing framework. Now each job *j* is associated with a deadline D_j that in turn is decided by application latency requirements. Each job *j* consists of a set of $\mathcal{N}_j$ tasks, and is deemed successful if all the $\mathcal{N}_j$ tasks are completed before the job deadline D_j. Let T_j denote job *j*'s completion time, and $T_{j,i}$ for $i = 1, \ldots, |\mathcal{N}_j|$ denote the (random) completion times of tasks belonging to job *j*. Based on our system model, job *j* meets its deadline if $T_j \leq D_j$, and its completion time is given by:

$$T_j = \max_{i=1,\ldots,|\mathcal{N}_j|} T_{j,i}, \ \forall j \tag{1}$$

Any task whose execution time is larger than the deadline is considered as a straggler. Our dynamic speculation approach mitigates the impact of stragglers by launching r_i extra attempts for each straggler. Therefore, each straggler includes r_i extra attempts/copies that begin execution simultaneously and process data independently of each other. A task gets completed when any one of the $r_i + 1$ attempts finishes the execution, and then the other copies are killed. As a result of the various sources of uncertainty causing stragglers, we model the completion time of attempt *k* (for $k = 1, \ldots, r_i + 1$) of task *i* and job *j* as a random variable $T_{j,i,k}$, using a known distribution. Hence, the completion time of task *i*, $T_{j,i}$, is determined by the completion time of the fastest attempt, i.e.,

$$T_{j,i} = \min_{k=1,\ldots,r_i+1} T_{j,i,k}, \ \forall i, j. \tag{2}$$

Execution times $T_{j,i,k}$ of different attempts are assumed to be independent because of resource virtulization, and $T_{j,i,k}$ follows a Pareto distribution, parameterized by $t_{\min}$ and $\beta_{j,i,k}$, where $t_{\min}$ denotes the minimum execution time and $\beta_{j,i,k}$ refers to the shape parameter, in accordance with current work characterizing task execution time in MapReduce [10], [11], [19], [23]. Unlike default Hadoop scheduling and similar to Hopper, our new dynamic speculation mechanism launches extra r_i copies for each straggler. After newly-arrived jobs progress $\xi\%$, all jobs within the system are notified to detect the straggling tasks; then these are jointly optimized to determine their optimal replication factors r_i for each straggler, under system capacity constraints. Moreover, the optimization continuously detects and re-adjusts the straggler cloning every θ seconds. This particular strategy avoids overwhelming the cluster with unnecessary extra copies (as in Dolly and Shed), while also ensuring that every straggler receives at least one attempt prior to the otherwise under-utilized system resources being apportioned among all active jobs/tasks.

III. Related Work

Much research has gone into improving the execution time of MapReduce-like systems to meet QoS (Quality of Service) [15], [17], [18], [24]–[50]. Some of the focus is directed on static resource provisioning, in order to meet a specific deadline in MapReduce; others present proposals concerning scaling resources as a response to resource demand, and cluster use to lower the total cost. Moreover, frameworks are also proposed to improve the performance of MapReduce jobs [15], [22], [51]–[53]. These are similar to our proposed work, owing to the need for optimizing resources to lower energy consumption as well as operating cost. However, these works, unlike our proposed approach, do not consider optimizing job execution time in the presence of stragglers. Meanwhile some studies have shown interest in augmenting the performance of MapReduce by postulating new scheduling techniques. These works range from deadline-aware scheduling [15], [16], [22], [54]–[59] to energy- and network-aware scheduling [60]–[66]. The proposed schedulers intend to improve both resource allocation and execution time whilst meeting the QoS. Meanwhile, other works [41], [67] aim to improve resource utilization whilst also adhering to completion time objectives. Our proposed scheduler also attempts to mitigate stragglers whilst maximizing cluster utilization through the use of a probabilistic approach.

The complexity of cloud computing is steadily on the rise, even as it is utilized for a gamut of fields. As a result, other researchers have shown interest in mitigating stragglers and augmenting the mechanism of default Hadoop speculation. They have proposed novel mechanisms to track stragglers, launch speculative tasks reactively and proactively [4]–[6], [10], [12], [13], [68]. Recent work [69] has derived expressions for a proactive approach to decide when and how much redundancy is given to jobs upon arrivals. Notably, reactive approaches generally create new copies of all slow-running tasks such as stragglers upon their detection, whereas proactive approaches launch multiple copies of each task at the start of task execution without waiting for the stragglers. These mechanisms are necessary to ensure a high reliability level in order to satisfactorily meet the QoS. Different from these works, we jointly maximize the probabilities of all jobs meeting their deadlines; it does so by intelligently optimizing the number of extra attempts given for each straggler on the basis of job size and progress.

IV. Joint PoCD and Resource Optimization

In this section, we use PoCD [17], [18], to quantify the probability of jobs meeting deadlines. We analyze PoCD for any given job *j* with $r_{j,i}$ extra attempts for each straggling task *i* and develop an online greedy algorithm to find the optimal $r_{j,i}$ for each straggler.

A. PoCD Analysis

We define PoCD as the probability that a job is completed prior to its deadline when launching r_i speculative copies for each straggling task *i*. For newly-arrived jobs (that are yet to start) and existing jobs (that may already have multiple attempts per straggler), their PoCDs are derived in Theorems 1 and 2, respectively. Recall that $T_{j,i,k}$, $t_{\min}$ and β denote the completion time of attempt *k* of task *i* and job *j*, the minimum execution time and the shape parameter, respectively.

Theorem 1: The PoCD of a newly-arrived job is given by

$$R_{\mathrm{su_j}} = \left[1 - \left(\frac{t_{\min}}{D_j}\right)^{\beta}\right]^{N_j}. \tag{3}$$

Proof: First, we find the probability that an attempt belonging to a newly submitted job will miss its deadline as follows:

$$P_{\mathrm{su_{j,i,k}}} = P\big(T_{j,i,k} > D_j\big) = \int_{D_j}^{\infty} \frac{\beta t_{\min}^{\beta}}{t^{\beta+1}}\, dt = \left(\frac{t_{\min}}{D_j}\right)^{\beta} \tag{4}$$

Note that for a newly submitted job there is only one attempt ($k = 1$) for every task *i*. Therefore, the probability that a job finishes before its deadline, PoCD, is given by

$$R_{\mathrm{su_j}} = \left[1 - \left(\frac{t_{\min}}{D_j}\right)^{\beta}\right]^{N_j} \tag{5}$$

where N_j is the number of tasks. ■

In our proposed dynamic speculation, during the process of speculating stragglers of a current job which may have already speculated and have multiple active attempts, we speculate the fastest attempt (i.e., the one which has made the greatest progress) in order to optimize our strategy's efficiency. The fastest attempt is only speculated if it happens to be a straggler at the decision point where we launch *r* extra copies. Otherwise, all extra attempts are killed, and the fastest attempt is kept running.

Let $\varphi_{j,i,k}$ denote the progress of attempt *k* in task *i* belonging to job *j*, and let $\varphi_{j,i}$ denote the largest progress (the percentage of data processed) of all task *i*'s attempts belonging to job *j*. That is, $\varphi_{j,i} = \max_{k=1,\ldots,r_j^i+1} \varphi_{j,i,k},\ \forall j, i$. Similarly, let $\beta_{j,i,k}$ denote the shape parameter of attempt *k* in task *i* belonging to job *j*, and let $\beta_{j,i}$ be the shape parameter of the fastest attempt. Let τ_j refer to the elapsed time of job *j*. We quantify the PoCD of an existing task as follows.

Theorem 2: The PoCD of an existing job

$$R_{\mathrm{ru_j}} = \prod_{i=1}^{|\mathcal{N}_j|} \left[1 - \left(\frac{\big(1-\varphi_{j,i}\big)t_{\min}}{D_j - \tau_j}\right)^{\beta_{j,i}\cdot\left(r_j^i+1\right)}\right]. \tag{6}$$

Proof: Let $\varphi_{j,i,k}$ be the progress of attempt *k* for task *i* belonging to job *j*. Recall that at the point of re-optimization, we keep the fastest attempt of each task for a running job, i.e., $\varphi_{j,i} = \max_k \varphi_{j,i,k}$.

Once the fastest attempt is determined, we launch *r* extra attempts for each straggling task; these extra attempts start executing from the last key-value pair processed by the fastest attempt. Therefore, the remaining execution time of each straggling task (denoted by $\hat{T}_{j,i,k}$) is i.i.d, and modeled as a Pareto distribution parameterized by $(1-\varphi_{j,i})t_{\min}$ and $\beta_{j,i}$, since $1-\varphi_{j,i,k}$ is the remaining fraction of data to be processed by attempt *k* belonging to task *i* of job *j*. Now, the probability that a running attempt will fail to finish before the deadline, can be computed as follows:

$$P_{\mathrm{ru_{j,i,k}}} = P\Big(\hat{T}_{j,i,k} > D_j - \tau_j\Big)$$
$$= \left(\frac{\big(1-\varphi_{j,i,k}\big)t_{\min}}{D_j - \tau_j}\right)^{\beta_{j,i,k}} \tag{7}$$

where $D_j - \tau_j$ is the remaining time before deadline, and $\hat{T}_{j,i,k}$ is the random remaining execution time of each attempt.

Therefore, the probability that a running job finishes before its deadline (denoted by $R_{\mathrm{ru_j}}$) is determined by the probability of all its tasks meeting the deadline, i.e.,

$$R_{\mathrm{ru_j}} = \prod_{i=1}^{|\mathcal{N}_j|} R_j\Big(r_j^i\Big) \tag{8}$$

where $R_j(r_j^i)$ is the probability of task *j* of job *i* meeting the deadline and is given by

$$R_j\Big(r_j^i\Big) = \left[1 - \left(\frac{\big(1-\varphi_{j,i}\big)t_{\min}}{D_j - \tau_j}\right)^{\beta_{j,i}\cdot\left(r_j^i+1\right)}\right]. \tag{9}$$

This is because for task *j* of job *i* to meet the deadline, it only requires one of its $r_j^i + 1$ attempts (i.e., r_j^i launched attempts plus one original attempt) to finish within $D_j - \tau_j$. For non-straggler jobs that do not require speculative execution at the time of re-optimization, it is easy to see that the PoCD remains the same as (8) and (9) by plugging in $r_j^i = 0$. ■

B. Joint PoCD Optimization

We first formulate the problem of joint PoCD maximization under system capacity constraints. Using dynamic speculation, each straggling task of job *j* has r_j^i+1 attempts, which includes an original attempt and r_j^i speculated attempts. Subsequently, the total number of VMs of job *j* is denoted by $\sum_i (r_j^i+1)+1$, where an extra VM is needed for running its job tracker/master. Recall that *m* is the total number of available VMs in the system, with λ being the fraction of these VMs permitted for task speculation. Hence, a cloud provider can balance the resource allocation and task execution by adjusting λ. A system capacity constraint $\sum_j \sum_i |\mathcal{N}_j|\cdot(r_j^i+1)+|\mathcal{J}| \le \lambda\cdot m$ needs to be satisfied at any given time. Let $R_j(r_j)$ (i.e., R_{su} for a newly-arrived job or R_{ru} for an existing job) denote the PoCD function for r_j^i extra attempts for task *i* in job *j*. Thus, we arrive at the following PoCD optimization:

$$\text{maximize} \quad \sum_{j=1}^{|\mathcal{J}|} U\big(p_j\big) \tag{10}$$

$$\text{s.t.} \quad \sum_{j=1}^{|\mathcal{J}|}\sum_{i=1}^{|\mathcal{N}_j|}\Big(r_j^i+1\Big) + |\mathcal{J}| \le \lambda \cdot m \tag{11}$$

$$p_j = \prod_{i=1}^{|\mathcal{N}_j|} R_j^i\Big(r_j^i\Big),\ \forall j \tag{12}$$

TABLE I
LIST OF SYMBOLS

Symbol	Description
$\mathcal{J}$	The set of all active jobs (submitted and running)
$\mathcal{J}_s$	The set of all jobs which contain stragglers
$\mathcal{N}_j$	The set of tasks for job j
$\mathcal{N}_j^s$	The set of straggling tasks for job j
ξ	Job progress of every submitted job
θ	Re-optimization interval
r_j^i	Number of speculative copies for task i in job j
λ	Fraction of VMs available for task execution and speculation
m	Total number of VMs available in the system
j'	Job with highest PoCD improvement
i'	Task with highest PoCD improvement
R_j^s	PoCD function for r_j^i extra attempts for task i for job j with stragglers
R_j^i	PoCD function for r_j^i extra attempts for task i for job j

$$r_j^i \geq 0, \ \forall j, \forall i \tag{13}$$

where p_j is the PoCD achieved by job j, and $R_j(r_j^i)$ is a PoCD function of a task i in job j that is monotonically increasing since larger r_j^i results in higher PoCD. The capacity constraint $\lambda \cdot m$ ensures that dynamic speculation can only utilize the fraction of cloud resources assigned for this purpose.

Here, $U(\cdot)$ refers to a utility function that guarantees our strategy's fairness. For instance, we can select a family of well-known α-fair utility functions that are parameterized by α [70]. Then, the solution for this PoCD optimization achieves a peak total PoCD (for $\alpha = 0$), proportional fairness (for $\alpha = 1$), or max-min fairness (for $\alpha = \infty$).

C. Our Proposed Algorithm

We present an online scheduling algorithm for solving the optimization problem in order to obtain the optimal r_j^i for each straggling task under cloud resource constraints. Upon job arrivals, the scheduler initially recalculates the remaining available resources for dynamic speculation and identifies all jobs along with their deadlines. The RM notifies all running jobs to check for stragglers, and for each running task, estimate $\beta_{j,i}$ to determine the likelihood of the task missing its deadline and becoming a straggler, i.e.,

$$\beta_{j,i} = \frac{t_{\text{rem}}}{t_{\text{rem}} - \left[\left(1 - \varphi_{j,i}\right) * t_{\min}\right]} \tag{14}$$

where t_{rem} is the expected remaining time, $t_{\min}$ is the minimum possible value of Pareto-distributed execution time, and $\varphi_{j,i}$ is the current task progress. It is easy to see that $\beta_{j,i} > 1$ and that a small value means the task will run longer [19] and is more likely to become a straggler. Then, our algorithm works in a greedy manner to assign VMs to stragglers with the highest utility improvement.

More precisely, we start by assigning $r_j^i = 0$ to each straggling task i in each job j with stragglers, and then calculate its utility R_j^s as a function of its PoCD. Let ω denote the total VMs assigned to all jobs and κ the available resources for all tasks. Iteratively, we identify the job j that has the minimum PoCD and then among all its stragglers $\mathcal{N}_j^s$, we find the straggler which has the minimum PoCD. We then increase the r_j^i of a straggler with minimum PoCD in job j by one. Steps 20-23 in the algorithm ensures that a straggling task i is removed once its assigned r_i reaches the maximum value, and a job j with straggling tasks is removed once all its tasks are processed by the algorithm. Then we update the utility function of every job j with respect to the current assignment of r_j^i. This process is repeated until the system capacity constraint (11) is reached or every straggler receives the maximum number of extra attempts.

D. Algorithm Complexity

To calculate the complexity of our algorithm, we need to find how often the else statement (line 15-23) can run in the worst case. In the worst case scenario, all tasks of all running jobs are straggling, and unlimited resources are available. Using an ordered array to represent $\mathcal{J}_s$ and $\mathcal{N}_j^s$, we get run times of $\mathcal{O}(1)$, $\mathcal{O}(1)$, $\mathcal{O}(n)$ for minimum operation (line 15 and 16), removing element (line 21 and 23), and updating the array (line 19), respectively. Taking all together with the while-loop gives a runtime of $\mathcal{O}(|\mathcal{J}_s|^2 * |\mathcal{N}_j^s|)$.

V. IMPLEMENTATION

We implement Shed+ as a pluggable scheduler in Hadoop YARN, which includes an RM (Resource Manager), an AM (Application Master) for each application (job) as well as an NM (Node Manager within each node). The AM issues a request to resource containers (VMs) to execute jobs/tasks and constantly tracks the progress of each task in NMs. The RM is responsible for monitoring as well as managing VMs within a cluster and scheduling jobs. More specifically, the scheduler optimizes and allocates resources to the requesting jobs. Figure 1 depicts our system architecture as well as the steps taken to attain optimality.

After a job submission, our scheduler waits for the job to progress $\xi\%$ before checking for stragglers. Then, the RM notifies each AM to detect stragglers. Once stragglers are found, the RM uses the job deadlines to calculate the optimal r for each straggler to maximize the utility as described in Algorithm 1. After r is obtained, it is sent by the RM to the corresponding AM to create r extra attempts (for each straggler). Subsequently, the AM negotiates the resources with the RM and then works in tandem with NMs to launch the attempts. The AM tracks the progress of all attempts and maintains the byte offset of the last processed record even as the submitted job continues to run. The checking process is repeated periodically every θ seconds or whenever a new job progresses $\xi\%$.

Since the AM monitors all running attempts, it responds to each probing from the RM by killing all the slow-running

Algorithm 1 Proposed Online Algorithm

Once a newly submitted job progresses $\xi\%$,
or θ seconds have passed since last check:
Kill all jobs that missed their deadlines
$\mathcal{J} = \{j_1, j_2, j_3, \dots\}$
for $j \in \mathcal{J}$ **do**
  Notify job j to check and estimate β_i for each straggler
end for
$r_j^i = 0 \ \forall j \in \mathcal{J}_s, \forall i \in \mathcal{N}_j^s$
$\omega = 0$
$\kappa = \lambda \cdot m - \sum_{j=1}^{|\mathcal{J}|} |\mathcal{N}_j| - |\mathcal{J}|$ \\No. of available VMs
Calculate $R_j^s \ \forall j \in \mathcal{J}_s$
while $\mathcal{J}_s \neq \{\emptyset\}$ **do**
  if $\omega + 1 > \kappa$ **then**
    break
  else
    $j' = \arg\min_j \{R_j^s\}$
    $i' = \arg\min_i \{R_{j'}^i\}$
    $r_{j'}^{i'} = r_{j'}^{i'} + 1$
    $\omega = \omega + 1$
    Calculate $R_j^s \ \forall j \in \mathcal{J}_s$
    if $r_{j'}^{i'} == MAX$ **then**
      $\mathcal{N}_{j'}^s = \mathcal{N}_{j'}^s - \{i'\}$
      if $\mathcal{N}_{j'}^s == \{\emptyset\}$ **then**
        $\mathcal{J}^s = \mathcal{J}^s - \{j'\}$
      end if
    end if
  end if
end while

attempts belonging to each task detected previously as a straggler, whereas the fastest one (i.e., that has processed the highest amount of data) is kept alive and continues to run. Notably, upon the arrival of a new job (or after θ seconds have passed since last optimization), our scheduler then re-optimizes resources and derives a new r_j^i for each straggling task, regardless of whether it is current or newly submitted. Thus, the AMs speculate/create new r_j^i copies for every single running attempt kept alive as the fastest attempt. To that end, we develop a new speculation mechanism that enables the preservation and transfer of existing task progress to speculative attempts. Specifically, the last known data offset processed by the straggling task gets passed on by the AM to new speculative attempts, which successfully continue the execution of tasks in a seamless manner. As a result, this approach significantly enhances the efficacy of dynamic speculation and in effect, the PoCD performance.

Figure 1 illustrates the manner in which our scheduler responds to new arrivals or after θ seconds have passed since the last optimization. Suppose that job 1 gets submitted to a cluster and is running. For the sake of simplicity, suppose that each job has only two tasks $A_{j,1}$ and $A_{j,2}$. Each task reports its progress to the AM including number of bytes processed. Thereafter, the AM reports the progress of the entire job to the RM. After $\xi\%$ of progress is achieved, the

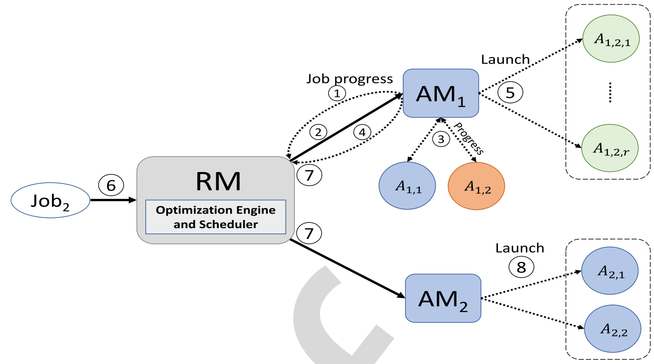


Fig. 1. System Architecture and steps taken upon new job arrival.

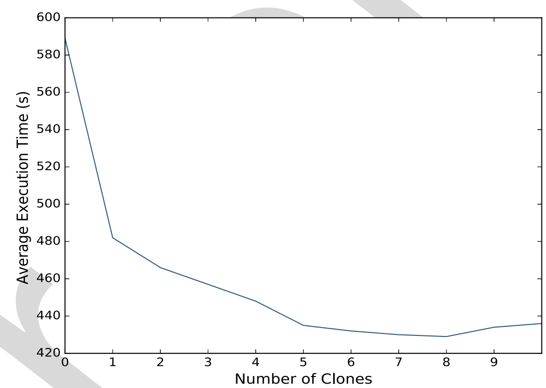


Fig. 2. Average job execution time versus number of clones.

RM notifies Job 1 to check for stragglers, in this case $A_{1,2}$, and report that to the RM. Then, the scheduler within the RM optimizes the cluster resources and derives r_1^2 for the straggling task $A_{1,2}$ on the basis of its PoCDs. After Job 1 obtains r_1^2, the AM speculates the straggling task, $A_{1,2}$, with r_1^2 extra attempts. Importantly, the total number of attempts for all the jobs is bounded by the available resources. That is, when RM calculates r_1^2 for the straggling task in Job 1, it considers all the running tasks including those in Job 2. This optimization process is repeated after Job 2 processes $\xi\%$ of the job or θ seconds elapse.

One challenge confronting us is that AMs must consider the time taken to launch new speculative attempts because in the case of clusters with high contention, the startup time of JVM (Java Virtual Machine) cannot be ignored [71]. Furthermore, the on-demand requests submitted to the RM cannot be predicted as they can arrive at any time. Therefore, all AMs take the JVM launching time into account when they pass the last offset processed to the new attempts [72]. The AMs specifically estimate the number of bytes b_{extra} to be processed by the speediest attempts. Although the last offset, b_{proc}, gets recorded upon the creation of new attempts, the AM will bypass the data processed during the time of launch, passing a new offset, b_{new}, to these new attempts. In case the AM finds that all the remaining bytes of data will be processed during the time of launch, all new attempts will be killed. This estimated number of bytes, b_{extra}, is obtained in the following manner:

$$b_{\text{extra}} = \frac{b_{\text{proc}}}{t_{\text{now}} - t_{\text{FP}}} \cdot (t_{\text{FP}} - t_{\text{lau}}) \tag{15}$$

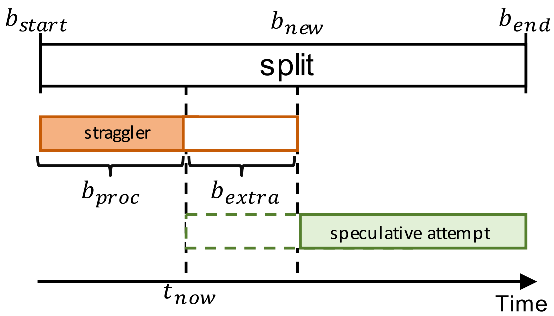


Fig. 3. Illustration of the byte offset estimation during speculative attempts' launching time. At t_{now}, the AM estimates the number of bytes (b_{extra}) to be processed during the speculative attempt's launching time. b_{new} is the start byte offset of the speculative attempt.

where t_{now} is the current time (i.e., time at which new attempts are about to be launched), t_{FP} is the time at which the first progress is reported for the original attempt, and t_{lau} is the amount of time to launch the original attempt (i.e., time elapsed from the instant the attempt is launched to the instant the attempt starts processing data). Here b_{proc} is the number of bytes processed by the original attempt until t_{now}. Thus, the new byte offset received by the new attempts is calculated as follows:

$$b_{\text{new}} = b_{\text{start}} + b_{\text{proc}} + b_{\text{extra}} \tag{16}$$

where b_{start} is the starting byte offset for the original attempt. Note that the straggling task keeps processing bytes until it reaches this new byte offset b_{new}. Figure 3 illustrates the byte offset estimation.

VI. Evaluation

The performance of our scheduler and algorithm are evaluated on a local cluster as well as Amazon EC2 cloud. In this section, we present the evaluation results. First, we give a description of the experimental setup, and then show our results comparing Shed+ with Shed, Hadoop with speculation enabled, Hopper, and Dolly.

A. Experimental Setup

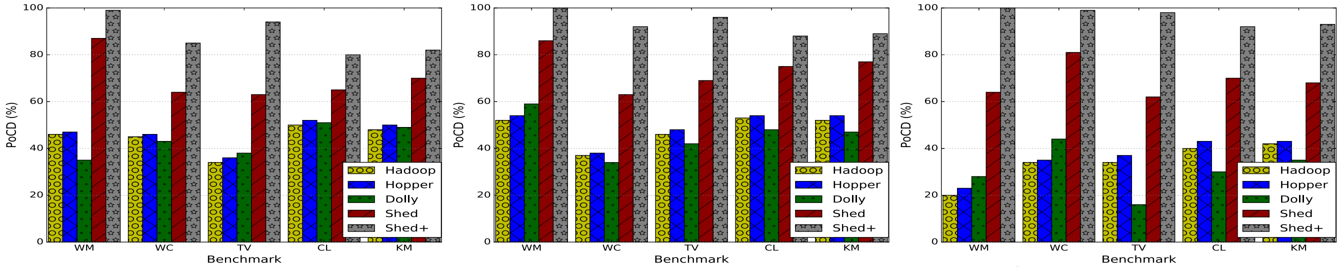
We deploy our proposed scheduler on a local cluster and Amazon EC2 consisting of 139 nodes – one master and 138 slaves. Because of the JVM launching time, we set the re-optimization interval to

$$\theta = \left(D - t_{avg}\right) \cdot \theta' + t_{avg}, \tag{17}$$

where t_{avg} is the average launching time overhead in the cluster, and θ' can be set from 0% to 100%. This is to ensure that the time between re-optimizations is never less than the launching time. t_{avg} is a cluster-specific variable, and it is obtained from Hadoop experiments. We set $\lambda = 100\%$, $\xi_j = 10\%$, $\theta' = 5\%$ and $t_{avg} = 60$ s. Each node is capable of running one task at a time. We evaluate our scheduler by using Map phases of three popular benchmarks, TermVector (TV), WordCount (WC), and WordMean (WM), as well as several Machine Learning benchmarks such as Classification (CL) and KMeans (KM) clustering benchmarks [73]. WordCount is an I/O and CPU-bound job while WordMean is CPU-bound. The ML benchmarks classify and cluster movies based on their ratings using anonymized movie ratings data. We assume that tasks of a job are executed in one wave in homogeneous nodes. We create three classes of jobs consisting of 5, 10, and 20 tasks [6]. We run 100 jobs for each experiment with varying job inter-arrival time. The baseline algorithms for comparison in our experiment are Shed, Hopper, Dolly, and Hadoop with speculation. Since Dolly does not consider deadlines, to make it comparable to our work, we set its straggler probability p equal to $1 - \mathit{PoCD}$, i.e., one minus the PoCD of default Hadoop, which is the probability of a job not meeting the deadline in Hadoop. Thus, Dolly assigns exactly $r + 1 = \log(1 - (1 - \epsilon)^{\frac{1}{N}})/\log p$ clones to each task for $\epsilon = 5\%$, regardless of their sizes and deadlines. ϵ is the acceptable risk of a job straggling. Unlike in Shed and Hopper, β is recalculated in Shed+ every time AM checks for stragglers and is task-dependent. We measure the PoCD of all strategies by calculating the percentage of jobs that completed before their deadlines. To emulate a realistic cloud cluster with resource contentions, we introduce background noise/tasks in each slave node, where noise shares resources with computation tasks. The task execution time measured in our cluster follows a Pareto distribution with an exponent $\beta \leq 2$ [10], [11], and $t_{\min} = 120$ sec. We choose deadlines relative to the median, $\tilde{x}$, of default Hadoop execution time, similar to the evaluations in [74]. Our proposed algorithm sets *r* for each straggling task equal to the *MAX* value. The value can be an environment-specific variable where there is not much improvement in execution time when *r* is large. To see what the maximum value of *r* is in our cluster, we run 10-task experiments with different values of *r*. Here, all attempts start at the same time [17]. We find that there is little improvement in execution time beyond *r* = 5, so we set the maximum number of attempts per straggling task to be 5. Figure 2 shows the average execution time with different values of *r*. Each point is the average of 100 runs. It can be seen that there is little improvement in execution time beyond $r = 5$, so we set the maximum number of attempts per task to be 5.

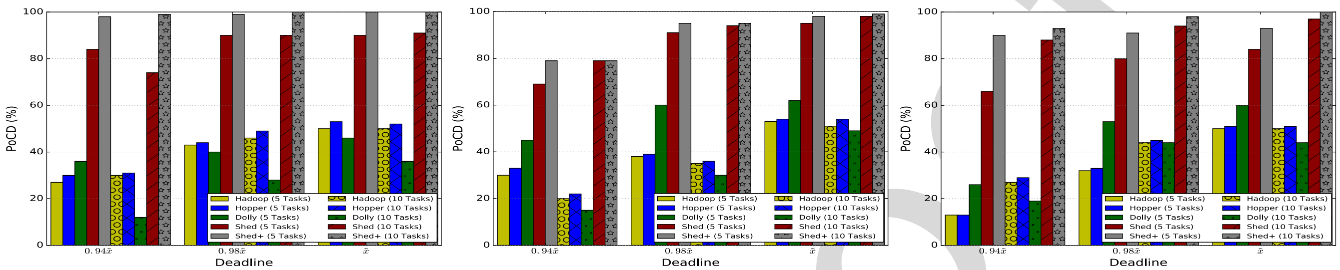
B. Results

Figure 4 compares the measured PoCD (percentage of jobs meeting deadline) of our proposed algorithm with Shed, Dolly, Hopper, and default Hadoop with speculation for various job sizes. In this figure we set the average load (each task with one copy) to 40%. We define the average load as the total number of running tasks to the total number of VMs. We tune the arrival rate to approximate the average load running. The figures show that Shed+ is able to achieve up to 100% PoCD, while Shed and Dolly are around 80% and 60%, respectively, in most experiments. The figure also shows that Shed+ can significantly outperform Shed due to the fact that Shed+ only speculates stragglers. This fine-grained speculation optimizes resources efficiently without the need to launch multiple copies for each task as in Shed. Moreover, Shed+ makes resources less contended which leads to faster processing. The performance difference increases for large jobs, i.e.,



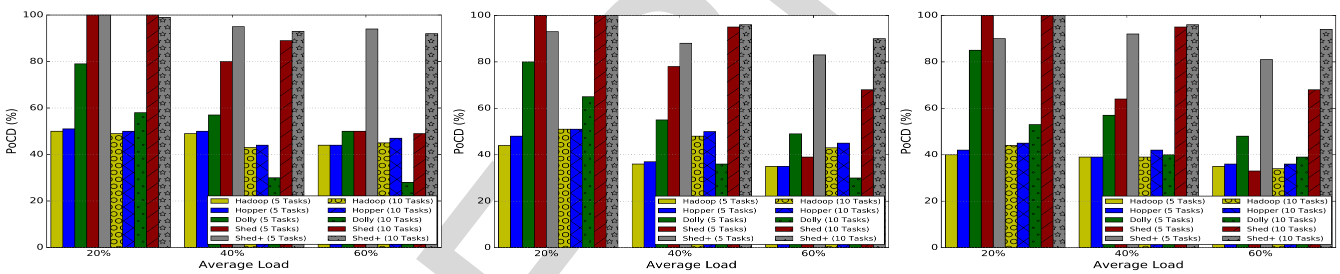
(a) (b) (c)

Fig. 4. Comparisons of Shed+, Shed, Hopper, Dolly and Hadoop in terms of PoCD with different benchmarks: (a) 5-task jobs (b) 10-task jobs (c) 20-task jobs.



(a) (b) (c)

Fig. 5. Comparisons of Shed+, Shed, Hopper, Dolly and Hadoop in terms of PoCD with a mix of workloads and benchmark combined with different deadlines: (a) WordMean (b) Classification (c) TermVector.



(a) (b) (c)

Fig. 6. Comparisons of Shed+, Shed, Hopper, Dolly and Hadoop in terms of PoCD with a mix of workloads and benchmark combined with different average load : (a) KMeans (b) WordCount (c) TermVector.

when the cloud utilization is extremely high, so there is no enough cloud resource to assign the number of clones needed to each job, making Shed less appealing. This demonstrates Shed+'s superiority in dealing with large jobs.

Figure 5 shows the PoCD of Shed+ compared with Shed, Hopper, Dolly, and Hadoop, for different deadlines. In this experiment, we run a mix of workloads and three benchmarks with various deadlines relative to the median. The figure presents each benchmark's separately. The results show that, even with mixed, heterogeneous workloads and deadlines, our algorithm achieves a PoCD of more than 100% in all cases (which is consistent with the homogeneous workload results), and significantly outperforms Dolly, Hopper, and Hadoop.

Moreover, the figures show that when job deadlines are relaxed, the PoCDs of all strategies increase, but Shed+ continues to perform significantly better than Hopper, Dolly, and Hadoop, demonstrating its superiority in dealing with hard application deadlines. Note that our numerical results compare Shed+, Shed, Hopper, Dolly, and Hadoop for various deadlines up to Hadoop's median job execution time, because Shed+ already achieves 100% PoCD in most cases due to more efficient utilization of system resources for running speculative copies needed for each straggler. This massive improvement over other strategies is also due to the fact that Shed+ periodically checks for stragglers and jointly optimizes available resources for speculative copies needed for all jobs with stragglers. Moreover, the new dynamic speculation mechanism guarantees that no repeated data processing is needed for any speculative attempts.

Figure 6 shows the PoCD of Shed+ compared with Shed, Hopper, Dolly, and Hadoop for different benchmarks for varying loads. Here, we fix the deadline and increase the average load (total running tasks) in the system. The figure shows that, as we increase the average load in the system, Shed+ continues to perform significantly better than all strategies. Shed+ is able to optimize resources and provide more VMs to the straggling tasks. The figure also shows that when the average load is low, both Shed and Shed+ perform relatively the same.

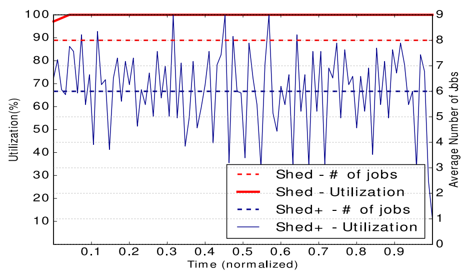


Fig. 7. Cluster utilization as a function of the decision variables calculated by the optimization model for Shed+ and Shed in addition to the average number of jobs being optimized.

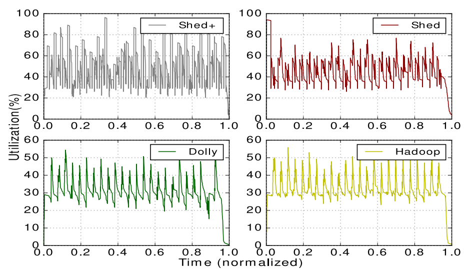


Fig. 8. Real time cluster utilization of Shed+, Shed, Dolly and Hadoop for 10-task jobs of WordCount benchmark.

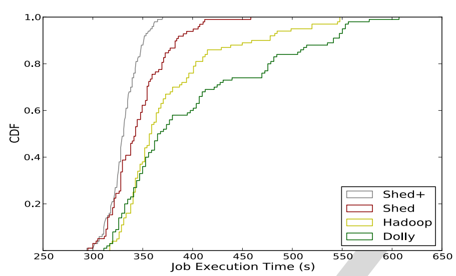


Fig. 9. The cumulative distribution function (CDF) of Shed+, Shed, Dolly and Hadoop for 10-task jobs of WordCount benchmark.

In Figure 7, we compare Shed and Shed+ in terms of the optimization model decision for 10-task WordCount jobs in a system with 40% average load. The figure depicts the system utilization as a function of total number of attempts/copies (including original ones) for all tasks, where each copy requires one VM. The figure shows that Shed always tries to fully utilize the cluster. However, this does not guarantee that all jobs receive the number of copies needed for each task. On the other hand, Shed+ is able to fully utilize the cluster when needed. This means that non-straggling tasks only run with one copy leaving remaining resources for straggling tasks. In other words, any extra attempt is given only to stragglers.

In addition, Figure 7 shows the average number of active jobs in the system being optimized. It can be clearly seen that Shed+ is able to improve job execution times and reduce resource competition for new arrivals to the system compared with Shed.

From the same experiment in Figure 7, Figure 8 depicts the cluster utilization under Shed+, Shed, Dolly and Hadoop in real time. It can be clearly seen that while Shed can only achieve about 75% utilization, Shed+ is able to optimize

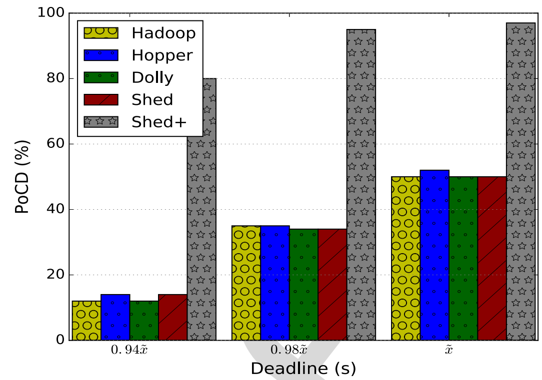


Fig. 10. Comparisons of Shed+, Shed, Hopper, Dolly and Hadoop in terms of PoCD with 20-job workload using WordMean benchmark and running in EC2.

the underutilized resources for stragglers and achieve much higher levels of utilization and fairness, where only stragglers receive more resources. The figure also shows how Shed+ exploits idle slots in order to mitigate the effect of stragglers and achieve better performance in meeting job deadlines. Similar results are evident for different workloads. On the other hand, Hadoop, Dolly and Hopper (not shown) are only able to achieve around 55% utilization. Hopper shows similar utilization to Hadoop.

Figure 9 shows the cumulative distribution function (CDF) of job execution times for the same experiment above. Notice that almost all jobs complete within 350 s under Shed+ whereas only 60%, 40%, and 40% of the jobs complete by 350 s under Shed, Dolly, and Hadoop, respectively, and it takes as much as 600 s and 550 s for some jobs to complete under Dolly and Hadoop, respectively. The average job execution time (not shown in the figure) for Shed+, Shed, Dolly, and Hadoop are 330 s, 345 s, 402 s, and 378 s, respectively.

Figure 10 and Figure 11 depict results from experiments on EC2. The figures show the PoCD of Shed+ compared with Shed, Hopper, Dolly and Hadoop with different deadlines. In these experiments, we increase the job size and arrival rate for WordMean and WordCount benchmarks. The figures show that Shed+ notably outperforms all baselines and is able to achieve nearly 100% PoCD. The figures also show that even Shed falls behind in meeting job deadlines. That is, with high arrival rate, Shed is not able to provide enough resources for the jobs in need, which makes its performance similar to Dolly, Hopper, and Hadoop. On the other hand, Shed+'s outstanding performance is due to the fact that only stragglers receive more resources according to their PoCDs.

Finally, in order to explore the potential tradeoffs between overhead due to frequent reoptimization and PoCD, we study the effects of algorithm parameter ξ_j and the re-optimization interval θ. Recall from equation (17) that θ was defined such that it is never less than the JVM launching time. In order to explore the tradeoff for the full range of re-optimization interval θ, we redefine θ for this experiment as $\theta = \theta' \cdot D$. Thus, for instance, $\theta' = 5\%$ means that θ is 5% of job deadline. Table II shows the PoCD of Shed+ for 10-task WordMean jobs with different values of ξ_j and θ. The results show that Shed+ is able to achieve large PoCDs for a wide range of parameters.

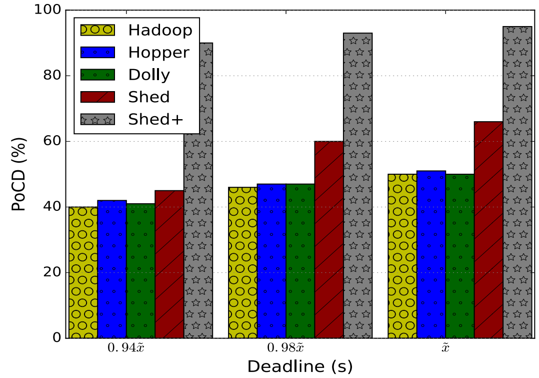


Fig. 11. Comparisons of Shed+, Shed, Hopper, Dolly and Hadoop in terms of PoCD with 20-job workload using WordCount benchmark and running in EC2.

TABLE II
POCDS FOR DIFFERENT ξ AND θ VALUES

ξ \ θ	5%	7%	10%	15%	20%	25%	30%	40%
10%	46	72	90	100	96	90	89	75
20%	42	67	85	100	94	89	81	74
30%	39	66	82	98	90	88	79	72

For instance, we can wait up to $\xi_j = 30\%$ of progress and optimize less often, and still achieve large PoCD. However, the results show a clear penalty of too frequent re-optimization when θ is 5–10% of the deadline. It can be seen that both very small and very large θ values lead to degraded PoCD. More frequent re-optimization (i.e., small θ) leads to killing replica attempts before getting the chance to start processing due to launching time overhead. This makes the PoCD results similar to Hadoop. On the other hand, very large θ values lead to too infrequent re-optimizations that may not respond to system dynamics quickly and lead to lower PoCD performance. This experiment provides valuable insights on how to exploit the tradeoff in practical systems. Note that our approach of setting θ according to equation (17) does not lead to the problem of re-optimizing too frequently.

VII. CONCLUSION

In this paper, we propose Shed+, a fine-grained optimization framework that leverages dynamic speculation to jointly maximize PoCD and cluster utilization. We also present an online scheduler that dynamically optimizes resources periodically. Our solution includes an online greedy algorithm to find the optimal number of speculative copies needed for each straggler. Our results show that Shed+ can achieve up to 100% PoCD compared to Shed, Dolly, Hopper, and Hadoop with speculation. The proposed algorithm is able to achieve more than 90% utilization of available cloud resources when needed, whereas Shed achieves 80%, but it is less efficient. Dolly, Hopper, and Hadoop achieve only about 55%.

In our future work, we will extend our work to consider energy utilization and energy efficiency in the joint optimization problem. Another extension would be to consider other architectures such as CPUs vs GPUs, and shared VMs vs dedicated VMs with different price units. In addition, we plan to investigate deadline-aware scheduling algorithms for multi-phase cloud systems, e.g., MapReduce, which involve communication and dependency among tasks. Moreover, we will expand our work to consider multi-cluster and geo-distributed environments. We plan to include heterogeneity in the network performance (including latency and bandwidth) into our model. Furthermore, modeling replication and related overhead will be considered for an online setting with dynamic job arrivals and departures.

REFERENCES

[1] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proc. EuroSys*, 2007, pp. 1–13.
[2] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[3] Apache Software Foundation. (Oct. 2019). *Hadoop*. [Online]. Available: https://hadoop.apache.org
[4] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *Proc. OSDI*, 2008, pp. 29–42.
[5] G. Ananthanarayanan *et al.*, "Reining in the outliers in map-reduce clusters using mantri," in *Proc. OSDI*, 2010, pp. 265–278.
[6] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones," in *Proc. NSDI*, 2013, pp. 185–198.
[7] V. Aggarwal, J. Fan, and T. Lan, "Taming tail latency for erasure-coded, distributed storage systems," in *Proc. INFOCOM*, 2017, pp. 1–9.
[8] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making sense of performance in data analytics frameworks," in *Proc. NSDI*, 2015, pp. 293–307.
[9] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: Measurements & analysis," in *Proc. SIGCOMM*, 2009, pp. 202–208.
[10] H. Xu and W. C. Lau, "Optimization for speculative execution in big data processing clusters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 2, pp. 530–545, Feb. 2017.
[11] D. Wang, G. Joshi, and G. Wornell, "Using straggler replication to reduce latency in large-scale parallel computing," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 43, no. 3, pp. 7–11, 2015.
[12] N. J. Yadwadkar, G. Ananthanarayanan, and R. Katz, "Wrangler: Predictable and faster jobs using fewer resources," in *Proc. SOCC*, 2014, pp. 1–14.
[13] S. Ibrahim, H. Jin, L. Lu, B. He, G. Antoniu, and S. Wu, "Maestro: Replica-aware map scheduling for mapreduce," in *Proc. CCGrid*, 2012, pp. 435–442.
[14] J. Rosen and B. Zhao, "Fine-grained micro-tasks for mapreduce skew-handling," Berkeley, CA, USA, Univ. Berkeley, White Paper, 2012.
[15] D. Cheng, J. Rao, C. Jiang, and X. Zhou, "Resource and deadline-aware job scheduling in dynamic hadoop clusters," in *Proc. IPDPS*, 2015, pp. 235–244.
[16] D. Li, C. Chen, J. Guan, Y. Zhang, J. Zhu, and R. Yu, "DCloud: Deadline-aware resource allocation for cloud computing jobs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 8, pp. 2248–2260, Aug. 2016.
[17] S. Alamro, M. Xu, T. Lan, and S. Subramaniam, "Shed: Optimal dynamic cloning to meet application deadlines in cloud," in *Proc. IEEE Int. Conf. Commun. (ICC)*, 2018, pp. 1–7.
[18] M. Xu, S. Alamro, T. Lan, and S. Subramaniam, "Chronos: A unifying optimization framework for speculative execution of deadline-critical mapreduce jobs," in *Proc. IEEE 38th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, 2018, pp. 718–729.
[19] X. Ren, G. Ananthanarayanan, A. Wierman, and M. Yu, "Hopper: Decentralized speculation-aware cluster scheduling at scale," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 379–392, 2015.
[20] M. Elteir, H. Lin, W.-C. Feng, and T. Scogland, "Streammr: An optimized mapreduce framework for AMD GPUs," in *Proc. ICPADS*, 2011, pp. 364–371.
[21] M. H. Almeer, "Cloud hadoop map reduce for remote sensing image analysis," *J. Emerg. Trends Comput. Inf. Sci.*, vol. 3, no. 4, pp. 637–644, 2012.

[22] S. Alamro, M. Xu, T. Lan, and S. Subramaniam, "CRED: Cloud right-sizing to meet execution deadlines and data locality," in *Proc. CLOUD*, 2016, pp. 686–693.

[23] M. Xu, S. Alamro, T. Lan, and S. Subramaniam, "Laser: A deep learning approach for speculative execution and replication of deadline-critical jobs in cloud," in *Proc. 26th Int. Conf. IEEE Comput. Commun. Netw. (ICCCN)*, 2017, pp. 1–8.

[24] Y. Ge, Z. Ding, M. Tang, and Y. Tian, "Resource provisioning for mapreduce computation in cloud container environment," in *Proc. IEEE 18th Int. Symp. Netw. Comput. Appl. (NCA)*, 2019, pp. 1–4.

[25] X. Ouyang, C. Wang, and J. Xu, "Mitigating stragglers to avoid QoS violation for time-critical applications through dynamic server blacklisting," *Future Gener. Comput. Syst.*, vol. 101, pp. 831–842, Dec. 2019.

[26] X. Xu, M. Tang, and Y.-C. Tian, "Theoretical results of QoS-guaranteed resource scaling for cloud-based mapreduce," *IEEE Trans. Cloud Comput.*, vol. 6, no. 3, pp. 879–889, Jul.–Sep. 2016.

[27] A. Verma, L. Cherkasova, and R. H. Campbell, "ARIA: Automatic resource inference and allocation for mapreduce environments," in *Proc. ICAC*, 2011, pp. 1–9.

[28] P. Lama and X. Zhou, "AROMA: Automated resource allocation and configuration of mapreduce environment in the cloud," in *Proc. ICAC*, 2012, pp. 63–72.

[29] K. Chen, J. Powers, S. Guo, and F. Tian, "CRESP: Towards optimal resource provisioning for mapreduce computing in public clouds," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 6, pp. 1403–1412, Jun. 2014.

[30] M. Malekimajd, A. M. Rizzi, D. Ardagna, M. Ciavotta, M. Passacantando, and A. Movaghar, "Optimal capacity allocation for executing mapreduce jobs in cloud systems," in *Proc. 16th Int. Symp. IEEE Symbolic Numer. Algorithms Sci. Comput. (SYNASC)*, 2014, pp. 385–392.

[31] H. Herodotou, F. Dong, and S. Babu, "No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics," in *Proc. 2nd ACM Symp. Cloud Comput.*, 2011, p. 18.

[32] E. Hwang and K. H. Kim, "Minimizing cost of virtual machines for deadline-constrained mapreduce applications in the cloud," in *Proc. ACM/IEEE 13th Int. Conf. Grid Comput.*, 2012, pp. 130–138.

[33] K. Kc and K. Anyanwu, "Scheduling hadoop jobs to meet deadlines," in *Proc. IEEE 2nd Int. Conf. Cloud Comput. Technol. Sci. (CloudCom)*, 2010, pp. 388–392.

[34] J. Polo, D. Carrera, Y. Becerra, M. Steinder, and I. Whalley, "Performance-driven task co-scheduling for mapreduce environments," in *Proc. IEEE Netw. Oper. Manag. Symp. (NOMS)*, 2010, pp. 373–380.

[35] W. Shi and B. Hong, "Clotho: An elastic mapreduce workload/runtime co-design," in *Proc. 12th Int. Workshop Adapt. Reflect. Middleware*, 2013, p. 5.

[36] W. Zhang, S. Rajasekaran, T. Wood, and M. Zhu, "MIMP: Deadline and interference aware scheduling of hadoop virtual machines," in *Proc. 14th IEEE/ACM Int. Symp. Clust. Cloud Grid Comput. (CCGrid)*, 2014, pp. 394–403.

[37] M. Abdelbaky, H. Kim, I. Rodero, and M. Parashar, "Accelerating mapreduce analytics using cometcloud," in *Proc. IEEE 5th Int. Conf. Cloud Comput. (CLOUD)*, 2012, pp. 447–454.

[38] M. Mattess, R. N. Calheiros, and R. Buyya, "Scaling mapreduce applications across hybrid clouds to meet soft deadlines," in *Proc. IEEE 27th Int. Conf. Adv. Inf. Netw. Appl. (AINA)*, 2013, pp. 629–636.

[39] M. Cardosa, P. Narang, A. Chandra, H. Pucha, and A. Singh, "Steamengine: Driving mapreduce provisioning in the cloud," in *Proc. IEEE 18th Int. Conf. High Perform. Comput.*, 2011, pp. 1–10.

[40] B. T. Rao and L. Reddy, "Scheduling data intensive workloads through virtualization on mapreduce based clouds," *Int. J. Comput. Sci. Netw. Security (IJCSNS)*, vol. 13, no. 6, p. 105, 2013.

[41] J. Polo *et al.*, "Resource-aware adaptive scheduling for mapreduce clusters," in *Proc. Middleware*, 2011, pp. 187–207.

[42] H. Li, X. Wei, Q. Fu, and Y. Luo, "Mapreduce delay scheduling with deadline constraint," *Concurrency Comput. Pract. Exp.*, vol. 26, no. 3, pp. 766–778, 2014.

[43] 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 *Proc. IEEE 34th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, 2014, pp. 93–103.

[44] Z. Tang, J. Zhou, K. Li, and R. Li, "A mapreduce task scheduling algorithm for deadline constraints," *Clust. Comput.*, vol. 16, no. 4, pp. 651–662, 2013.

[45] J. Wang, Q. Li, and Y. Shi, "Slo-driven task scheduling in mapreduce environments," in *Proc. 10th IEEE Web Inf. Syst. Appl. Conf. (WISA)*, 2013, pp. 308–313.

[46] C.-H. Chen, J.-W. Lin, and S.-Y. Kuo, "Mapreduce scheduling for deadline-constrained jobs in heterogeneous cloud computing systems," *IEEE Trans. Cloud Comput.*, vol. 6, no. 1, pp. 127–140, Jan.–Mar. 2018.

[47] Z. Liu, Q. Zhang, M. F. Zhani, R. Boutaba, Y. Liu, and Z. Gong, "Dreams: Dynamic resource allocation for mapreduce with data skew," in *Proc. IFIP/IEEE Int. Symp. Integr. Netw. Manag. (IM)*, 2015, pp. 18–26.

[48] B. Palanisamy, A. Singh, and L. Liu, "Cost-effective resource provisioning for mapreduce in a cloud," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 5, pp. 1265–1279, May 2015.

[49] N. Lim, S. Majumdar, and P. Ashwood-Smith, "MRCP-RM: A technique for resource allocation and scheduling of mapreduce jobs with deadlines," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 5, pp. 1375–1389, May 2017.

[50] L. Gu, D. Zeng, P. Li, and S. Guo, "Cost minimization for big data processing in geo-distributed data centers," *IEEE Trans. Emerg. Topics Comput.*, vol. 2, no. 3, pp. 314–323, Sep. 2014.

[51] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. Vijaykumar, "Tarazu: optimizing mapreduce on heterogeneous clusters," in *Proc. ACM SIGARCH Comput. Archit. News*, vol. 40, no. 1, 2012, pp. 61–74.

[52] B. Heintz, A. Chandra, and J. Weissman, "Cross-phase optimization in mapreduce," in *Proc. Cloud Comput. Data Intensive Appl.*, 2014, pp. 277–302.

[53] Y. Ying, R. Birke, C. Wang, L. Y. Chen, and N. Gautam, "Optimizing energy, locality and priority in a mapreduce cluster," in *Proc. IEEE Int. Conf. Auton. Comput. (ICAC)*, 2015, pp. 21–30.

[54] S. Kaur and P. Saini, "Deadline-aware mapreduce scheduling with selective speculative execution," in *Proc. Int. Conf. Comput. Commun. Netw. Technol. (ICCCNT)*, 2017, pp. 6–9.

[55] M. L. Della Vedova, D. Tessera, and M. C. Calzarossa, "Probabilistic provisioning and scheduling in uncertain cloud environments," in *Proc. IEEE Symp. Comput. Commun. (ISCC)*, 2016, pp. 797–803.

[56] F. Teng, H. Yang, T. Li, F. Magoulès, and X. Fan, "MUS: A novel deadline-constrained scheduling algorithm for hadoop," *Int. J. Comput. Sci. Eng.*, vol. 11, no. 4, pp. 360–367, 2015.

[57] Z. Tang, L. Jiang, J. Zhou, K. Li, and K. Li, "A self-adaptive scheduling algorithm for reduce start time," *Future Gener. Comput. Syst.*, vols. 43–44, pp. 51–60, Feb. 2015.

[58] S. Li, S. Hu, and T. Abdelzaher, "The packing server for real-time scheduling of mapreduce workflows," in *Proc. 21st IEEE Real Time Embedded Technol. Appl. Symp.*, 2015, pp. 51–62.

[59] K. Bok, J. Hwang, J. Lim, Y. Kim, and J. Yoo, "An efficient mapreduce scheduling scheme for processing large multimedia data," *Multimedia Tools Appl.*, vol. 76, pp. 1–24, Oct. 2016.

[60] A. C. Zhou, T.-D. Phan, S. Ibrahim, and B. He, "Energy-efficient speculative execution using advanced reservation for heterogeneous clusters," in *Proc. 47th Int. Conf. Parallel Process.*, 2018, pp. 1–10.

[61] P. Li, L. Ju, Z. Jia, and Z. Sun, "SLA-aware energy-efficient scheduling scheme for hadoop yarn," in *Proc. IEEE 7th Int. Symp. High Perform. Comput. Commun. (HPCC) IEEE 12th Cyberspace Safety Security (CSS) IEEE 17th Int. Conf. Int. Conf. Embedded Softw. Syst. (ICESS)*, 2015, pp. 623–628.

[62] J. Wang, X. Li, and J. Yang, "Energy-aware task scheduling of mapreduce cluster," in *Proc. IEEE Int. Conf. Service Sci. (ICSS)*, 2015, pp. 187–194.

[63] A. Gregory and S. Majumdar, "A constraint programming based energy aware resource management middleware for clouds processing mapreduce jobs with deadlines," in *Proc. Companion Pub. ACM/SPEC Int. Conf. Perform. Eng.*, 2016, pp. 15–20.

[64] L. Mashayekhy, M. M. Nejad, D. Grosu, Q. Zhang, and W. Shi, "Energy-aware scheduling of mapreduce jobs for big data applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 10, pp. 2720–2733, Oct. 2015.

[65] J. Wang and X. Li, "Task scheduling for mapreduce in heterogeneous networks," *Clust. Comput.*, vol. 19, no. 1, pp. 197–210, 2016.

[66] J. Wolf *et al.*, "FLEX: A slot allocation scheduling optimizer for mapreduce workloads," in *Proc. Middleware*, 2010, pp. 1–20.

[67] Z. Guo and G. Fox, "Improving mapreduce performance in heterogeneous network environments and resource utilization," in *Proc. CCGrid*, 2012, pp. 714–716.

[68] M. F. Aktas, P. Peng, and E. Soljanin, "Straggler mitigation by delayed relaunch of tasks," *SIGMETRICS Perform. Eval. Rev.*, vol. 45, no. 3, pp. 1–52, 2018.

[69] M. F. Aktas and E. Soljanin, "Optimizing redundancy levels in master-worker compute clusters for straggler mitigation," 2019. [Online]. Available: https://arxiv.org/abs/1906.05345

[70] T. Lan, D. Kao, M. Chiang, and A. Sabharwal, “An axiomatic theory of fairness in network resource allocation,” in *Proc. INFOCOM*, 2010, pp. 1343–1351.

[71] M. Mao and M. Humphrey, “A performance study on the vm startup time in the cloud,” in *Proc. CLOUD*, 2012, pp. 423–430.

[72] M. Xu, S. Alamro, T. Lan, and S. Subramaniam, “Optimizing speculative execution of deadline-sensitive jobs in cloud,” in *Proc. SIGMETRICS*, 2017, pp. 17–18.

[73] Apache Software Foundation. (Nov. 2019). *PUMA: Purdue Maprduce Benchmark Suite*. [Online]. Available: https://engineering.purdue.edu/∼puma/pumabenchmarks.htm

[74] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu, “GRASS: Trimming stragglers in approximation analytics,” in *Proc. NSDI*, 2014, pp. 289–302.

Sultan Alamro (Student Member, IEEE) received the B.S. degree in electronics and communications engineering from Qassim University, Saudi Arabia, in 2008, and the M.S. degree in electrical engineering from the Polytechnic Institute of New York University, Brooklyn, NY, USA, in 2013. He is currently pursuing the Ph.D. degree in electrical engineering with the George Washington University. He is with the Department of Electrical Engineering, College of Engineering, Qassim University, Buraidah, Saudi Arabia. His research interests include cloud computing and resource optimization.

Maotong Xu (Student Member, IEEE) received the B.S. degree from Northwestern Polytechnical University, China, in 2012, and the M.S. and Ph.D. degrees from George Washington University in 2014 and 2019, respectively. He is currently a Research Scientist with Facebook. His research interests include cloud computing and data center networks.

Tian Lan (Member, IEEE) received the B.A.Sc. degree from Tsinghua University, China, in 2003, the M.A.Sc. degree from the University of Toronto, Canada, in 2005, and the Ph.D. degree from Princeton University in 2010. He is currently an Associate Professor of electrical and computer engineering with George Washington University. His research interests include network optimization and algorithms, cyber security, cloud, and edge computing. He received the 2008 IEEE Signal Processing Society Best Paper Award, the 2009 IEEE GLOBECOM Best Paper Award, and the 2012 INFOCOM Best Paper Award.

Suresh Subramaniam (Fellow, IEEE) received the Ph.D. degree in electrical engineering from the University of Washington, Seattle, in 1997.

He is a Professor and the Chair of electrical and computer engineering with George Washington University, Washington, DC, USA, where he directs the Lab for Intelligent Networking and Computing. His research interests are in the architectural, algorithmic, and performance aspects of communication networks, with current emphasis on optical networks, cloud computing, data center networks, and IoT. He has published over 200 peer- reviewed papers in these areas.

Prof. Subramaniam received the 2017 SEAS Distinguished Researcher Award from George Washington University. He is a co-editor of three books on optical networking. He has served in leadership positions for several top conferences, including IEEE ComSoc’s flagship conferences of ICC, Globecom, and INFOCOM. He serves on the editorial boards for the IEEE/ACM TRANSACTIONS ON NETWORKING and the IEEE/OSA JOURNAL OF OPTICAL COMMUNICATIONS AND NETWORKING. From 2012 and 2013, he served as the Elected Chair for the IEEE Communications Society Optical Networking Technical Committee. He has been an IEEE Distinguished Lecturer since 2018.