# Shed: Optimal Dynamic Cloning to Meet Application Deadlines in Cloud

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

***Abstract*—As cloud applications are becoming increasingly deadline-sensitive, meeting desired deadlines is more critical, especially in shared clusters. It has been shown that a few slow tasks, called stragglers, could significantly adversely impact job execution times. Moreover, poor scheduling of data analytics applications can lead to inefficient resource usage, and eventually hurt system performance. One way to mitigate stragglers is by launching extra attempts (clones) for each task upon job submission. In this paper, we propose Shed, an optimization framework that leverages dynamic cloning to jointly maximize jobs' Probability of Completion before Deadline (PoCD) by fully utilizing the available resources. Our work includes a novel online scheduler that dynamically recomputes and reallocates resources during a job's execution for PoCD maximization. The results show that Shed is able to leverage cloud resources and maximize the percentage of jobs that meet their deadlines – up to $100\%$ in our experiments compared to typically around $60\%$ and $40\%$ for another cloning approach called Dolly, and Hadoop with speculation enabled, respectively.**

## I. INTRODUCTION

Modern applications such as enterprise IT, financial services, social networks, and data analytics are increasingly dependent on distributed cloud computing frameworks, such as MapReduce [1], to meet critical performance objectives. Massive amounts of data are split into blocks and stored distributedly in an underlying file system, to support parallel processing of computation jobs across clusters and nodes in the cloud. For instance, Hadoop, a popular open-source software framework for analyzing big data [2], uses the MapReduce programming model.

However, the performance of these parallel cloud processing frameworks when processing data are often negatively affected by *stragglers*, i.e., slow running tasks that lead to a long job execution time, which make them unsuitable to latency-sensitive applications that requires job completion time guarantees. Previous research has shown that stragglers can be 8 times slower than the median task [3–5]. Thus, just a few stragglers could have a large impact on the overall performance of job completion times and result in the violation of Service Level Agreements (SLAs). There are a number of reasons that lead to stragglers in cloud. First, the nodes in cloud are mostly made up of commodity components, with varying degrees of heterogeneity. This causes nodes to process tasks at varying speed. Second, the large-scale nature of datacenters is associated with errors both in software and hardware, leading to failures of machines and interruptions of task execution. Third, resource sharing and virtualization mandates co-scheduling of tasks, which especially if done on the same machines, result in resource interference and stragglers [4, 6]. Previous research also shows that congested links in datacenter networks can last for very long periods, and is another cause of stragglers [7].

In this paper, we propose Shed, an optimization framework that leverages *dynamic cloning* to maximize the probability of meeting individual job deadlines. Recently, reactive and proactive approaches to mitigating the effect of stragglers have been suggested by researchers. Reactive strategies are intended to detect stragglers after they occur and then launch extra or speculative copies of slow tasks [3, 4], while proactive strategies launch clones which are replicas of the original tasks [5]. They avoid waiting for stragglers to be detected, and launch speculative tasks preemptively. Our dynamic cloning approach differs from these solutions in the sense that it considers individual job deadlines and optimizes the number of task clones (and therefore, available cloud resources) assigned to each job, to jointly maximize the total probability of meeting job deadlines. Moreover, the number of task clones is adjusted on the fly with respect to both cloud load and current job progress. We develop and implement a new clone launching mechanism that preserves the work already completed by the original task before cloning, to enable seamless task execution and progress transfer to task clones. The proposed framework allows us to jointly maximize all jobs' probability to meet deadlines by optimizing and leveraging the underutilized cloud resources.

The ability to meet deadlines is crucial for latency-sensitive and mission-critical applications [8]. However, none of the existing techniques (both reactive and proactive) can provide formal performance guarantees in terms of meeting application deadlines, since these approaches are deadline-oblivious. Not only does our dynamic cloning approach react to concurrent job demands and progress on the fly, it also presents a novel scheduler that employs a new metric, Probability of Completion before Deadlines (PoCD), to quantify the probability that a MapReduce job meets its desired deadline. By analyzing PoCD based on cloud processing models, we formulate an optimization problem to jointly maximize the total PoCD of all active jobs, by determining the optimal number of clones (i.e., extra attempts) with respect to job progress and cloud resource constraints. Unlike Dolly [5] that relies on a fixed number of clones for each job/task, Shed dynamically optimizes the number of clones assigned to each job and enables an optimization of total PoCD in the cloud. Upon job arrivals and departures, Shed readjusts the number of clones, depending on current system load and the collective job progress with respect to deadlines.

The proposed solution is prototyped on Hadoop and evaluated using realistic workload in an Amazon EC2 testbed consisting of 121 nodes. In particular, Shed is implemented as a plug-in scheduler in Hadoop, and it is evaluated with both I/O- and CPU-bound benchmarks. We compare our solution through intensive experiments with Dolly and Hadoop's default strategy for speculative execution. In this work, we focus on meeting application deadlines, while other objectives such as energy are left for future consideration. The results validate that our proposed dynamic cloning strategy is able to leverage underutilized cloud resources and to maximize the percentage of jobs meeting their deadlines – improving from 60% and 40% under Dolly and Hadoop's default speculative and straggler mitigation strategy, respectively, to up to 100% under our optimized dynamic cloning strategy.

## II. Related Work

A lot of research has gone into improving MapReduce-like systems' execution time, in order to guarantee that the Quality of Service (QoS) will be met [9, 10]. Some of the focus is directed on static resource provisioning, so that a given deadline in MapReduce could be met, while others present proposals for scaling resources in response to resource demand and cluster use as a way to reduce the total cost. Furthermore, frameworks are proposed to improve MapReduce jobs' performance [8, 11]. These works are similar to the work we propose, due to the need to optimize resources in order to reduce both energy consumption and operating cost. Nonetheless, unlike our work, these works do not consider optimizing job execution time in the presence of stragglers.

There are studies that have shown interest in improving MapReduce performance by proposing new scheduling techniques [12]. The proposed schedulers aim to improve the resource allocation and execution time of jobs while meeting the QoS. In addition to these works' goals, our proposed scheduler also tries to mitigate stragglers and maximize the cluster utilization using a probabilistic approach.

Other researchers have developed an interest in mitigating stragglers and enhancing the mechanism of the default Hadoop speculation. They proposed new mechanisms for detecting stragglers, reactively and proactively and launch speculative tasks accordingly [3–5]. In particular, reactive approaches typically launch new copies of all slow-running tasks (i.e., stragglers) once they are detected, while proactive approaches launch multiple copies for each task at the beginning of task execution without waiting for stragglers to occur. These mechanisms are essential in order to ensure a high level of reliability to satisfy a given QoS. Different from these existing works, this paper presents an optimization framework that jointly maximizes the probabilities that all jobs meet their deadlines, by intelligently optimizing the number of clones based on job's size and progress.

## III. Background and System Model

We consider a parallel computation framework, such as MapReduce, that splits computation jobs into small tasks to run in parallel cross multiple nodes. Each node is capable of executing a number of tasks based on its capacity. The framework consists of *map* and *reduce* tasks; the output of the map tasks are passed as input to the reduce tasks. In this paper, we consider a system with limited resource capacity $m$ Virtual Machines (VMs), and a fraction $\lambda$ of this capacity, as determined by the cloud provider, is available for allocation to jobs. Similar to [11, 13], we focus on map-only jobs/tasks that are executed in one wave in homogeneous nodes.

Consider $J$ jobs submitted to the MapReduce processing framework. Each job $j$ is associated with a deadline $D_j$ that is determined by application latency requirements. Each job $j$ consists of $N_j$ tasks, and it is considered successful if all its $N_j$ tasks are executed and completed before the job deadline $D_j$. Let $T_j$ denote job $j$'s completion time, and $T_{j,i}$ for $i = 1, \ldots, N_j$ be 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:

$$T_j = \max_{i=1,\ldots,N_j} T_{j,i},\ \forall j \tag{1}$$

Any task whose execution time exceeds the deadline is considered a straggler. Our dynamic cloning approach mitigates the effect of stragglers by proactively launching $r_j$ extra attempts for each task. Thus, for each task, there are $r_j + 1$ attempts/copies that start execution at the same time and process data independently of each other. A task is finished once any one of the $r_j + 1$ attempts finishes execution, and then the other copies are killed. Due to various sources of uncertainty causing stragglers, we model the completion time of attempt $k$ (for $k = 1, \ldots, r_j + 1$) of task $i$ and job $j$ as a random variable $T_{j,i,k}$, with known distribution. Thus, task $i$'s completion time $T_{j,i}$ is determined by the completion time of the fastest attempt, i.e.,

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

We assume that execution times $T_{j,i,k}$ of different attempts are independent because of resource virtulization, and that $T_{j,i,k}$ follows a Pareto distribution, parameterized by $t_{\min}$ and $\beta$, where $t_{\min}$ is the minimum execution time and $\beta$ is a shape parameter, according to existing work characterizing task execution time in MapReduce [14]. Unlike default Hadoop scheduling, our new dynamic cloning mechanism is able to proactively launch an extra $r_j$ copies for each task on the fly. When a new job arrives, all jobs in the system are jointly optimized to determine their optimal replication factors $r_j$ for $j = 1, \ldots, J$, under system capacity constraints. In this way, the optimization is able to continuously re-adjust task cloning with respect to system dynamics. This strategy not only avoids time overhead for detecting stragglers (required by default Hadoop speculative execution), it also ensures that each active task receives at least one attempt before the otherwise under-utilized system resources are apportioned among all active jobs/tasks.

## IV. Joint PoCD and Resource Optimization

### A. PoCD Analysis

We define PoCD as the probability that a job finishes before its deadline, when launching $r_j$ extra attempts. For newly-arrived jobs (that are yet to start) and existing jobs (that may already have multiple attempts per task), we derive their PoCDs in Theorems 1 and 2, respectively. For simplicity, we temporarily drop the subscript for task, and let $T_{j,k}$ denote the completion time of an attempt in job $j$. Recall that $\beta$ and $t_{\min}$ are the shape parameter and the minimum execution time, respectively, of the Pareto distribution characterizing $T_{j,k}$.

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

$$R_{\text{su}} = \left[1 - \left(\frac{t_{\min}}{D_j}\right)^{\beta\cdot(r_j+1)}\right]^{N_j} \tag{3}$$

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

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

Therefore, the probability that a job finishes before its deadline, PoCD, is given by

$$R_{\text{su}} = \left[1 - \left(\frac{t_{\min}}{D_j}\right)^{\beta\cdot(r_j+1)}\right]^{N_j} \tag{5}$$

where $N_j$ is the number of tasks. This shows that as we increase $r_j$, a high PoCD can be obtained. □

In our proposed dynamic cloning, when cloning an existing job that may already have multiple active attempts for each task, we clone the fastest attempt (i.e., having maximum progress) of each task to optimize the efficiency of our strategy. Let $\phi_i$ be the largest progress (i.e., the percentage of data processed) of all task $i$'s attempts, and $\tau_j$ be the elapsed time of job $j$. Under this strategy, we quantify the PoCD of an existing task as follows:

**Theorem 2.** *The PoCD of an existing job*

$$R_{\text{ru}} = \left[1 - \left(\frac{(1-\phi_i)t_{\min}}{D_j - \tau_j}\right)^{\beta\cdot(r_j+1)}\right]^{N_j} \tag{6}$$

*Proof.* To compute the probability that a running attempt finishes before its deadline, we need to find how much progress it has made. We denote the progress as $\varphi_k$. Note that, upon a new job submission, we keep that fastest attempt of each task for a running job. We denote the execution time of the fastest attempt of task $i$ as $T_{i,f}$. It can be determined by:

$$T_{i,f} = \min_{k=1,\ldots,r_j+1} T_{i,k},\ \forall i. \tag{7}$$

Once $T_{i,f}$ is determined, we launch $r$ extra attempts for each task; these extra attempts start executing from the last key-value pair processed by the fastest attempt. Now, the probability that a running attempt will fail to finish before the deadline, can be computed as follows:

$$P_{\rm ru} = P((1-\varphi_k)T_{j,k} > D_j - \tau_j) = \left(\frac{(1-\varphi_k)t_{\min}}{D_j-\tau_j}\right)^{\beta} \quad (8)$$

where $D_j - \tau_j$ and $1-\varphi_k$ are the remaining time before deadline and the remaining fraction of data to be processed, respectively. Note that when we calculate the PoCD of a running job, we use the progress of the slowest task (where the task progress is given by the progress of its fastest attempt). We denote the progress as $\phi_i$.

Therefore, the probability that a running job finishes before its deadline is given by

$$R_{\rm ru} = \left[1 - \left(\frac{(1-\phi_i)t_{\min}}{D_j-\tau_j}\right)^{\beta\cdot(r_j+1)}\right]^{N_j} \quad (9)$$

□

### B. Joint PoCD Optimization

We formulate the problem of joint PoCD maximization under system capacity constraints. Under dynamic cloning, each task of job $j$ has $r_j+1$ attempts, including one original attempt and $r_j$ cloned attempts. Then, the total number of attempts of job $i$ is $N_j\cdot(r_j+1)+1$, where an extra VM is required to run its job tracker/master. Recall that $m$ is the total number of VMs available in the system, and $\lambda$ is the fraction of these VMs that are allowed for task cloning. Thus, by adjusting $\lambda$, a cloud provider can balance the resource allocation for clones and task execution. A system capacity constraint $\sum_j N_j\cdot(r_j+1)+|J| \le \lambda\cdot m$ must be satisfied at any given time. Let $R_j(r_j)$ (i.e., $R_{\rm su}$ for a newly-arrived job or $R_{\rm ru}$ for an existing job) be the PoCD function for $r_j$ extra attempts. We have the following PoCD optimization:

$$\text{maximize} \quad \sum_{j=1}^{J} U(p_j), \quad (10)$$

$$\text{s.t.} \quad \sum_{j=1}^{J} N_j\cdot(r_j+1)+|J| \le \lambda\cdot m \quad (11)$$

$$p_j = R_j(r_j),\ \forall j \quad (12)$$

$$r_j \ge 0,\ \forall j \quad (13)$$

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

Here, $U(\cdot)$ is a utility function to guarantee fairness of our strategy. For example, we can choose a family of well-known $\alpha$-fair utility functions parameterized by $\alpha$ [15]. Then, the solution to this PoCD optimization is able to achieve maximum 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 to obtain the optimal $r_j$ for each job under cloud resource constraints. Upon job arrivals and departures, the scheduler first recalculates remaining resources available for dynamic cloning, and identifies all jobs as well as their upcoming deadlines. The algorithm then works in a greedy fashion to assign VMs to jobs with highest utility improvement. More precisely, we start by assigning $r_j=0$ to each job, and calculate each job's utility as a function of its PoCD. We use $\omega$ to denote the total resource assigned to all jobs and $\kappa$ the available resource for all tasks. We iteratively find the job with minimum PoCD utility and increase its $r_j$ by one. The process is repeated until the system capacity constraint (11) is reached.

**Algorithm 1:** Proposed Online Algorithm

```
Upon submission of a new job:
Kill all jobs which missed their deadlines
J = {j_1, j_2, j_3, ...}
if |J| == 1 then
    r_max = ⌊(λ·m − N_1 − 1)/N_1⌋
    r_1 = r_max
else
    r_j = 0 ∀j
    ω = 0
    κ = λ·m − Σ_{j=1}^{J} N_j − |J|
    Calculate R_j ∀j
    while J ≠ {∅} do
        j' = arg min_j {R_j}
        if N_{j'} + ω > κ then
            J = J − {j'}
        else
            r_{j'} = r_{j'} + 1
            ω = ω + N_{j'}
            Calculate R_j ∀j
        end if
    end while
end if
```

## V. Implementation

We implement Shed as a pluggable scheduler in Hadoop YARN, which consists of a Resource Manager

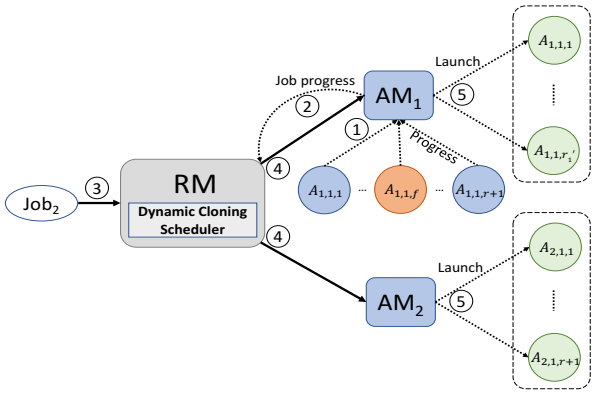



Fig. 1: System Architecture and steps taken upon new job arrival.

(RM), an Application Master (AM) for each application (job) and a Node Manager (NM) in each node. The AM requests resource containers (VMs) to execute jobs/tasks and continuously monitors the progress of each tasks in NMs. The RM is responsible for tracking and managing VMs in a cluster and scheduling jobs. In particular, the scheduler in the RM optimizes and assigns resources to requesting jobs. Figure 1 illustrates our system architecture and steps taken to achieve optimality.

Upon submission of a job, our scheduler uses the job's deadline and number of tasks to calculate the optimal $r$ which maximizes its PoCD. Once $r$ is obtained, the RM sends it to the corresponding AM to create $r+1$ attempts for each task. Then, the AM negotiates resources with the RM and works with NMs to launch attempts. While the submitted job is running, the AM keeps track of all attempts' progress and maintains the last record's offset processed.

As the AM keeps track of all running attempts, it reacts to every new arrival (job) by killing all slow attempts belonging to every task, while the fastest one (i.e., which processed the most data) is kept alive and will continue running. Note that when a new job arrives, our scheduler reoptimizes resources and obtains a new $r$ for each job (running or newly submitted) that maximizes the overall PoCD. Therefore, the AMs speculate/create new $r$ copies for each running attempt that is kept alive as the fastest attempt. We develop a new clone launching mechanism, which allows existing task progress to be preserved and transferred to clone attempts. In particular, the last known data offset that has been processed by the original task is passed by AM to its new clone attempts, which are able to continue the task execution in a smooth, seamless fashion. This significantly improves the effectiveness of dynamic cloning and thus the performance of PoCD optimization.

In Figure 1, we demonstrate how our scheduler reacts to new arrivals. Consider job 1 is submitted to a cluster and running. For simplicity, we assume each job has only one task $A_{j,1}$. Job 1 creates $r_1$ extra attempts as determined by the scheduler upon submission. Each attempt, $A_{1,1,k}$ for $k = 1, \ldots, r_1+1$, reports its progress to the AM including number of bytes processed. Then the AM reports the whole job's progress to the RM. When job 2 arrives, the scheduler in the RM reoptimizes cluster resources and obtains $r$ for jobs 1 and 2 based on their PoCDs. Once job 1 receives a new value $r_1'$, the AM kills all slow attempts and clones the fastest attempt, $A_{1,1,f}$. The total number of attempts for all jobs are bounded by the available resources.

One challenge that arises is that AMs need to consider the time it takes to launch new $r$ attempts. The reason is because in highly contended clusters, Java Virtual Machine (JVM) startup time is significant and cannot be ignored [16]. Moreover, these on-demand requests submitted to the RM can not be predicted and could arrive at any time. Thus, each AM takes JVM launching time into consideration when passing the last offset processed to the new attempts [17]. In particular, the AMs estimate the number of bytes, $b_{\text{extra}}$, that will be processed by the fastest attempts. Even though the last offset, $b_{\text{proc}}$, is recorded when the new attempts are created, the AM will skip the data processed during launching time and pass a new offset, $b_{\text{new}}$, to the new attempts. If the AM finds that all remaining bytes of the data will be processed during launching time, the new attempts will be killed. The estimated number of bytes, $b_{\text{extra}}$, can be obtained as follows:

$$\frac{b_{\text{proc}} - b_{\text{start}}}{t_{\text{now}} - t_{\text{proc}}} \cdot (t_{\text{FP}} - t_{\text{lau}}) \tag{14}$$

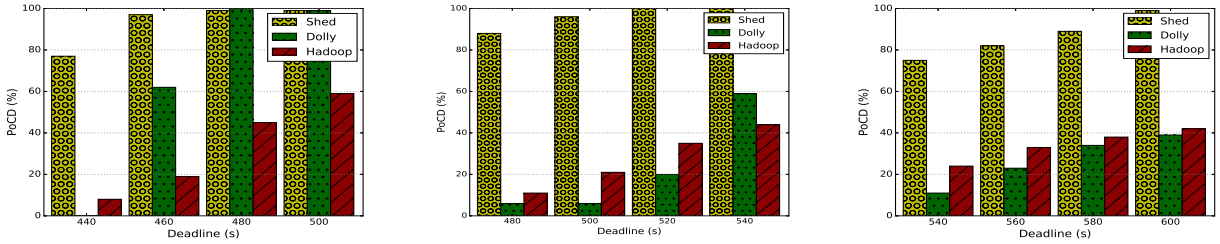
where $t_{\text{proc}}$, $t_{\text{FP}}$ and $t_{\text{lau}}$ are attempt start processing time, first progress report time and attempt launch time, respectively. Thus, the new byte offset received by the new attempts is calculated as follows:

$$b_{\text{new}} = b_{\text{start}} + b_{\text{proc}} + b_{\text{extra}} \tag{15}$$

## VI. Evaluation

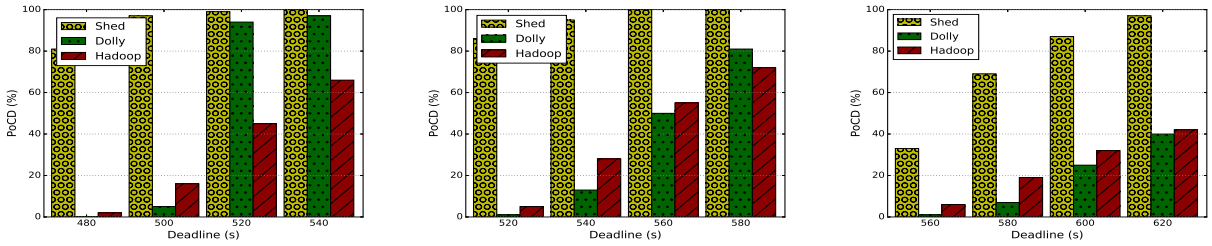
### A. Experimental Setup

We deploy our proposed scheduler on Amazon EC2 consisting of 121 nodes, one master and 120 slaves. We set $\lambda = 100\%$. Each node is capable of running one task at a time. We evaluate our scheduler by using Map phases of two benchmarks, WordCount and WordMean. WordCount is an I/O and CPU-bound job while WordMean is CPU-bound. We assume that tasks of a job are executed in one wave in homogeneous nodes. We create three classes of jobs consisting of 5,




(a) (b) (c)

Fig. 2: Comparisons of Shed, Dolly and Hadoop in terms of PoCD with different workloads using WordCount benchmark: (a) 5-task jobs (b) 10-task jobs (c) 20-task jobs.




(a) (b) (c)

Fig. 3: Comparisons of Shed, Dolly and Hadoop in terms of PoCD with different workloads using WordMean benchmark: (a) 5-task jobs (b) 10-task jobs (c) 20-task jobs.

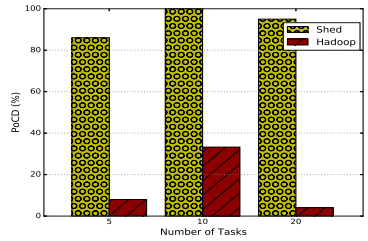



Fig. 4: Comparison of Shed and Hadoop in terms of PoCD with hybrid workloads and deadlines.

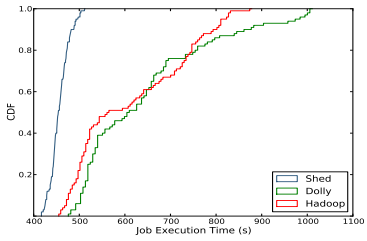



Fig. 5: The cumulative distribution function (CDF) of Shed, Dolly and Hadoop for 10-task jobs of WordCount benchmark.

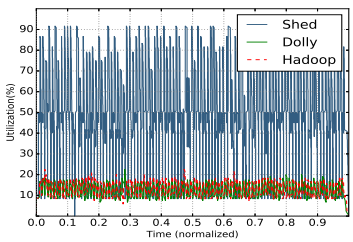



Fig. 6: Cluster utilization of Shed, Dolly and Hadoop for 10-task jobs of WordCount benchmark.

10, and 20 tasks. Each task processes a chunk of data with size 128MB. We run 100 jobs for each experiment and set average job inter-arrival time to 5 minutes. The baseline in our experiment is 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 $1 - PoCD$, 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. We measure the PoCD of all strategies by calculating the percentage of jobs that completed before their deadline. 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 $\beta \leq 2$ [14], and $t_{\min} = 120$ sec. Even though our proposed algorithm sets $r$ equal to the maximum value if there is only one job, we find that $r$ 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 $r$. We find 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

Figures 2 and 3 compare the measured PoCD (percentage of jobs meeting deadline) of our proposed algorithm with Dolly and default Hadoop with speculation.

The figures show that our algorithm is able to achieve up to 100% PoCD, while default Hadoop is around 40% in most experiments. The figures also show that Shed can significantly outperform Dolly with large jobs or tight deadlines. The performance difference reduces for small jobs with large deadlines, i.e., when the cloud utilization is extremely low, so there exists enough cloud resource to assign the maximum needed number of clones to each job, making optimization less appealing.

Moreover, the figures show that when job deadlines are relaxed, the PoCDs of all strategies increase, but Shed continues to perform significantly better than Dolly and default Hadoop, demonstrating its superiority in dealing with hard application deadlines. Note that our numerical results compare Shed, Dolly and default Hadoop for various deadlines up to 620s, because Shed already achieves nearly 100% PoCD due to more efficient utilization of system resources for running clone copies. This massive improvement over both Dolly and default Hadoop is also due to the fact that Shed proactively launches clones before stragglers occur in the cloud and jointly optimizes the number of clones for all jobs. Moreover, the new clone launching mechanism guarantees that no repeated data processing is needed for any clone attempts.

Figure 4 shows the PoCD of Shed compared with Hadoop for hybrid workloads and deadlines. We test our algorithm by running WordCount benchmark with 100 jobs: 50 5-task jobs, 30 10-task jobs and 20 20-task jobs with deadlines 460 s, 500 s and 560 s, respectively. The results show that, even with mixed, heterogeneous workloads and deadlines, our algorithm achieves a PoCD of more than 85% in all cases (which is consistent with the homogeneous workload results), and significantly outperforms Hadoop.

Figure 5 shows the cumulative distribution function (CDF) of job execution times for 10-task WordCount jobs and a deadline of 500s. Notice that almost all jobs complete within 500s under Shed whereas only 10% and 20% of the jobs complete by 500s under Dolly and Hadoop, respectively, and it takes as much as 1000s and 900s for some jobs to complete under Dolly and Hadoop, respectively. The average job execution time (not shown in the figure) for Shed, Dolly and Hadoop are 457s, 645s and 615s, respectively. Figure 6 depicts the cluster utilization under Shed and Hadoop strategies. It can be clearly seen that while Hadoop can only achieve 20% utilization (similar to Dolly), Shed is able to optimize the underutilized resources and achieve much higher levels of utilization. The figure also shows how Shed exploits idle slots in order to achieve better performance in meeting job deadlines. Similar results are evident for different workloads.

## VII. Conclusion

In this paper, we propose Shed, an optimization framework that leverages dynamic cloning to jointly maximize PoCD and cluster utilization. We also present an online scheduler that dynamically optimize resources upon new job arrival. Our solution includes an online greedy algorithm to find the optimal number of clones needed for each job. Our results show that Shed can achieve up to 100% PoCD compared to Dolly and Hadoop with speculation. The proposed algorithm is able to achieve more than 90% utilization of available cloud resources, whereas Dolly and Hadoop achieves only about 22%.

## References


[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] Apache Software Foundation, "Hadoop." [Online]. Available: https://hadoop.apache.org

[3] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments." in *OSDI'08*, 2008.

[4] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters using mantri." in *OSDI'10*.

[5] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones," in *NSDI'13*, 2013.

[6] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making sense of performance in data analytics frameworks," in *NSDI'15*, 2015.

[7] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: measurements & analysis," in *SIGCOMM'09*.

[8] D. Cheng, J. Rao, C. Jiang, and X. Zhou, "Resource and deadline-aware job scheduling in dynamic hadoop clusters," in *IPDPS'15*.

[9] X. Xu, M. Tang, and Y.-C. Tian, "Theoretical results of qos-guaranteed resource scaling for cloud-based mapreduce," *IEEE Trans. on Cloud Computing*, vol. PP, no. 99, pp. 1–1, 2016.

[10] A. Verma, L. Cherkasova, and R. H. Campbell, "Aria: automatic resource inference and allocation for mapreduce environments," in *ICAC'11*.

[11] S. Alamro, M. Xu, T. Lan, and S. Subramaniam, "Cred: Cloud right-sizing to meet execution deadlines and data locality," in *CLOUD'16*.

[12] J. Wolf, D. Rajan, K. Hildrum, R. Khandekar, V. Kumar, S. Parekh, K.-L. Wu, and A. Balmin, "Flex: A slot allocation scheduling optimizer for mapreduce workloads," in *Middleware'10*.

[13] M. Elteir, H. Lin, W.-c. Feng, and T. Scogland, "Streammr: an optimized mapreduce framework for amd gpus," in *ICPADS'11*.

[14] H. Xu and W. C. Lau, "Optimization for speculative execution in big data processing clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 2, pp. 530–545, 2017.

[15] T. Lan, D. Kao, M. Chiang, and A. Sabharwal, "An axiomatic theory of fairness in network resource allocation," in *INFOCOM'10*.

[16] M. Mao and M. Humphrey, "A performance study on the vm startup time in the cloud," in *CLOUD'12*.

[17] M. Xu, S. Alamro, T. Lan, and S. Subramaniam, "Optimizing speculative execution of deadline-sensitive jobs in cloud," in *SIGMETRICS' 17*.