Straggler Handling Techniques

MD Akik Khan
Louisiana State University
Baton Rouge, LA 70803
akhan12@lsu.edu

ABSTRACT

Distributed processing frameworks split large chunks of data into smaller parts so that it is easier to process the task. These data chunks are then executed in parallel to achieve faster job completion. During this execution, there are some jobs that are slow and takes a while to complete the task. These smaller slow running tasks are called *stragglers*. Stragglers increase the average job duration by 47% in production data clusters at Facebook and Microsoft Bing [1, 2, 4]. This is because current mitigation techniques all involve an element of waiting and speculation.

In this paper, we will analyze three systems and their techniques to handle stragglers and compare them with other scheduling algorithms to justify why these are better solutions for a given situation. First method is *full cloning of small jobs to avoid waiting and speculation*, which is the baseline for a system called *Dolly* [1]. This will be compared with another system called *Wrangler* [4] that proactively avoids situations that cause stragglers. It has the capability of automatically predicting such situations by using statistical learning techniques based on cluster resource utilization counters. The last method is called *GRASS*, which carefully uses speculation to mitigate the impact of stragglers in approximation jobs [2]. This finds a balance between immediateness of improving approximation goal and long term implication of using extra resources for speculation.

1. INTRODUCTION

As technology progresses, newer systems and frameworks are able to process large datasets by breaking them into small tasks and execute them in parallel on compute slots on different machines. Cloud computing is another example that follows the same pattern. However, these technologies face a barrier when slower tasks bog down the system and prevent other jobs to finish. Tasks that are much slower than other tasks are called *stragglers*. As a result, the overall job completion time increases since a job finishes only when its last task finishes.

Two of the major techniques in straggler mitigation are *blacklisting* and *speculative execution*. *Blacklisting* is determining the health of a machine so that it can divide it into two groups, health machine and unhealthy (bad) machine. If the machine has bad health, it will avoid to schedule tasks in that machine. But this is not that efficient in weeding out stragglers because they also occur in non-blacklisted machines because of intrinsically complex reasons such as I/O contentions, interference by periodic maintenance operations and background services, and hardware behaviors [5]. *Speculative execution* means waiting to observe the progress of the tasks and launching duplicate copies of slower tasks. This problem is exacerbated when some tasks start straggling when they are well into their execution. "Spawning a speculative copy at that point might be too late to help [1]."

To handle this issues, researchers suggested to use a variation of *speculative execution*. This will launch multiple *clones* of every task of a job and consider the one that finishes first rather than waiting and analyzing the tasks. But cloning has two main challenges, usage of extra resources and contention on intermediate jobs. From the production trace analysis, it was observed that 90% of jobs consume almost as little as 6% resources [1]. As a result, we can use few extra resources for interactive jobs as they fall in the category of small jobs. When extra clones are created, there is a chance of having potential contention for the intermediate data passed between tasks of different phases (e.g. map, reduce, join). A solution to this problem is to make each downstream clone read exclusively from only a single upstream clone. But this staggers the start times of the downstream clones. To handle the contention problem, a hybrid solution is suggested called delay assignment. Its performance was tested using the workloads of Facebook and Bing, which showed an improvement in the average completion time of the small jobs by 34% to 46%, respectively, with LATE [6] and Mantri [3] as baselines.

Another approach is to launch speculative copies (*approximation jobs*) instead of cloning every task in a job. They focus mainly on providing timely result and gives this the highest priority even if it only part of a dataset. To implement this, we can use *Greedy Speculative* (GS) scheduling and *Resource Aware Speculative* (RAS) scheduling. GS greedily picks the task that improves the approximation goal whereas RAS considers the opportunity cost and schedules a speculative copy only if it saves time and resources. By combining this two scheduling techniques, GRASS was created that works in two steps. It first uses RAS to schedule tasks and then GS when the job gets close to approximation bound. It results in 47% improvement in accuracy of deadline-bound jobs and 38% speed-up of error-bound jobs compared to state-of-the-art straggler mitigation techniques (LATE and Mantri).

However, without any additional information, such reactive techniques can not differentiate between nodes that are inherently slow and nodes that are temporarily overloaded. Such techniques lead to unnecessary over-utilization of resources without necessarily improving the job completion times. Though proactive, Dolly is still a replication-based approach that focuses only on interactive jobs and incurs extra resources. Being doubtful to the correlations between stragglers and nodes' status, replication-based approaches are wasteful.

2. THE CASE OF CLONING

A job has different phases that execute the same type of tasks in parallel. Based on the progress rates of tasks, we can identify stragglers. Progress rates are expected to be similar to IO and compute operations when we do not have stragglers. It is used instead of task's duration to remain agnostic to skews in work assignment among tasks.

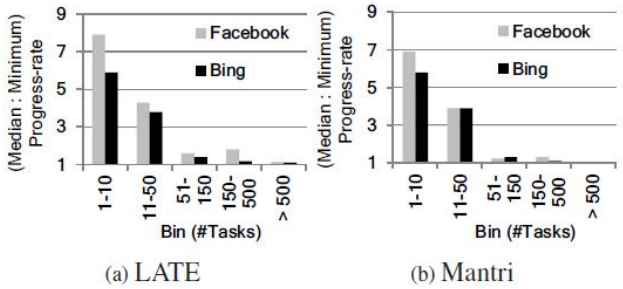


Figure 1: Slowdown ration *after* applying LATE and Mantri. Small jobs see a higher prevalence of stragglers.

Slowdown ratio was measured for each phase to find a comparison with stragglers. It is the ratio of progress rate of the median task to the slowest task. "The negative impact of the stragglers increases as the slowdown ratio increases [1]." The slowest task is up to 8 times slower than the median task in the job for LATE whereas it is 7 times slower for Mantri (Fig. 1). Another noticeable thing for two cases (LATE and Mantri) is that speculation technique is not effective in mitigating stragglers in small jobs, but it is effective for large jobs. That is why it is not that beneficial since large amount of jobs (almost 80%) consist of small ($\leq$ 10) tasks and they dominate the production traces of Facebook and Bing.

Blacklisting is Insufficient: A solution for mitigating stragglers is to black-list machines that will give leverage to stragglers. For this case, a straggler is classified as tasks that have progress rate lower than half of the median progress rate among tasks that take place. Two time windows, 5 minutes and 1 hour, were tested to analyze the effects of stragglers. "The best case eliminates only 12% of the stragglers and improves the average completion time by only 8.4% (in the Bing trace, 11% of stragglers are eliminated leading to an improvement of 6.2%) [1]." Even though there is a small improvement, it will not help in large production traces.

Heavy tail in Job Sizes: Small interactive jobs dominate the cluster and have stringent latency demands. "In the Facebook and Bing traces, jobs with $\leq$ 10 tasks account for 82% and 61% of all the jobs, respectively [1]." On the other hand, they are the ones that are most affected by stragglers. As job sizes have a *heavy-tail* distribution, we can clone small jobs using few extra resources. Figure 2a shows, 90% of the smallest jobs consume only 6% and 11% of the total cluster resources in the Facebook and Bing clusters, respectively. Indeed, the distribution of resources consumed by jobs follows a power law (Figure 2b). In fact, at any point in time, the small jobs do not use more than 2% of the overall cluster resources.

Cloning of Parallel Jobs: Dolly suggests launching multiple clones of a job and consider the result of the first clone that finishes. Cloning is a better option than speculation because i) it does not have to wait and observe a task before acting and ii) it is free from the risk of speculating wrong tasks or missing the stragglers.

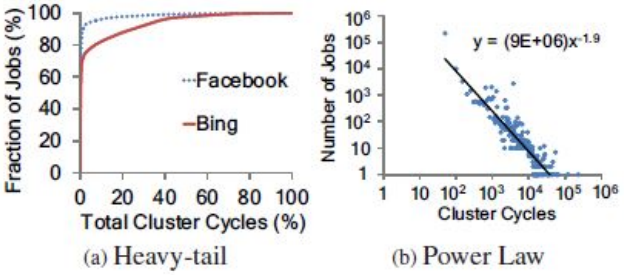


Figure 2: (a) shows the heavy tail in the fraction of total resources used and (b) shows that the distribution of cluster resources consumed by jobs, in the Facebook trace, follows a power law.

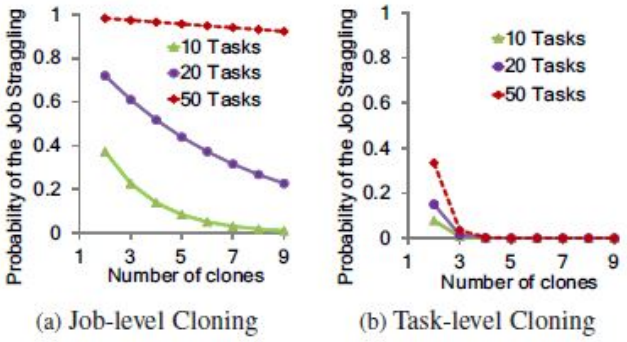


Figure 3: Probability of a job straggling for varying number of clones, and sample jobs of 10, 20, and 50 tasks. Task-level cloning requires fewer clones than job-level cloning to achieve the same probability of the job straggling.

Granularity of Cloning: Granularity of cloning plays a crucial role in achieving efficiency. There are two types of cloning possible: job-level cloning and task-level cloning. For job-level cloning, multiple clones of the entire job are launched for every job submitted to the cluster. It is appealing due to its simplicity and ease of implementation. Task-level cloning means cloning at the granularity of individual tasks and launching multiple clones of each task. A clone group is formed with different clones of the same task. We will then consider the clone from a clone group that finishes first. That is why task-level cloning requires internal changes to the execution engine of the framework.

From figure 3, we can see that task-level cloning has higher gains/clone and the probability of the job straggling drops rapidly. But there is a problem in this design, storage. As a result, task-level cloning in Dolly is preferred over job-level cloning.

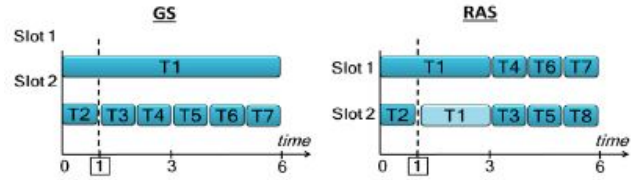
3. APPROXIMATION JOBS

Approximation jobs are explored across two dimensions: *deadline-bound* jobs and *error-bound* jobs. *Deadline-bound*

jobs maximize the accuracy of their result within a specified time limit whereas *error-bound* jobs strive to minimize the time taken to reach a specified error limit in the result. It is important because of cluster heterogeneities and multi-waved jobs (number of tasks is greater than available compute slots).

Deadline-Bound Jobs In the absence of speculation, a different policy is considered known as Shortest Job First (SJF) that schedules the task with the smallest processing time. In many cases, it can be proven to minimize the number of incomplete tasks, thus maximizing the number of tasks completed. "Thus, without speculation, SJF finishes the most tasks before the deadline [2]."

If speculation is allowed, a neutral approach would be to allow the currently running tasks to be placed in the queue, and to choose the smallest sized task. So, the next task to run is still chosen according to SJF, only now speculative copies are also considered. This policy is called *Greedy Speculative (GS) scheduling*, because it picks the next task to schedule greedily (the one that finishes the quickest), and thus improve the accuracy. But *Resource Aware Speculative (RAS) scheduling* speculates only if it saves both time and resources. Thus, the sum of the resources used by the speculative and original copies, when running simultaneously, must be less than letting just the original copy finish.

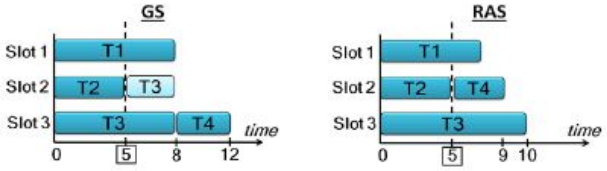


Task ID	T1	T2	T3	T4	T5	T6	T7	T8	T9
t_{rem}	5	---	--	---	---	---	---	---	---
t_{new}	2	---	1	1	1	1	1	1	3

Figure 4: GS and RAS for a deadline-bound job with 9 tasks. The t_{rem} and t_{new} values are when T2 finishes. The example illustrates deadline values of 3 and 6 time units.

Two important measures we need to be familiar about are t_{rem} (remaining duration of a running task) and t_{new} (duration of a new copy). Even though figure 4 shows better performance for RAS, it is not uniformly better than GS. In particular, RAS's cautious approach can backfire if it overestimates the opportunity cost. Looking at the figure, if the deadline of the jobs were reduced from 6 time units to 3 time units instead, GS performs better than RAS. At the end of 3 time units, GS has led to three completed tasks while RAS has little to show for its resource gains by speculating T1. So, we can say that the value of deadline and the number of waves are two important factors in determining scheduling technique.

Error-Bound Jobs: The goal of error-bound jobs is to minimize the makespan of the tasks needed to achieve the error limit. Thus, instead of SJF, Longest Job First (LJF) is more preferred. Figure 5 presents an illustration of GS and RAS for an error-bound job with 6 tasks and 3 compute slots. The t_{rem} and t_{new} values are at 5 time units. GS decides to launch a copy of T3 as it has the highest t_{rem}. RAS conservatively avoids doing so. Consequently, when the error limit is high (say, 40%) GS is quicker, but RAS is better when the limit decreases (to, say, 20%).



Task ID	T1	T2	T3	T4	T5
t_{rem}	3	---	5	---	---
t_{new}	3	---	3	4	4

Figure 5: GS and RAS for error-bound job with 6 tasks. The t_{rem} and t_{new} values are when T2 finishes. The example illustrates error limit of 40% (3 tasks) and 20% (4 tasks).

4. TWO OPPOSITE STRATEGIES

A fundamental challenge of cloning is the potential contention it creates in reading data. To take care of this case, two pure strategies at opposite ends are suggested. They are Contention-Avoidance Cloning (CAC) and Contention Cloning (CC). CAC completely avoids contention by assigning each upstream clone, as it finishes, to a new downstream task clone. This avoid contention because it guarantees that every upstream task clone only transfers data to a single clone per downstream clone group. On the other hand, CC ignores the extra contention caused and assumes that the first finished upstream clone in every clone group can sustain transferring its intermediate output to all downstream task clones. None of these clones are handicapped as all the downstream clones start at the same time. Also, only one of the upstream clones in a clone group need to be a non-straggler for a job to succeed without straggling.

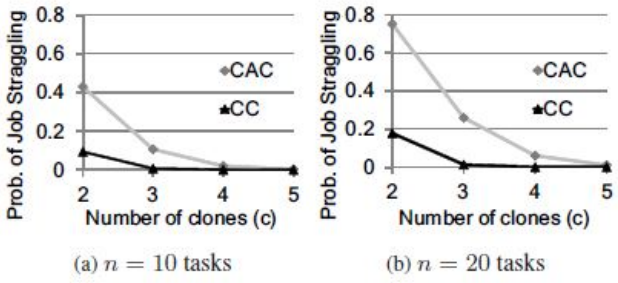


Figure 6: CAC vs. CC: Probability of a job straggling.

Comparison between CAC and CC: Part (a) has 10 upstream and downstream jobs whereas part (b) has 20. "With three clones per task, the probability of the job straggling increases by over 10% and 30% with CAC compared to CC [1]." The gap between the two diminishes for higher numbers of clones. For this case, CAC is more likely to have jobs that will be stragglers compared to CC. But CC is not free from downsides. For smaller jobs (< 50 tasks), transfers of jobs slow down by 32% and 39% at median, third quartile values are 50%. Transfers of large jobs are less hurt because tasks of large jobs are often not cloned because of lack of

cloning budget. Overall, contentions cause significant slowdown of transfers and are worth avoiding.

Delay Assignment: A deficiency with both CAC and CC is that they do not distinguish stragglers from tasks that have normal variations in their progress. They have other negative impacts that are crucial in determining stragglers. For this reason, a hybrid approach called delay assignment was proposed that first waits to assign the early upstream clones (like CAC), and thereafter proceeds without waiting for any remaining stragglers (like CC). The setting of wait time plays an important role in delay assignment's performance. The objective is to minimize the expected duration of a downstream task, which is the minimum of the durations of its clones. They were able to pick a wait duration that minimized the completion time by following these three steps:

1. Calculate the clone's expected duration for reading each upstream output using T_C and T_E.

2. Use read durations of all clones of a task to estimate the overall duration of the task.

3. Find the delay that minimizes the task's duration.

5. GRASS SPECULATION ALGORITHM

Another approach to handle stragglers is GRASS speculation algorithm. There are two possibilities we have to look into to implement GRASS: deadline-bound jobs and error-bound jobs. For deadline-bound jobs, we switch from RAS to GS when the time to the deadline is sufficient for at most two waves of tasks. One the other hand, for error-bound tasks, we switch when the number of (unique) scheduled tasks needed to satisfy the error-bound makes up to two waves.

But there are some challenges when implementing these ideas. First, identifying the final two waves of tasks is difficult in practice because tasks are not scheduled at explicit wave boundaries (scheduled when slots open up). In addition, the wave-width of jobs varies considerably depending on cluster utilization, so does task duration. In light of these difficulties, we interpret the guideline as follows: RAS is better when the deadline is loose, or the error limit is low; otherwise GS performs better.

An ideal approach would be to gather enough samples of job performance based on switching to GS at different points. For deadline-bound jobs, this is decided by remaining time to the deadline. For error-bound jobs, this is decided by the number of tasks to complete towards meeting the error.

In an ideal situation, an incoming job starts with RAS and periodically compares samples of jobs smaller than its size during its execution to check if it is better to switch to GS. It checks by using its remaining work at any given point and continues with RAS until the optimal switching point turns out to be at present. This process is performed periodically during the job's execution. But the size of job alone is insufficient to calculate the optimal switching point. As a result, it is augmented with the number of waves, which is approximated using current cluster utilization. Another measure is the estimation accuracy of t_{rem} and t_{new}. Because of these reasons, GRASS obtains samples of job performance with both GS and RAS across values of deadline/error-bound, estimation accuracy of t_{rem} and t_{new}, and cluster utilization to decide when (and if) to switch from RAS to GS.

6. PERFORMANCE EVALUATION

In this part, we will describe and compare the algorithms presented here and also some other algorithms that are used to mitigate stragglers. This evaluation was mainly based on the paper [4] because it explains the necessary details for each algorithm and compares [1] and [2].

Dolly improves the average completion time of jobs by 42% compared to LATE and 40% compared to Mantri, in the Facebook workload. The corresponding improvements are 27% and 23% in the Bing workload. Small jobs benefit the most, improving by 46% and 37% compared to LATE and 44% and 34% compared to Mantri, in the Facebook and Bing workloads. This was possible because of the power-law in job sizes and also for admission control policy.

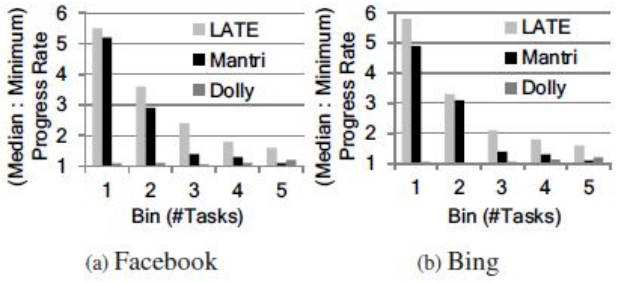


Figure 7: Ratio of median to minimum progress rates of tasks within a phase.

Figure 7 presents supporting evidence for the improvements. The ratio of medium to minimum progress rates of tasks, which is over 5 with LATE and Mantri in our deployment, drops to as low as 1.06 with Dolly. Even at the 95th percentile, this ratio is only 1.17, thereby indicating that Dolly effectively mitigates nearly all stragglers.

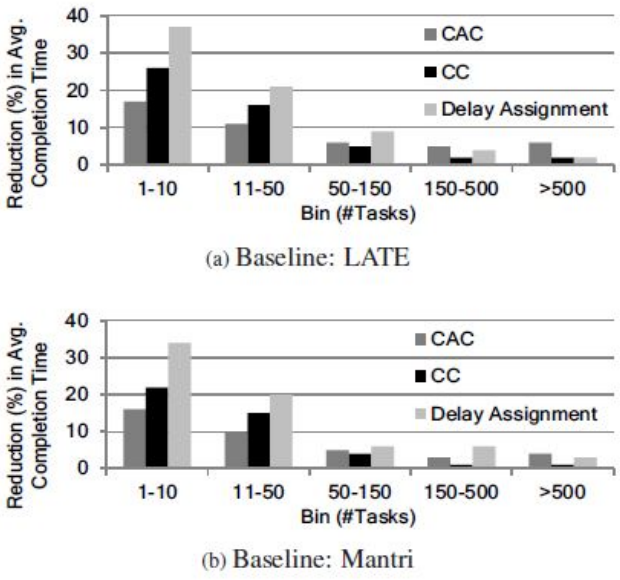


Figure 8: Intermediate data contention. Delay Assignment is 2.1x better than CAC and CC (Bing workload).

CC and CAC: We now compare [1] delay assignment

to the two static assignment schemes, Contention Cloning (CC) and Contention Avoidance Cloning (CAC) in Figure 11, for the Bing workload. With LATE as the baseline, CAC and CC improve the small jobs by 17% and 26%, in contrast to delay assignment's 37% improvement (or up to 2.1x better). With Mantri as the baseline, delay assignment is again up to 2.1x better. In the Facebook workload, delay assignment is at least 1.7x better. The main reason behind its better performance is its accurate estimation of the effect of contention and the likelihood of stragglers.

GRASS was implemented on top of two data-analytics frameworks, Hadoop and Spark, representing batch jobs and interactive jobs, respectively. Hadoop jobs read data from HDFS while Spark read from in-memory RDDs. As a result, Spark tasks finish faster than Hadoop even with same sized inputs. Implementing GRASS required two changes: task executor and job scheduler. Task executors were augmented to periodically report progress and job scheduler collects these reports, maintains samples of completed tasks and jobs, and decides the switching point. While the techniques are simple, the downside is the error in estimation. According to [2], their estimates of t_{rem} and t_{new} achieve moderate accuracies of 72% and 76%, respectively, on average.

GRASS was evaluated on a 200 node EC2 cluster. The results from [2] are listed below:

1. GRASS increases accuracy of deadline-bound jobs by 47% and speeds up error-bound jobs by 38%. Even non-approximation jobs speed up by 34%. Also, it nearly matches the optimal performance.

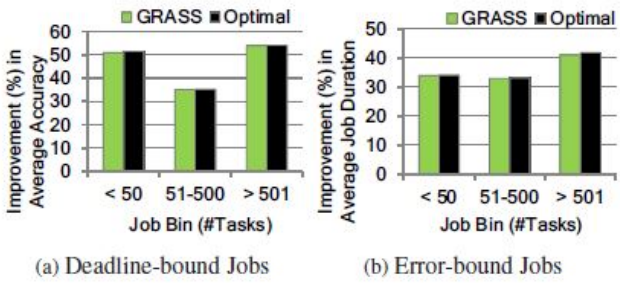
2. GRASS's learning based approach to determine the switching point from RAS to GS is over 30% better than simple strawman techniques.

To evaluate the performance of GRASS, Facebook's production Hadoop cluster and Microsoft Bing's production Dryad cluster were used. The traces capture over half a million jobs that ran across many months (appendix Table 1). An important thing to note here is that they did not use any approximation queries, which required them to complete all their tasks. Job Bins: Jobs were binned by their number of tasks. They used three distinctions "small" ($<$ 50 tasks), "medium" (51 - 500 tasks), and "large" ($>$ 500 tasks).

Deadline-Bound Jobs: GRASS improves the accuracy of deadline-bound jobs by 34% to 40% in the Hadoop prototype. Gains in both the Facebook and Bing workloads are similar.The gains compared (Appendix Figure 10) to LATE as baseline are consistently higher than Mantri. Also, the gains in larger jobs are pronounced compared to small and medium jobs because their higher number of waves of tasks provide plenty of potential for GRASS. The Spark prototype improves accuracy by 43% to 47%. The gains are higher because Spark's task sizes are much smaller than Hadoop's due to in-memory inputs. Again, large jobs gain the most, improving by over 50%. Better improvement of large multi-waved jobs is encouraging because smaller task sizes in future will ensure that multi-waved executions will be the norm.

Error-bound jobs: Similar to deadline-bound jobs, improvements with the Spark prototype (33% - 37%) are higher compared to the Hadoop prototype (24% - 30%). This shows that GRASS works well with both Hadoop and Spark. (App. Fig. 11)

Optimality of GRASS: Even though there were much better improvements compared to baseline, they tried to find if more improvement was possible after that. To understand the room available for improvement beyond GRASS, they compare its performance with an optimal scheduler that knows task duration and slot availabilities in advance.



(a) Deadline-bound Jobs (b) Error-bound Jobs

Figure 9: GRASS's gains matches the optimal scheduler.

Figure 8 shows that GRASS's performance matches with Optimal for both deadline and error-bound jobs. Thus, it is a near-optimal solution to the NP-hard problem of scheduling tasks for approximation jobs with speculative copies.

The Value of Switching and Learning: According to the experiment, switching from RAS to GS and the timing of this switch are crucial points. Results (App. Fig. 12 and 13) showed that GRASS's overall improvement in accuracy is over 20% better than the best of RAS or GS alone. It is also important to know when to switch from RAS to GS without affecting the performance. For static switching, small and medium jobs lag the most as wrong estimation of switching point affects a large fraction of their tasks. Thus, the benefit of adaptively determining the switching point is significant. Deadline/error-bound provides the best result when only one factor (deadline/error-bound or cluster utilization or estimation accuracy) is used to switch. When two factors are used, in addition to the deadline/error-bound, cluster utilization matters more for the Hadoop prototype while estimation accuracy is important for the Spark prototype. Thus, in the absence of a detailed model for job execution, the three factors act as good predictors.

7. CONCLUSION

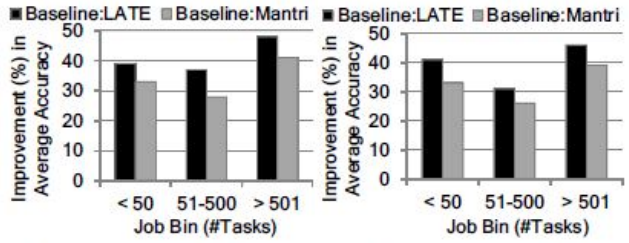
We mentioned about Dolly, a system that launches multiple clones of jobs and removing waiting from straggler mitigation. The main challenge of cloning was making the intermediate data transfer efficient, i.e., avoiding multiple tasks downstream in the job from contending for the same upstream output. Then, delay assignment was used to efficiently avoid such contention using a cost-benefit model. Evaluation using production workloads showed that Dolly sped up small jobs by 34% to 46% on average, after applying LATE and Mantri, using only 5% extra resources. Another speculation model we mentioned was GRASS that uses opportunity cost to determine when to speculate early in the job and then switches to more aggressive speculation as the job nears its approximation bound. GRASS improves accuracy for deadline-bound jobs by 47% and speeds up error-bound jobs by 38% in production workloads from Facebook and Bing. Further, the evaluation highlights that GRASS is a unified speculation solution for both approximation and exact computations, since it also provides a 34% speed up for exact jobs.

8. APPENDIX

	Facebook	Microsoft Bing
Dates	Oct 2010	May−Dec* 2009
Framework	Hadoop	Dryad
File System	HDFS	Cosmos
Script	Hive	Scope
Jobs	375K	200K
Cluster Size	3500	Thousands
Straggler mitigation	LATE	Mantri

* *One week in each month*

Table 1: Frequency of Special Characters



(a) Facebook Workload–Hadoop (b) Bing Workload–Hadoop

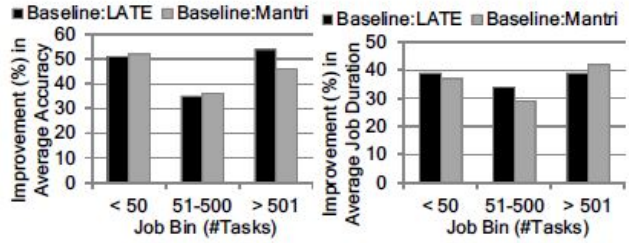
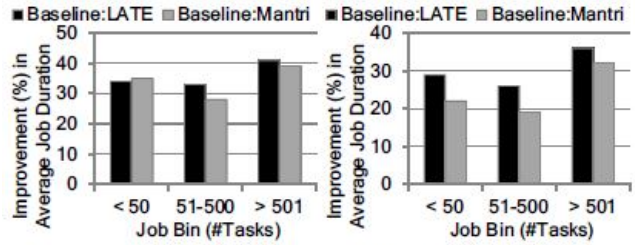
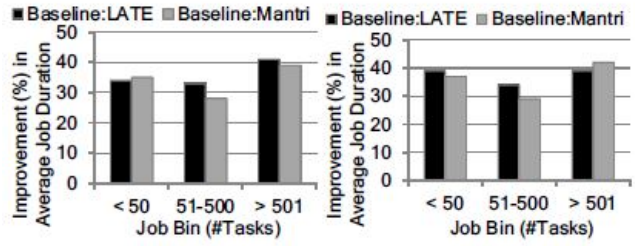


Figure 10: Accuracy Improvement in deadline-bound jobs with LATE and Mantri as baselines.

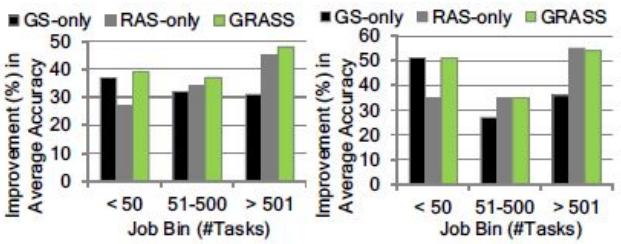


(a) Facebook Workload–Hadoop (b) Bing Workload–Hadoop



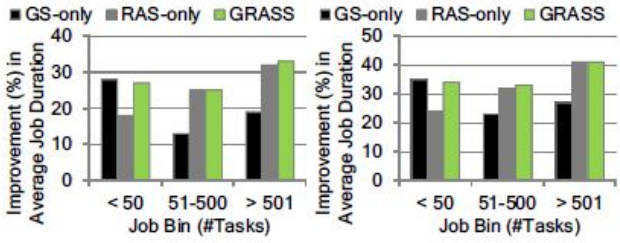
(c) Facebook Workload–Spark (d) Bing Workload–Spark

Figure 11: Speedup in error-bound jobs with LATE and Mantri as baselines.



(a) Hadoop (b) Spark

Figure 12: GRASS's switching is 25% better than using GS or RAS all through for deadline-bound jobs. We use the Facebook workload and LATE as baseline.



(a) Facebook Workload–Hadoop (b) Facebook Workload–Spark

Figure 13: GRASS's switching is 20% better than using GS or RAS all through for error-bound jobs. We use the Facebook workload and LATE as baseline.

9. REFERENCES

[1] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective straggler mitigation: Attack of the clones. *NSDI*, pages 185–198, April 2013.

[2] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu. Grass: Trimming stragglers in approximation analytics. *NSDI*, pages 289–302, April 2014.

[3] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, E. Harris, and B. Saha. Reining in the outliers in map-reduce clusters using mantri. *USENIX OSDI*, 2010.

[4] A. Bhandare, J. George, S. Deshpande, and Y. Karle. Review and analysis of straggler handling techniques. *IJCSIT*, 7(5):2270–2276, 2016.

[5] J. Dean. Achieving rapid response times in large online services. http://research.google.com/people/jeff/latency.html.

[6] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. *USENIX OSDI*, 2008.