GRASS: Trimming Stragglers in Approximation Analytics

Ganesh Ananthanarayanan, *University of California, Berkeley;* **Michael Chien-Chun Hung,** *University of Southern California;* **Xiaoqi Ren,** *California Institute of Technology;* **Ion Stoica,** *University of California, Berkeley;* **Adam Wierman,** *California Institute of Technology;* **Minlan Yu,** *University of Southern California*

https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/ananthanarayanan

This paper is included in the Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14).

April 2–4, 2014 • Seattle, WA, USA

ISBN 978-1-931971-09-6

Open access to the Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14) is sponsored by USENIX

GRASS: Trimming Stragglers in Approximation Analytics

Ganesh Ananthanarayanan[1], Michael Chien-Chun Hung[2], Xiaoqi Ren[3], Ion Stoica[4], Adam Wierman[3], Minlan Yu[2]

[1]*Microsoft Research*, [2]*University of Southern California*, [3]*California Institute of Technology*,
[4]*University of California, Berkeley*

ga@microsoft.com, {chienchun.hung, minlanyu}@usc.edu, {xren, adamw}@caltech.edu, istoica@cs.berkeley.edu

Abstract

In big data analytics, timely results, even if based on only part of the data, are often *good enough*. For this reason, *approximation* jobs, which have deadline or error bounds and require only a subset of their tasks to complete, are projected to dominate big data workloads. Straggler tasks are an important hurdle when designing approximate data analytic frameworks, and the widely adopted approach to deal with them is speculative execution. In this paper, we present GRASS, which carefully uses speculation to mitigate the impact of stragglers in approximation jobs. GRASS's design is based on first principles analysis of the impact of speculation. GRASS delicately balances immediacy of improving the approximation goal with the long term implications of using extra resources for speculation. Evaluations with production workloads from Facebook and Microsoft Bing in an EC2 cluster of 200 nodes shows that GRASS increases accuracy of deadline-bound jobs by 47% and speeds up error-bound jobs by 38%. GRASS's design also speeds up exact computations (zero error-bound), making it a *unified solution for straggler mitigation*.

1 Introduction

Large scale data analytics frameworks automatically compose *jobs* operating on large data sets into many small *tasks* and execute them in parallel on *compute slots* on different machines. A key feature catalyzing the widespread adoption of these frameworks is their ability to guard against failures of tasks, both when tasks fail outright as well as when they run slower than the other tasks of the job. Dealing with the latter, referred to as *stragglers*, is a crucial design component that has received widespread attention across prior studies [1, 2, 3].

The dominant technique to mitigate stragglers is *speculation*—launching speculative copies for the slower tasks, where a speculative copy is simply a duplicate of the original task. It then becomes a race between the original and the speculative copies. Such techniques are state-of-the-art and deployed in production clusters at Facebook and Microsoft Bing, thereby significantly speeding up jobs. The focus of this paper is on speculation for an emerging class of jobs: *approximation* jobs.

Approximation jobs are starting to see considerable interest in data analytics clusters [4, 5, 6]. These jobs are based on the premise that providing a timely result, even if only on part of the dataset, is more important than processing the entire data. These jobs tend to have *approximation bounds* on two dimensions—deadline and error [7]. *Deadline-bound* jobs strive to maximize the accuracy of their result within a specified time deadline. *Error-bound* jobs, on the other hand, strive to minimize the time taken to reach a specified error limit in the result. Typically, approximation jobs are launched on a large dataset and require only a *subset* of their tasks to finish based on the bound [8, 9, 10].

Our focus is on the problem of speculation for approximation jobs.[1] Traditional speculation techniques for straggler mitigation face a fundamental limitation when dealing with approximation jobs, since they do not take into account approximation bounds. Ideally, when the job has many more tasks than compute slots, we want to prioritize those tasks that are likely to complete within the deadline or those that contribute the earliest to meeting the error bound. By not considering the approximation bounds, state-of-the-art straggler mitigation techniques in production clusters at Facebook and Bing fall significantly short of optimal mitigation. They are 48% lower in average accuracy for deadline-bound jobs and 40% higher in average duration of error-bound jobs.

Optimally prioritizing tasks of a job to slots is a classic scheduling problem with known heuristics [11, 12, 13]. These heuristics, unfortunately, do not directly carry over to our scenario for the following reasons. First, they calculate the optimal ordering statically. Straggling of tasks, on the other hand, is unpredictable and necessitates dynamic modification of the priority ordering of tasks according to the approximation bounds. Second, and most importantly, traditional prioritization techniques assign tasks to slots assuming every task to occupy only one slot. Spawning a speculative copy, however, leads to the same task using two (or multiple) slots simultaneously. Hence, this distills our challenge

[1]Note that an error-bound job with error of zero is the same as an exact job that requires all its tasks to complete. Hence, by focusing on approximation jobs, we automatically subsume exact computations.

to *achieving the approximation bounds by dynamically weighing the gains due to speculation against the cost of using extra resources for speculation*.

Scheduling a speculative copy helps make immediate progress by finishing a task faster. However, while not scheduling a speculative copy results in the task running slower, many more tasks may be completed using the saved slot. To understand this *opportunity cost*, consider a cluster with one unoccupied slot and a straggler task. Letting the straggler complete takes five more time units while a new copy of it would take four time units. While scheduling a speculative copy for this straggler speeds it up by one time unit, if we were not to, that slot could finish another task (taking five time units too).

This simple intuition of opportunity cost forms the basis for our two design proposals. First, Greedy Speculative (GS) scheduling is an algorithm that *greedily* picks the task to schedule next (original or speculative) that most improves the approximation goal at that point. Second, Resource Aware Speculative (RAS) scheduling considers the opportunity cost and schedules a speculative copy only if doing so saves both time *and* resources.

These two designs are motivated by first principles analysis within the context of a theoretical model for studying speculative scheduling. An important guideline from our model is that the value of being greedy (GS) increases for smaller jobs while considering opportunity cost of speculation (RAS) helps for larger jobs. As our model is generic, a nice aspect is that the guideline holds not only for approximation jobs but also for exact jobs that require all their tasks to complete.

We use the above guideline to dynamically combine GS and RAS, which we call GRASS. At the beginning of a job's execution, GRASS uses RAS for scheduling tasks. Then, as the job gets *close* to its approximation bound, it switches to GS, since our theoretical model suggests that the opportunity cost of speculation diminishes with fewer unscheduled tasks in the job. GRASS learns the point to switch from RAS to GS using job and cluster characteristics.

We demonstrate the generality of GRASS by implementing it in both Hadoop [14] (for batch jobs) and Spark [15] (for interactive jobs). We evaluate GRASS using production workloads from Facebook and Bing on an EC2 cluster with 200 machines. GRASS increases accuracy of deadline-bound jobs by 47% and speeds up error-bound jobs by 38% compared to state-of-the-art straggler mitigation techniques deployed in these clusters (LATE [2] and Mantri [1]). In fact, GRASS results in near-optimal performance. In addition, GRASS also speeds up exact jobs, that require all their tasks to complete, by 34%. Thus, it is a *unified speculation solution for both approximation as well as exact computations*.

2 Challenges and Opportunities

Before presenting our system design, it is important to understand the challenges and opportunities for speculating straggler tasks in the context of approximation jobs.

2.1 Approximation Jobs

Increasingly, with the deluge of data, analytics applications no longer require processing entire datasets. Instead, they choose to tradeoff accuracy for response time. *Approximate* results obtained early from just part of the dataset are often *good enough* [4, 6, 5]. Approximation is explored across two dimensions—time for obtaining the result (deadline) and error in the result [7].

- *Deadline-bound* jobs strive to maximize the accuracy of their result within a specified time limit. Such jobs are common in real-time advertisement systems and web search engines. Generally, the job is spawned on a large dataset and accuracy is proportional to the fraction of data processed [8, 9, 10] (or tasks completed, for ease of exposition).

- *Error-bound* jobs strive to minimize the time taken to reach a specified error limit in the result. Again, accuracy is measured in the amount of data processed (or tasks completed). Error-bound jobs are used in scenarios where the value in reducing the error below a limit is marginal, *e.g.,* counting of the number of cars crossing a section of a road to the nearest thousand is sufficient for many purposes.

Approximation jobs require schedulers to *prioritize the appropriate subset of their tasks* depending on the deadline or error bound. Prioritization is important for two reasons. First, due to cluster heterogeneities [2, 3, 16], tasks take different durations even if assigned the same amount of work. Second, jobs are often *multi-waved*, i.e., their number of tasks is much more than available compute slots, thereby they run only a fraction of their tasks at a time [17]. For example, when a job with 1000 tasks is given only 100 slots simultaneously (due to, say, fair scheduling), it runs only one-tenth of its tasks at a time. These tasks, though, are independent and can be scheduled in any order. The trend of multi-waved jobs is expected to grow with smaller tasks [18].

2.2 Challenges

The main challenge in prioritizing tasks of approximation jobs arises due to some of them *straggling*. Even after applying many proactive techniques, in production clusters in Facebook and Microsoft Bing, the average

job's slowest task is eight times slower than the median.[2] It is difficult to model all the complex interactions in clusters to prevent stragglers [3, 20]. Ananthanarayanan et al. (Section 2.1.2 in [3]) also show that blacklisting machines based on their likeliness to cause stragglers (in both the short- as well as long-term) has little benefits; machines are neither consistently problematic nor exhibit simple correlations with task durations.

The widely adopted technique to deal with straggler tasks is *speculation*. This is a reactive technique that spawns speculative copies for tasks deemed to be straggling. The earliest among the original and speculative copies is picked while the rest are killed. While scheduling a speculative copy makes the task finish faster and thereby increases accuracy, they also compete for compute slots with the unscheduled tasks.

Therefore, our problem is to *dynamically prioritize tasks based on the deadline/error-bound while choosing between speculative copies for stragglers and unscheduled tasks*. This problem is, unfortunately, NP-Hard and devising good heuristics (i.e., with good approximation factors) is an open theoretical problem.

2.3 Potential Gains

Given the challenges posed by stragglers discussed above, it is not surprising that the potential gains from mitigating their impact are significant. To highlight this we use a simulator with an optimal bin-packing scheduler. Our baselines are the the state-of-the-art mitigation strategies (LATE [2] and Mantri [1]) in the production clusters. Optimally prioritizing the tasks while correctly balancing between speculative copies and unscheduled tasks presents the following potential gains. Deadline-bound jobs improve their accuracy by 48% and 44%, in the Facebook and Bing traces, respectively. Error-bound jobs speed up by 32% and 40%. We next develop an online heuristic to achieve these gains.

3 Speculation Algorithm Design

The key choice made by a cluster scheduling algorithm is to pick the next task to schedule given a vacant slot. Traditionally, this choice is made among the set of tasks that are queued; however when speculation is allowed, the choice also includes speculative copies of tasks that are already running. This extra flexibility means that a design must determine a prioritization that carefully weighs the gains from speculation against the cost of extra resources while still meeting the approximation goals. Thus, we first focus on tradeoffs in the design of the speculation policy. Specifically, using both small examples and analytic modeling we motivate the use of two simple heuristics, Greedy Speculative (GS) scheduling and Resource Aware Speculative (RAS) scheduling that together make up the core of GRASS.

[2]Task durations are normalized by their input sizes to be resistant to data skews [19, 1].

3.1 Speculation Alternatives

For simplicity, we first introduce GS and RAS in the context of deadline-bound jobs and then briefly describe how they can be adapted to error-bound jobs.

3.1.1 Deadline-bound Jobs

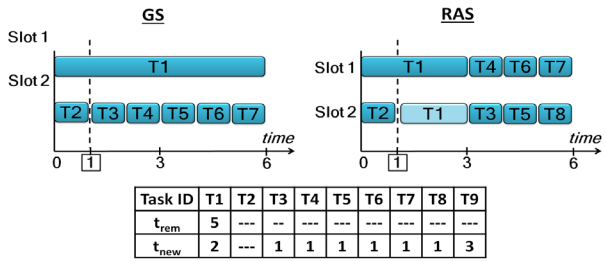
If speculation was not allowed, there is a natural, well-understood policy for the case of deadline-bound jobs: Shortest Job First (SJF), which schedules the task with the smallest processing time. In many settings, SJF can be proven to minimize the number of incomplete tasks in the system, and thus maximize the number of tasks completed, at all points of time among the class of non-preemptive policies [11, 12]. Thus, without speculation, SJF finishes the most tasks before the deadline.

If one extends this idea to the case where speculation is allowed, then a natural approach is to allow the currently running tasks to also be placed in the queue, and to choose the task with the smallest size, i.e., t_{new} (requiring, of course, that the task finishes before the deadline). If the chosen task has a copy currently running, we check that the speculative copy being considered provides a benefit, i.e., $t_{new} < t_{rem}$. So, the next task to run is still chosen according to SJF, only now speculative copies are also considered. We term this policy *Greedy Speculative (GS) scheduling*, because it picks the next task to schedule *greedily*, i.e., the one that will finish the quickest, and thus improve the accuracy the earliest *at present*.

Figure 1 (left) presents an illustration of GS for a simple job with nine tasks and two concurrent slots. Tasks T1 and T2 are scheduled first, and when T2 finishes, the t_{rem} and t_{new} values are as indicated. At this point, GS schedules T3 next as it is the one with the lowest t_{new}, and so forth. Assuming the deadline was set to 6 time units, the obtained accuracy is $\frac{7}{9}$ (or 7 completed tasks).

Picking the earliest task to schedule next is often optimal when a job has no unscheduled tasks (i.e., either single-waved jobs or the last wave of a multi-waved job). However, when there are unscheduled tasks it is less clear. For example, in Figure 1 (right) if we schedule a speculative copy of T1 when T2 finished, instead of T3, 8 tasks finish by the deadline of 6 time units.

The previous example highlights that running a speculative copy has resource implications which are important to consider. If the speculative copy finishes early, both slots (of the speculative copy and the original) be-



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 1: **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.**

come available sooner to start the other tasks. This *opportunity cost* of speculation is an important tradeoff to consider, and leads to the second policy we consider: *Resource Aware Speculative (RAS) scheduling*.

To account for the opportunity cost of scheduling a speculative copy, RAS speculates only if it saves both time *and* resources. Thus, not only must t_{new} be less than t_{rem} to spawn a speculative copy but 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. In other words, for a task with c running copies, its resource savings, defined as $c \times t_{rem} - (c+1) \times t_{new}$, must be positive.

By accounting for the opportunity cost of resources, RAS can out-perform GS in many cases. As mentioned earlier, in Figure 1 (right) where RAS achieves an accuracy of $\frac{8}{9}$ versus GS's $\frac{7}{9}$ in the deadline of 6 time units. This improvement comes because, when T2 finishes, speculating on T1 saves 1 unit of resource.

However, RAS is not uniformly better than GS. In particular, RAS's cautious approach can backfire if it overestimates the opportunity cost. In the same example in Figure 1, if the deadline of the job 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.

As the example alludes to, the value of the deadline and the number of waves are two important factors impact whether GS or RAS is a better choice. A third important factor, which we discuss later in §4.1, is the estimation accuracy of t_{rem} and t_{new}.

Pseudocode 1 describes the details of GS and RAS. The set T consists of all the running and unscheduled tasks of the jobs. There are two stages in the scheduling process: (i) *Pruning Stage:* In this stage (lines $5-12$), tasks that are not slated to complete by the deadline are removed from consideration. Further, GS removes those tasks whose speculative copy is not expected to finish earlier than the running copy. RAS removes those tasks

```
1:  procedure DEADLINE(⟨Task⟩ T, float δ, bool OC)
                          ▷ OC = 1 → use RAS; 0 → use GS
2:      if OC then
3:          for each Task t in T do
4:              if t.running then
                    t.saving = t.c ×t.t_rem − (t.c+1) × t_new
                                                ▷ PRUNING STAGE
        δ' ← Remaining Time to δ
        ⟨Task⟩Γ ← φ
5:      for each Task t in T do
6:          if t.t_new > δ' then continue    ▷ Exceeds deadline
7:          if OC then
8:              if t.saving > 0 then Γ.add(t)
9:          else
10:             if t.running then
11:                 if t.t_new < t.t_rem then Γ.add(t)
12:             else Γ.add(t)
                                            ▷ SELECTION STAGE
13:     if Γ ≠ null then
14:         if OC then SortDescending(Γ, "saving")
15:         else SortAscending(Γ, t_new)
            return Γ.first()
```

Pseudocode 1: **GS and RAS algorithms for deadline-bound jobs (deadline of δ). T is the set of unfinished tasks with the following fields per task: t_{rem}, t_{new}, and a boolean "running" to denote if a copy of it is currently executing. RAS is used when OC is set. At default, both algorithms schedule the task with the lowest t_{new} within the deadline.**

which do not save on resources by speculation. (ii) *Selection Stage:* From the pruned set, GS picks the task with the lowest t_{new} while RAS picks the task with the highest resource savings (lines $13-15$).

3.1.2 Error-bound Jobs

Though error-bound jobs require a different form of prioritization than deadline-bound jobs, the speculative core of the GS and RAS algorithms are again quite natural. Specifically, 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 the natural prioritization of tasks. In particular, LJF minimizes the makespan among the class of non-preemptive policies in many settings [11, 12]. This again represents a "greedy" prioritization for this setting.

Despite the above change to the prioritization of which task to schedule, the form of GS and RAS remain the same as in the case of deadline-bound jobs. In particular, speculative copies are evaluated in the same manner, e.g., RAS's criterion is still to pick the task whose speculation leads to the highest resource savings. Pseudocode 2 presents the details. The pruning stage (lines $5-11$) will remove from consideration those tasks that are not the earliest to contribute to the desired error bound. The

```
1:  procedure ERROR(⟨Task⟩ T, float ϵ, bool OC)
                          ▷ OC = 1 → use RAS; 0 → use GS
                                  ▷ Error ϵ is in #tasks
2:      for each Task t in T do
            t.duration = min(t.t_rem, t.t_new)
3:          if OC then
4:              if t.running then
                    t.saving = t.c × t.t_rem − (t.c+1) × t_new
                                          ▷ PRUNING STAGE
        SortAscending(T, "duration")
        ⟨Task⟩Γ ← ϕ
5:      for each Task t in T[0 : T.count() (1 − ϵ)] do
                                          ▷ Earliest tasks
6:          if OC then
7:              if t.saving > 0 then Γ.add(t)
8:          else
9:              if t.running then
10:                 if t.t_new < t.t_rem then Γ.add(t)
11:             else Γ.add(t)
                                      ▷ SELECTION STAGE
12:     if Γ ≠ null then
13:         if OC then SortDescending(Γ, "saving")
14:         else SortDescending(Γ, t_rem)
            return Γ.first()
```

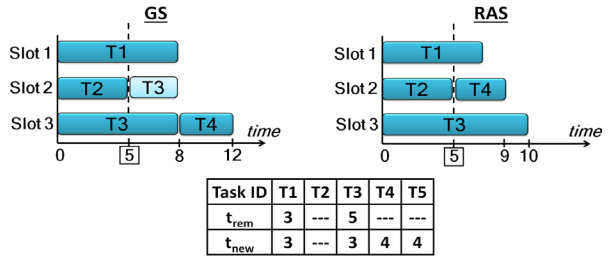
Pseudocode 2: **GS and RAS speculation algorithms for error-bound jobs (error-bound of ϵ). T is the set of unfinished tasks with the following fields per task: t_{rem}, t_{new}, and a boolean "running" to denote if a copy of it is currently executing. The t_{rem} of the task is the minimum of all its running copies. RAS is used when OC is set. At default, both algorithms schedule the task with the highest t_{rem}.**

list of earliest tasks is based on the effective duration of every task, i.e., the minimum of t_{rem} and t_{new}. During selection (lines $12-14$), GS picks the task with the highest t_{rem} while RAS picks the task with the highest saving.

Figure 2 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%).

3.2 Contrasting GS and RAS

To this point, we have seen that GS and RAS are two natural approaches for integrating speculation into a cluster scheduler for approximation jobs. However, the examples we have considered highlight that neither of GS or RAS is uniformly better. In order to develop a better understanding of these two algorithms, as well as other possible alternatives, we have developed a simple analytic model for speculation in approximation jobs. The model assumes wave-based scheduling and constant



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

Figure 2: **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).**

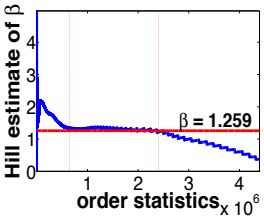


Figure 3: **Hill plot of Facebook task durations.**

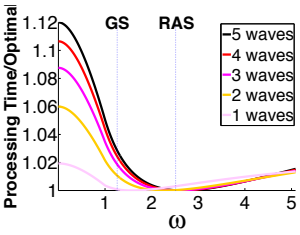


Figure 4: **Near-optimality of GS & RAS under Pareto task durations ($\beta = 1.259$).**

wave-width for a job (see §A for details along with formal results). For readability, here we present only the three major guidelines from our analysis. Most importantly, these guidelines highlight that different speculation policies are required during the early waves of a job than during the final wave.

Guideline 1 *During the early waves of a job, speculation is only valuable if task durations are extremely heavy tailed, e.g., Pareto with infinite variance (i.e., with shape parameter $\beta < 2$). In this case, it is optimal to speculate conservatively, using ≤ 2 copies of a task.*

This guideline is relevant because task durations are indeed heavy-tailed for the Facebook and Bing traces (see the Hill plot in Figure 3), which suggests that task durations have a Pareto tail (i.e., $P(\tau > x) = \theta(x^{-\beta})$) with shape parameter $\beta = 1.259$.[3] While both GS and RAS speculate during early waves, RAS is more conservative than GS and thus outperforms it during early waves.

Guideline 2 *During the final wave of a job, speculate aggressively to fully utilize the allotted capacity.*

[3] A Hill plot provides a more robust estimation of Pareto distributions than the, more commonly used, regression on a log-log plot [21]. To interpret the plot, a flat region corresponds to an estimate of β. The fact that the curve in Figure 3 is flat over a large range of order statistics (on the x-axis), but not all order statistics, indicates that the distribution of task sizes is not exactly Pareto distribution in its body, but is well-approximated by a Pareto (power-law) tail.

This guideline says that, even if all tasks are currently scheduled, if a slot becomes available it should be filled with a speculative copy. While both GS and RAS do this to some extent, GS speculates more aggressively than RAS and thus, outperforms RAS during the final wave.

The previous two guidelines highlight a tradeoff between RAS and GS, which we formalize next.

Guideline 3 *For jobs that require more than two waves RAS is near-optimal, while for jobs that require fewer than two waves GS is near-optimal.*

To make this point more salient, consider the general class of speculative replication policies that waits until a task has run ω time before starting a speculative copy. We study this broad class in §A, and GS and RAS correspond to particular rules for how to choose ω. To see this, we can define $t_{\text{new}} = E[\tau]$ and $t_{\text{rem}} = E[\tau - \omega | \tau > \omega]$, where τ is a random task size. Then, under GS, ω is the time when $E[\tau] = E[\tau - \omega | \tau > \omega]$, and, under RAS, ω is the time when $2E[\tau] = E[\tau - \omega | \tau > \omega]$.

Figure 4 contrasts the performance of all the replication policies in this more general class. Specifically, it shows the ratio of the response time of the replication policy with parameter ω normalized to the optimal response time. It illustrates this ratio for jobs of differing numbers of waves, and for $\omega \in [0, 5]$. To highlight GS and RAS, they are shown via vertical lines. The response times shown in the figure are computed using the model and analysis described in §A. The main conclusion from this figure is, as described in the guideline above, that neither GS or RAS is universally optimal, but each is near-optimal for jobs with a certain number of waves: RAS for jobs with large numbers of waves and GS for jobs with small numbers of waves.

4 GRASS Speculation Algorithm

In this section, we build our speculation algorithm called GRASS.[4] Our theoretical analysis summarized in §3.2 motivates a design that uses RAS during the early waves of jobs and GS during the final two waves. A simple *strawman* solution to achieve this would be as follows. For deadline-bound jobs, switch from RAS to GS when the time to the deadline is sufficient for at most two waves of tasks. Similarly, for error-bound jobs, switch when the number of (unique) scheduled tasks needed to satisfy the error-bound makes up two waves.

Identifying the final two waves of tasks is difficult in practice. Tasks are not scheduled at explicit wave boundaries but rather as and when slots open up. In addition, the wave-width of jobs does not stay constant but varies considerably depending on cluster utilization. Finally, task durations are varied and hard to estimate.

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, while otherwise GS performs better. This mimics the intuition from the examples in §3.1. Therefore, GRASS seeks to switch from RAS to GS as it gets *close to the job's approximation bound*.

The complexities in these systems mean that precise estimates of the optimal switching point cannot be obtained from our model. Instead, we adopt an indirect learning based approach where inferences are made based on executions of previous jobs (with similar number of tasks) and cluster characteristics (utilization and estimation accuracy). We compare our learning approach to the strawman described above in §6.3.

4.1 Learning the Switching Point

An ideal approach would accumulate enough samples of job performance (accuracy or completion time) based on switching to GS at different points. For deadline-bound jobs, this is decided by the remaining time to the deadline. For error-bound jobs, this is decided by the number of tasks to complete towards meeting the error. To speed up our sample collection, instead of accumulating samples of switching to GS, we simply generate samples of job performance using GS or RAS *throughout* the job (described shortly in §4.2).

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 point (measured in time remaining or tasks to complete). It steps through all possible points in its remaining work at which it could switch and estimates the optimal point using job samples of appropriate sizes. It continues with RAS until the optimal switching point turns out to be *at present*. The above calculation for the optimal switching point is performed periodically during the job's execution.

For example, when a deadline-bound job has 6s of its deadline remaining, GRASS compares the potential accuracy obtained if it were to switch at each point in its future (at 1s granularity). The accuracy if it were to switch after, say, 2s is the sum of accuracies of jobs with deadlines of 2s that used only RAS and those with 4s that used only GS. Switching happens if among all such points, the best accuracy is obtained by switching now.

The size of the job alone is insufficient to calculate the optimal switching point. Even jobs of comparable size might have different number of waves depending on the number of available slots. Therefore, we augment our samples of job performance with the number of waves, simply approximated using current *cluster utilization*.

[4] GRASS is a concatenation of GS and RAS

Finally, *estimation accuracy* of t_{rem} and t_{new} also decides the optimal switching point. RAS's cautious approach of considering the opportunity cost of speculating a task is valuable when task estimates are erroneous. In fact, at low estimation accuracies (along with certain values of utilization and deadline/error-bound), it is better to not switch to GS at all and employ RAS all along. §6.3.2 analyzes the impact of these three factors.

Therefore, 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. It uses these three factors collectively to decide when (and if) to switch from RAS to GS. We next describe how the samples are collected.

4.2 Generating Samples

As described above, GRASS compares samples of job performance that use only GS or RAS throughout, to decide when to switch. These samples have to be updated continuously to stay abreast with dynamic changes in clusters. To continuously generate such samples, we introduce a *perturbation* in GRASS's switching decision. With a small probability ξ, GRASS decides to *not* switch and instead picks one of GS or RAS for the entire duration of the job (both GS and RAS are equally probable). Such perturbation helps us obtain comparable samples.

The crucial trade-off in setting ξ is in balancing the benefit of obtaining such comparable samples with the performance loss incurred by the job due to not making the right switching decision. Theoretical analyses of such multi-armed bandit problems in prior work defines an optimal value of ξ by making stochastic assumptions about the distribution of the costs and the associated rewards [22]. Our setup, however, does not yield itself to such assumptions as the underlying distribution can be arbitrary. Another class of techniques that we considered modified ξ with time [23]. Over time, the value of ξ is gradually reduced using a damping function, thus indicating higher confidence in the learned value. We decided against such damping of ξ because clusters constantly evolve with new software and hardware modules, leading to newer interactions between them.

Therefore, we pick a constant value of ξ using empirical analysis. A job is marked for generating performance samples with a probability of ξ, and we pick GS or RAS with equal probability. In practice, we bucket jobs by their number of tasks and compare only within jobs of the same bucket.

5 Implementation

We implement GRASS on top of two data-analytics frameworks, Hadoop (version 0.20.2) [14] and Spark (version 0.7.3) [15], representing batch jobs and interactive jobs, respectively. Hadoop jobs read data from HDFS while Spark jobs read from in-memory RDDs. Consequently, Spark tasks finished quicker than Hadoop tasks, even with the same input size. Note that while Hadoop and Spark use LATE[2] currently, we also implement Mantri[1] to use as a second baseline.

Implementing GRASS required two changes: task executors and job scheduler. Task executors were augmented to periodically report progress. We piggyback on existing update mechanisms of tasks that conveyed only their start and finish. Progress reports were configured to be sent every 5% of data read/written. The job scheduler collects these reports, maintains samples of completed tasks and jobs, and decides the switching point.

5.1 Task Estimators

GRASS uses two estimates for tasks: remaining duration of a running task (t_{rem}) and duration of a new copy (t_{new}).
Estimating t_{rem}: Tasks periodically update the scheduler with *progress reports* containing the fraction of input read and output written. Since tasks of analytics jobs are IO-intensive, we extrapolate the remaining duration of the task based on the time elapsed thus far.
Estimating t_{new}: We estimate the duration of a new task by sampling from durations of completed tasks (normalized to input and output sizes). The t_{new} values of all tasks are updated whenever a task completes.
Accuracy of estimation: While the above techniques are simple, the downside is the error in estimation. Our estimates of t_{rem} and t_{new} achieve moderate accuracies of 72% and 76%, respectively, on average. When a task completes, we update the accuracy using the estimated and actual durations. GRASS uses the accuracy of estimation to appropriately switch from RAS to GS.

5.2 DAG of Tasks

Jobs are typically composed as a DAG of tasks with *input* tasks (*e.g.,* map or extract) reading data from the underlying storage and *intermediate* tasks (*e.g.,* reduce or join) aggregating their outputs. Even in DAGs of tasks, the accuracy of the result is decided by the fraction of completed input tasks. This makes GRASS's functioning straightforward in error-bound jobs—complete as many input tasks as required to meet the error-bound and all intermediate tasks further in the DAG.

For deadline-bound jobs, we use a widely occurring property that intermediate tasks perform similar functions across jobs. Further, they have relatively fewer stragglers. Thus, we estimate the time taken for intermediate tasks by comparing jobs of similar sizes and then subtract it to obtain the deadline for the input tasks.

Input tasks of a job, typically, read equal amounts of data. Thus, the fraction of tasks completed represents fraction of data processed too, thus making it a good indicator of the result's accuracy.

6 Evaluation

We evaluate GRASS on a 200 node EC2 cluster. Our focus is on quantifying the performance improvements compared to current designs, i.e., LATE [2] and Mantri [1], and on understanding how close to the optimal performance GRASS comes. Our main results can be summarized as follows.

1. GRASS increases accuracy of deadline-bound jobs by 47% and speeds up error-bound jobs by 38%. Even non-approximation jobs (i.e., error-bound of zero) speed up by 34%. Further, GRASS nearly matches the optimal performance. (§6.2)

2. GRASS's learning based approach for determining when to switch from RAS to GS is over 30% better than simple strawman techniques. Further, the use of all three factors discussed in §4.1 is crucial for inferring the optimal switching point. (§6.3)

6.1 Methodology

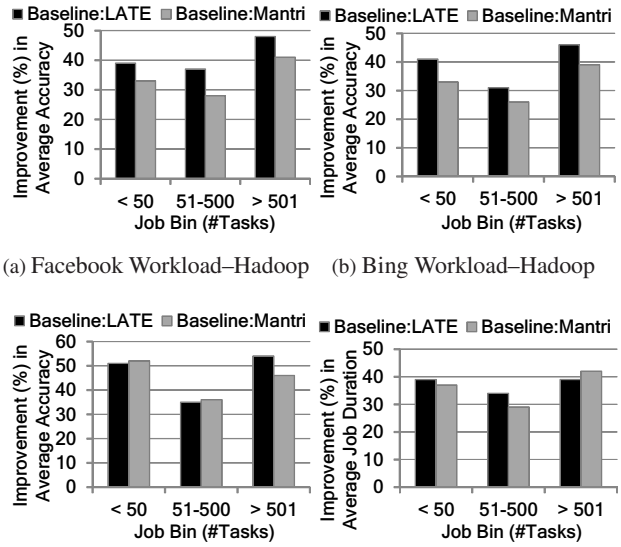
Workload: Our evaluation is based on traces from Facebook's production Hadoop [14] cluster and Microsoft Bing's production Dryad [24] cluster. The traces capture over half a million jobs running across many months (Table 1). The clusters run a mix of interactive and production jobs whose performance have significant impact on productivity and revenue. The jobs had diverse resource requirements of CPU, memory and IO. To create our experimental workload, we retain the inter-arrival times, input files and number of tasks of jobs. The jobs were, however, not approximation queries and required all their tasks to complete. Hence, we convert the jobs to mimic deadline- and error-bound jobs as follows.

For experiments on error-bound jobs, we pick the error tolerance of the job randomly between 5% and 30%. This is consistent with the experimental setup in recently reported research [4, 25]. Prior work also recommends setting deadlines by calibrating task durations [4, 9]. For the purpose of calibration, we obtain the ideal duration of a job in the trace by substituting the duration of each of its task by the median task duration in the job, again, as per recent work on straggler mitigation [3]. We set the deadline to be an additional factor (randomly between 2% to 20%) on top of this ideal duration.

Job Bins: We bin jobs by their number of tasks. We use three distinctions "small" (< 50 tasks), "medium" ($51 - 500$ tasks), and "large" (> 500 tasks).

	Facebook	Microsoft Bing
Dates	Oct 2012	May-Dec 2011
Framework	Hadoop	Dryad
Script	Hive [26]	Scope [27]
Jobs	575K	500K
Cluster Size	3,500	Thousands
Straggler–mitigation	LATE [2]	Mantri [1]

Table 1: **Details of Facebook and Bing traces.**



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

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

Figure 5: **Accuracy Improvement in deadline-bound jobs with LATE [2] and Mantri [1] as baselines.**

EC2 Deployment: We deploy our Hadoop and Spark prototypes on a 200-node EC2 cluster and evaluate them using the workloads described above. Each experiment is repeated five times and we pick the median. We measure improvement in the average accuracy for deadline-bound jobs and average duration for error-bound jobs.

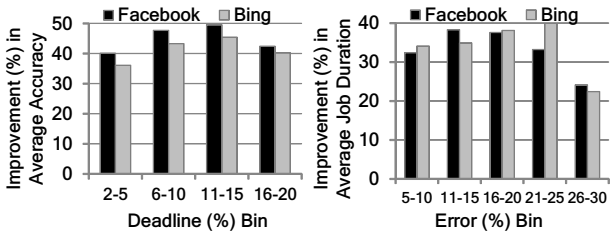
We also use a trace-driven simulator to evaluate at larger scales and over longer durations. The simulator replays all the task properties including their straggling.
Baseline: We contrast GRASS with two state-of-the-art speculation algorithms—LATE [2] and Mantri [1].

6.2 Improvements from GRASS

We contrast GRASS's performance with that of LATE [2], Mantri [1], and the optimal scheduler.

6.2.1 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. Figure 5a and 5b split the gains by job size. The gains compared



(a) Deadline Bins (b) Error Bins

Figure 6: GRASS**'s overall gains (compared to LATE) binned by the deadline and error bound. Deadlines are binned by the factor over ideal job duration (see §6.1)**

to LATE as baseline are consistently higher than Mantri. Also, the gains in large jobs are pronounced compared to small and medium sized jobs because their many waves of tasks provides 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. This makes the effect of stragglers more distinct. Again, large jobs gain the most, improving by over 50% (Figure 5c and 5d). Large multi-waved jobs improving more is encouraging because smaller task sizes in future [18] will ensure that multi-waved executions will be the norm. Unlike the Hadoop case, gains compared to both LATE and Mantri are similar because both have only limited effect when the impact of stragglers is high.

Figure 6a dices the improvements by the deadline (specifically, the additional factor over the ideal job duration (see §6.1)). Note that gains are nearly uniform across deadline values. This indicates that GRASS can not only cope with stringent deadlines but be valuable even when the deadline is lenient.

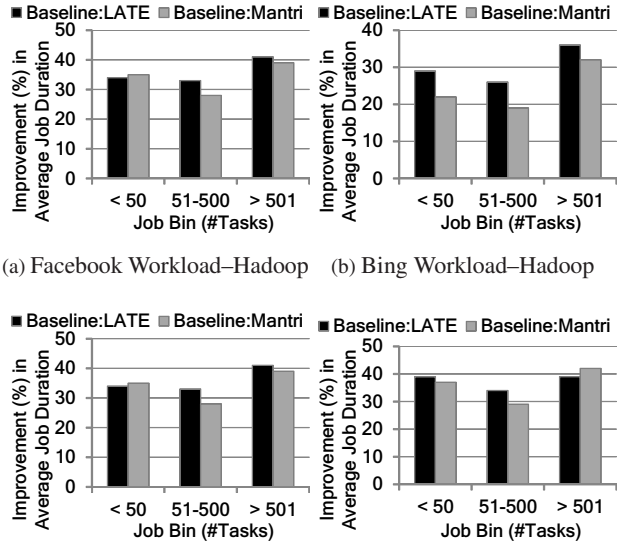
Gains with simulations are consistent with deployment, indicating not only that GRASS's gains hold over longer durations but also the simulator's robustness.

6.2.2 Error-bound jobs

Similar to deadline-bound jobs, improvements with the Spark prototype (33% to 37%) are higher compared to the Hadoop prototype (24% to 30%). This shows that GRASS works well not only with established frameworks like Hadoop but also upcoming ones like Spark.

Note that the gains are indistinguishable among different job bins (Figures 7a and 7b) in the Spark prototype; large jobs gain a touch more in the Hadoop prototype (Figures 7c and 7d). Again, our simulation results are consisten with deployment, and so are omitted.

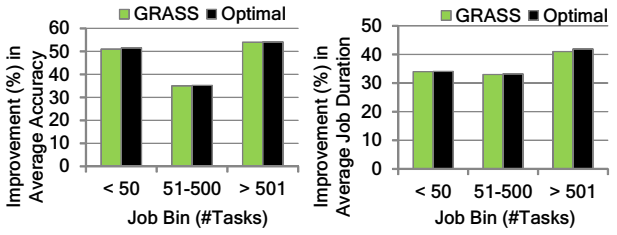
As Figure 6b shows, GRASS's gains persist at both tight as well as moderate error bounds. At high error bounds, there is smaller scope for GRASS beyond LATE.



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

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

Figure 7: **Speedup in error-bound jobs with LATE [2] and Mantri [1] as baselines.**



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

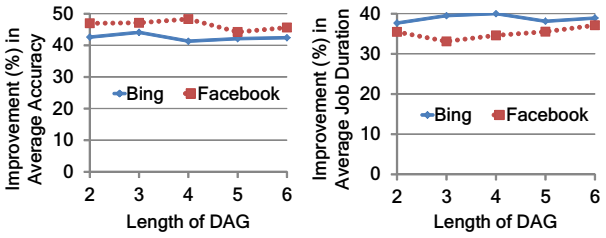
Figure 8: GRASS**'s gains matches the optimal scheduler.**

The gains at tight error bounds is noteworthy because these jobs are closer to *exact* jobs that require all (or most of) their tasks to complete. In fact, exact jobs speed up by 34%, thus making GRASS valuable even in clusters that are yet to deploy approximation analytics.

6.2.3 Optimality of GRASS

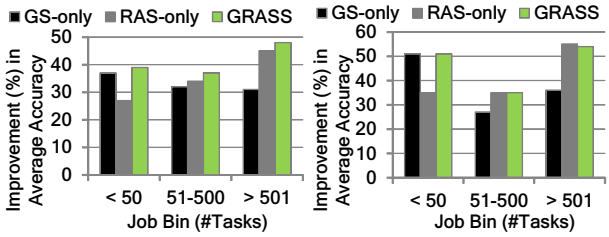
While the results above show the speed up GRASS provides, the question remains as to whether further improvements are possible. To understand the room available for improvement beyond GRASS, we compare its performance with an optimal scheduler that knows task durations and slot availabilities in advance.

Figure 8 shows the results for the Facebook workload with Spark. It highlights that GRASS's performance matches the optimal for both deadline as well as error-bound jobs. Thus, GRASS is an efficient near-optimal solution for the NP-hard problem of scheduling tasks for approximation jobs with speculative copies.



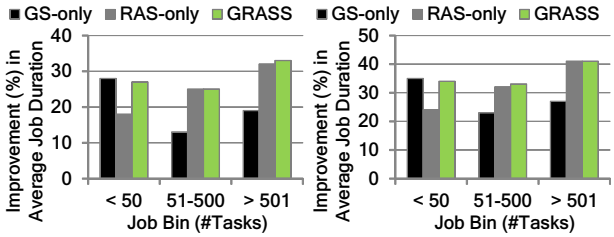
(a) Deadline-bound Jobs. (b) Error-bound Jobs.

Figure 9: GRASS's gains holds across job DAG sizes.



(a) Hadoop (b) Spark

Figure 10: 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 11: 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.

6.2.4 DAG of tasks

To complete the evaluation of GRASS we investigate how performance gains depend on the length of the job's DAG. Intuitively, as long as our estimation of intermediate phases is accurate, GRASS's handling of the input phase should remain unchanged, and Figure 9 confirms this for both deadline and error-bound jobs. Gains from GRASS remain stable with the length of the DAG.

6.3 Evaluating GRASS's Design Decisions

To understand the impact of the design decisions made in GRASS, we focus on three questions. First, how important is it that GRASS switches from RAS to GS? Second, how important is it that this switching is learned adaptively rather than fixed statically? Third, how sensitive is GRASS to the perturbation factor ξ? In the interest of space, we present results on these topics for only the Facebook workload using LATE as a baseline; results for the Bing workload with Mantri are similar.

6.3.1 The value of switching

To understand the importance of switching between RAS and GS we compare GRASS's performance with using only GS and RAS all through the job. Figure 10 performs the comparison for deadline-bound jobs. GRASS's improvements, both on average as well as in individual job bins, are strictly better than GS and RAS. This shows that if using only one of them is the best choice, GRASS automatically avoids switching. Further, GRASS's overall improvement in accuracy is over 20% better than the best of GS or RAS, demonstrating the value of switching as the job nears its deadline. The above trends are consistent with error-bound jobs as well (Figure 11), though GRASS's benefit is slightly lower.

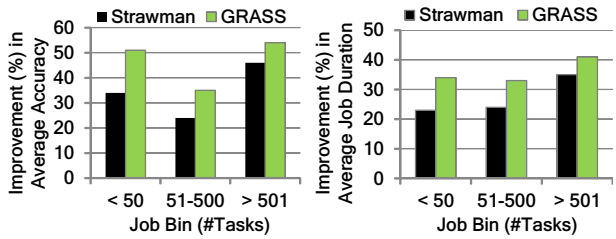
The contrast of GS and RAS is also interesting. GS outperforms RAS for small jobs but loses out as job sizes increase. The significant degradation in performance in the unfavorable job bin (medium and large jobs for GS, versus small jobs for RAS) illustrates the pitfalls of statically picking the speculation algorithm.

6.3.2 The value of learning

Given the benefit of switching, the question becomes when this switching should occur. GRASS does this adaptively based on three factors: deadline/error-bound, cluster utilization and estimation accuracy of t_{rem} and t_{new}. Now, we illustrate the benefit of this approach compared to simpler options, i.e., choosing the switching point statically or based on a subset of these three factors. Note that we have already seen that these three factors are enough to be near optimal (Figure 8).

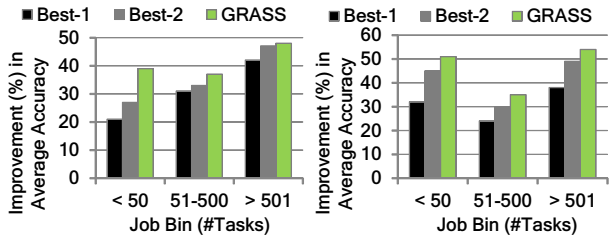
Static switching: First, when considering a static design, a natural "strawman" based on our theoretical analysis is to estimate the point when there are two remaining waves as follows. For deadline-bound jobs, it is the point when the time to the deadline is sufficient for at most two waves of tasks. For error-bound jobs, it is when the number of (unique) scheduled tasks sufficient to satisfy the error-bound make up two waves. The strawman uses the current wave-width of the job and assumes task durations to be median of completed tasks.

Figure 12 compares GRASS with the above strawman. Gains with the strawman are 66% and 73% of the gains with GRASS for deadline-bound and error-bound jobs,



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

Figure 12: **Comparing GRASS's learning based switching approach to a strawman that approximates two waves of tasks. GRASS is $30\% - 40\%$ better than the strawman.**



(a) Hadoop (b) Spark

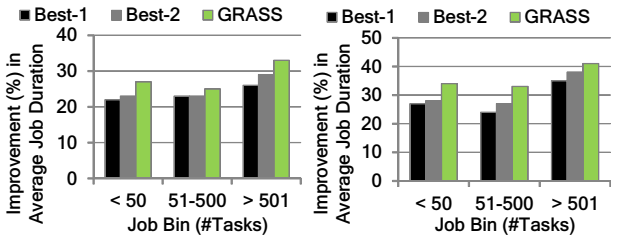
Figure 13: **Using all three factors for deadline-bound jobs compared to only one or two is $18\% - 30\%$ better.**

respectively. 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.

Adaptive switching: Next, we study the impact of the three factors used to adaptively learn the switching threshold. To do this, Figures 13 and 14 compares the designs using the best one or two factors with GRASS.

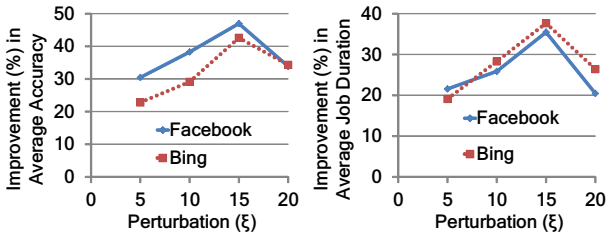
When only one factor can be used to switch, picking the deadline/error-bound provides the best results. This is intuitive given the importance of the approximation bound to the ordering of tasks. When two factors are used, in addition to the deadline/error-bound, cluster utilization matters more for the Hadoop prototype while estimation accuracy is important for the Spark prototype. Tasks of Hadoop jobs are longer and more sensitive to slot allocations, which is dictated by the utilization. While the smaller Spark tasks are more fungible, this also makes them sensitive to estimation errors.

Using only one factor is significantly worse than using all three factors. The performance picks up with deadline-bound jobs when two factors are used, but error-bound jobs' gains continue to lag until all three are used. Thus, in the absence of a detailed model for job executions, the three factors act as good predictors.



(a) Hadoop (b) Spark

Figure 14: **Using all three factors for error-bound jobs compared to one or two factors is $15\% - 25\%$ better.**



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

Figure 15: **Sensitivity of GRASS's performance to the perturbation factor ξ. Using $\xi = 15\%$ is empirically best.**

6.3.3 Sensitivity to Perturbation

The final aspect of GRASS that we evaluate is the perturbation factor, ξ, which decides how often the scheduler does *not* switch during a job's execution (described in §4.2). This perturbation is crucial for GRASS's learning of the optimal switching point. All results shown previously set ξ to 15%, which was picked empirically.

Figure 15 highlights the sensitivity of GRASS to this choice. Low values of ξ hamper learning because of the lack of sufficient samples, while high values incur performance loss resulting from not switching from RAS to GS often enough. Our results show, that this exploration–exploitation tradeoff is optimized at $\xi = 15\%$, and that performance drops off sharply around this point. Deadline-bound jobs are more sensitive to poor choice of ξ than error-bound jobs. Using ξ of 15% is consistent with studies on multi-armed bandit problems [28], which is related to our learning problem.

7 Related Work

The problem of stragglers was identified in the original MapReduce paper [29]. Since then solutions have been proposed to mitigate them using speculative executions [2, 1, 24]. These solutions, however, are not for approximation jobs. These jobs require proritizing the right subset of tasks by carefully considering the opportunity cost of speculation. Further, our evaluations show

that GRASS speeds up even for exact jobs that require all their tasks to complete. Thus, it is a unified solution that cluster schedulers can deploy for both approximation as well as non-approximation computations.

Prioritizing tasks of a job is a classic scheduling problem with known heuristics [11, 12]. These heuristics assume accurate knowledge of task durations and hence do not require speculative copies to be scheduled dynamically. Estimating task durations accurately, however, is still an open challenge as acknowledged by many studies [3, 20]. This makes speculative copies crucial and we develop a theoretically backed solution to optimally prioritize tasks with speculative copies.

Modeling real world clusters has been a challenge faced by other schedulers too. Recently reported research has acknowledged the problem of estimating task durations [16], predicting stragglers [3, 20] as well as modeling multi-waved job executions [17]. Their solutions primarily involve sidestepping the problem by not predicting stragglers and upfront replicating the tasks [3], or approximating number of waves to file sizes [17]. Such sidestepping, however, is not an option for GRASS and hence we build tailored approximations.

Finally, replicating tasks in distributed systems have a long history [30, 31, 32] with extensive studies in prior work [33, 34, 35]. These studies assume replication *upfront* as opposed to *dynamic* replication in reaction to stragglers. The latter problem is both harder and unsolved. In this work, we take a stab at this problem that yields near-optimal results in our production workloads.

8 Concluding Remarks and Future Work

This paper explores speculative task scheduling in the context of approximation jobs. From the analysis of a generic analytic model, we develop a speculation algorithm, 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. Prototypes on Hadoop and Spark, deployed on a 200 node EC2 cluster shows that GRASS improves accuracy fo 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.

A topic that requires further work is scheduling speculative copies for stragglers *across* jobs. While GRASS intelligently picks between scheduling a speculative copy for a running task versus scheduling a new task of a job, it does so within the slots allocated to the job (typically, based on fair allocations). The next step to GRASS is to weigh the impact of speculating a running task with scheduling a new task of *any* job. Answering this question will not only involve comparing across jobs but also revisit existing fairness based schedulers.

A Modeling and Analyzing Speculation

In this section we introduce the model and analysis that led to the guidelines described in §3.2. The model focuses on one job that has T tasks[5] and S slots out of a total capacity normalized to 1. Let the initial job size be x and the remaining amount of work in the job at time t be $x(t)$. We use $W = T/S$ to denote the (fractional) number of waves necessary to complete the job, and throughout we assume $W \geq 1$.

We focus our analysis on the rate at which work is completed, which we denote by $\mu(t; x, S, T)$ or $\mu(t)$ for short. Note that by focusing on the service rate we are ignoring ordering of the tasks and focusing primarily on the impact of speculation.

In our analysis we begin with proactive speculation, and then move to reactive speculation. This progression is natural since the analysis of proactive speculation serves as a stepping stone to the design of reactive speculation policies. Further, in the case of proactive speculation we can precisely specify the optimal policy, whereas in the case of reactive speculation, we must resort to numerical optimization.

A.1 Proactive speculation

We start by considering a general class of proactive policies that launch $k(x(t))$ speculative copies of tasks when the job has remaining size $x(t)$. We propose the following *approximate model* for $\mu(t)$ in this case.

$$\left[\left(\frac{x(t)}{x}\right)\left(\frac{T}{S}\right)k(x(t))\right]^{1}_{\frac{k(x(t))}{S}} \cdot \left(\frac{E[\tau]}{k(x(t))E\left[\min(\tau_1, \ldots, \tau_{k(x(t))}\right]}\right) \quad (1)$$

where τ is a random task size. Note that we assume task sizes are i.i.d.

To understand this approximate model, note that the first term approximates the completion rate of work and the second term approximates the "blow up factor," i.e., the ratio of the expected work completed without duplications to the amount of work done with duplications. To understand the first term, note that $(x(t)/x)T$ is the fractional number of tasks that remain to be completed at time t, and thus there are $(x(t)/x)Tk(x(t))$ tasks available to schedule at time t including speculative copies. Recalling that the capacity of a slot is $1/S$, that the maximum capacity that can be allocated is 1, and that the minimum number of slots is the number of copies $k(x(t))$,

[5] For approximation jobs T should be interpreted as the number of tasks that are completed before the deadline or error limit is reached.

we obtain the first term in (1). The second term computes the "blow up factor," which is the the expected amount of work done per task without speculation ($E[\tau]$) divide by the expected work done per task with speculation ($k(x(t))E[\min(\tau_1, \tau_2, \ldots, \tau_{k(x(t))})]$, since $k(x(t))$ copies are created and they are stopped when the first copy completes. Perhaps the most important aspect of this approximation is the fact that task sizes are i.i.d. in this manner, and this is what leads both to stragglers and to the benefits of replication. While this is certainly simplistic, the value of the model is highlights be the usefulness of the guidelines that follow from our analysis.

Given the model in (1), the question is: What proactive speculation policy minimizes job duration? As discussed in §3.2, the distribution of task sizes shows considerable evidence of a Pareto-tail, and so we focus our analysis on this setting. The following theorem follows from first deriving the response time of a job given the model for $\mu(t)$ in (1), and then deriving the $k(x(t))$ that minimizes the response time. Each of these steps requires significant, but not difficult, analysis, which we omit due to space.

Theorem 1 *When task sizes are Pareto(x_m,β), the proactive speculation policy that minimizes the completion time of the job is*

$$k(x(t)) = \begin{cases} \sigma, & \frac{x(t)}{x} T\sigma \geq S \\ S/(\frac{x(t)}{x}T), & S > \frac{x(t)}{x} T\sigma \text{ and } \frac{x(t)}{x}T \geq 1; \\ S, & 1 > \frac{x(t)}{x}T. \end{cases} \quad (2)$$

where $\sigma = \max(2/\beta, 1)$.

This theorem leads to Guidelines 1 and 2. Specifically, the first line corresponds to the "early waves" and the second and third lines correspond to the "last wave". During the "early waves" the optimal policy may or may not speculate, depending on the task size distribution – speculation happens only when $\beta < 2$, which is when task sizes have infinite variance. In contrast, during the "last wave", regardless of the task size distribution, the optimal policy speculates to ensure all slots are used.

A.2 Reactive speculation

We now turn to reactive speculation policies, which launches copies of a task only after it has completed ω work. Both GS and RAS are examples of such policies and can be translated into choices for ω as shown in §3.2.

Our analysis of proactive policies provides important insight into the design of reactive policies. In particular, during early waves the the optimal proactive policy runs at most two copies of each task, and so we limit our reactive policies to this level of speculation. Additionally, the previous analysis highlights that during the last wave the it is best to speculate aggressively in order to use up the full capacity, and thus it is best to speculated immediately without waiting ω time. This yields the following approximation for $\mu(t)$:

$$\begin{cases} \frac{E[\tau_1]}{E[\tau_1 | 0 \leq \tau_1 < \omega] \Pr(0 \leq \tau_1 < \omega) + (2E[Z - \omega | \tau_1 \geq \omega] + \omega) \Pr(\tau_1 > \omega)}, \\ \quad \text{when } \frac{x(t)}{x} T(\Pr(0 \leq \tau_1 < \omega) + 2\Pr(\tau_1 \geq \omega)) \geq S. \\ \\ \text{optimal proactive speculation (from (1))}, \\ \quad \text{when } \frac{x(t)}{x} T(\Pr(0 \leq \tau_1 < \omega) + 2\Pr(\tau_1 \geq \omega)) < S. \end{cases} \quad (3)$$

τ_1, τ_2 are random task sizes and $Z = \min(\tau_1, \tau_2 + \omega)$.

Again, the first line in (3) approximates the service rate during the early waves of the job, while the second line approximates the service rate during the final wave of the job. To understand the first line, note that during early waves there are enough tasks to use capacity 1 (in expectation) as long as $\frac{x(t)}{x} T(\Pr(0 \leq \tau < \omega) + 2\Pr(T \geq \omega)) \geq S$. Thus, all that remains is the "blow up factor." As before, the numerator is the expected amount of work per task without speculation ($E[\tau]$) and the denominator is the expected amount of work per task with reactive speculation. This is $E[\tau | \tau < \omega]$ if the initial copy finishes before ω, and $2E[Z - \omega | \tau > \omega] + \omega$ if the initial copy takes longer than ω.

Within this model, our design problem can now be reduced to finding ω *that minimizes the response time of the job*. The complicated form of (3) makes it difficult to understand the optimal ω analytically, and thus we use numerical calculations. Figure 4 presents a numerical optimization by comparing GS and RAS to other reactive policies. It leads go Guideline 3, which highlights that GS is near optimal if the number of waves in the job is < 2, while RAS is near-optimal if the number of waves in the job is ≥ 2. Note that the results in Figure 4 are for Pareto task sizes with $\beta = 1.259$, but the finding is robust for $\beta \in (1, 2)$.

Acknowledgments

We thank our shepherd Nina Taft and the anonymous reviewers for their suggestions to improve this work. We also thank Rohan Gandhi for his feedback on our early drafts. This research was partially funded by research grant NSF CNS-1319820, NSF CISE Expeditions award CCF-1139158, the DARPA XData Award FA8750-12-2-0331, and gifts from Qualcomm, Amazon Web Services, Google, SAP, Blue Goji, Cisco, Clearstory Data, Cloudera, Ericsson, Facebook, General Electric, Hortonworks, Huawei, Intel, Microsoft, NetApp, Oracle, Quanta, Samsung, Splunk, VMware and Yahoo!.

References

[1] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, E. Harris, and B. Saha. Reining in the Outliers in Map-Reduce Clusters Using Mantri. In *USENIX OSDI*, 2010.

[2] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *USENIX OSDI*, 2008.

[3] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective Straggler Mitigation: Attack of the Clones. In *USENIX NSDI*, 2013.

[4] S.Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *EuroSys*. ACM, 2013.

[5] T. Condie, N. Conway, P. Alvaro, J. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce Online. In *USENIX NSDI*, 2010.

[6] Interactive Big Data analysis using approximate answers, 2013. http://tinyurl.com/k5favda.

[7] J. Liu, K. Shih, W. Lin, R. Bettati, and J. Chung. Imprecise Computations. *Proceedings of the IEEE*, 1994.

[8] S. Lohr. *Sampling: design and analysis*. Thomson, 2009.

[9] J. Hellerstin, P. Haas, and H. Wang. Online Aggregation. In *ACM SIGMOD*, 1997.

[10] M. Garofalais and P. Gibbons. Approximate Query Processing: Taming the Terabytes. In *VLDB*, 2001.

[11] M. Pinedo. *Scheduling: theory, algorithms, and systems*. Springer, 2012.

[12] L. Kleinrock. *Queueing systems, volume II: computer applications*. John Wiley & Sons New York, 1976.

[13] C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard-real-time Environment. *Journal of the ACM (JACM)*, 1973.

[14] Hadoop. http://hadoop.apache.org.

[15] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *USENIX NSDI*, 2012.

[16] E. Bortnikov, A. Frank, E. Hillel, S. Rao. Predicting Execution Bottlenecks in Map-Reduce Clusters. In *USENIX HotCloud*, 2012.

[17] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. PACMan: Coordinated Memory Caching for Parallel Jobs. In *USENIX NSDI*, 2012.

[18] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica. The Case for Tiny Tasks in Compute Clusters. In *USENIX HotOS*, 2013.

[19] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. A Study of Skew in MapReduce Applications. In *Open Cirrus Summit*, 2011.

[20] J. Dean. Achieving Rapid Response Times in Large Online Services. In *Berkeley AMPLab Cloud Seminar*, 2012.

[21] S. Resnick. *Heavy-tail phenomena: probabilistic and statistical modeling*. Springer, 2007.

[22] J. C. Gittins. Bandit Processes and Dynamic Allocation Indices. *Journal of the Royal Statistical Society. Series B (Methodological)*, 1979.

[23] I. Sonin. A Generalized Gittins Index for a Markov Chain and Its Recursive Calculation. *Statistics & Probability Letters*, 2008.

[24] M. Isard, M. Budiu, Y. Yu, A. Birrell and D. Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *ACM Eurosys*, 2007.

[25] W. Baek and T. Chilimbi. Green: a Framework for Supporting Energy-conscious Programming Using Controlled Approximation. In *ACM Sigplan Notices*, 2010.

[26] Hive. http://wiki.apache.org/hadoop/Hive.

[27] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Datasets. In *VLDB*, 2008.

[28] M. Tokic and G. Palm. Value-difference Based Exploration: Adaptive Control between Epsilon-greedy and Softmax. In *KI 2011: Advances in Artificial Intelligence*. Springer, 2011.

[29] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 2008.

[30] A. Baratloo, M. Karaul, Z. Kedem, and P. Wycko. Charlotte: Metacomputing on the Web. In *9th Conference on Parallel and Distributed Computing Systems*, 1996.

[31] E. Korpela D. Anderson, J. Cobb. SETI@home: An Experiment in Public-Resource Computing. In *Comm. ACM*, 2002.

[32] M. Rinard and P. Diniz. Commutativity Analysis: a New Analysis Framework for Parallelizing Compilers. In *ACM PLDI*, 1996.

[33] D. Paranhos, W. Cirne, and F. Brasileiro. Trading Cycles for Information: Using Replication to Schedule Bag-of-Tasks Applications on Computational Grids. In *Euro-Par*, 2003.

[34] G. Ghare and S. Leutenegger. Improving Speedup and Response Times by Replicating Parallel Programs on a SNOW. In *JSSPP*, 2004.

[35] W. Cirne, D. Paranhos, F. Brasileiro, L. Goes, and W. Voorsluys. On the Efficacy, Efficiency and Emergent Behavior of Task Replication in Large Distributed Systems. In *Parallel Computing*, 2007.