Effective Straggler Mitigation: Attack of the Clones

Ganesh Ananthanarayanan[1], Ali Ghodsi[1,2], Scott Shenker[1], Ion Stoica[1]

[1] *University of California, Berkeley* [2] *KTH/Sweden*

{ganesha,alig,shenker,istoica}@cs.berkeley.edu

Abstract

Small jobs, that are typically run for interactive data analyses in datacenters, continue to be plagued by disproportionately long-running tasks called *stragglers*. In the production clusters at Facebook and Microsoft Bing, even after applying state-of-the-art straggler mitigation techniques, these latency sensitive jobs have stragglers that are on average 8 times slower than the median task in that job. Such stragglers increase the average job duration by 47%. This is because current mitigation techniques all involve an element of waiting and speculation. We instead propose full *cloning* of small jobs, avoiding waiting and speculation altogether. Cloning of small jobs only marginally increases utilization because workloads show that while the majority of jobs are small, they only consume a small fraction of the resources. The main challenge of cloning is, however, that extra clones can cause contention for intermediate data. We use a technique, *delay assignment*, which efficiently avoids such contention. Evaluation of our system, Dolly, using production workloads shows that the small jobs speedup by 34% to 46% *after* state-of-the-art mitigation techniques have been applied, using just 5% extra resources for cloning.

1 Introduction

Cloud computing has achieved widespread adoption due to its ability to automatically parallelize a *job* into multiple short *tasks*, and transparently deal with the challenge of executing these tasks in a distributed setting. One such fundamental challenge is *straggling tasks*, which is faced by all cloud frameworks, such as MapReduce [1], Dryad [2], and Spark [3]. Stragglers are tasks that run much slower than other tasks, and since a job finishes only when its last task finishes, stragglers delay job completion. Stragglers especially affect *small jobs*, *i.e.,* jobs that consist of a few tasks. Such jobs typically get to run all their tasks at once. Therefore, even if a single task is slow, *i.e.,* straggle, the whole job is significantly delayed.

Small jobs are pervasive. Conversations with datacenter operators reveal that these small jobs are typically used when performing interactive and exploratory analyses. Achieving low latencies for such jobs is critical to enable data analysts to efficiently explore the search space. To obtain low latencies, analysts already restrict their queries to small but carefully chosen datasets, which results in jobs consisting of only a few short tasks. The trend of such exploratory analytics is evident in traces we have analyzed from the Hadoop production cluster at Facebook, and the Dryad cluster at Microsoft Bing. Over 80% of the Hadoop jobs and over 60% of the Dryad jobs are small with fewer than ten tasks[1]. Achieving low latencies for these small interactive jobs is of prime concern to datacenter operators.

The problem of stragglers has received considerable attention already, with a slew of *straggler mitigation* techniques [1, 4, 5] being developed. These techniques can be broadly divided into two classes: *black-listing* and *speculative execution*. However, our traces show that *even after* applying state-of-the-art blacklisting and speculative execution techniques, the small jobs have stragglers that, on average, run eight times slower than that job's median task, slowing them by 47% on average. Thus, stragglers remain a problem for small jobs. We next explain the limitations of these two approaches.

Blacklisting identifies machines in bad health (*e.g.,* due to faulty disks) and avoids scheduling tasks on them. The Facebook and Bing clusters, in fact, blacklist roughly 10% of their machines. However, stragglers occur on the non-blacklisted machines, often due to intrinsically complex reasons like IO contentions, interference by periodic maintenance operations and background services, and hardware behaviors [6].

For this reason, speculative execution [1, 4, 5, 7] was explored to deal with stragglers. Speculative execution *waits* to observe the progress of the tasks of a job and launches duplicates of those tasks that are slower. However, speculative execution techniques have a fundamental limitation when dealing with small jobs. Any meaningful comparison requires waiting to collect statistically significant samples of task performance. Such waiting limits their agility when dealing with stragglers in small jobs as they often start all their tasks simultaneously. The 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.

In this paper, we propose a different approach. Instead of waiting and trying to predict stragglers, we take speculative execution to its extreme and propose launching multiple *clones* of every task of a job and only use the result of the clone that finishes first. This technique is both general and robust as it eschews waiting, speculating, and finding complex correlations. Such *proactive*

[1] The length of a task is mostly invariant across small and large jobs.

cloning will significantly improve the agility of straggler mitigation when dealing with small interactive jobs.

Cloning comes with two main challenges. The first challenge is that extra clones might use a prohibitive amount of extra resources. However, our analysis of production traces shows a strong *heavy-tail distribution* of job sizes: the smallest 90% of jobs consume as less as 6% of the resources. The interactive jobs whose latency we seek to improve all fall in this category of small jobs. We can, hence, improve them by using few extra resources.

The second challenge is the potential *contention* that extra clones create on intermediate data, possibly hurting job performance. Efficient cloning requires that we clone each task and use the output from the clone of the task that finishes first. This, however, can cause contention for the intermediate data passed between tasks of the different phases (*e.g.,* map, reduce, join) of the job; frameworks often compose jobs as a graph of *phases* where tasks of downstream phases (*e.g.,* reduce) read the output of tasks of upstream phases (*e.g.,* map). If all downstream clones read from the upstream clone that finishes first, they contend for the IO bandwidth. An alternate that avoids this contention is making each downstream clone read exclusively from only a single upstream clone. But this staggers the start times of the downstream clones.

Our solution to the contention problem, *delay assignment*, is a hybrid solution that aims to get the best of both the above pure approaches. It is based on the intuition that most clones, except few stragglers, finish nearly simultaneously. Using a cost-benefit analysis that captures this small variation among the clones, it checks to see if clones can obtain exclusive copies before assigning downstream clones to the available copies of upstream outputs. The cost-benefit analysis is generic to account for different communication patterns between the phases, including all-to-all (MapReduce), many-to-one (Dryad), and one-to-one (Dryad and Spark).

We have built Dolly, a system that performs cloning to mitigate the effect of stragglers while operating within a resource budget. Evaluation on a 150 node cluster using production workloads from Facebook and Bing shows that Dolly improves the average completion time of the small jobs by 34% to 46%, respectively, with LATE [5] and Mantri [4] as baselines. These improvements come with a resource budget of merely 5% due to the aforementioned heavy-tail distribution of job-sizes. By picking the fastest clone of every task, Dolly effectively reduces the slowest task from running $8\times$ slower on average to $1.06\times$, thus, effectively eliminating all stragglers.

2 The Case for Cloning

In this section we quantify: (i) magnitude of stragglers and the potential in eliminating them, and (ii) power law distribution of job sizes that facilitate aggressive cloning.

	Facebook	Microsoft Bing
Dates	Oct 2010	May-Dec* 2009
Framework	Hadoop	Dryad
File System	HDFS [9]	Cosmos
Script	Hive [10]	Scope [11]
Jobs	375K	200K
Cluster Size	3,500	Thousands
Straggler–mitigation	LATE [5]	Mantri [4]

* *One week in each month*

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

Production Traces: Our analysis is based on traces from Facebook's production Hadoop [8] cluster and Microsoft Bing's production Dryad [2] cluster. These are large clusters with thousands of machines running jobs whose performance and output have significant impact on productivity and revenue. Therefore, each of the machines in these clusters is well-provisioned with tens of cores and sufficient (tens of GBs) memory. The traces capture the characteristics of over half a million jobs running across many months. Table 1 lists the relevant details of the traces. The Facebook cluster employs the LATE straggler mitigation strategy [5], while the Bing cluster uses the Mantri straggler mitigation strategy [4].

2.1 Stragglers in Jobs

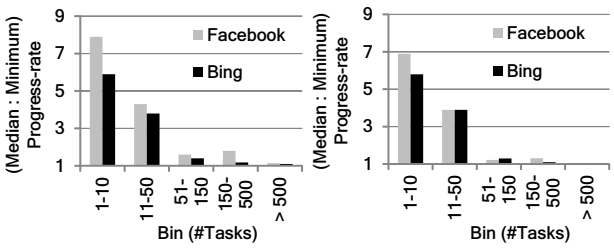
We first quantify the magnitude and impact of stragglers, and then show that simple blacklisting of machines in the cluster is insufficient to mitigate them.

2.1.1 Magnitude of Stragglers and their Impact

A job consists of a graph of *phases* (*e.g.,* map, reduce, and join), with each phase executing the same type of tasks in parallel. We identify stragglers by comparing the *progress rates* of tasks within a phase. The progress rate of a task is defined as the size of its input data divided by its duration. In absence of stragglers, progress rates of tasks of a phase are expected to be similar as they perform similar IO and compute operations. We use the progress rate instead of the task's duration to remain agnostic to skews in work assignment among tasks [4]. Techniques have been developed to deal with the problem of data skews among tasks [12, 13, 14] and our approach is complementary to those techniques.

Within each phase, we measure the *slowdown ratio*, *i.e.,* the ratio of the progress rate of the median task to the slowest task. The negative impact of stragglers increases as the slowdown ratio increases. We measure the slowdown ratio after applying the LATE and Mantri mitigations; a what-if simulation is used for the mitigation strategy that the original trace did not originally deploy.

Figure 1a plots the slowdown ratio by binning jobs according to their number of tasks, with LATE in effect.



(a) LATE (b) Mantri

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

p_b	Blacklisted Machines (%)		Job Improvement (%)	
	5 min	1 hour	5 min	1 hour
0.3	4%	6%	7.1%	8.4%
0.5	1.6%	2.8%	4.4%	5.2%
0.7	0.8%	1.2%	2.3%	2.8%

Table 2: **Blacklisting by predicting straggler probability. We show the fraction of machines that got blacklisted and the improvements in completion times by avoiding them.**

Phases in jobs with fewer than ten tasks, have a median value of this ratio between 6 and 8, *i.e.,* the slowest task is up to $8\times$ slower than the median task in the job. Also, small jobs are hit harder by stragglers.[2] This is similar even if Mantri [4] was deployed. Figure 1b shows that the slowest task is still $7\times$ slower than the median task, with Mantri. However, both LATE and Mantri effectively mitigate stragglers in large jobs.

Speculation techniques are not as effective in mitigating stragglers in small jobs as they are with large jobs because they rely on comparing different tasks of a job to identify stragglers. Comparisons are effective with more samples of task performance. This makes them challenging to do with small jobs because not only do these jobs have fewer tasks but also start all of them simultaneously.

Impact of Stragglers: We measure the potential in speeding up jobs in the trace using the following crude analysis: replace the progress rate of every task of a phase that is slower than the median task with the median task's rate. If this were to happen, the average completion time of jobs improves by 47% and 29% in the Facebook and Bing traces, respectively; small jobs (those with ≤ 10 tasks) improve by 49% and 38%.

2.1.2 Blacklisting is Insufficient

An intuitive solution for mitigating stragglers is to *blacklist* machines that are likely to cause them and avoid scheduling tasks on them. For this analysis, we classify a task as a straggler if its progress rate is less than half of the median progress rate among tasks in its phase. In our trace, stragglers are not restricted to a small set of machines but are rather spread out uniformly through the cluster. This is not surprising because both the clusters already blacklist machines with faulty disks and other hardware troubles using periodic diagnostics.

We enhance this blacklisting by monitoring machines at finer time intervals and employing temporal prediction techniques to warn about straggler occurrences. We use an EWMA to predict stragglers—the probability of a machine causing a straggler in a time window is equally dependent on its straggler probability in the previous window and its long-term average. Machines with a predicted straggler probability greater than a threshold (p_b) are blacklisted for that time window but considered again for scheduling in the next time window.

We try time windows of 5 minutes and 1 hour. Table 2 lists the fraction of machines that get blacklisted and the resulting improvement in job completion times by eliminating stragglers on them, in the Facebook trace. 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%). This is in harsh contrast with potential improvements of 29% to 47% if all stragglers were eliminated, as shown in §2.1.1.

The above results do not prove that effective blacklisting is impossible, but shows that none of the blacklisting techniques that we and, to our best knowledge, others [6] have tried effectively prevent stragglers, suggesting that such correlations either do not exist or are hard to find.

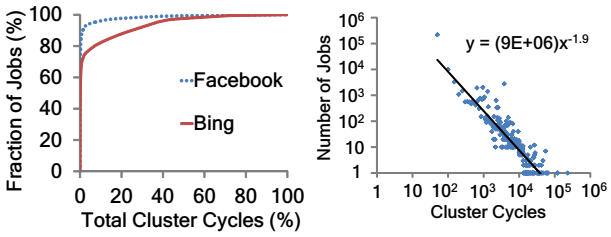
2.2 Heavy Tail in Job Sizes

We observed that smaller jobs are most affected by stragglers. These jobs were submitted by users for iterative experimental purposes. For example, researchers tune the parameters of new mining algorithms by evaluating it on a small sample of the dataset. For this reason, these jobs consist of just a few tasks. In fact, in both our traces, we have noted a correlation between a job's duration and the number of tasks it has, *i.e.,* jobs with shorter durations tend to have fewer tasks. Short and predictable response times for these jobs is of prime concern to datacenter operators as they significantly impact productivity.

On the one hand, small interactive jobs absolutely dominate the cluster and have stringent latency demands. In the Facebook and Bing traces, jobs with ≤ 10 tasks account for 82% and 61% of all the jobs, respectively. On the other hand, they are the most affected by stragglers.

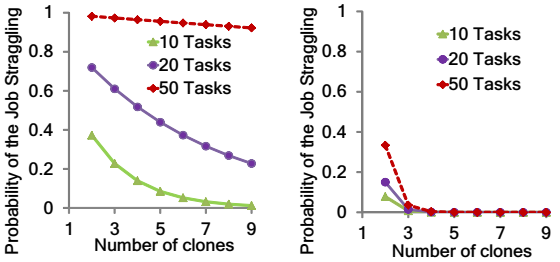
Despite this, we can clone all the small jobs using few extra resources. This is because job sizes have a *heavy-tail* distribution. Just a few large jobs consume most of

[2] Implicit in our explanation is that small interactive jobs consist of just a few tasks. While we considered alternate definitions based on input size and durations, in both our traces, we see a high correlation between jobs running for short durations and the number of tasks they contain along with the size of their input.



(a) Heavy-tail (b) Power Law

Figure 2: **Heavy tail. Figure (a) shows the heavy tail in the fraction of total resources used. Figure (b) shows that the distribution of cluster resources consumed by jobs, in the Facebook trace, follows a power law. Power-law exponents are** 1.9 **and** 1.8 **when fitted with least squares regression in the Facebook and Bing traces.**



(a) Job-level Cloning (b) Task-level Cloning

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.**

the resources in the cluster, while the cluster is dominated by small interactive jobs. As 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 (see Figure 2b). In fact, at any point in time, the small jobs do not use more than 2% of the overall cluster resources.

The heavy-tail distribution offers potential to speed up these jobs by using few extra resources. For instance, cloning each of the smallest 90% of the jobs three times increases overall utilization by merely 3%. This is well within reach of today's underutilized clusters which are heavily over-provisioned to satisfy their peak demand of over 99%, that leaves them idle at other times [15, 16].

Google recently released traces from their cluster job scheduler that schedules a mixed workload of MapReduce batch jobs, interactive queries and long-running services [17]. Analysis of these traces again reveal a heavy-tail distribution of job sizes, with 92% of the jobs accounting for only 2% of the overall resources [18].

3 Cloning of Parallel Jobs

We start this section by describing the high-level idea of cloning. After that (§3.1) we determine the granularity of cloning, and settle for cloning at the granularity of tasks, rather than entire jobs, as the former requires fewer clones. Thereafter (§3.2), we investigate the number of clones needed if we desire the probability of a job straggling to be at most ϵ, while staying within a cloning budget. Finally (§3.3), as we are unlikely to have room to clone every job in the cluster, we show a very simple admission control mechanism that decides when to clone jobs. An important challenge of cloning—handling data contention between clones—is dealt with in §4.

In contrast to *reactive* speculation solutions [1, 4, 5], Dolly advocates a *proactive* approach—straightaway launch multiple clones of a job and use the result of the first clone that finishes. Cloning makes straggler mitigation agile as it does not have to wait and observe a task before acting, and also removes the risk inherent in speculation—speculating the wrong tasks or missing the stragglers. Similar to speculation, we assume that picking the earliest clone does not bias the results, a property that generally holds for data-intensive computations.

3.1 Granularity of Cloning

We start with a job consisting of a single phase. A crucial decision affecting efficiency is the granularity of cloning. A simple option is to clone at the granularity of jobs. For every job submitted to the cluster, multiple clones of the entire job are launched. Results are taken from the earliest job that finishes. Such job-level cloning is appealing due to its simplicity and ease of implementation.

A fine-grained alternative is to clone at the granularity of individual tasks. Thus, multiple clones of each task are launched. We refer to the different clones of the same task as a *clone group*. In every clone group, we then use the result of the clone that finishes first. Therefore, unlike job-level cloning, task-level cloning requires internal changes to the execution engine of the framework.

As a result of the finer granularity, for the same number of clones, task-level cloning provides better probabilistic guarantees for eliminating stragglers compared to job-level cloning. Let p be the probability of a task straggling. For a single-phased job with n parallel tasks and c clones, the probability that it straggles is $(1-(1-p)^n)^c$ with job-level cloning, and $1-(1-p^c)^n$ with task-level cloning. Figure 3 compares these probabilities. Task-level cloning gains more per clone and the probability of the job straggling drops off faster.

Task-level cloning's resource efficiency is desirable because it reduces contention on the input data which is read from file systems like HDFS [9]. If replication of input data does not match the number of clones, the clones contend for IO bandwidth in reading the data. Increas-

ing replication, however, is difficult as clusters already face a dearth of storage space [19, 20]. Hence, due to its efficiency, we opt for task-level cloning in Dolly.

3.2 Budgeted Cloning Algorithm

Pseudocode 1 describes the cloning algorithm that is executed at the scheduler per job. The algorithm takes as input the cluster-wide probability of a straggler (p) and the acceptable risk of a job straggling (ϵ). We aim for an ϵ of 5% in our experiments. The probability of a straggler, p, is calculated every hour, where the straggler progresses at less than half the median task in the job. This coarse approach suffices for our purpose.

Dolly operates within an allotted resource budget. This budget is a configurable fraction (β) of the total capacity of the cluster (C). At no point does Dolly use more than this cloning budget. Setting a hard limit eases deployment concerns because operators are typically nervous about increasing the average utilization by more than a few percent. Utilization and capacity are measured in number of slots (computation units allotted to tasks).

The pseudocode first calculates the desired number of clones per task (step 2). For a job with n tasks, the number of clones desired by task-level cloning, c, can be derived to be at least $\log\left(1-(1-\epsilon)^{(1/n)}\right)/\log p$. [3] The number of clones that are eventually spawned is limited by the resource budget ($C \cdot \beta$) and a utilization threshold (τ), as in step 3. The job is cloned only if there is room to clone all its tasks, a policy we explain shortly in §3.3. Further, cloning is avoided if the cluster utilization after spawning clones is expected to exceed a ceiling τ. This ceiling avoids cloning during heavily-loaded periods.

Note that Pseudocode 1 spawns the same number of clones to all the tasks of a job. Otherwise, tasks with fewer clones are more likely to lag behind. Also, there are no conflicts between jobs in updating the shared variables B_U and U because the centralized scheduler handles cloning decisions one job at a time.

Multi-phased Jobs: For multi-phased jobs, Dolly uses Pseudocode 1 to decide the number of clones for tasks of *every* phase. However, the number of clones for tasks of a downstream phase (*e.g.,* reduce) never exceeds the number of clones launched its upstream phase (*e.g.,* map). This avoids contention for intermediate data (we revisit this in §4). In practice, this limit never applies because small jobs have equal number of tasks across their phases. In both our traces, over 91% of the jobs with ≤ 10 tasks have equal number of tasks in their phases.

3.3 Admission Control

The limited cloning budget, β, should preferably be utilized to clone the small interactive jobs. Dolly achieves this using a simple policy of *admission control*.

```
1: procedure CLONE(n tasks, p, ε)
       C: Cluster Capacity, U: Cluster Utilization
       β: Budget in fraction, B_U: Utilized budget in #slots
2:     c = ⌈log(1 − (1 − ε)^(1/n)) / log p⌉
3:     if (B_U + c · n) ≤ (C · β) and (U + c · n) ≤ τ then
                ▷ Admission Control: Sufficient capacity to
                  create c clones for each task
4:         for each task t do
               Create c clones for t
           B_U ← B_U + c · n
```

Pseudocode 1: **Task-level cloning for a single-phased job with n parallel tasks, on a cluster with probability of straggler as p, and the acceptable risk of straggler as ϵ.**

Whenever the first task of a job is to be executed, the admission control mechanism computes, as previously explained, the number of clones c that would be required to reach the target probability ϵ of that job straggling. If, at that moment, there is room in the cloning budget for creating c copies of all the tasks, it admits cloning the job. If there is not enough budget for c clones of all the tasks, the job is simply denied cloning and is executed without Dolly's straggler mitigation. The policy of admission control implicitly biases towards cloning small jobs—the budget will typically be insufficient for creating the required number of clones for the larger jobs. Step 3 in Pseudocode 1 implements this policy.

Many other competing policies are possible. For instance, a job could be partially cloned if there is not enough room for c clones. Furthermore, preemption could be used to cancel the clones of an existing job to make way for cloning another job. It turns out that these competing policies buy little performance compared to our simple policy. We compare these policies in §5.5.

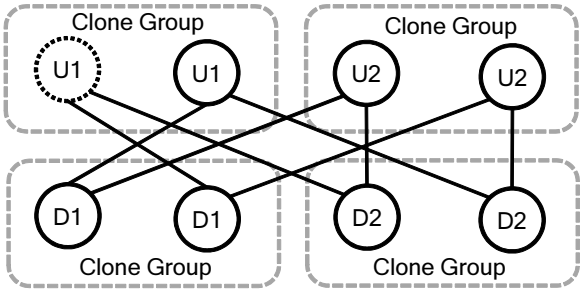
4 Intermediate Data Access with Dolly

A fundamental challenge of cloning is the potential contention it creates in reading data. Downstream tasks in a job read intermediate data from upstream tasks according to the communication pattern of that phase (all-to-all, many-to-one, one-to-one). The clones in a downstream clone group would ideally read their intermediate data from the upstream clone that finishes first as this helps them all start together.[4] This, however, can create contention at the upstream clone that finishes first. Dealing with such contentions is the focus of this section.

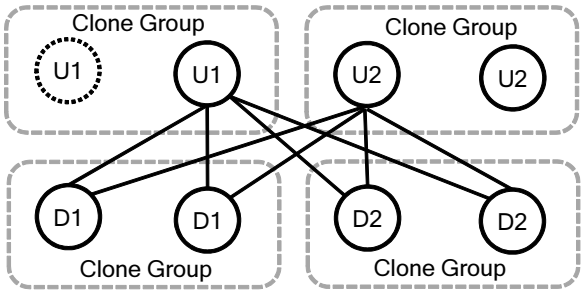
We first (§4.1) explore two pure strategies at opposite ends of the spectrum for dealing with intermediate data contention. At one extreme, we completely avoid con-

[3] The probability of a job straggling can be at most ϵ, *i.e.,* $1-(1-p^c)^n \leq \epsilon$. The equation is derived by solving for c.

[4] Intermediate data typically only exists on a single machine, as it is not replicated to avoid time and resource overheads. Some systems do replicate intermediate data [4, 21] for fault-tolerance but limit this to replicating only a small fraction of the data.



(a) Contention-Avoidance Cloning (CAC)



(b) Contention Cloning (CC)

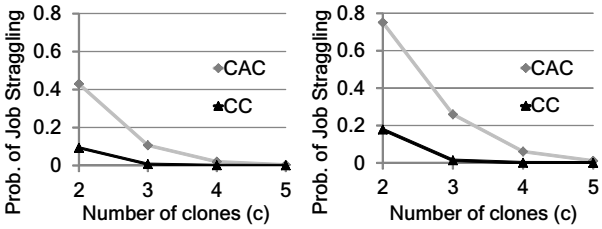
Figure 4: **Intermediate data contention. The example job contains two upstream tasks (U1 and U2) and two downstream tasks (D1 and D2), each cloned twice. The clone of U1 is a straggler (marked with a dotted circle). CAC waits for the straggling clone while CC picks the earliest clone.**

tention by assigning each upstream clone, as it finishes, to a new downstream task clone. This avoids contention because it guarantees that every upstream task clone only transfers data to a single clone per downstream clone group. At another extreme, the system 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. As we show (§4.2), the latter better mitigates stragglers compared to the former strategy. However, we show (§4.3) that the latter may lead to congestion whereas the former completely avoids it. Finally (§4.4), we settle on a hybrid between the two (§4.4), *delay assignment* that far outperforms these two pure strategies.

4.1 Two Opposite Strategies

We illustrate two approaches at the opposite ends of the spectrum through a simple example. Consider a job with two phases (see Figure 4) and an all-to-all (*e.g.,* shuffle) communication pattern between them (§4.4 shows how this can be generalized to other patterns). Each of the phases consist of two tasks, and each task has two clones.

The first option (Figure 4a), which we call Contention-Avoidance Cloning (CAC) eschews contention altogether. As soon as an upstream task clone finishes, its output is sent to exactly one downstream task clone per clone group. Thus, the other downstream task clones have to wait for another upstream task clone to finish before they can start their computation. We call this Contention-Avoidance Cloning (CAC). Note that in CAC an upstream clone will send its intermediate data to the exact same number of other tasks as if no cloning was done, avoiding contention due to cloning. The disadvantage with CAC is that when some upstream clones straggle, the corresponding downstream clones that read data from them automatically lag behind.



(a) $n = 10$ tasks (b) $n = 20$ tasks

Figure 5: **CAC vs. CC: Probability of a job straggling.**

The alternate option (Figure 4b), Contention Cloning (CC), alleviates this problem by making all the tasks in a downstream clone group read the output of the upstream clone that finishes first. This ensures that no downstream clone is disadvantaged, however, all of them may slow down due to contention on disk or network bandwidth.

There are downsides to both CAC and CC. The next two sub-sections quantify these downsides.

4.2 Probability of Job Straggling: CAC vs. CC

CAC increases the vulnerability of a job to stragglers by negating the value of some of its clones. We first analytically derive the probability of a job straggling with CAC and CC, and then compare them for some representative job sizes. We use a job with n upstream and n downstream tasks, with c clones of each task.

CAC: A job straggles with CAC when either the upstream clones straggle and consequently handicap the downstream clones, or the downstream clones straggle by themselves. We start with the upstream phase first before moving to the downstream phase.

The probability that at least d upstream clones of every clone group will succeed without straggling is given by the function Ψ; p is the probability of a task straggling.

$$\Psi(n,c,d) = \text{Probability}[n \text{ upstream tasks of } c \text{ clones with } \geq d \text{ non-stragglers per clone group}]$$

$$\Psi(n,c,d) = \left(\sum_{i=0}^{c-d} \binom{c}{i} p^i (1-p)^{c-i}\right)^n \qquad (1)$$

Therefore, the probability of exactly d upstream clones not straggling is calculated as:

$$\Psi(n, c, d) - \Psi(n, c, d-1)$$

Recall that there are n downstream tasks that are cloned c times each. Therefore, the probability of the whole job straggling is essentially the probability of a straggler occurring in the downstream phase, conditional on the number of upstream clones that are non-stragglers.

$$\text{Probability[Job straggling with CAC]} = 1 - \sum_{d=1}^{c} \left[\Psi(n, c, d) - \Psi(n, c, d-1)\right] \left(1 - p^d\right)^n \quad (2)$$

CC: CC assigns all downstream clones to the output of the first upstream task that finishes in every clone group. As all the downstream clones start at the same time, none of them are handicapped. For a job to succeed without straggling, it only requires that one of the upstream clones in each clone group be a non-straggler. Therefore, the probability of the job straggling is:

$$\text{Probability[Job straggling with CC]} = 1 - \Psi(n, c, 1)\left(1 - p^c\right)^n \quad (3)$$

CAC vs. CC: We now compare the probability of a job straggling with CAC and CC for different job sizes. Figure 5 plots this for jobs with 10 and 20 upstream and downstream tasks each. With three clones per task, the probability of the job straggling increases by over 10% and 30% with CAC compared to CC. Contrast this with our algorithm in §3.2 which aims for an ϵ of 5%. The gap between CAC and CC diminishes for higher numbers of clones but this is contradictory to our decision to pick task-level cloning as we wanted to limit the number of clones. In summary, CAC significantly increases susceptibility of jobs to stragglers compared to CC.

4.3 I/O Contention with CC

By assigning all tasks in a downstream clone group to read the output of the earliest upstream clone, CC causes contention for IO bandwidth. We quantify the impact due to this contention using a micro-benchmark rather than using mathematical analysis to model IO bandwidths, which for contention is likely to be inaccurate.

With the goal of realistically measuring contention, our micro-benchmark replicates the all-to-all data shuffle portion of jobs in the Facebook trace. The experiment is performed on the same 150 node cluster we use for Dolly's evaluation (§5). Every downstream task reads its share of the output from each of the upstream tasks. All the reads start at exactly the same relative time as in the original trace and read the same amount of data from every upstream task's output. The reads of all the downstream tasks of a job together constitute a *transfer* [22].

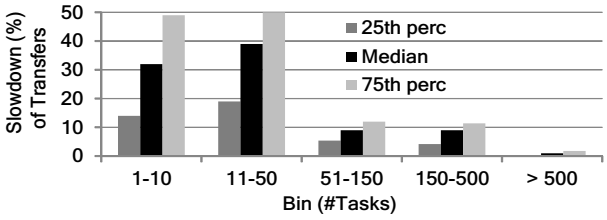


Figure 6: **Slowdown (%) of transfer of intermediate data between phases (all-to-all) due to contention by CC.**

The number of clones per upstream and downstream task is decided as in §3. In the absence of stragglers, there would be as many copies of the upstream outputs as there are downstream clones. However, a fraction of the upstream clones will be stragglers. When upstream clones straggle, we assume their copy of the intermediate data is not available for the transfer. Naturally, this causes contention among the downstream clones.

Reading contended copies of intermediate data likely results in a lower throughput than when there are exclusive copies. Of interest to us is the slowdown in the transfer of the downstream phase due to such contentions, compared to the case where there are as many copies of the intermediate data as there are downstream clones.

Figure 6 shows the slowdown of transfers in each bin of jobs. Transfers of jobs in the first two bins 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, we see that contentions cause significant slowdown of transfers and are worth avoiding.

4.4 Delay Assignment

The analyses in §4.2 and §4.3 conclude that both CAC and CC have downsides. Contentions with CC are not small enough to be ignored. Following strict CAC is not the solution either because it diminishes the benefits of cloning. A deficiency with both CAC and CC is that they do not distinguish stragglers from tasks that have normal (but minor) variations in their progress. CC errs on the side of assuming that all clones other than the earliest are stragglers, while CAC assumes all variations are normal.

We develop a hybrid approach, *delay assignment*, that first waits to assign the early upstream clones (like CAC), and thereafter proceeds without waiting for any remaining stragglers (like CC). Every downstream clone waits for a small window of time (ω) to see if it can get an exclusive copy of the intermediate data. The wait time of ω allows for normal variations among upstream clones. If the downstream clone does not get its exclusive copy even after waiting for ω, it reads with contention from one of the finished upstream clone's outputs.

Crucial to delay assignment's performance is setting the wait time of ω. We next proceed to discuss the analysis that picks a balanced value of ω.

Setting the delay (ω)**:** The objective of the analysis is to minimize the expected duration of a downstream task, which is the minimum of the durations of its clones.

We reuse the scenario from Figure 4. After waiting for ω, the downstream clone either gets its own exclusive copy, or reads the available copy with contention with the other clone. We denote the durations for reading the data in these two cases as T_E and T_C, respectively. In estimating read durations, we eschew detailed modeling of systemic and network performance. Further, we make the simplifying assumption that all downstream clones can read the upstream output (of size r) with a bandwidth of B when there is no contention, and αB in the presence of contention ($\alpha \leq 1$).

Our analysis, then, performs the following 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 ω that minimizes the task's duration.

Step (1)**:** We first calculate T_C, *i.e.,* the case where the clone waits for ω but does not get its exclusive copy, and contends with the other clone. The downstream clone that started reading first will complete its read in $\left(\omega + \left(\frac{r-Bw}{\alpha B}\right)\right)$, *i.e.,* it reads for ω by itself and contends with the other clone for the remaining time. The other clone takes $\left(2\omega + \left(\frac{r-Bw}{\alpha B}\right)\right)$ to read the data.

Alternately, if the clone gets its exclusive copy, then the clone that began reading first reads without interruption and completes its read in $\left(\frac{r}{B}\right)$. The other clone, since it gets its own copy too, takes $\left(\frac{r}{B} + \min(\frac{r}{B}, \omega)\right)$ to read the data.[5] Now that we have calculated T_C and T_E, the expected duration of the task for reading this upstream output is simply $p_c T_C + (1 - p_c)\, T_E$, where p_c is the probability of the task not getting an exclusive copy. Note that, regardless of the number of clones, every clone is assigned an input source latest at the end of ω. Unfinished upstream clones at that point are killed.

Step (2)**:** Every clone may have to read the outputs of multiple upstream clones, depending on the intermediate data communication pattern. In all-to-all communication, a task reads data from each upstream task's output. In one-to-one or many-to-one communications, a task reads data from just one or few tasks upstream of it. Therefore, the total time T_i taken by clone i of a task is obtained by considering its read durations from each of the relevant upstream tasks, along with the expected time for computation. The expected duration of the task is the minimum of all its clones, $\min_i (T_i)$.

Step (3)**:** The final step is to find ω that minimizes this expected task duration. We sample values of B and α, p_c and the computation times of tasks from samples of completed jobs. The value of B depends on the number of active flows traversing a machine, while the p_c is inversely proportional to ω. Using these, we pick ω that minimizes the duration of a task calculated in step (2). The value of ω is calculated periodically and automatically for different job bins (see §5.2). A subtle point with our analysis is that it automatically considers the option where clones read from the available upstream output, one after the other, without contending.

A concern in the strategy of delaying a task is that it is not work-conserving and also somewhat contradicts the observation in §2 that waiting before deciding to speculate is harmful. Both concerns are ameliorated by the fact that we eventually pick a wait duration that minimizes the completion time. Therefore, our wait is not because we lack data to make a decision but precisely because the data dictates that we wait for the duration of ω.

5 Evaluation

We evaluate Dolly using a prototype built by modifying the Hadoop framework [8]. We deploy our prototype on a 150-node cluster and evaluate it using workloads derived from the Facebook and Bing traces (§2), indicative of Hadoop and Dryad clusters. In doing so, we preserve the inter-arrival times of jobs, distribution of job sizes, and the DAG of the jobs from the original trace. The jobs in the Dryad cluster consist of multiple phases with varied communication patterns between them.

5.1 Setup

Prototype Implementation: We modify the job scheduler of Hadoop 0.20.2 [8] to implement Dolly. The two main modifications are launching clones for every task and assigning map outputs to reduce clones such that they read the intermediate data without contention.

When a job is submitted, its tasks are queued at the scheduler. For every queued task, the scheduler spawns many clones. Clones are indistinguishable and the scheduler treats every clone as if it were another task.

The all-to-all transfer of intermediate data is implemented as follows in Hadoop. When map tasks finish, they notify the scheduler about the details of their outputs. The scheduler, in turn, updates a synchronized list of available map outputs. Reduce tasks start after a fraction of the map tasks finish [23]. On startup, they poll on the synchronized list of map outputs and fetch their data as and when they become available. There are two changes we make here. First, every reduce task differen-

[5] The wait time of ω is an upper limit. The downstream clone can start as soon as the upstream output arrives.

Bin	1	2	3	4	5
Tasks	1–10	11–50	51–150	151–500	> 500

Table 3: **Job bins, binned by their number of tasks.**

tiates between map clones and avoids repetitive copying. Second, tasks in a reduce clone group notify each other when they start reading the output of a map clone. This helps them wait to avoid contention.

Deployment: We deploy our prototype on a private cluster with 150 machines. Each machine has 24GB of memory, 12 cores, and 2TB of storage. The machines have 1Gbps network links connected in a topology with full bisection bandwidth. Each experiment is repeated five times and we present the median numbers.

Baseline: Our baselines for evaluating Dolly are the state-of-the-art speculation algorithms—LATE [5] and Mantri [4]. Additionally, with each of these speculation strategies, we also include a blacklisting scheme that avoids problematic machines (as described in §2.1.2).

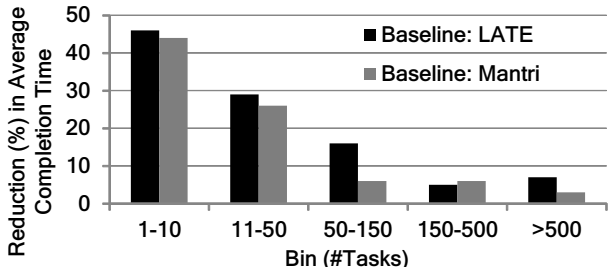
In addition to overall improvement in average completion time of jobs, we bin jobs by their number of tasks (see Table 3) and report the average improvement in each bin. The following is a summary of our results.

- Average completion time of small jobs improves by 34% to 46% compared to LATE and Mantri, using fewer than 5% extra resources (§5.2 and §5.4).
- Delay assignment outperforms CAC and CC by 2×. Its benefit increases for jobs with higher number of phases and all-to-all intermediate data flow (§5.3).
- Admission control of jobs is a good approximation for preemption in favoring small jobs (§5.5).

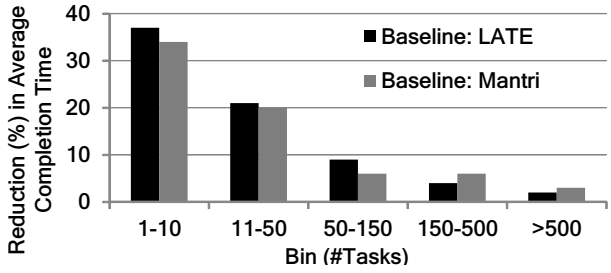
5.2 Does Dolly mitigate stragglers?

We first present the improvement in completion time using Dolly. Unless specified otherwise, the cloning budget β is 5% and utilization threshold τ is 80%.

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. Figure 7 plots the improvement in different job bins. Small jobs (bin-1) 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 is because of the power-law in job sizes and the policy of admission control. Figures 8a and 8b show the average duration of jobs in the smallest two bins with LATE and Mantri, and its reduction due to Dolly's cloning, for the Facebook workload. Figure 8c shows the distribution of gains for jobs in bin-1. We see that jobs improve by



(a) Facebook workload.



(b) Bing workload.

Figure 7: Dolly**'s improvement for the Facebook and Bing workloads, with LATE and Mantri as baselines.**

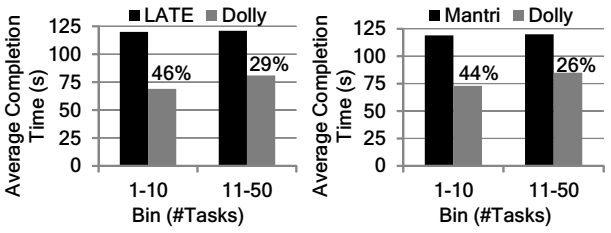
nearly 50% and 60% at the 75$^{\text{th}}$ and 90$^{\text{th}}$ percentiles, respectively. Note that even at the 10$^{\text{th}}$ percentile, there is a non-zero improvement, demonstrating the seriousness and prevalence of the problem of stragglers in small jobs.

Figure 9 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 95$^{\text{th}}$ percentile, this ratio is only 1.17, thereby indicating that Dolly effectively mitigates nearly all stragglers.

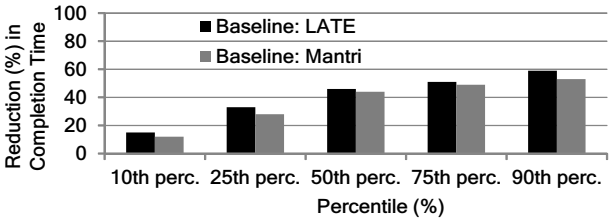
The ratio not being exactly 1 shows that some stragglers still remain. One reason for this is that while our policy of admission control is a good approximation (§3.3), it does not explicitly prioritize small jobs. Hence a few large jobs possibly deny the budget to some small jobs. Analyzing the consumption of the cloning budget shows that this is indeed the case. Jobs in bin-1 and bin-2 together consume 83% of the cloning budget. However, even jobs in bin-5 get a small share (2%) of the budget.

5.3 Delay Assignment

Setting ω: Crucial to the above improvements is delay assignment's dynamic calculation of the wait duration of ω. The value of ω, picked using the analysis in §4.4, is updated every hour. It varied between 2.5s and 4.7s for jobs in bin-1, and 3.1s and 5.2s for jobs in bin-2. The value of ω varies based on job sizes because the number of tasks in a job influences B, α and p_c. Figure 10 plots the variation with time. The sensitivity of ω to the periodicity of updating its value is low—using values between

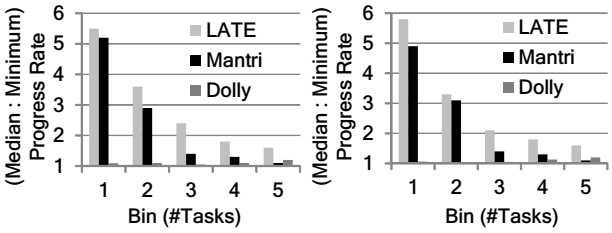


(a) Job Durations. (b) Job Durations.



(c) Distribution of improvements ($\leq$ 10 tasks).

Figure 8: **Dissecting Dolly's improvements for the Facebook workload. Figures (a) and (b) show the duration of the small jobs before and after Dolly. Figure (c) expands on the distribution of the gains for jobs with $\leq$ 10 tasks.**



(a) Facebook (b) Bing

Figure 9: **Ratio of median to minimum progress rates of tasks within a phase. Bins are as per Table 3.**

30 minutes to 3 hours causes little change in its value.

CC and CAC: We now compare 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 2.1× better). With Mantri as the baseline, delay assignment is again up to 2.1× better. In the Facebook workload, delay assignment is at least 1.7× better.

The main reason behind delay assignment's better performance is its accurate estimation of the effect of contention and the likelihood of stragglers. It uses sampling from prior runs to estimate both. Bandwidth estimation is 93% accurate without contention and 97% accurate with contention. Also, the probability of an upstream

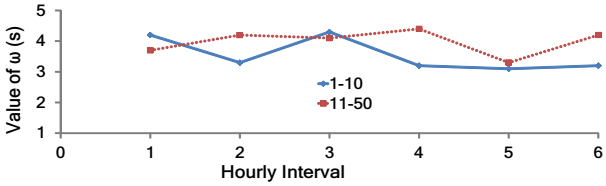
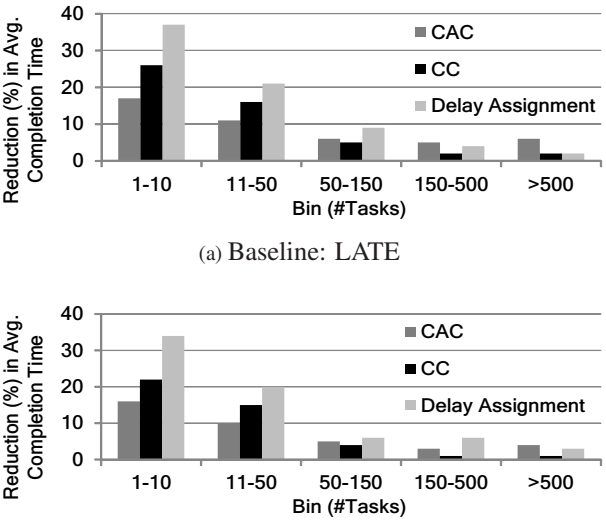


Figure 10: **Variation in ω when updated every hour.**



(a) Baseline: LATE

(b) Baseline: Mantri

Figure 11: **Intermediate data contention. Delay Assignment is 2.1× better than CAC and CC (Bing workload).**

clone straggling is estimated to an accuracy of 95%.

Between the two, CC is a closer competitor to delay assignment than CAC, for small jobs. This is because they transfer only moderate amounts of data. However, contentions hurt large jobs as they transfer sizable intermediate data. As a result, CC's gains drop below CAC.

Number of Phases: Dryad jobs may have multiple phases (maximum of 6 in our Bing traces), and tasks of different phases have the same number of clones. More phases increases the chances of there being fewer exclusive copies of task outputs, which in turn worsens the effect of both waiting as well as contention. Figure 12 measures the consequent drop in performance. CAC's gains drop quickly while CC's performance drops at a moderate rate. Importantly, delay assignment's performance only has a gradual and relatively small drop. Even when the job has six phases, improvement is at 31%, a direct result of its deft cost-benefit analysis (§4.4).

Communication Pattern: Delay assignment is generic to handle any communication pattern between phases. Figure 13 differentiates the gains in completion times of the *phases* based on their communication pattern. Results show that delay assignment is significantly more

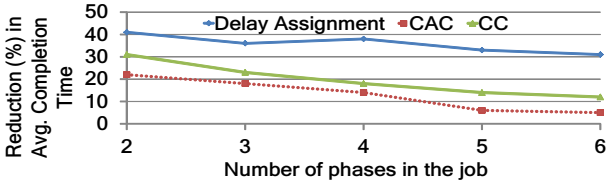


Figure 12: Dolly's gains as the number of phases in jobs in bin-1 varies in the Bing workload, with LATE as baseline.

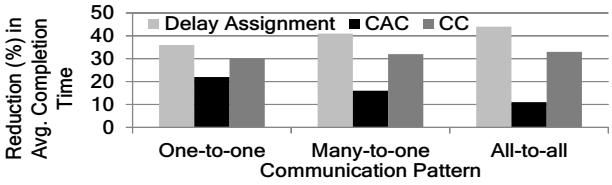
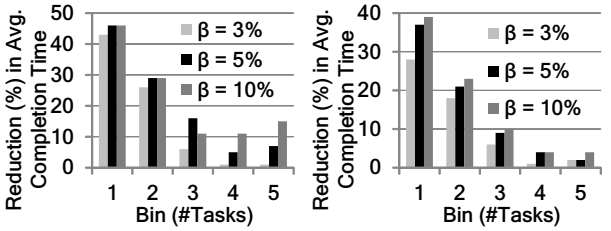


Figure 13: Performance of Dolly across phases with different communication patterns in bin-1, in the Bing workload.



(a) Facebook (b) Bing

Figure 14: Sensitivity to cloning budget (β). Small jobs see a negligible drop in performance even with a 3% budget.

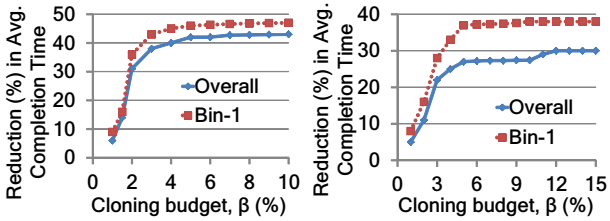
valuable for all-to-all communication patterns than the many-to-one and one-to-one patterns. The higher the dependency among communicating tasks, the greater the value of delay assignment's cost-benefit analysis.

Overall, we believe the above analysis shows the applicability and robust performance of Dolly's mechanisms to different frameworks with varied features.

5.4 Cloning Budget

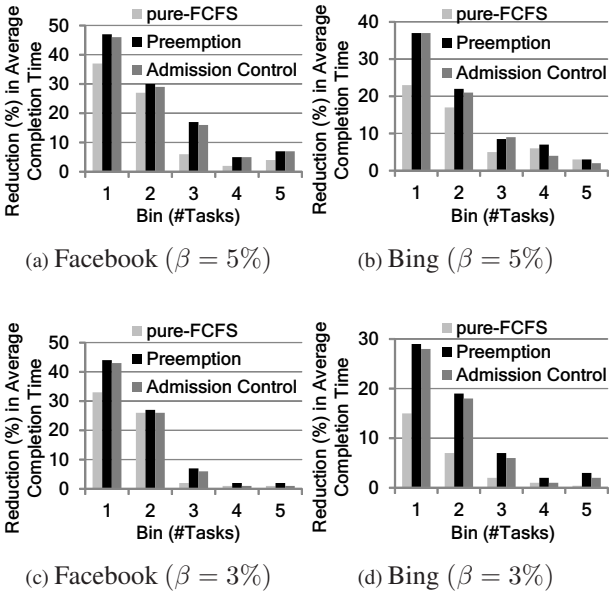
The improvements in the previous sections are based on a cloning budget β of 5%. In this section, we analyze the sensitivity of Dolly's performance to β. We aim to understand whether the gains hold for lower budgets and how much further gains are obtained at higher budgets.

In the Facebook workload, overall improvement remains at 38% compared to LATE even with a cloning budget of only 3% (Figure 14a). Small jobs, in fact, see a negligible drop in gains. This is due to the policy of admission control to favor small jobs. Large jobs take a non-negligible performance hit though. In fact, in the Bing workload, even the small jobs see a drop of 7%



(a) Facebook (b) Bing

Figure 15: Sweep of β to measure the overall average completion time of all jobs and specifically those within bin-1.



(a) Facebook ($\beta = 5\%$) (b) Bing ($\beta = 5\%$)

(c) Facebook ($\beta = 3\%$) (d) Bing ($\beta = 3\%$)

Figure 16: **Admission Control. The policy of admission control well approximates the policy of preemption and outperforms pure-FCFS in utilizing the cloning budget.**

when the budget is reduced from 5% to 3%. This is because job sizes in Bing are less heavy-tailed. However, the gains still stand at a significant 28% (Figure 14b).

Increasing the budget to 10% does not help much. Most of the gains are obtained by eliminating stragglers in the smaller jobs, which do not require a big budget.

In fact, sweeping the space of β (Figure 15) reveals that Dolly requires a cloning budget of at least 2% and 3% for the Facebook and Bing workloads, below which performance drops drastically. Gains in the Facebook workload plateau beyond 5%. In the Bing workload, gains for jobs in bin-1 plateau at 5% but the overall gains cease to grow only at 12%. While this validates our setting of β as 5%, clusters can set their budgets based on their utilizations and the jobs they seek to improve with cloning.

5.5 Admission Control

A competing policy to admission control (§3.3) is to preempt clones of larger jobs for the small jobs. Preemption is expected to outperform admission control as it explicitly prioritizes the small jobs; we aim to quantify the gap.

Figure 16 presents the results with LATE as the baseline and cloning budgets of 5% and 3%. The gains with preemption is 43% and 29% in the Facebook and Bing workloads, compared to 42% and 27% with the policy of admission control. This small difference is obtained by preempting 8% and 9% of the tasks in the two workloads. Lowering the cloning budget to 3% further shrinks this difference, even as more tasks are preempted. With a cloning budget of 3%, the improvements are nearly equal, even as 17% of the tasks are preempted, effectively wasting cluster resources. Admission control well approximates preemption due to the heavy tailed distribution. Note the near-identical gains for small jobs.

Doing neither preemption or admission control in allocating the cloning budget ("pure-FCFS") reduces the gains by nearly 14%, implying this often results in larger jobs denying the cloning budget to the smaller jobs.

6 Related Work

Replicating tasks in distributed systems have a long history [24, 25, 26], and have been studied extensively [27, 28, 29] in prior work. These studies conclude that modeling running tasks and using it for predicting and comparing performance of other tasks is the hardest component, errors in which often cause degradation in performance. We concur with a similar observation in our traces.

The problem of stragglers was identified in the original MapReduce paper [1]. Since then solutions have been proposed to fix it using speculative executions [2, 4, 5]. Despite these techniques, stragglers remain a problem in small jobs. Dolly addresses their fundamental limitation—wait to observe before acting—with a proactive approach of cloning jobs. It does so using few extra resources by relying on the power-law of job sizes.

Based on extensive research on detecting faults in machines (*e.g.,* [30, 31, 32, 33, 34]), datacenters periodically check for faulty machines and avoid scheduling jobs on them. However, stragglers continue to occur on the non-blacklisted machines. Further improvements to blacklisting requires a root cause analysis of stragglers in small jobs. However, this is intrinsically hard due to the complexity of the hardware and software modules, a problem recently acknowledged in Google's clusters [6].

In fact, Google's clusters aim to make jobs "predictable out of unpredictable parts" [6]. They overcome vagaries in performance by scheduling backup copies for every job. Such backup requests are also used in Amazon's Dynamo [35]. This notion is similar to Dolly. However, these systems aim to overcome variations in scheduling delays on the machines, not runtime stragglers. Therefore, they cancel the backup copies once one of the copies starts. In contrast, Dolly has to be resilient to runtime variabilities which requires functioning within utilization limits and efficiently handle intermediate data.

Finally, our delay assignment model is similar to the idea of delay scheduling [36] that delays scheduling tasks for locality. We borrow this idea in Dolly, but crucially, pick the value of the delay based on a cost-benefit analysis weighing contention versus waiting for slower tasks.

7 Conclusions and Future Work

Analysis of production traces from Facebook and Microsoft Bing show that straggler tasks continue to affect small interactive jobs by 47% even after applying state-of-the-art mitigation techniques [4, 5]. This is because these techniques wait before launching speculative copies. Such waiting bounds their agility for small jobs that run all their tasks at once.

In this paper we developed a system, Dolly, that launches multiple clones of jobs, completely removing waiting from straggler mitigation. Cloning of small jobs can be achieved with few extra resources because of the heavy-tail distribution of job sizes; the majority of the jobs are small and can be cloned with little overhead. 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. We developed *delay assignment* 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.

Going forward, we plan to evaluate Dolly's compatibility with caching systems proposed for computation frameworks. These systems rely on achieving *memory locality*—scheduling a task on the machine that caches its input—along with cache replacement schemes targeted for parallel jobs [37]. Analyzing (and dealing with) the impact of multiple clones for every task on both these aspects is a topic for investigation.

We also plan to extent Dolly to deal with clusters that deploy multiple computation frameworks. Trends indicate a proliferation of frameworks, based on different computational needs and programming paradigms (*e.g.,* [3, 7]). Such specialized frameworks may, perhaps, lead to homogeneity of job sizes within them. Challenges in extending Dolly to such multi-framework clusters includes dealing with any weakening of the heavy-tail distribution, a crucial factor behind Dolly's low overheads.

References

[1] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *USENIX OSDI*, 2004.

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

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

[4] 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.

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

[6] J. Dean. *Achieving Rapid Response Times in Large Online Services*. `http://research.google.com/people/jeff/latency.html`.

[7] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, T. Vassilakis. Dremel: Interactive Analysis of Web-Scale Datasets. In *VLDB*, 2010.

[8] Hadoop. `http://hadoop.apache.org`.

[9] Hadoop distributed file system. `http://hadoop.apache.org/hdfs`.

[10] Hive. `http://wiki.apache.org/hadoop/Hive`.

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

[12] Y. Yu *et al.* Distributed Aggregation for Data-Parallel Computing: Interfaces and Implementations. In *ACM SOSP*, 2009.

[13] G. Ananthanarayanan, C. Douglas, R. Ramakrishnan, S. Rao, and I. Stoica. True Elasticity in Multi-Tenant Clusters through Amoeba. In *ACM SoCC*, 2012.

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

[15] L. A. Barroso. Warehouse-scale computing: Entering the teenage decade. In *ISCA*, 2011.

[16] Y. Chen, S. Alspaugh, D. Borthakur, R. Katz. Energy Efficiency for Large-Scale MapReduce Workloads with Significant Interactive Analysis. In *ACM EuroSys*, 2012.

[17] J. Wilkes and C. Reiss., 2011. `https://code.google.com/p/googleclusterdata/wiki/ClusterData2011_1`.

[18] C. Reiss, A. Tumanov, G. Ganger, R. H. Katz, M. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *ACM SoCC*, 2012.

[19] A. Thusoo . Data warehousing and analytics infrastructure at facebook. In *SIGMOD*, 2010.

[20] G. Ananthanarayanan, A. Ghodsi, S. Shenker, I. Stoica. Disk Locality Considered Irrelevant. In *USENIX HotOS*, 2011.

[21] S. Ko, I. Hoque, B. Cho, I. Gupta. Making Cloud Intermediate Data Fault-Tolerant. In *ACM SOCC*, 2010.

[22] M. Chowdhury, M. Zaharia, J. Ma, M. Jordan, I. Stoica. Managing Data Transfers in Computer Clusters with Orchestra. In *ACM SIGCOMM*, 2011.

[23] Hadoop Slowstart. `https://issues.apache.org/jira/browse/MAPREDUCE-1184/`.

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

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

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

[27] 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.

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

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

[30] A. Merchant, M. Uysal, P. Padala, X. Zhu, S. Singhal, and K. Shin. Maestro: Quality-of-Service in Large Disk Arrays. In *ACM ICAC*, 2011.

[31] E. Ipek, M. Krman, N. Krman, and J. F. Martinez. Core Fusion: Accommodating Software Diversity in Chip Multiprocessors. In *ISCA*, 2007.

[32] J. G. Elerath and S. Shah. Dependence upon fly-height and quantity of heads. In *Annual Symposium on Reliability and Maintainability*, 2003.

[33] J. G. Elerath and S. Shah. Server class disk drives: How reliable are they? In *Annual Symposium on Reliability and Maintainability*, 2004.

[34] J. Gray and C. van Ingen. Empirical measurements of disk failure rates and error rates. In *Technical Report MSR-TR-2005-166*, 2005.

[35] G. DeCandia and D. Hastorun and M. Jampani and G. Kakulapati and A. Lakshman and A. Pilchin and S. Sivasubramanian and P. Vosshall and W. Vogels. Dynamo: Amazons Highly Available Key-value Store. In *ACM SOSP*, 2007.

[36] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *ACM EuroSys*, 2010.

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