Reining in the Outliers in Map-Reduce Clusters using Mantri

Ganesh Ananthanarayanan[†◇] Srikanth Kandula[†] Albert Greenberg[†]
Ion Stoica[◇] Yi Lu[†] Bikas Saha[‡] Edward Harris[‡]
[†]*Microsoft Research* [◇] *UC Berkeley* [‡] *Microsoft Bing*

Abstract– Experience from an operational Map-Reduce cluster reveals that outliers significantly prolong job completion. The causes for outliers include run-time contention for processor, memory and other resources, disk failures, varying bandwidth and congestion along network paths and, imbalance in task workload. We present Mantri, a system that monitors tasks and culls outliers using *cause-* and *resource-aware* techniques. Mantri's strategies include restarting outliers, network-aware placement of tasks and protecting outputs of valuable tasks. Using real-time progress reports, Mantri detects and acts on outliers early in their lifetime. Early action frees up resources that can be used by subsequent tasks and expedites the job overall. Acting based on the causes and the resource and opportunity cost of actions lets Mantri improve over prior work that only duplicates the laggards. Deployment in Bing's production clusters and trace-driven simulations show that Mantri improves job completion times by 32%.

1 Introduction

In a very short time, Map-Reduce has become the dominant paradigm for large data processing on compute clusters. Software frameworks based on Map-Reduce [1, 11, 13] have been deployed on tens of thousands of machines to implement a variety of applications, such as building search indices, optimizing advertisements, and mining social networks.

While highly successful, Map-Reduce clusters come with their own set of challenges. One such challenge is the often unpredictable performance of the Map-Reduce jobs. A job consists of a set of tasks which are organized in phases. Tasks in a phase depend on the results computed by the tasks in the previous phase and can run in parallel. When a task takes longer to finish than other similar tasks, tasks in the subsequent phase are delayed. At key points in the job, a few such *outlier* tasks can prevent the rest of the job from making progress. As the size of the cluster and the size of the jobs grow, the impact of outliers increases dramatically. Addressing the outlier problem is critical to speed up job completion and improve cluster efficiency.

Even a few percent of improvement in the efficiency of a cluster consisting of tens of thousands of nodes can save millions of dollars a year. In addition, finishing production jobs quickly is a competitive advantage. Doing so predictably allows SLAs to be met. In iterative modify/ debug/ analyze development cycles, the ability to iterate faster improves programmer productivity.

In this paper, we characterize the impact and causes of outliers by measuring a large Map-Reduce production cluster. This cluster is up to two orders of magnitude larger than those in previous publications [1, 13, 20] and exhibits a high level of concurrency due to many jobs simultaneously running on the cluster and many tasks on a machine. We find that variation in completion times among functionally similar tasks is large and that outliers inflate the completion time of jobs by 34% at median.

We identify three categories of root causes for outliers that are induced by the interplay between storage, network and structure of Map-Reduce jobs. First, *machine characteristics* play a key role in the performance of tasks. These include static aspects such as hardware reliability (e.g., disk failures) and dynamic aspects such as contention for processor, memory and other resources. Second, *network characteristics* impact the data transfer rates of tasks. Datacenter networks are over-subscribed leading to variance in congestion among different paths. Finally, the specifics of Map-Reduce leads to *imbalance* in work – partitioning data over a low entropy key space often leads to a skew in the input sizes of tasks.

We present Mantri[1], a system that monitors tasks and culls outliers based on their causes. It uses the following techniques: (i) Restarting outlier tasks cognizant of resource constraints and work imbalances, (ii) Network-aware placement of tasks, and (iii) Protecting output of tasks based on a cost-benefit analysis.

The detailed analysis and decision process employed by Mantri is a key departure from the state-of-the-art for outlier mitigation in Map-Reduce implementations [11, 13, 20]; these focus only on duplicating tasks. To our knowledge, none of them protect against data loss induced recomputations or network congestion induced outliers. Mantri places tasks based on the locations of their data sources as well as the current utilization of network links. On a task's completion, Mantri replicates its output if the

[1]From Sanskrit, a minister who keeps the king's court in order

benefit of not having to recompute outweighs the cost of replication.

Further, Mantri performs intelligent restarting of outliers. A task that runs for long because it has more work to do will not be restarted; if it lags due to reading data over a low-bandwidth path, it will be restarted only if a more advantageous network location becomes available. Unlike current approaches that duplicate tasks only at the end of a phase, Mantri uses real-time progress reports to act early. While early action on outliers frees up resources that could be used for pending tasks, doing so is nontrivial. A duplicate may finish faster than the original task but has the opportunity cost of consuming resources that other pending work could have used.

In summary we make the following contributions. First, we provide an analysis of the causes of outliers in a large production Map-Reduce cluster. Second, we develop Mantri, that takes early actions based on understanding the causes and the opportunity cost of actions. Finally, we perform an extensive evaluation of Mantri and compare it to existing solutions.

Mantri runs live in all of Bing's production clusters since May 2010. Results from a deployment of Mantri on a production cluster of thousands of servers and from replaying several thousand jobs collected on this cluster in a simulator show that:

- Mantri reduces the completion time of jobs by 32% on average on the production clusters. Extensive simulations show that job phases are quicker by 21% and 42% at the 50th and 75th percentiles. Mantri's median reduction in completion time improves on the next best scheme by 3.1x while using fewer resources.
- By placing *reduce* tasks to avoid network hotspots, Mantri improves the completion times of the reduce phases by 60%.
- By preferentially replicating the output of tasks that are more likely to be lost or expensive to recompute, Mantri speeds up half of the jobs by at least 20% each while only increasing the network traffic by 1%.

2 Background

We monitored the cluster and software systems that support the Bing search engine for over twelve months. This is a cluster of tens of thousands of commodity servers managed by Cosmos [8], a proprietary upgraded form of Dryad [13]. Despite a few differences, implementations of Map-Reduce [1, 8, 11, 13] are broadly similar.

Most of the jobs in the examined cluster are written in Scope [8], a mash-up language that mixes SQL-like declarative statements with user code. The Scope compiler transforms a job into a workflow– a directed acyclic graph where each node is a phase and each edge joins a phase that produces data to another that uses it. A phase is a set of one or more tasks that run in parallel and perform the same computation on different parts of the input stream. Typical phases are map, reduce and join. The number of tasks in a phase is chosen at compile time. A task will read its input over the network if it is not available on the local disk but outputs are written to the local disk. The eventual outputs of a job (as well as raw data) are stored in a reliable block storage system implemented on the same servers that do computation. Blocks are replicated n-ways for reliability. A run-time scheduler assigns tasks to machines, based on data locations, dependence patterns and cluster-wide resource availability. The network layout provides more bandwidth within a rack than across racks.

We obtain detailed logs from the Scope compiler and the Cosmos scheduler. At each of the job, phase and task levels, we record the execution behavior as represented by begin and end times, the machines(s) involved, the sizes of input and output data, the fraction of data that was read across racks and a code denoting the success or type of failure. We also record the workflow of jobs. Table 1 depicts the random subset of logs that we analyze here. Spanning eighteen days, this dataset is at least one order of magnitude larger than prior published data along many dimensions, e.g., number of jobs, cluster size.

3 The Outlier Problem

We begin with a first principles approach to the outlier problem, then analyze data from the production cluster to quantify the problem and obtain a breakdown of the causes of outliers (§4). Beginning at the first principles motivates a distinct approach (§5) which as we show in §6 significantly improves on prior art.

3.1 Outliers in a Phase

Assume a phase consists of n tasks and has s slots. Slot is a virtual token, akin to a quota, for sharing cluster resources among multiple jobs. One task can run per slot at a time. On our cluster, the median ratio of $\frac{n}{s}$ is 2.11 with a stdev of 12.37. The goal is to minimize the phase completion time, *i.e.*, the time when the last task finishes.

Based on data from the production cluster, we model t_i, the completion time of task i, as a function of the size of the data it processes, the code it runs, the resources available on the machine it executes and the bandwidth available on the network paths involved:

$$t_i = f\left(\text{datasize}, \text{code}, \text{machine}, \text{network}\right). \qquad (1)$$

Large variation exists along each of the four variables leading to considerable difference in task completion times. The amount of data processed by tasks in the same phase varies, sometimes widely, due to limitations in dividing work evenly. The code is the same for tasks in a

Dates 2009-'10	Phases x 10^3	Jobs	Compute (years)	Data (PB)	Network (PB)
May 25,26	19.0	938	49.1	12.6	.66
Jun 16,17	16.5	991	88.0	22.7	1.22
Jul 20,21	22.0	1183	51.6	14.3	.67
Aug 20,21	29.2	1873	60.6	18.7	.76
Sep 15,16	27.4	1653	73.0	22.8	.73
Oct 15,16	20.4	1362	84.1	25.3	.86
Nov 16,17	37.8	1834	88.4	25.0	.68
Dec 10,11	18.7	1777	96.2	18.6	.72
Jan 11,12	24.4	1842	79.5	21.5	1.99

Table 1: Details of the logs from a production cluster consisting of thousands of servers.

phase, but differs significantly across phases (*e.g.*, map and reduce). Placing a task on a machine that has other resource hungry tasks inflates completion time, as does reading data across congested links.

In the ideal scenario, where every task takes the same amount of time, say T, scheduling is simple. Any work-conserving schedule would complete the phase in $\left(\lceil \frac{n}{s} \rceil \times T\right)$. When the task completion time varies, however, a naive work-conserving scheduler can take up to $\left(\frac{\sum_n t_i}{s} + \max t_i\right)$. A large variation in t_i increases the term $\max t_i$ and manifests as outliers.

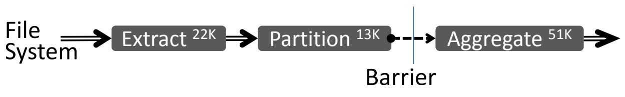
The goal of a scheduler is to minimize the phase completion time and make it closer to $\frac{\sum_n t_i}{s}$. Sometimes, it can do even better. By placing tasks at less congested machines or network locations, the t_i's themselves can be lowered. The challenge lies in recognizing the aspects that can be changed and scheduling accordingly.

3.2 Extending from a phase to a job

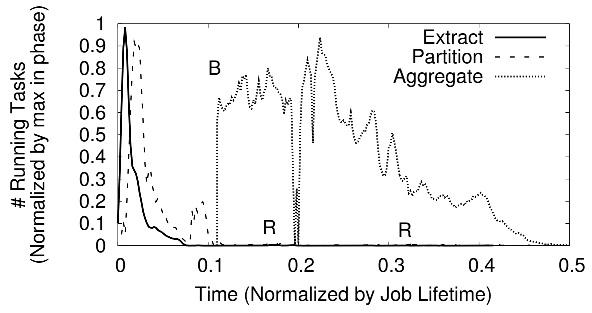
The phase structure of Map-Reduce jobs adds to the variability. An outlier in an early phase, by delaying when tasks that use its output may start, has cumulative effects on the job. At *barriers* in the workflow, where none of the tasks in successive phase(s) can begin until all of the tasks in the preceding phase(s) finish, even one outlier can bring the job to a standstill[2]. Barriers occur primarily due to reduce operations that are neither commutative nor associative [28], for instance, a reduce that computes the median of records that have the same key. In our cluster, the median job workflow has eight phases and eleven edges, 47% are barriers (number of edges exceeds the number of phases due to table joins).

Dependency across phases also leads to outliers when task output is lost and needs to be *recomputed*. Data loss happens due to a combination of disk errors, software errors (*e.g.*, bugs in garbage collectors) and timeouts due to machines going unresponsive at times of high load. In fact, recomputes cause some of the longest waiting times observed on the production cluster. A recompute can cascade into earlier phases if the inputs for the recomputed task are no longer available and need to be regenerated.

[2]There is a variant in implementation where a slot is reserved for a task before all its inputs are ready. This is either to amortize the latency of network transfer by moving data over the network as soon as it is generated [1, 11], or compute partial results and present answers *online* even before the job is complete [9]. Regardless, pre-allocation of slots hogs resources for longer periods if the input task(s) straggle.



(a) Partial workflow with the number of tasks in each phase



(b) Time lapse of task execution (R=Recomputes, B=Barrier).

Figure 1: An example job from the production cluster

3.3 Illustration of Outliers

Figure 1(a) shows the workflow for a job whose structure is typical of those in the cluster. The job reads a dataset of search usage and derives an index. It consists of two Map-Reduce operations and a join, but for clarity we only show the first Map-Reduce here. Phase names follow the Dryad [13] convention– *extract* reads raw blocks, *partition* divides data on the key and *aggregate* reduces items that share a key.

Figure 1(b) depicts a timeline of an execution of this workflow. It plots the number of tasks of each phase that are active, normalized by the maximum tasks active at any time in that phase, over the lifetime of the job. Tasks in the first two phases start in quick succession to each other at ∼.05, whereas the third starts after a barrier.

Some of the outliers are evident in the long lulls before a phase ends when only a few of its tasks are active. In particular, note the regions before x∼.1 and x∼.5. The spike in phase #2 here is due to the outliers in phase #1 holding on to the job's slots. At the barrier, x∼.1, just a few outliers hold back the job from making forward progress. Though most aggregate tasks finish at x∼.3, the phase persists for another 20%.

The worst cases of waiting immediately follow recomputations of lost intermediate data marked by R. Recomputations manifest as tiny blips near the x axes for phases that had finished earlier, *e.g.*, phase #2 sees recomputes at x∼.2 though it finished at x∼.1. At x∼.2, note that aggregate almost stops due to a few recomputations.

We now quantify the magnitude of the outlier problem, before presenting our solution in detail.

4 Quantifying the Outlier Problem

We characterize the prevalence and causes of outliers and their impact on job completion times and cluster resource usage. We will argue that three factors – dynamics, concurrency and scale, that are somewhat unique to large Map-Reduce clusters for efficient and economic operation, lie at the core of the outlier problem. To our knowledge, we are the first to report detailed experiences from a large production Map-Reduce cluster.

4.1 Prevalence of Outliers

Figure 2(a) plots the fraction of high runtime outliers and recomputes in a phase. For exposition, we arbitrarily say that a task has high runtime if its time to finish is longer than 1.5x the median task duration in its phase. By recomputes, we mean instances where a task output is lost and dependent tasks wait until the output is regenerated.

We see in Figure 2(a) that 25% of phases have more than 15% of their tasks as outliers. The figure also shows that 99% of the phases see no recomputes. Though rare, recomputes have a widespread impact (§4.3). Two out of a thousand phases have over 50% of their tasks waiting for data to be recomputed.

How much longer do outliers run for? Figure 2(b) shows that 80% of the runtime outliers last less than 2.5 times the phase's median task duration, with a uniform probability of being delayed by between 1.5x to 2.5x. The tail is heavy and long– 10% take more than 10x the median duration. Ignoring these if they happen early in a phase, as current approaches do, appears wasteful.

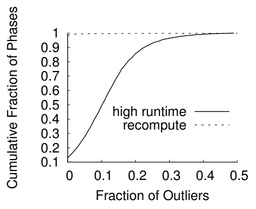
Figure 2(b) shows that most recomputations behave normally, 90% of them are clustered about the median task, but 3% take over 10x longer.

4.2 Causes of Outliers

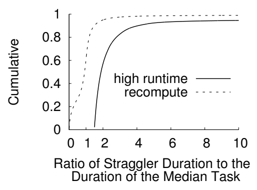
To tease apart the contributions of each cause, we first determine whether a task's runtime can be explained by the amount of data it processes or reads across the network[3]. If yes, then the outlier is likely due to workload imbalance or poor placement. Otherwise, the outlier is likely due to resource contention or problematic machines.

Figure 3(a) shows that in 40% of the phases (top right), all the tasks with high runtimes (i.e., over 1.5x the me-

[3]For each phase, we fit a linear regression model for task lifetime given the size of input and the volume of traffic moved across low bandwidth links. When the residual error for a task is less than 20%, i.e., its run time is within [.8, 1.2]x of the time predicted by this model, we call it explainable.



(a) What fraction of tasks in a phase are outliers?



(b) How much longer do outliers take to finish?

Figure 2: Prevalence of Outliers.

dian task) are well explained by the amount of data they process or move on the network. Duplicating these tasks would not make them run faster and will waste resources. At the other extreme, in 18% of the phases (bottom left), none of the high runtime tasks are explained by the data they process. Figure 3(b) shows tasks that take longer than they should, as predicted by the model, but do not take over 1.5x the median task in their phase. Such tasks present an opportunity for improvement. They may finish faster if run elsewhere, yet current schemes do nothing for them. 20% of the phases (on the top right) have over 55% of such improvable tasks.

Data Skew: It is natural to ask why data size varies across tasks in a phase. Across phases, the coefficient of variation ($\frac{stdev}{mean}$) in data size is .34 and 3.1 at the 50^{th} and 90^{th} percentiles. From experience, dividing work evenly is non-trivial for a few reasons. First, scheduling each additional task has overhead at the job manager. Network bandwidth is another reason. There might be too much data on a machine for a task to process, but it may be worse to split the work into multiple tasks and move data over the network. A third reason is poor coding practice. If the data is partitioned on a key space that has too little entropy, i.e., a few keys correspond to a lot of data, then the partitions will differ in size. Some reduce tasks are not amenable to splitting (neither commutative nor associative [27]), and hence each partition has to be processed by one task. Some joins and sorts are similarly constrained. Duplicating tasks that run for long because they have a lot of work to do is counter-productive.

Crossrack Traffic: Reduce phases contribute over 70% of the cross rack traffic in the cluster, while most of the rest is due to joins. We focus on cross rack traffic because the links upstream of the racks have less bandwidth than the cumulative capacity of servers in the rack.

We find that crossrack traffic leads to outliers in two ways. First, in phases where moving data across racks is avoidable (through locality constraints), a task that ends up in a disadvantageous network location runs slower than others. Second, in phases where moving data across racks is unavoidable, not accounting for the competition among tasks within the phase (self-interference) leads to outliers. In a reduce phase, for example, each task reads

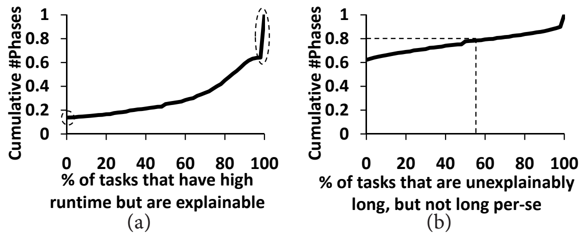


Figure 3: Contribution of data size to task runtime (see §4.2)

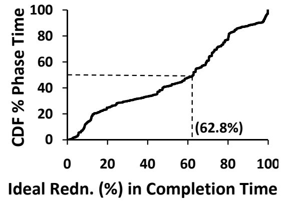


Figure 4: For reduce phases, the reduction in completion time over the current placement by placing tasks in a network-aware fashion.

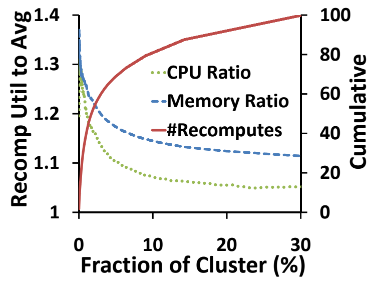


Figure 5: The ratio of processor and memory usage when recomputations happen to the average at that machine (y1). Also, the cumulative percentage of recomputations across machines (y2).

from every map task. Since the maps are spread across the cluster, regardless of where a reduce task is placed, it will read a lot of data from other racks. Current implementations place reduce tasks on any machine with spare slots. A rack that has too many reduce tasks will be congested on its downlink leading to outliers.

Figure 4 compares the current placement with an ideal one that minimizes the impact of network transfer. When possible it avoids reading data across racks and if not, places tasks such that their competition for bandwidth does not result in hotspots. In over 50% of the jobs, reduce phases account for 17% of the job's lifetime. For the reduce phases, the figure shows that the median phase takes 62% longer under the current placement.

Bad and Busy Machines: We rarely find machines that persistently inflate runtimes. Recomputations, however, are more localized. Half of them happen on 5% of the machines in the cluster. Figure 5 plots the cumulative share of recomputes across machines on the axes on the right. The figure also plots the ratio of processor and memory utilization during recomputes to the overall average on that machine. The occurrence of recomputes is correlated

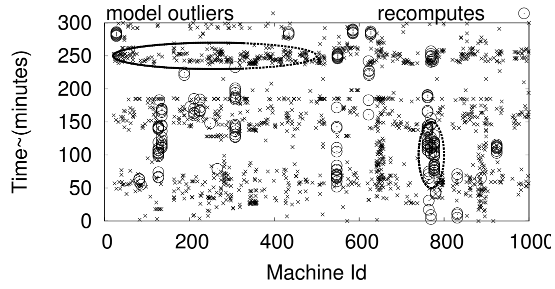


Figure 6: Clustering recomputations and outliers.

with increased use of resources by at least 20%. The subset of machines that triggers most of the recomputes is steady over days but varies over weeks, likely indicative of changing hotspots in data popularity or corruption in disks [7].

Figure 6 investigates the occurrence of "spikes" in outliers. For legibility, we only plot a subset of the machines. We find that runtime outliers (shown as stars) cluster by time. If outliers were happening at random, there should not be any horizontal bands. Rather it appears that jobs contend for resources at some times. Even at these busy times, other lightly loaded machines exist. Recomputations (shown as circles) cluster by machine. When a machine loses the output of a task, it has a higher chance of losing the output of other tasks.

Rarely does an entire rack of servers experience the same anomaly. When an anomaly happens, the fraction of other machines within the rack that see the same anomaly is less than $\frac{1}{20}$ for recomputes, and $\frac{4}{20}$ for runtime with high probability. So, it is possible to restart a task, or replicate output to protect against loss on another machine within the same rack as the original machine.

4.3 Impact of Outliers

We now examine the impact of outliers on job completion times and cluster usage. Figure 7 plots the CDF for the ratio of job completion times, with different types of outliers included, to an ideal execution that neither has skewed run times nor loses intermediate data. The y-axes weighs each job by the total cluster time its tasks take to run. The hypothetical scenarios, with some combination of outliers present but not the others, do not exist in practice. So we replayed the logs in a trace driven simulator that retains the structure of the job, the observed task durations and the probabilities of the various anomalies (details in §6). The figure shows that at median, the job completion time would be lower by 15% if runtime outliers did not happen, and by more than 34% when none of the outliers happen. Recomputations impact fewer jobs than runtime outliers, but when they do, they delay completion time by a larger amount.

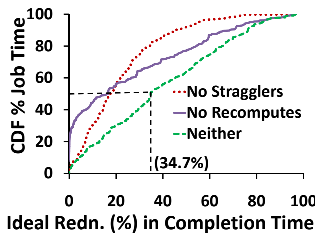


Figure 7: Percentage speed-up of job completion time in the ideal case when (some combination of) outliers do not occur.

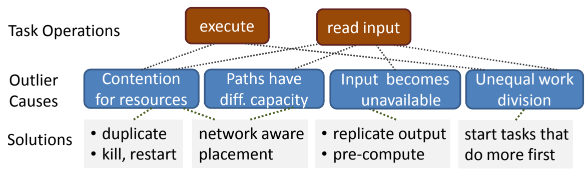


Figure 8: The Outlier Problem: Causes and Solutions

By inducing high variability in repeat runs of the same job, outliers make it hard to meet SLAs. At median, the ratio of $\frac{stdev}{mean}$ in job completion time is 0.8, i.e., jobs have a non-trivial probability of taking twice as long or finishing half as quickly.

To summarize, we take the following lessons from our experience.

- High running times of tasks do not necessarily indicate slow execution - there are multiple reasons for legitimate variation in durations of tasks.
- Every job is guaranteed some slots, as determined by cluster policy, but can use idle slots of other jobs. Hence, judicious usage of resources while mitigating outliers has collateral benefit.
- Recomputations affect jobs disproportionately. They manifest in select faulty machines and during times of heavy resource usage. Nonetheless, there are no indications of faulty racks.

5 Mantri Design

Mantri identifies points at which tasks are unable to make progress at the normal rate and implements targeted solutions. The guiding principles that distinguish Mantri from prior outlier mitigation schemes are *cause awareness* and *resource cognizance*.

Distinct actions are required for different causes. Figure 8 specifies the actions Mantri takes for each cause. If a task straggles due to contention for resources on the machine, restarting or duplicating it elsewhere can speed it up (§5.1). However, not moving data over the low bandwidth cross rack links, and if unavoidable, doing so while avoiding hotspots requires systematic placement (§5.2). To speed up tasks that wait for lost input to be recomputed, we find ways to protect task output (§5.3). Finally, for tasks with a work imbalance, we schedule the large tasks before the others to avoid being stuck with the large ones near completion (§5.4).

There is a subtle point with outlier mitigation: reducing the completion time of a task may in fact increase the job completion time. For example, replicating the output of every task will drastically reduce recomputations–both copies are unlikely to be lost at the same time, but can slow down the job because more time and bandwidth are used up for this task denying resources to other tasks that are waiting to run. Similarly, addressing outliers early in a phase vacates slots for outstanding tasks and can speed up completion. But, potentially uses more resources per task. Unlike Mantri, none of the existing approaches act early or replicate output. Further, naively extending current schemes to act early without being cognizant of the cost of resources, as we show in §6, leads to worse performance.

Closed-loop action allows Mantri to act optimistically by bounding the cost when probabilistic predictions go awry. For example, even when Mantri cannot ascertain the cause of an outlier, it experimentally starts copies. If the cause does not repeatedly impact the task, the copy can finish faster. To handle the contrary case, Mantri continuously monitors running copies and kills those whose cost exceeds the benefit.

Based on task progress reports, Mantri estimates for each task the remaining time to finish, t_{rem}, and the predicted completion time of a new copy of the task, t_{new}. Tasks report progress once every 10s or ten times in their lifetime, whichever is smaller. We use Δ to refer to this period. We defer details of the estimation to §5.5 and proceed to describe the algorithms for mitigating each of the main causes of outliers. All that matters is that t_{rem} be an accurate estimate and that the predicted distribution t_{new} account for the underlying work that the task has to do, the appropriateness of the network location and any persistent slowness of the new machine.

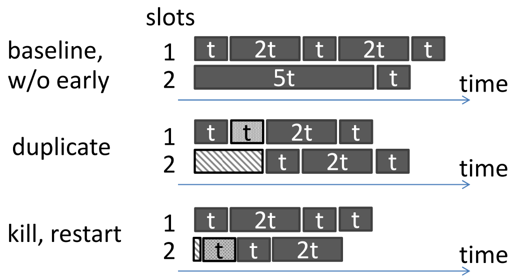


Figure 9: A stylized example to illustrate our main ideas. Tasks that are eventually killed are filled with stripes, repeat instances of a task are filled with a lighter mesh.

5.1 Resource-aware Restart

We begin with a simple example to help exposition. Figure 9 shows a phase that has seven tasks and two slots.

```
1: let $\Delta$ = period of progress reports
2: let $c$ = number of copies of a task
3: periodically, for each running task, kill all but the fastest $\alpha$ copies
       after $\Delta$ time has passed since begin
4: **while** slots are available **do**
5:     **if** tasks are waiting for slots **then**
6:         kill, restart task if $t_{rem} > \mathbb{E}(t_{new}) + \Delta$, stop at $\gamma$ restarts
7:         duplicate if $\mathbb{P}(t_{rem} > t_{new}\frac{c+1}{c}) > \delta$
8:         start the waiting task that has the largest data to read
9:     **else**                                    ▷ all tasks have begun
10:        duplicate iff $\mathbb{E}(t_{new} - t_{rem}) > \rho\Delta$
11:    **end if**
12: **end while**
```

Pseudocode 1: Algorithm for Resource-aware restarts (simplified).

Normal tasks run for times t and $2t$. One outlier has a runtime of $5t$. Time increases along the x axes.

The timeline at the top shows a baseline which ignores outliers and finishes at $7t$. Prior approaches that only address outliers at the end of the phase also finish at $7t$.

Note that if this outlier has a large amount of data to process letting the straggling task be is better than killing or duplicating it, both of which waste resources.

If however, the outlier was slowed down by its location, the second and third timelines compare duplication to a restart that kills the original copy. After a short time to identify the outlier, the scheduler can duplicate it at the next available slot (the middle time-line) or restart it in-place (the bottom timeline). If prediction is accurate, restarting is strictly better. However, if slots are going idle, it may be worthwhile to duplicate rather than incur the risk of losing work by killing.

Duplicating the outlier costs a total of $3t$ in resources ($2t$ before the original task is killed and t for the duplicate) which may be wasteful if the outlier were to finish in sooner than $3t$ by itself.

Restart Algorithm: Mantri uses two variants of restart, the first kills a running task and restarts it elsewhere, the second schedules a duplicate copy. In either method, Mantri restarts only when the probability of success, i.e., $\mathbb{P}(t_{new} < t_{rem})$ is high. Since t_{new} accounts for the systematic differences and the expected dynamic variation, Mantri does not restart tasks that are normal (e.g., runtime proportional to work). Pseudocode 1 summarizes the algorithm. Mantri kills and restarts a task if its remaining time is so large that there is a more than even chance that a restart would finish sooner. In particular, Mantri does so when $t_{rem} > \mathbb{E}(t_{new}) + \Delta$[4]. To not thrash on inaccurate estimates, Mantri kills a task no more than $\gamma = 3$ times.

The "kill and restart" scheme drastically improves the job completion time without requiring extra slots as we show analytically in [5]. However, the current job scheduler incurs a queueing delay before restarting a task, that

[4]Since the median of the heavy tailed task completion time distribution is smaller than the mean, this check implies that $\mathbb{P}(t_{new} < t_{rem}) > \mathbb{P}(t_{new} < \mathbb{E}(t_{new})) \geq .5$

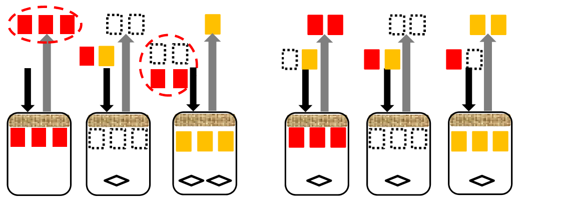


Figure 10: Three reduce tasks (rhombus boxes) are to be placed across three racks. The rectangles indicate their input. The type of the rectangle indicates the map that produced this data. Each reduce task has to process one shard of each type. The ad-hoc placement on the left creates network bottlenecks on the cross-rack links (highlighted). Tasks in such racks will straggle. If the network has no other traffic, the even placement on the right avoids hotspots.

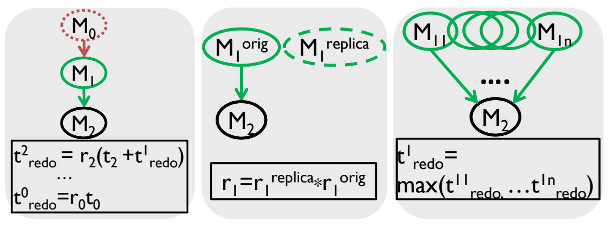
can be large and highly variant. Hence, we consider scheduling duplicates.

Scheduling a duplicate results in the minimum completion time of the two copies and provides a safety net when estimates are noisy or the queueing delay is large. However, it requires an extra slot and if allowed to run to finish, consumes extra computation resource that will increase the job completion time if outstanding tasks are prevented from starting. Hence, when there are outstanding tasks and no spare slots, we schedule a duplicate only if the total amount of computation resource consumed decreases. In particular, if c copies of the task are currently running, a duplicate is scheduled only if $\mathbb{P}(t_{rem} > t_{new}\frac{c+1}{c}) > \delta$. By default, $\delta = .25$. For example, a task with one running copy is duplicated only if t_{new} is less than half of t_{rem}. For stability, Mantri does not re-duplicate a task for which it launched a copy recently. Any copy that has run for some time and is slower than the second fastest copy of the task will be killed to conserve resources. Hence, there are never more than three running copies of a task[5]. When spare slots are available, as happens towards the end of the job, Mantri schedules duplicates more aggressively, i.e., whenever the reduction in the job completion time is larger than the start up time, $\mathbb{E}(t_{new} - t_{rem}) > \rho\Delta$. By default, $\rho = 3$. Note that in all the above cases, if more than one task satisfies the necessary conditions, Mantri breaks ties in favor of the task that will benefit the most.

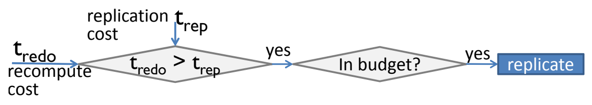
Mantri's restart algorithm is independent of the values for its parameters. Setting γ to a larger and ρ, δ to a smaller value trades off the risk of wasteful restarts for the reward of a larger speed-up. The default values that are specified here err on the side of caution.

By scheduling duplicates conservatively and pruning aggressively, Mantri has a high success rate of its restarts. As a result, it reduces completion time and conserves resources (§6.2).

[5]The two fastest copies and the copy that has recently started.


M0
M1
M2
$t^2_{redo} = r_2(t_2 + t^1_{redo})$
...
$t^0_{redo} = r_0 t_0$
M_1^{orig}
$M_1^{replica}$
M2
$r_1 = r_1^{replica} * r_1^{orig}$
M_{11}
M_{1n}
....
M2
$t^1_{redo} = \max(t^{11}_{redo}, \dots t^{1n}_{redo})$

(a) (b) (c)



(d) Decision Process

Figure 11: Avoiding costly recomputations: The cost to redo a task includes the recursive probability of predecessor tasks having to be re-done (a). Replicating output reduces the effective probability of loss (b). Tasks with many-to-one input patterns have high recomputation cost and are more valuable (c).

5.2 Network-Aware Placement

Reduce tasks, as noted before (§4.2), have to read data across racks. A rack with too many reduce tasks is congested on its downlink and such tasks will straggle. Figure 10 illustrates such a scenario.

Given the utilization of all the network links and the locations of inputs for all the tasks (and jobs) that are waiting to run, optimally placing the tasks to minimize job completion time is a form of the centralized traffic engineering problem [14, 18]. However achieving up-to-date information of network state and centralized co-ordination across all jobs in the cluster are challenging. Instead, Mantri approximates the optimal placement by a local algorithm that does not track bandwidth changes nor require co-ordination across jobs.

With Mantri, each job manager places tasks so as to minimize the load on the network and avoid self-interference among its tasks. If every job manager takes this independent action, network hotspots will not cause outliers. Note that the sizes of the map outputs in each rack are known to the job manager prior to placing the tasks of the subsequent reduce phase. For a reduce phase with n tasks running on a cluster with r racks, let its input matrix $I_{n,r}$ specify the size of input in each rack for each of the tasks[6]. For any placement of reduce tasks to racks, let the data to be moved out (on the uplink) and read in (on the downlink) on the i^{th} rack be d^i_u, d^i_v, and the corresponding available bandwidths be b^i_u and b^i_d respectively. For each rack, we compute two terms $c_{2i-1} = \frac{d^i_u}{b^i_u}$ and $c_{2i} = \frac{d^i_v}{b^i_d}$. The first term is the ratio of outgoing traffic and available uplink bandwidth, and the second term is the ratio of incoming traffic and available downlink bandwidth. The algorithm computes the optimal value over all placement permutations, i.e., the rack location for each task that minimizes the maximum data transfer time, as $\arg\min \max_j c_j,\ j = 1, \cdots, 2n,$.

Rather than track the available bandwidths b^i_u and b^i_d as they change with time and as a function of other jobs in the cluster, Mantri uses these estimates. Reduce phases with a small amount of data finish quickly, and the bandwidths can be assumed to be constant throughout the execution of the phase. For phases with a large amount of data, the bandwidth averaged over their long lifetime can be assumed to be equal for all links. We see that with these estimates Mantri's placement comes close to the ideal in our experiments (see §6.4).

For phases other than reduce, Mantri complements the Cosmos policy of placing a task close to its data [23]. By accounting for the cost of moving data over low bandwidth links in t_{new}, Mantri ensures that no copy is started at a location where it has little chance of finishing earlier thereby not wasting resources.

5.3 Avoiding Recomputation

To mitigate costly recomputations that stall a job, Mantri protects against interim data loss by replicating task output. It acts early by replicating those outputs whose cost to recompute exceeds the cost to replicate. Mantri estimates the cost to recompute as the product of the probability that the output will be lost and the time to repeat the task. The probability of loss is estimated for a machine over a long period of time. The time to repeat the task is t_{redo} with a recursive adjustment that accounts for the task's inputs also being lost. Figure 11 illustrates the calculation of t_{redo} based on the data loss probabilities (r_i's), the time taken by the tasks (t_i's) and recursively looks at prior phases. Replicating the output reduces the likelihood of recomputation to the case when all replicas are unavailable. If a task reads input from many tasks (e.g., a reduce), t_{redo} is higher since any of the inputs needing to be recomputed will stall the task's recomputation [7]. The cost to replicate, t_{rep}, is the time to move the data to another machine in the rack.

In effect, the algorithm replicates tasks at key places in a job's workflow – when the cumulative cost of not replicating many successive tasks builds up or when tasks ran on very flaky machines (high r_i) or when the output is so small that replicating it would cost little (low t_{rep}).

Further, to avoid excessive replication, Mantri limits the amount of data replicated to 10% of the data processed by the job. This limit is implemented by granting tokens proportional to the amount of data processed by each task. Task output that satisfies the above cost-benefit check is

[6] In I, the row sum indicates the data to be read by the task, whereas the column sum indicates the total input present in that rack.

[7] In Fig. 11(c), we assume that if multiple inputs are lost, they are recomputed in parallel and the task is stalled by the longest input. Since recomputes are rare (Fig. 2(a)), this is a fair approximation of practice.

replicated only if an equal number of tokens are available. Tokens are deducted on replication.

Mantri proactively recomputes tasks whose output and replicas, if any, have been lost. From §4, we see that recomputations on a machine cluster by time, hence Mantri considers a recompute to be the onset of a temporal problem which will cause future requests for data on this machine to fail and pre-computes such output. Doing so decreases the time that a dependent task will have to wait for lost input to be regenerated. As before, Mantri imposes a budget on the extra cluster cycles used for pre-computation. Together, probabilistic replication and pre-computation approximate the ideal scheme in our evaluation (§6.5).

5.4 Data-aware Task Ordering

Workload imbalance causes tasks to straggle. Mantri does not restart outliers that take a long time to run because they have more work to do. Instead, Mantri improves job completion time by scheduling tasks in a phase in descending order of their input size. Given n tasks, s slots and input sizes $d[1 \cdots n]$, if the optimal completion time is T_O, scheduling tasks in inverse order of their input sizes will take T, where $\frac{T}{T_O} \leq \frac{4}{3} - \frac{1}{3s}$ [12]. This means that scheduling tasks with the longest processing time first is at most 33% worse than the optimal schedule; computing the optimal is NP-hard [12].

5.5 Estimation of t_{rem} and t_{new}

Periodically, every running task informs the job scheduler of its status, including how many bytes it has read, d_{read}, thus far. Mantri combines the progress reports with the size of the input data that each task has to process, d, and predicts how much longer the task would take to finish using this model:

$$t_{rem} = t_{elapsed}\frac{d}{d_{read}} + t_{wrapup}. \tag{2}$$

The first term captures the remaining time to process data. The second term is the time to compute after all the input has been read and is estimated from the behavior of earlier tasks in the phase. Tasks may speed up or slow down and hence, rather than extrapolating from each progress report, Mantri uses a moving average. To be robust against lost progress reports, when a task hasn't reported for a while, Mantri increases t_{rem} by assuming that the task has not progressed since its last report. This linear model for estimating the remaining time for a task is well suited for data-intensive computations like Map-Reduce where a task spends most of its time reading the input data. We seldom see variance in computation time among tasks that read equal amounts of data [26].

Mantri estimates t_{new}, the distribution over time that a new copy of the task will take to run, as follows:

$$t_{new} = \textit{processRate} * \textit{locationFactor} * d + \textit{schedLag}. \tag{3}$$

The first term is a distribution of the process rate, i.e., $\frac{\Delta time}{\Delta data}$, of all the tasks in this phase. The second term is a relative factor that accounts for whether the candidate machine for running this task is persistently slower (or faster) than other machines or has smaller (or larger) capacity on the network path to where the task's inputs are located. The third term, as before, is the amount of data the task has to process. The last term is the average delay between a task being scheduled and when it gets to run. We show in §6.2 that these estimates of t_{rem} and t_{new} are sufficiently accurate for Mantri's functioning.

6 Evaluation

We deployed and evaluated Mantri on Bing's production cluster consisting of thousands of servers. Mantri has been running as the outlier mitigation module for all the jobs in Bing's clusters since May 2010. To compare against a wider set of alternate techniques, we built a trace driven simulator that replays logs from production.

6.1 Setup

Clusters: The production cluster consists of thousands of server-class multi-core machines with tens of GBs of RAM that are spread roughly 40 servers to a rack. This cluster is used by Bing product groups. The data we analyzed earlier is from this cluster, so the observations from §4 hold here.

Workload: Mantri is the default outlier mitigation solution for the production cluster. The jobs submitted to this cluster are independent of us, enabling us to evaluate Mantri's performance in a live cluster across a variety of production jobs. We compare Mantri's performance on all jobs in the month of June 2010 with prior runs of the same jobs in April-May 2010 that ran with the earlier build of Cosmos.

In addition, we also evaluate Mantri on four hand-picked applications that represent common building blocks. *Word Count* calculates the number of unique words in the input. *Table Join* inner joins two tables each with three columns of data on one of the columns. *Group By* counts the number of occurrences of each word in the file. Finally, *grep* searches for string patterns in the input. We vary input sizes from 53 GB to 500 GB.

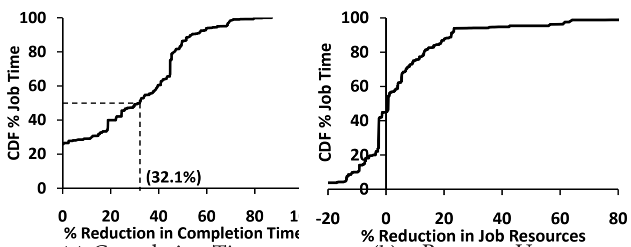
Prototype: Mantri builds on the Cosmos job scheduler and consists of about 1000 lines of C++ code. To compute t_{rem}, Mantri maintains an execution record for each of the running tasks that is updated when the task reports progress. A phase-wide data structure stores the necessary statistics to compute t_{new}. When slots become available, Mantri runs Pseudocode 1 and restarts or duplicates the task that would benefit the most or starts new tasks in descending order of data size. To place tasks appropriately, name builds on the per-task *affinity list*, a preferred set of machines and racks that the task can run on. At run-time the job manager attempts to place the task at its preferred locations in random order, and when none of them are available runs the task at the first available slot. The affinity list for map tasks has machines that have replicas of the input blocks. For reduce tasks, to obtain the desired proportional spread across racks (see §5.2), we populate the affinity list with a proportional number of machines in those racks.

Trace-driven Simulator: The simulator replays the logs shown in Table 1. For each phase, it faithfully repeats the observed distributions of task completion times, data read by each task, size and location of inputs, probability of failures and recomputations, and fairness based evictions. Restarted tasks have their execution times and failure probabilities sampled from the same distribution of tasks in their phase. The simulator also mimics the job workflow including semantics like barriers before phases, the permissible concurrent slots per phase and the input/output relationships between phases. It mimics cluster characteristics like machine failures, network congestion and availability of computation slots. For the network, it uses a fluid model rather than simulating individual packets. Doing the latter, at petabyte scale, is out of scope for this work.

Compared Schemes: Our results on the production cluster uses the current Dryad implementation as the baseline (§6.2). It contains state-of-the-art outlier mitigation strategies and runs thousands of jobs daily.

Our simulator performs a wider and detailed comparison. It compares Mantri with the outlier mitigation strategies in Hadoop [1], Dryad [13], Map-Reduce [11], LATE [20], and a modified form of LATE that acts on stragglers early in the phase. As the current Dryad build already has modules for straggler mitigation, we compare all of these schemes to a baseline that does not mitigate any stragglers (§6.3). On the other hand, since these schemes do not do network-aware placement or recompute mitigation, we use the current Dryad implementation itself as their baseline (§6.4 and §6.5).

We also compare Mantri against some ideal benchmarks. *NoSkew* mimics the case when all tasks in a phase take the same amount of time, set to the average over the observed task durations. *NoSkew + ChopTail* goes even further, it removes the worst quartile of the observed durations, and sets every task to the average of remaining durations. *IdealReduce* assumes perfect up-to-date knowledge of available bandwidths and places reduce tasks accordingly. *IdealRecompute* uses future knowledge of which tasks will have their inputs recomputed and protects those inputs.



(a) Completion Time (b) # Resource Usage

Figure 12: Evaluation of Mantri as the default build for all jobs on the production cluster for twenty-five days.

Metrics: As our primary metrics, we use the reduction in completion time and resource usage[8], where

$$\text{Reduction} = \frac{\text{Current} - \text{Modified}}{\text{Current}}. \quad (4)$$

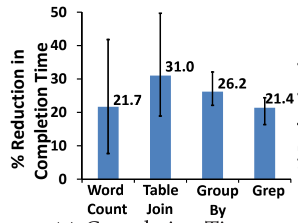
Summary: Our results are summarized as follows:

- In live deployment in the production cluster Mantri sped up the median job by 32%. 55% of the jobs experienced a net reduction in resources used. Further Mantri's network-aware placement reduced the completion times of typical reduce phases by 31%.
- Simulations driven from production logs show that Mantri's restart strategy reduces the completion time of phases by 21% (and 42%) at the 50th (and 75th) percentile. Here, Mantri's reduction in completion time improves on Hadoop by 3.1x while using fewer resources than Map-Reduce, each of which are the current best on those respective metrics.
- Mantri's network-aware placement of tasks speeds up half of the reduce phases by at least 60% each.
- Mantri reduces the completion times due to recomputations of jobs that constitute 25% (or 50%) of the workload by at least 40% (or 20%) each while consuming negligible extra resources.

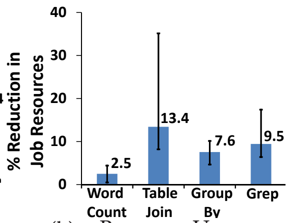
6.2 Deployment Results

Jobs in the Wild: We compare one month of jobs in the Bing production cluster that ran after Mantri was turned live with runs of the same job(s) on earlier builds. We use only those recurring jobs that have roughly similar amounts of input and output across runs. Figure 12(a) plots the CDF of the improvement in completion time. The y axes weighs each job by the total time its tasks take to run since improvement on larger jobs adds more value

[8]A reduction of 50% implies that the property in question, completion time or resources used, decreases by half. Negative values of reduction imply that the modification uses more resources or takes longer.



(a) Completion Time



(b) # Resource Usage

Figure 13: Comparing Mantri's straggler mitigation with the baseline implementation on a production cluster of thousands of servers for the four representative jobs.

	% reduction in completion time		
	avg	min	max
Phase	31.5	28.4	34.2
Job	12.6	7.0	19.2

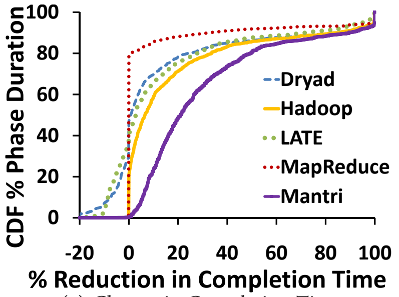
Table 2: Comparing Mantri's network-aware spread of tasks with the baseline implementation on a production cluster of thousands of servers.

to the cluster. Jobs that occupy the cluster for half the time sped up by at least 32.1%. Figure 12(b) shows that 55% of jobs see a *reduction* in resource consumption while the others use up a few extra resources. These gains are due to Mantri's ability to detect outliers early and accurately. The success rate of Mantri's copies, i.e., the fraction of time they finish before the original copy, improves by 2.8x over the earlier build. At the same time, Mantri expends fewer resources, it starts .47x fewer copies. Further, Mantri acts early, over 50% of its copies are started before the original task has completed 42% of its work as opposed to 77% with the earlier build.

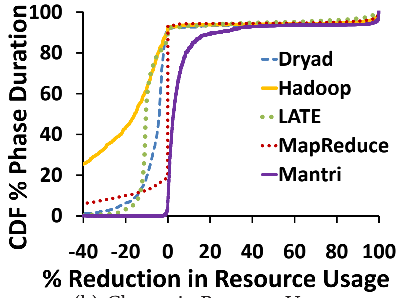
Straggler Mitigation: To cross-check the above results on standard jobs, we ran four prototypical jobs with and without Mantri twenty times each. Figure 13 shows that job completion times improve by roughly 25% and resource usage falls by roughly 10%. The histograms plot the average reduction, error bars are the 10^{th} and 90^{th} percentiles of samples. Further, we logged all the progress reports for these jobs. We find that Mantri's predictor, based on reports from the recent past, estimates t_{rem} to within a 2.9% error of the actual completion time.

Placement of Tasks: To evaluate Mantri's network-aware spreading of reduce tasks, we ran *Group By*, a job with a long-running reduce phase, ten times on the production cluster. Table 2 shows that the reduce phase's completion time reduces by 28.4% on average causing the job to speed up by an average of 12.6%. To understand why, we measure the *spread* of tasks, i.e., the ratio of the number of concurrent reduce tasks to the number of racks they ran in. High spread implies that some racks have more tasks which interfere with each other while other racks are idle. Mantri's spread is 1.5 compared to 5.5 for the earlier build.

To compare against alternative schemes and to piece apart gains from the various algorithms in Mantri, we present results from the trace-driven simulator.



(a) Change in Completion Time



(b) Change in Resource Usage

Figure 14: Comparing straggler mitigation strategies. Mantri provides a greater speed-up in completion time while using fewer resources than existing schemes.

6.3 Can Mantri mitigate stragglers?

Figure 14 compares straggler mitigation strategies in their impact on completion time and resource usage. The y-axes weighs phases by their lifetime since improving the longer phases improves cluster efficiency. The figures plot the cumulative reduction in these metrics for the 210K phases in Table 1 with each repeated thrice. For this section, our common baseline is the scheduler that takes no action on outliers. Recall from §6.1 that the simulator replays the task durations and the anomalies observed in production.

Figures 14(a) and 14(b) show that Mantri improves completion time by 21% and 42% at the 50^{th} and 75^{th} percentiles and reduces resource usage by 3% and 7% at these percentiles. From Figure 14(a), at the 50^{th} percentile, Mantri sped up phases by an additional 3.1x over the 6.9% improvement of Hadoop, the next best scheme. To achieve the smaller improvement Hadoop uses 15.9% more resources (Fig. 14(b)). Map-Reduce and Dryad have no positive impact for 80% and 50% of the phases respectively. Up to the 30^{th} percentile Dryad increases the completion time of phases. LATE is similar in its time improvement to Hadoop but uses fewer resources.

The reason for poor performance is that they miss outliers that happen early in the phase and by not knowing the true causes of outliers, the duplicates they schedule are mostly not useful. Mantri and Dryad schedule .2 restarts per task for the average phase (.06 and .56 for LATE and

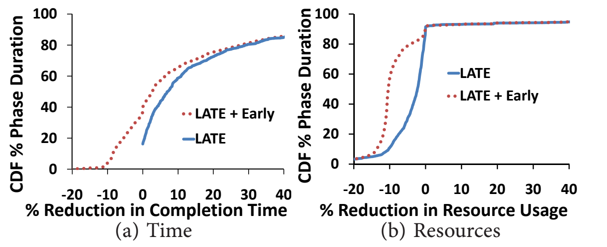


Figure 15: Extending LATE to speculate early results in worse performance

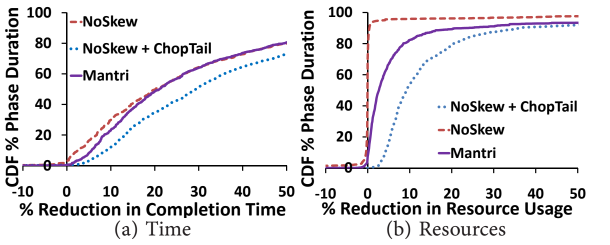


Figure 16: Mantri is on par with an ideal *NoSkew* benchmark and slightly worse than *NoSkew+ChopTail* (see end of §6.3)

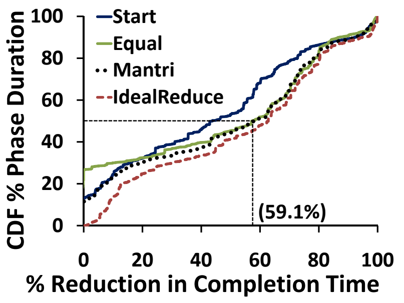


Figure 17: By being network aware, Mantri speeds up the median reduce phase by 60% over the current placement.

Hadoop). But, Mantri's restarts have a success rate of 70% compared to the 15% for LATE. The other schemes have lower success rates.

While the insight of *early action* on stragglers is valuable, it is nonetheless non trivial. We evaluate this in Figures 15(a) and 15(b) that present a form of LATE that is identical in all ways except that it addresses stragglers early. We see that addressing stragglers early increases completion time up to the 40th percentile, uses more resources and is worse than vanilla LATE. Being resource aware is crucial to get the best out of early action (§5.1).

Finally, Fig. 16 shows that Mantri is on par with the ideal benchmark that has no variation in tasks, *NoSkew*, and is slightly worse than the variant that removes all durations in the top quartile, *NoSkew+ChopTail*. The reason is that Mantri's ability to substitute long running tasks with their faster copies makes up for its inability to act with perfect future knowledge of which tasks straggle.

6.4 Does Mantri improve placement?

Figure 17 plots the reduction in completion time due to Mantri's placement of reduce tasks as a CDF over all reduce phases in the dataset in Table 1. As before, the y-axes weighs phases by their lifetime. The figure shows that Mantri provides a median speed up of 59% or a 2.5x improvement over the current implementation.

The figure also compares Mantri against strategies that estimate available bandwidths differently. The *IdealReduce* strategy tracks perfectly the changes in available bandwidth of links due to the other jobs in the cluster. The *Equal* strategy assumes that the available bandwidths are equal across all links whereas *Start* assumes that the available bandwidths are the same as at the start of the phase. We see a partial order between *Start* and *Equal* (the two solid lines). Short phases are impacted by transient differences in the available bandwidths and *Start* is a good choice for these phases. However, these differences even out over the lifetime of long phases for whom *Equal* works better. Mantri is a hybrid of *Start* and *Equal*. It achieves a good approximation of *IdealReduce* without re-sampling available bandwidths.

To capture how Mantri's placement differs from Dryad, Figure 18 plots the ratio of the throughput obtained by the median task in each reduce phase to that obtained by the slowest task. With Mantri, this ratio is 1.05 at median and never larger than 2. In contrast, with Dryad's policy of placing tasks at the first available slot, this ratio is 5.25 (or 14.33) at the 50th (or 75th) percentile. Note that duplicating tasks that are delayed due to network congestion without considering the available bandwidths or where other tasks are located would be wasteful.

6.5 Does Mantri help with recomputations?

The best possible protection against loss of output would (a) eliminate all the increase in job completion time due to tasks waiting for their inputs to be recomputed and (b) do so with little additional cost. Mantri approximates both goals. Fig. 19 shows that Mantri achieves parity with *IdealRecompute*. Recall that IdealRecompute has perfect future knowledge of loss. The improvement in job completion time is 20% (40%) at the 50th (75th) percentile.

The reason is that Mantri's policy of selective replication is both accurate and biased towards the more expensive recomputations. The probability that task output that was replicated will be used because the original data becomes unavailable is 84%. Similarly, the probability that a pre-computation becomes useful is 76%, which increases to 93% if pre-computations are triggered only when two recomputations happen at a machine in quick succession. Figure 20 shows the complementary contributions from replication and pre-computation– each contribute

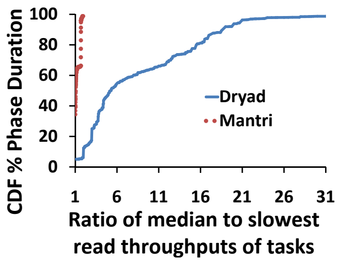


Figure 18: Unlike Dryad, Mantri's placement provides more consistent throughput to tasks in reduce phases.

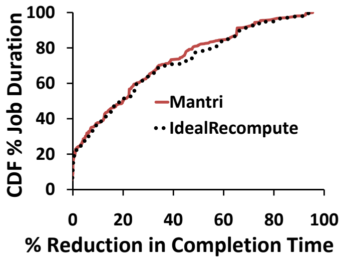


Figure 19: By probabilistically replicating task output and recomputing lost data before it is needed Mantri speeds up jobs by an amount equal to the ideal case of no data loss.

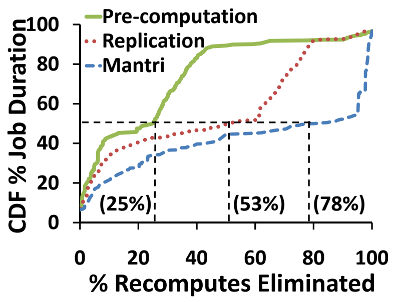


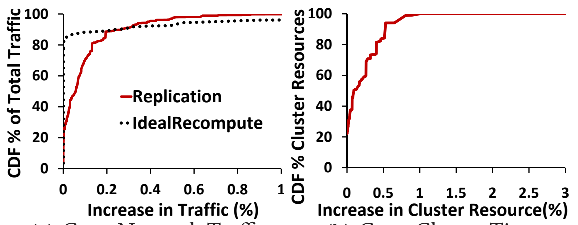
Figure 20: Fraction of recomputations that are eliminated due to Mantri's recomputation mitigation strategy, along with individual contributions from replication and pre-computation.

roughly 66% and 33% to the total. Cumulatively, the figure shows that Mantri eliminates 78% of recomputations for the median job. We note that Mantri ignores 75% of the recomputations in the bottom quartile of jobs since their impact on job completion time is small.

Fig. 21(a) shows that the extra network traffic due to replication is (overall negligible and) comparable to *IdealReduce*. Mantri sometimes replicates more data than the ideal, and at other times misses some tasks that should be replicated. Fig. 21(b) shows that pre-computations take only a few percentage extra resources.

7 Related Work

Much recent work focuses on large scale data parallel computing. Following on the Map-Reduce [11] paper, there has been work in improving workflows [1, 13], language design [8, 27], and fair schedulers [18]. Our work



(a) Cost: Network Traffic (b) Cost: Cluster Time

Figure 21: The cost to protect against recomputes is fewer than a few percentage points in both the extra traffic on the network and cluster time for pre-computation.

here takes the next step of understanding how such production clusters behave and can be improved.

Run-time stragglers have been identified by past work [11, 20]. However, we are the first to characterize the prevalence of stragglers in production and their causes. By understanding the causes, addressing stragglers early and scheduling duplicates only when there is a fair chance that the speculation saves both time and resources, our approach provides a greater reduction in job completion time while using fewer resources than prior strategies that duplicate tasks towards the end of a phase. Also, we uniquely avoid network hotspots and protect against loss of task output, two further causes of outliers.

By only acting at the end of a phase, current schemes [1, 11, 13] miss early outliers. They vary in the choice of which tasks to duplicate. After a threshold number of tasks have finished, Map-Reduce [11] duplicates all the tasks that remain. Dryad [13] duplicates those that have been running for longer than the 75th percentile of task durations. After all tasks have started, Hadoop [1] uses slots that free up to duplicate any task that has read less data than the others, while LATE [20] duplicates only those reading at a slow rate.

Though some recent proposals do away with capacity over-subscription in data centers [3, 17], today's networks remain over-subscribed albeit at smaller levels. It is common to place tasks near their input (same machine, rack etc.) for map and at the first free slot for reduce [1, 11, 13]. Our approach to eliminate outliers by a network-aware placement is orthogonal to recent work that packs tasks requiring different resources on to a machine [25], or trades-off fairness with efficiency [18]. Quincy accounts for capacity but not for runtime variations in bandwidth due to competition from other tasks.

ISS [15] protects intermediate data by replicating locally-consumed data. In particular, this does not include map output, since Hadoop transfers map output to reduce tasks as it is produced. ISS's replication strategy runs the risk of being both wasteful (when very few machines are error-prone) and insufficient (when the transfer of map output fails). In contrast, Mantri presents a broader solution that (a) replicates task output based on the probability of data loss and the recursive cost of recomputing inputs and (b) pre-computes lost data.

The Map-Reduce paradigm is similar to parallel databases in its goal of analyzing large data [22] and to dedicated HPC clusters and parallel programs [16] by presenting similar optimization opportunities. In the context of multiple processors, studies have been done on the classic problem of dynamic task scheduling [4, 6] as well as task duplication [24]. Star-MPI [2] adapts parameters like network topology between a set of communicating processors by observing performance over time. Prior work has also focused on modeling and optimizing the communication in parallel programs [10, 19, 21] that have one-to-all or all-to-all traffic, i.e., where every receiver processes all of the output of tasks in earlier stages. In the context of the many-to-many traffic, typical of Map-Reduce, we present practical techniques for bandwidth estimation and task placement that realizes near-optimal performance.

8 Conclusion

Mantri delivers effective mitigation of outliers in Map-Reduce networks. It is motivated by, what we believe is, the first study of a large production Map-Reduce cluster. The root of Mantri's advantage lies in integrating static knowledge of job structure and dynamically available progress reports into a unified framework that identifies outliers early, applies cause-specific mitigation and does so only if the benefit is higher than the cost. In our implementation on a cluster of thousands of servers, we find Mantri to be highly effective.

Outliers are an inevitable side-effect of parallelizing work. They hurt Map-Reduce networks more due to the structure of jobs as graphs of dependent phases that pass data from one to the other. Their many causes reflect the interplay between the network, storage and, computation in Map-Reduce. Current systems shirk this complexity and assume that a duplicate would speed things up. Mantri embraces it to mitigate a broad set of outliers.

Acknowledgments– For feedback on drafts, we thank members of the RAD lab, the Cosmos product group and the OSDI reviewers. Alexei Polkhanov and Juhan Lee were invaluable in taking Mantri to production clusters.

References

[1] Hadoop distributed filesystem. http://hadoop.apache.org.

[2] A. Faraj, X. Yuan, D. Lowenthal. STAR-MPI: Self Tuned Adaptive Routines for MPI Collective Operations. In *SC*, 2006.

[3] A. Greenberg, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM*, 2009.

[4] I. Ahmad and M. K. Dhodhi. Semi-distributed load balancing for massively parallel multicomputer systems. In *IEEE TSE.*, 1991.

[5] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, and Y. Lu. Reigning in the outliers in map-reduce clusters. Technical Report MSR-TR-2010-69, Microsoft Research, 2010.

[6] B. Ucar, C. Aykanat, K. Kaya, M. Ikinci. Task assignment in Heterogeneous Computing Systems. In *JPDC*, 2006.

[7] L. N. Bairavasundaram, G. R. Goodson, B. Schroeder, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. An analysis of data corruption in the storage stack. In *FAST*, 2008.

[8] 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.

[9] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmleegy, and R. Sears. Mapreduce online. In *NSDI*, 2010.

[10] D. Culler et al. LogP: Towards a Realistic Model of Parallel Computation. In *SIGPLAN PPoPP*, 1993.

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

[12] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2), 1969.

[13] M. Isard et al. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Eurosys*, 2007.

[14] S. Kandula, D. Katabi, B. Davie, and A. Charny. Walking the Tightrope: Responsive Yet Stable Traffic Engineering. In *SIGCOMM*, 2005.

[15] S. Ko, I. Hoque, B. Cho, and I. Gupta. Making cloud intermediate data fault-tolerant. In *SOCC*, 2010.

[16] A. Krishnamurthy and K. Yelick. Analysis and optimizations for shared address space programs. *JPDC*, 1996.

[17] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*, 2008.

[18] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *SOSP*, 2009.

[19] M. Lauria and A. Chien. MPI-FM: High Performance MPI on Workstation Clusters. In *JPDC*, 1997.

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

[21] P. Patarasuk, A. Faraj, X. Yuan. Pipelined Broadcast on Ethernet Switched Clusters. In *IEEE IPDPS*, 2006.

[22] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. R. Madden, and M. Stonebraker. A comparison of approaches to large scale data analysis. In *SIGMOD*, 2009.

[23] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, R. Chaiken. Nature of Datacenter Traffic: Measurements and Analysis. In *IMC*, 2009.

[24] S. Manoharan. Effect of task duplication on assignment of dependency graphs. In *Parallel Comput.*, 2001.

[25] T. Sandholm and K. Lai. Mapreduce optimization using regulated dynamic prioritization. In *SIGMETRICS*, 2009.

[26] Y. Kwon, M. Balazinska, B. Howe, J. Rolia. Skew-Resistant Parallel Processing of Feature-Extracting Scientific User-Defined Functions. In *SOCC*, 2010.

[27] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, J. Currey. DryadLINQ: A System for General-Purpose Data-Parallel Computing Using a High-Level Language. In *OSDI*, 2008.

[28] Y. Yu, P. K. Gunda, and M. Isard. Distributed Aggregation for Data-Parallel Computing: Interfaces, Impl. In *SOSP*, 2009.