Learning from failure across multiple clusters: A trace-driven approach to understanding, predicting, and mitigating job terminations

Nosayba El-Sayed[1] Hongyu Zhu[2] Bianca Schroeder[2]
[1]*Computer Science and Artificial Intelligence Lab (CSAIL), MIT, nosayba@csail.mit.edu*
[2]*Department of Computer Science, University of Toronto, {serailhydra, bianca}@cs.toronto.edu*

***Abstract*—In large-scale computing platforms, jobs are prone to interruptions and premature terminations, limiting their usability and leading to significant waste in cluster resources. In this paper, we tackle this problem in three steps. First, we provide a comprehensive study based on log data from multiple large-scale production systems to identify patterns in the behaviour of unsuccessful jobs across different clusters and investigate possible root causes behind job termination. Our results reveal several interesting properties that distinguish unsuccessful jobs from others, particularly w.r.t. resource consumption patterns and job configuration settings. Secondly, we design a machine learning-based framework for predicting job and task terminations. We show that job failures can be predicted relatively early with high precision and recall, and also identify attributes that have strong predictive power of job failure. Finally, we demonstrate in a concrete use case how our prediction framework can be used to mitigate the effect of unsuccessful execution using an effective task-cloning policy that we propose.**

I. Introduction

In the era of big data, scientists, businesses and governments alike depend critically on systems and frameworks for the effective processing of ever growing amounts of data. While the capability of large-scale systems in terms of the volume of data they can handle has increased at breakneck speeds, reliability and dependability have not quite kept up at the same pace. In fact, reliability concerns have been on the rise for different types of large-scale systems: whether in traditional high-performance computing (HPC) platforms where applications are expected to abort every few minutes in next-generation machines [16], or in cloud computing environments that run diverse workloads on top of complex software and heterogenous hardware, increasing a job's proneness to faults and errors [5], [8], [10], [13].

Several recent studies that analyzed a large workload trace made publicly available by Google [17] for one of their multi-purpose clusters, found that a large fraction of cluster time was spent executing jobs that terminate unsuccessfully (e.g. fail or get killed) [5], [8], [10], [13], highlighting the need for an in-depth understanding of how and why jobs terminate in large-scale systems.

While these studies provide some insights into the characteristics of unsuccessful executions in Google's cluster, there has been no follow-up work that investigates the generalizability of the reported findings to other computing clusters, or to other classes of large-scale systems. Moreover, there has been no work that demonstrates how the results of these studies can be used to mitigate unsuccessful job executions or improve clusters in other ways.

In this work, our first goal is to provide a comprehensive analysis of the characteristics and root causes of job termination, using workload traces collected at different systems. We utilize traces from three different sources: a multi-purpose cluster at Google [17], a Hadoop cluster at Carnegie Mellon University (CMU) [2], and a HPC cluster at Los Alamos National Labs (LANL) [3]. Using the traces, we investigate how and why large-scale jobs terminate unsuccessfully in parallel clusters, and based on our observations we propose and evaluate frameworks for job failure prediction and mitigation.

The main contributions of this paper are as follows:

- We provide a detailed characterization of factors that differentiate unsuccessful from successful job executions (Section III) and explore different hypotheses for root causes that might explain unsuccessful executions (Section IV). Our work extends previous work to include data sets for two new large-scale systems and covers factors that have not been explored in previous work. For some previously studied factors, we come to different conclusions from prior work.
- We use machine learning techniques to predict unsuccessful execution at the job and the task level. We show that unsuccessful execution can be predicted relatively early on in a job's lifetime and with high precision and recall (Section V).
- We propose and evaluate a practical use case for how our findings and predictor can be used to mitigate job failure using task cloning, and discuss a number of other potential use cases (Section 6).

II. Description of the job traces

Our study is based on the following job traces collected at three different organizations:

a) Google: This trace is publicly available [17] and captures all 349K jobs submitted to a 12,000 node multi-purpose compute cluster at Google over a one-month period. Jobs can consist of one or multiple tasks and for each task, as well as for the job as a whole, the *exit status* is recorded: jobs and tasks were marked as 'finished' if they completed successfully; 'killed' if they were aborted by the user or due to a dependency on another job that is no longer satisfied;

TABLE I
OVERVIEW OF THE WORKLOAD TRACES IN OUR DATASET.

Data Source	Data Timespan	#Nodes	#Users	#Scheduled Jobs	%Success Jobs	%Failed Jobs	%Killed or Aborted Jobs	%Other
Google	29 days	12K	227	349K	63%	1.3%	35.5%	0.003%
CMU	883 days	64	73	78K	88.8%	8.11%	3%	0.04%
LANL	1,022 days	256	446	290K	67.5%	0.62%	30.7%	1%

TABLE II
COMPARISON OF JOB-DURATION QUARTILES BETWEEN SUCCESSFUL AND UNSUCCESSFUL JOBS AT MULTIPLE CLUSTERS.

Data Source	Job Status	Job-duration distribution			
		25tile	Median	75tile	99tile
Google	`COMPLETED`	1.0225	2.8158	7.7739	357.6081
	`FAILED`	2.128	5.9519	38.908	31462
	`KILLED/ABORTED`	17.4708	69.5155	328.106	52144
CMU	`COMPLETED`	0.8532	8.7213	75.1883	14241
	`FAILED`	2.3509	22.9641	204.8583	54868
	`KILLED/ABORTED`	9.2011	61.7547	550.1506	80018
LANL	`COMPLETED`	76	300	5165	3156100
	`FAILED`	2737.5	15934	107578	7901700
	`KILLED/ABORTED`	1339	16034	236352	7396864

'evict' if the task was descheduled due to the arrival of a higher priority task on the same machine; and 'failed', typically as a result of a software crash. Tasks can have multiple *attempts* before the final exit, where each attempt has a status as well. Details about the trace are available in [9].

b) CMU OpenCloud: OpenCloud [2] is a 64-node Hadoop cluster at Carnegie Mellon University used by researchers in areas such as computational biology, image processing and machine learning. The traces were collected over a period of 30 months, cover 78K jobs and include information on job configuration and resource usage, as well as the exit status of each job in the cluster: 'finished' if it completes successfully; 'killed' if it is aborted manually by the user; and 'failed' if the job crashes. Details about the trace are available in [2].

c) LANL HPC Cluster: LANL made publicly available three years worth of job traces for a 256 node HPC system (system with ID 20 on their webpage [3]), which was used by scientists, typically to run CPU and memory-intensive scientific simulations. The traces contain data on job launch time and exit time, as well as the final status of each job: 'finished' when all application processes complete successfully; 'aborted' if the user canceled the job; 'killed' if an application process was terminated by a signal; 'syskill' if a job was killed by an administrative user, either due to maintenance or due to a problem; and 'failed' if one or more nodes running this job crashed. Details about the traces are available at [3], [15].

Table I summarizes the characteristics of the traces as well as the breakdown of the job exit status in each cluster. In terms of cluster time, we find that unsuccessful executions consume significant portions of compute cycles (more than 50% of cluster time in each trace).

III. WHAT CHARACTERIZES UNSUCCESSFUL JOBS IN LARGE CLUSTERS?

We begin by studying the characteristics of unsuccessful jobs across different clusters. Our goal is to utilize the traces described above to learn how failed or killed job executions exhibit patterns that distinguish them from successful executions. We examine various job attributes, including job duration, degree of parallelism, and utilization of cluster resources.

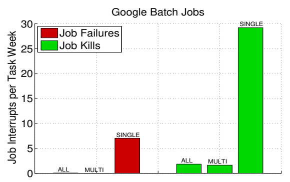
a) Job duration: We define the duration of a job as the total amount of task-minutes spent by the job in the cluster before exiting. For the LANL cluster where jobs did not consist of tasks, we use node-minutes to denote job duration. In Table II, we compare in each cluster the distribution of job durations between jobs that failed, jobs that got killed, and jobs that completed successfully. We observe that in all three clusters, successful jobs consistently run for the shortest durations among all jobs scheduled in the same cluster, while killed jobs consistently spend the longest durations before they are terminated. This observation is intuitive since hanging jobs, for example, are aborted by users when they spend longer than usual to complete or when they stop responding to health-monitoring signals. Finally, we find that for the LANL HPC cluster, failed jobs report comparable durations to killed jobs, but in Google and OpenCloud, failed jobs are shorter than killed jobs.

b) Degree of parallelism: In the Google cluster, 35% of batch jobs are multi-task jobs, 89.3% of which end up being killed, while more than 90% of single-task jobs complete. The portion of failed jobs is around 1.5% for both single- and multi-task jobs. In OpenCloud, tasks are either 'mappers' or 'reducers': more than 80% of the jobs had at least 2 mappers, and nearly 60% had 1 or 0 reducers. We use the number of mappers to indicate a job's parallelism. Unlike Google, we find that the breakdown of job status is almost identical for single- and multi-task jobs at OpenCloud, and reflects the overall breakdown (recall Table I). In the LANL cluster, 79% of the jobs ran on a single node. We observe 70% and 50% success rates in single-node and multi-node jobs, respectively. The fraction of jobs that are killed or syskilled within single-node jobs (22%) is almost doubled in parallel jobs (45%).

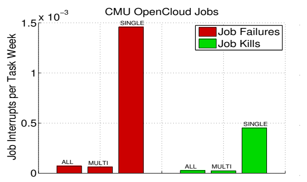
Since parallel jobs spend more time in the cluster, they are naturally more likely to encounter problems. To account for this effect, we normalize the number of job interruptions observed in the traces by the total amount of task-weeks (or node-weeks) consumed by the jobs in each category. Figure 1 shows the results for all jobs in our dataset.

Surprisingly, we find that when normalized by system time, parallel jobs exhibit *lower* rates of job interruptions per unit time compared to single jobs (with the only exception of killed jobs in the LANL cluster).

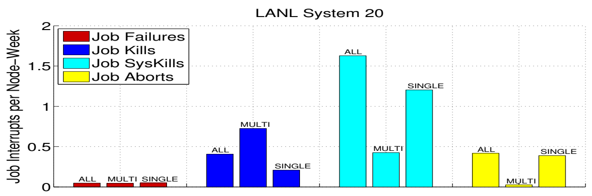
c) Usage of cluster resources: We now study how cluster resources are used by completed jobs versus unsuccessful jobs. We begin by examining the amount of resources initially *requested* by a job at submission time. In Google, data on the amount of CPU, memory, and disk space requested by a job's



(a) Google Batch Jobs.



(b) CMU OpenCloud Jobs.



(c) LANL Jobs.

Fig. 1. The rate of job interruptions normalized by system time for all jobs, multi-task jobs and single-task jobs.

tasks is made available. In the OpenCloud cluster, data on the amount of virtual memory (VM) required by each mapper and by each reducer in a job are available in the job's config files. In the LANL cluster, data is available on the number of requested processors by each job submission.

Figure 2 plots the distribution of the average requested resources by tasks in a job (or just the requested number of processors in the case of LANL's jobs), for jobs that complete, fail or get killed, separately.

The first thing we observe is that resource requests varied distinctly between successful and unsuccessful jobs across all the clusters in our dataset. In Google, failed jobs requested significantly more memory and disk-space capacities than the rest of the jobs. In the OpenCloud cluster, the amounts of requested VM for mappers and reducers were left at the default value (1024 MB [1]) in more than 99% of jobs that completed or failed. Killed jobs were the exception, as their configured VM requests deviated significantly from the default settings. At LANL, jobs that failed or got killed reported significantly higher requests for processors in the cluster than other jobs. Our observations suggest that knowledge of the amount of cluster resources a job requests can potentially be used to predict the job's final status. (We study the prediction of unsuccessful executions in Section V.)

After examining resource requests, we now turn our attention to how these jobs actually *consumed the cluster's resources* during runtime. Data on resource utilization is available from Google and OpenCloud only. In Google, data on the CPU, memory and disk usage is available for each task, aggregated over 5-minute intervals.

The OpenCloud traces contain data on I/O task usage only, collected from Hadoop's Distributed File System (HDFS) and node-local filesystem counters. We examine four I/O counters: the amount of bytes read by mappers when a job starts (`HDFS_BYTES_READ`); the amount of bytes written by reducers when the final output is produced (`HDFS_BYTES_WRITTEN`); the amount of bytes read from the local filesystem on a reducer node (`FILE_BYTES_READ`); and the amount of bytes written by mappers to local disks when the map output is produced, combined with the amount of bytes spilled by reducers during the shuffle phase (`FILE_BYTES_WRITTEN`).

Figure 3 studies the distribution of the average utilization of cluster resources by jobs at the Google and OpenCloud clusters. We observe different resource-usage behaviours between successful and unsuccessful jobs. At Google, killed jobs used more CPU and memory than the rest, while failed jobs reported dramatically higher I/O utilization rates. It is worth noting that I/O consumption here refers to node-local disk usage; Google's traces do not contain any information on the GFS (Google File System) usage.

In OpenCloud, we also observe different I/O-usage behaviours between completed and interrupted jobs; most notably, we find that failed jobs report the highest rates for I/O writes to local disks in worker nodes. This agrees with our observation for the Google cluster, where failed jobs also report the highest local-disk usage rates across all jobs.

IV. What are some possible root-causes behind unsuccessful executions?

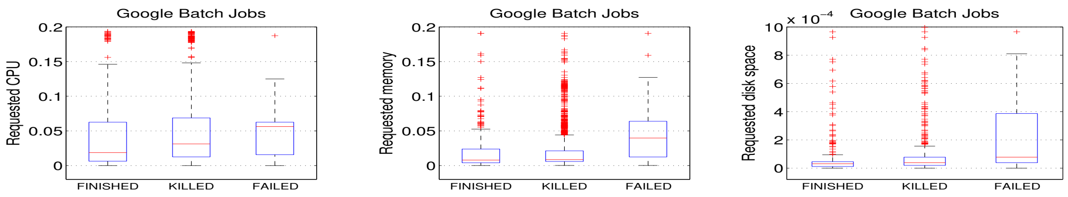
We now use the traces to investigate different hypotheses that could shed light on the possible root causes behind job termination in large-scale clusters.

A. *Resource-Limit Violation*

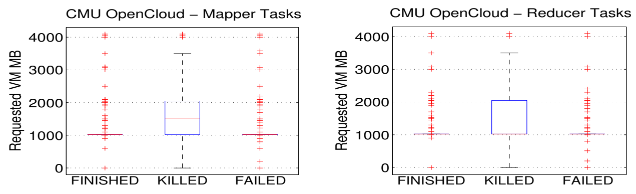
One possible hypothesis is that jobs do not execute successfully because of insufficient resources. For example, at Google a task that uses more memory than what it requested at start time can be terminated by the resource manager, and tasks that use more CPU time than their requested amount may be throttled [9]. We are interested in learning how frequently such resource violation issues happen in the field, leading to job or task termination. We use the traces from Google to explore this question, as it is the only dataset where information is made available for both CPU/memory *limits* and CPU/memory *usage* for each task in the cluster.

We compute for each task in the trace the ratio between the average amount of CPU/memory consumed by the task over its lifetime to the amount of CPU/memory requested by the task at submission time. Figure 4-(a) plots the distribution of this ratio in Google's tasks, broken down by task exit status. Additionally, we consider the ratio between the *maximum* amount of CPU/memory ever reported by a task during execution to the task's requested CPU/memory, in Figure 4-(b). The data points above the horizontal dashed line in the graphs (which refers to ratio=1) correspond to tasks that exceeded their resource limits.

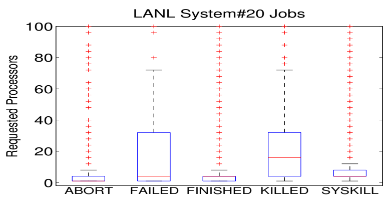
We find that in the average case, tasks rarely exceed their memory limits, but 25% of *failed* tasks did exceed their memory limits at some point in their lifetime (see right plot in Figure 4-(b)). On the other hand, consuming more CPU cycles than initially requested was not problematic for the tasks in



(a) Google Batch Jobs.

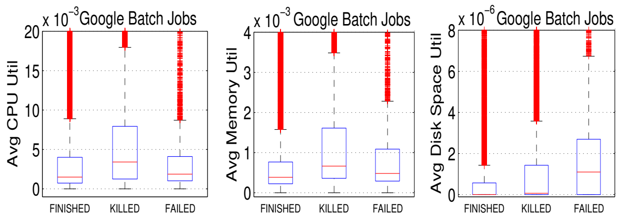


(b) CMU OpenCloud Jobs.

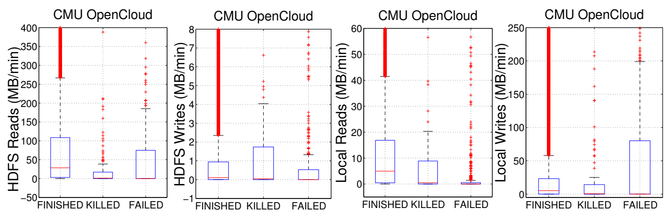


(c) LANL Jobs.

Fig. 2. Comparison of cluster resource *requests* between successful and unsuccessful jobs. (Recall that in a box plot the bottom and top of the box are the 25th and 75th percentiles, respectively, the band near the middle of the box is the median, and the '+' markers represent outliers.)

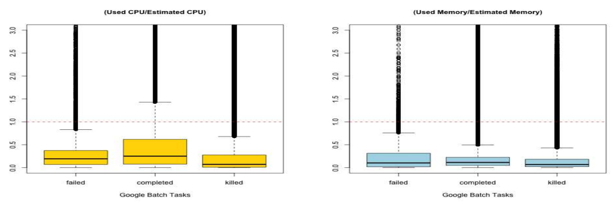


(a) Google Batch Jobs.

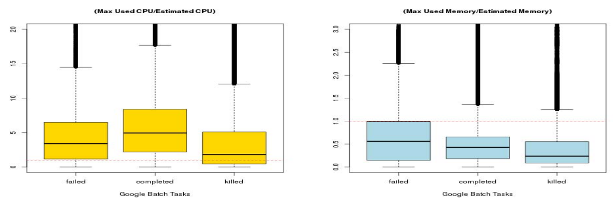


(b) CMU OpenCloud Jobs.

Fig. 3. Comparison of cluster resource *utilization* between successful and unsuccessful jobs.



(a) Ratio of average-to-requested CPU usage (left) and memory usage (right).



(b) Ratio of maximum-to-requested CPU usage (left) and memory usage (right).

Fig. 4. Resource-limit violation by Google's tasks.

this cluster. In fact, tasks that completed successfully were more likely to exceed their CPU requests than other tasks.

To better understand when a task's memory usage soars during runtime, e.g. due to memory leaks, we computed the ratio between the average memory usage in the second half of a task's lifetime to the average memory usage in its first half, for all of Google's batch tasks. We found that 22.87% of failed tasks have more than *doubled* their average memory consumption over time (indicated by a ratio greater than 2); meanwhile, 13.5% and 9.68% of killed tasks and completed tasks, respectively, report a ratio greater than 2.

Observation: For the Google cluster in our dataset, memory-management issues show a correlation with task failure. Even though failed tasks request significantly more memory than other tasks (see Section III), they are still more likely to exceed their memory limits and to experience dramatic growth in memory usage over time.

B. Job Fault-Tolerance Configuration

The next question we ask is whether the way a job is configured to deal with problems at runtime, such as slow or failing tasks, is correlated with job termination.

In the Google and OpenCloud clusters, reliability and fault-tolerance are provided by the underlying framework through data replication and task re-execution (i.e. task retries upon failures). In the LANL cluster, jobs represent tightly-coupled HPC applications where fault-tolerance is achieved mainly through application-level checkpointing. The LANL traces do not provide any data on the checkpointing parameters for these jobs. Google also do not provide information on job fault-tolerance configuration, but the effect of task re-execution during runtime on job reliability can be analyzed using the logs on task events. The CMU OpenCloud trace however contains detailed information on job and task configurations (collected from Hadoop's job_config XML files). We next examine the fault-tolerance knobs that are used to configure Hadoop's 'speculative execution' feature, handle bad data records, and manage task retries.

a) Speculative Execution (SE) in OpenCloud: SE in Hadoop is a way of dealing with cases where a mapper or a reducer task in a parallel job ends up on a slow node, therefore slowing down the entire job. The framework will then schedule redundant clones of the slow task on multiple machines and eventually collect the output from the first clone that completes. Users have the option to disable the SE feature for their jobs by unsetting these SE flags: `map.tasks.speculative.execution` and `reduce.tasks.speculative.execution`.

By analyzing the OpenCloud config files, we find that 99% of scheduled jobs left this feature enabled for both mappers and reducers. When separating jobs by exit status, we find that while only 0.02% and 0.5% of completed jobs and killed jobs, respectively, disabled SE for mappers and/or reducers, a much higher fraction of 2% of *failed* jobs had disabled the feature (divided almost equally between jobs that disabled SE for both mappers and reducers and jobs that disabled SE only for mappers).

b) 'Skip' Mode in OpenCloud: The 'skip' mode is one of Hadoop's ways of identifying bad input data that might be causing tasks to fail. The framework enables this mode after the number of attempts by a task reaches a certain limit (configured through the parameter `mapred.skip.attempts.to.start.skipping`), after which the task must report to the framework which data records it will process next. If the task ends up failing, the framework would know which records are possibly the problematic ones in order to skip them in future attempts.

In the OpenCloud cluster, we find that the task-attempt limit after which skip mode is enabled was left at its default value of 2 attempts in 96% of the jobs. When considering the exit status of the job, we find that 96.5%, 97.6% and 95.5% of completed jobs, killed jobs and failed jobs, respectively, had the default configuration for this parameter, with the second most popular value being 16 in all job-status groups.

c) Limit on Task Retries in OpenCloud: We now consider the maximum number of attempts a mapper task or a reducer task can make before the framework terminates it, configured using the parameters `mapred.map.max.attempts` and `mapred.reduce.max.attempts`. In the OpenCloud cluster, 4% of all jobs modified the default limit (4 attempts) for

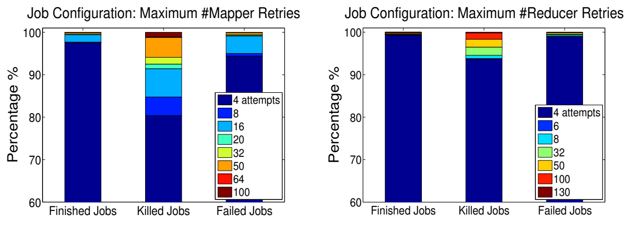
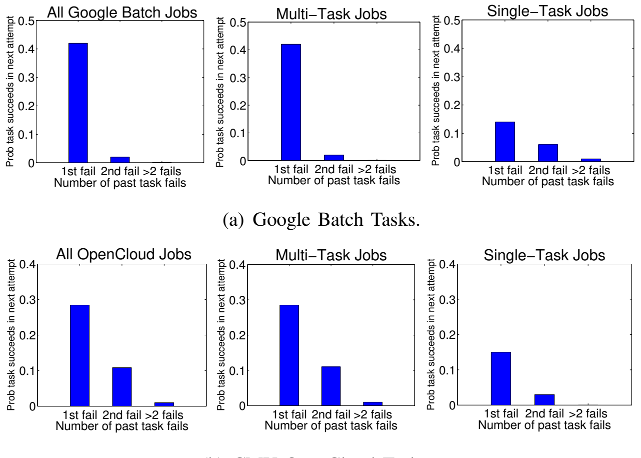


Fig. 5. Breakdown of the maximum number of task retries configured in the OpenCloud Hadoop jobs.



(a) Google Batch Tasks.

(b) CMU OpenCloud Tasks.

Fig. 6. The probability that a task succeeds in a retry as a function of the number of past failed attempts.

mapper tasks, while only 1% of the jobs modified the retry limit for reducer tasks. Figure 5 shows the detailed breakdown of the task-retry limits configured in OpenCloud's jobs. We find that the value for this limit went as high as 100 for mappers and 130 for reducers. The bar plots show that killed jobs had the largest fraction of jobs with high limits on task retries that exceeded the default setting, followed by failed jobs, for both mappers and reducers.

Observation: For the CMU Hadoop cluster where configuration data is available, unsuccessful jobs were more likely to have fault-tolerance configurations that deviate from the default framework settings; e.g. the disabling of speculative execution and the increase in the allowed limit on task retries.

C. Task failures causing job failures

In this section we investigate what role task failures play in job failures. More than 98% and 95% of failed jobs at Google and CMU, respectively, had at least one failed task. In contrast, only 0.25% and 20% of completed jobs at Google and CMU, respectively, had one or more failed tasks. That motivates us to review the effectiveness of the key mechansims for task-level fault-tolerance, which consists of retrying a failed task.

We begin by asking the question of how likely a task retry is to succeed, following one or more failed attempts. Figure 6 plots the probability of a task succeeding in the next attempt as a function of the number of past failed attempts, for both Google (top) and OpenCloud (bottom).

It is worth noting that the probability of a task succeeding in the first attempt without past failures (not shown in the graphs) is 98% and 97% in Google and OpenCloud, respectively. We observe from the left-most graphs in Figure 6 that this probability drops dramatically to 42% and 28% in Google and OpenCoud, respectively, in the first task retry (i.e. after making one failed attempt). In the second retry, a task's success chance drops further to only 2% in Google and 11% in OpenCloud. The probability of succeeding after more than two or three attempts becomes negligible in both clusters.

Interestingly, when considering single-task jobs only (right-most graphs), we find that the probability of succeeding in the first task retry is much lower than in parallel jobs (middle graphs): 13% and 15% in Google and OpenCloud, respectively. Note that the probability of a single-task succeeding without failures (not shown in the graphs), is 97% in Google and 90% in CMU, so these lower retry chances are not explained by lower initial success chances. One possible explanation of this observation in single-task jobs is the lack of race conditions that may affect the completion of parallel jobs, therefore making single-task jobs fail more deterministically.

While our observations above indicate that retrying tasks more than twice is almost futile, we find that many users are optimistic beyond hope, wasting significant cluster resources in the process: We studied the distribution of #task retries in failed tasks at Google and OpenCloud and found that 70–90% of failed tasks tried re-executing more than once and 15–30% tried re-executing more than twice. Some failed tasks tried re-executing up to 70–100 times before their terminal failure.

Observation: For the clusters in our dataset where task re-execution is the recovery mechanism, we find that attempting to re-execute a task more than once or twice is futile. Yet, we find that larger numbers of retries are common in tasks that terminally fail, and that task failure is strongly associated with the entire job failing.

D. *Machine Outages*

Another possible root cause behind job interruption is the failure of the physical machine (node) hosting the job. In our dataset, information on node outages are available for the Google and LANL clusters. At Google, records of all machine removals from the cluster that happened during the data collection period are available. The 'removal' of a machine could either be due to planned maintenance or unplanned failure. According to Google [17], whenever a machine is removed, any task running on it will have a task 'evict' event logged and the framework will try to re-schedule the task on another machine. At LANL, records of all node outages that happened during the trace collection period are also available. According to the LANL website [3], when a node fails, all jobs running on it should exit the cluster with a 'failed' status.

We examine the effect of node outages on job reliability in the Google and LANL clusters, by correlating the logs on *machine events* with the traces on *job events*. We are particularly interested in quantifying the fraction of job terminations that can be attributed to machine outages. Therefore, we use the recorded timestamps to identify all tasks at Google and all jobs at LANL whose execution was interrupted by a machine outage event.

TABLE III
BREAKDOWN OF TASK AND JOB EVENTS CORRELATED WITH MACHINE OUTAGES AT THE GOOGLE AND LANL CLUSTERS.

Data Source	Event Type-X	Percent% of events correlated with machine removals	Percent% of all Type-X events in the cluster
Google	`EVICT`	91%	16%
	`KILL`	8.4%	0.04%
	`FAIL`	0.3%	0.004%
	`LOST`	0.3%	1%
LANL	`SYSKILL`	48%	3%
	`KILL`	29%	4.8%
	`ABORT`	14%	3%
	`FAIL`	2%	5.4%
	(Unknown)	7%	7%

Table III presents the breakdown of the types of job/task events correlated with machine removals (left column); we also show for each event type, what fraction of the total number of events of this type are correlated with machine outages (right column). For example, in the Google cluster, we find that 91% of the task events that were interrupted by machine removals at Google are task 'evicts', and that these machine-related evicts constitute 16% of all task evicts that appear in the trace (the remaining evicts are most likely due to preemption by higher-priority tasks).

The second-highest event type that appears with machine removals at the Google cluster are task kills (8.4%). One hypothesis is that these kills could be related to sudden machine failures where no proper termination signal was sent to the tasks before the machine shut down. Interestingly, when we turn to the LANL cluster, we find that more than 90% of job events correlated with machine shutdowns were either 'syskilled', 'killed', or 'aborted'.

Task and job 'fails' constituted only 0.3% and 2% of all machine-removal related events at Google and LANL, respectively. In the Google cluster, these task fails represent a tiny fraction (0.004%) of all task-attempt failures that appear in the trace. At LANL, job failures correlated with machine outages explain 5.4% of all job failures in the cluster.

Observation: For the clusters in our dataset where machine-outage logs are available, we find that the majority of task/job events correlated with machine removals are evicts or kills. Machine removals can be identified as the root cause behind task *failures* very rarely at the Google cluster, and explain around 5% of job failures at the LANL cluster.

V. PREDICTION OF UNSUCCESSFUL EXECUTIONS

After studying the properties of unsuccessful jobs and investigating several possible root causes behind their termination, we now ask the question of whether we can predict the unsuccessful termination of a job during runtime. We focus primarily on the Google traces to design and evaluate our prediction framework, since these traces contain detailed information on job and task executions and high-resolution logs of CPU, memory, and I/O usage.

A. Design of Prediction Framework

Our goal is to design a framework that is capable of predicting the unsuccessful termination of a job or a task running in the cluster. Table IV summarizes the input variables we use as predictors. Fields under 'Config' refer to variables whose values are known at job configuration time, i.e. before execution. Fields under 'Counters' and 'Usage' are known only during runtime.

Prediction technique: We use Random Forests [4] (RF), the decision-trees based machine-learning technique, to make predictions about jobs and tasks. We configure the RF algorithm to use 50 trees in each iteration, and we split the data such that 70% of Google's jobs in the traces are used for training and 30% for testing.

Prediction knobs: We use a default classification (decision) threshold of 0.5 to predict class membership, but in some of our experiments we vary the threshold from as small as 0.1 and up to 0.9. For our training set, we experiment with various ratios between the positive and negative classes of data points in order to correct for the bias in the original data (since the ratio between successful and unsuccessful events is not balanced)–a technique commonly referred to as 'data oversampling'.

Metrics: The metrics we use to quantify the quality of our predictions are *precision* and *recall*, two standard machine learning metrics. Precision refers to the number of true positives divided by the number of true and false positives. Recall is the number of true positives divided by the number of true positives and false negatives. For example, when predicting whether a job is going to fail, precision refers to the fraction of jobs that did actually fail out of all the jobs that were predicted to fail, and recall refers to the fraction of jobs that were predicted to fail out of all the jobs that did actually fail.

Note that another common metric, *accuracy*, which was used in some prior work [6], [11], [12], is not an interesting metric for the type of events we are trying to predict. Accuracy is true positives plus true negatives divided by the entire population (= *1 - missclassification rate*). For rare events, a predictor that always makes negative predictions will have high accuracy, but is not very useful in practice.

Prediction scenarios: We experiment with different scenarios, beginning with the simplest one: can we predict a job's unsuccessful termination using only data available at job submission time? We call this scenario '`Config only`'. The second scenario we explore assumes knowledge of early resource-usage indicators of the job, particularly the average and standard deviation in the CPU, memory, and disk consumption of the job's tasks over the first 5 minutes of execution (in addition to configuration parameters). For jobs that ran for shorter than 5 minutes, we noticed that a subset of these short-lived jobs had valid usage logs recorded in the traces (e.g. over the first 2 or 3 minutes of execution). We included these short-lived jobs with usage data in our experiments. We refer to this second scenario as '`Config + 5min Usage`'.

In the third prediction scenario, we add a runtime flag that is set to `TRUE` once a single task-failure is detected across all tasks in a job. We refer to this setup as '`Config + TaskFailFlag`'. Finally, the fourth scenario assumes knowledge of all the parameters discussed above: '`Config + 5min Usage + TaskFailFlag`'.

B. Job-Kill Prediction

We begin by exploring if job 'kills' can be predicted accurately in Google. Recall that a kill in the Google cluster can indicate either a manual abort by the user or a termination due to an unsatisfied dependency. Once a job is killed, all its tasks are also killed immediately. Figure 7 shows the precision and recall obtained under our different predictors;

TABLE IV
SUMMARY OF PREDICTION VARIABLES IN OUR FRAMEWORK.

Variable	Category	Description
`job_status` or `task_status`	Response	The unsuccessful outcome we are interested in predicting, which could be either `FAILED` or `KILLED`.
	Input Variables (Predictors)	
`userID`	Config	ID of the engineer who submitted the job.
`logical_job_name`	Config	ID of the application that this job is running.
`scheduling_class`	Config	Each job has a scheduling class that drives machine-local policies for allocating resources.
`priority`	Config	Job and task priorities determine if they are given preference for resources.
`num_tasks`	Config	Degree of parallelism in a job.
`different_machines_rest`	Config	This flag, when enabled, means that all tasks in a job must be scheduled on different physical machines.
`requested_cpu`	Config	Amount of requested CPU capacity by a task.
`requested_memory`	Config	Amount of memory requested by a task.
`requested_disk`	Config	Amount of disk space requested by a task.
`early_cpu_usage_avg`	Usage	Average amount of CPU used by a job's tasks over the first 5 minutes.
`early_memory_usage_avg`	Usage	Average amount of memory used by a job's tasks over the first 5 minutes.
`early_IO_usage_avg`	Usage	Average amount of I/O used by a job's tasks over the first 5 minutes.
`early_cpu_usage_CoV`	Usage	Variability in CPU usage among co-tasks in a job over the first 5 minutes.
`early_memory_usage_CoV`	Usage	Variability in memory usage among co-tasks in a job over the first 5 minutes.
`early_IO_usage_CoV`	Usage	Variability in I/O usage among co-tasks in a job over the first 5 minutes.
`TaskFailFlag`	Counters	Per-job: this flag is set to `TRUE` if at least one task failed in the job.
`AttemptFailFlag`	Counters	Per-task: this flag is set to `TRUE` if at least one attempt failed in the task.
`PastAttemptsFlag`	Counters	Per-attempt: this is `TRUE` if there are past failed attempts made before this current attempt.

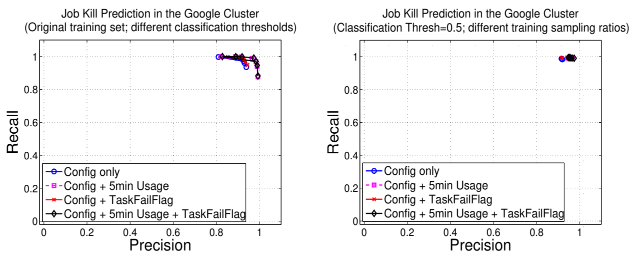


Fig. 7. Predicting Google job kills using random forests while varying the decision threshold (left) and the training sampling ratio (right) .

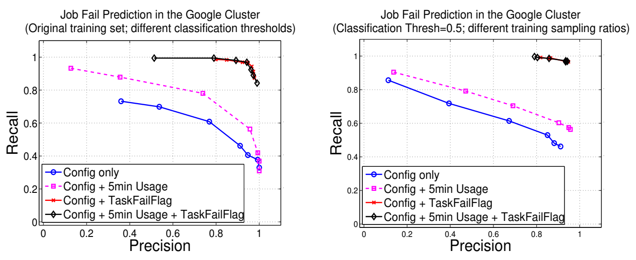


Fig. 8. Predicting Google job failures using random forests while varying the decision threshold (left) and the training sampling ratio (right) .

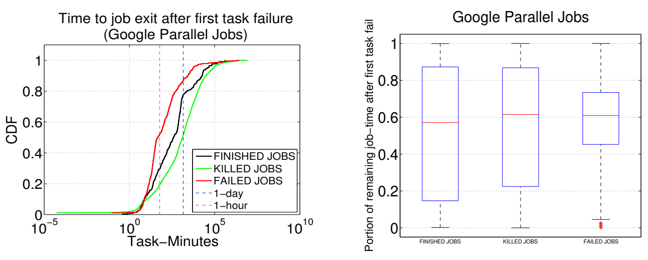


Fig. 9. Distribution of the remaining time (left) and the remaining portion (right) of a parallel Google job, after the first task failure.

the left graph shows the results when varying the classification threshold from [0.1–0.9], and the right graph shows the results under a fixed threshold of 0.5 while varying the sampling ratio in the training set.

We find that job kills can be predicted very accurately even under the simplest scenario of using only job configuration parameters. When taking a closer look at the output of the decision trees, we find that the amount of CPU, memory, and disk requested by the job plus the job's degree of parallelism were marked as the most significant predictors of a job kill. We also observe that the quality of the predictions for job kills are more sensitive to the classification threshold than they are for the training sampling ratio. We next explore if the same level of high-quality predictions can be achieved for predicting a job failure.

C. Job-Failure Prediction

Figure 8 shows the precision and recall obtained when running our job-failure predictor, using the same configurations described above for job-kill predictions. We observe much higher variability in the quality of the predictions obtained under different settings, as well as a broader spectrum of precision-recall tradeoffs. Using job config data alone no longer produces the same excellent results as was the case with job kills. Nevertheless, it is interesting to observe that just by looking at how a job is configured we can predict with up to 79.5% precision that a job will fail, recalling more than 50% of all failed jobs. A closer look reveals that the amount of disk-space requested by a job is the significant predictor used by the decision trees.

We find that considering a job's first five-minute resource utilization improves the predictions to some extent, but the most significant improvements are achieved when we include the `TaskFailFlag`. Using this flag, we can successfully recall up to 94% of all failed jobs with at least 95% precision.

To better understand how much benefit can be reaped by this predictor in practice, we examine how early in a job's lifetime does the first task failure happen. Figure 9 plots the distribution of the remaining time in a parallel job (left) and remaining portion of a parallel job's lifetime (right), after the first task failure happens. We find that the median time-to-exit in failed jobs is around 60 minutes, and in killed jobs a whole day. In terms of remaining job portions (Figure 9-(right)), we find that the median is around 60% across all job-status groups; i.e. half of the jobs with failed tasks run for roughly 40% or less of their total execution time before the first task failure event. This result shows that job failure can be predicted early and accurately.

Given that a single task failure is a strong predictor of a job failure, the next natural question to ask is whether we can predict the failure of a *task* within a job.

D. Task-Failure Prediction

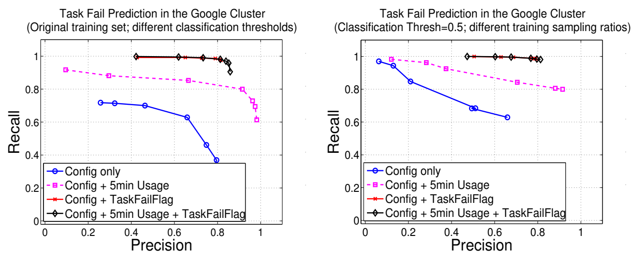
Recall that in case the first attempt at executing a task ends in failure, the task might be retried a few more times. We first look at predicting a task's final status, i.e. the status when the tasks exits (possibly after a number of task retries) in Section V-D1, and later, in Section V-D2, at predicting the status of intermediate retry attempts.

1) Predicting final task status: In addition to the scenarios described above, we add an `AttemptFailFlag` predictor, which we set to true if at least one attempt for this task failed in the past. Figure 10(a) shows the task-level results. We observe that using config data alone produces less precise predictions compared to jobs, and that knowledge of the task's resource usage in the first few minutes improves the predictions more significantly than when used for job-level predictions. The reason might be that the job's view has lower resolution than the task as it considers the average consumption of all co-tasks in the job. Finally, we find that using the `AttemptFailFlag` produces the highest improvement in the predictions, particularly for our recall rate. This observation led us to our next question: can we predict failures on the *task-attempt* level?

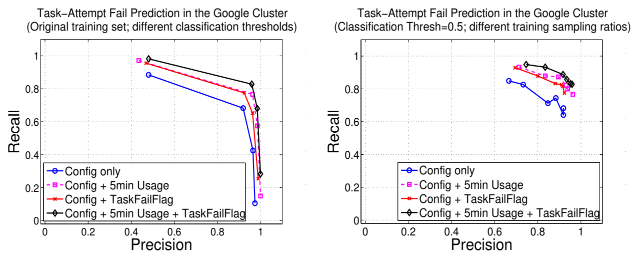
2) Predicting task-attempt failure: Figure 10(b) shows that the failure of individual task attempts can be predicted with higher precision than task-level and job-level failures. The amount of requested memory, requested disk space, and degree of parallelism were marked as most significant predictors by the decision trees.

VI. Mitigating the Effect of Unsuccessful Executions

In this section, we explore how the findings of our work and the prediction framework could be used to mitigate the effect of unsuccessful job executions. We start with a series of different suggestions in Section VI-A and then present one specific use case in Section VI-B where we develop one



(a) Final task status.



(b) Task-attempt status.

Fig. 10. Predicting Google task failures using random forests while varying the decision threshold (left) and the training sampling ratio (right).

of the suggestions into a concrete algorithm and evaluate its effectiveness in trace-driven simulations.

A. Policy recommendations

Previous work on predicting task failures and kills has focused exclusively on using those predictions to proactively terminate jobs that are likely to be unsuccessful. We believe that the usefulness of this approach is questionable in practice for at least two reasons: First, a user will likely be unhappy if their job is being killed by the system with the only reason being that the system believes the job is unlikely to succeed. It also makes the possibly incorrect assumption that a job which eventually fails does not have any value for the user: jobs might generate output continuously throughout their runtime and those intermediate results can be useful to the user. Second, proactively killing a job makes it very difficult for the user to determine what the problem was with that job. Letting the job run until it fails by itself might generate error messages or debugging information that the user can use for trouble shooting and fixing their job.

Instead, we make a number of alternative suggestions, which we believe to be more practical and impactful:

• *Limiting the number of retries:* Our first suggestion is simple: Given that the success rate of retrying a failed task drops to zero quickly with the number of attempts (recall Figure 6), we recommend putting a cap R on the number of times a task is retried after failure, to limit the amount of wasted resources.

• *Turning on additional monitoring:* Often jobs are able to generate a lot of additional monitoring and debugging information, but collection of such information is typically disabled during production runs for efficiency reasons. One potential use for our prediction framework would be to dynamically enable the collection of additional information once unsuccessful execution is predicted, for the user to use this information later for trouble shooting.

• *Increasing frequency of checkpoints:* Many programs rely on periodic checkpoints that can be used for recovery in the case of failure or to provide intermediate results for killed/failed jobs. Our prediction framework could be used to turn on checkpointing or increase checkpoint frequency once unsuccessful execution is being predicted.

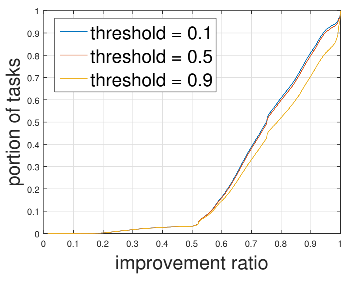
• *Adjusting scheduling priority or allocated resources:* Given that both a job's scheduling priority [13] and its allocated resources are correlated to its exit status, one could consider adjusting these attributes based on exit status predictions. The specific details of such a policy would depend on system internal details on what those jobs are and what makes them likely to fail. One could argue that priority and resources should be reduced for jobs that are unlikely to succeed in order to reduce the negative effect they might have on other (more promising) concurrent jobs and on overall resource usage. Alternatively, one could argue that for jobs predicted to fail that are deemed important, their priority and resources should be increased in order to improve their success chances.

• *Scheduling redundant tasks (speculative execution):* Since it is difficult without further system-level information to make recommendations that reduce failure rates, we instead focus on strategies that minimize the damage done by failed tasks. One harmful side effect of failed tasks is that they are responsible for a significant amount of job slowdown. We therefore propose to use a strategy similar to the speculative execution used to deal with stragglers [1]: Once a task is predicted to likely end in failure, we proactively start a clone of the task in parallel, rather than waiting until the task fails and then start a retry. While this will not increase the task's chances of success, it means that the final execution status is reached faster. We work out the details of such a strategy, including a trace-driven evaluation of its benefits in the next section.

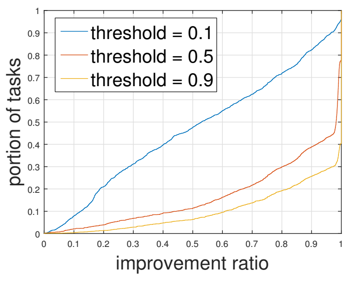
B. Use case: Proactive cloning

To minimize the slowdowns caused by failing tasks [13], our goal is to ensure that tasks that fail do so quickly. We propose to use our prediction framework from Section V to preemptively start early retry attempts for tasks that are likely to fail (i.e. overlap execution of the retry attempt with the first task attempt). For example, for a task that ends in failure after two failed attempts, our goal is to preemptively start a clone of the task in parallel with the first task attempt, rather than waiting until the first attempt fails and then serially starting a second attempt. If the clone is started early enough, one would expect to shorten the time to completion significantly. We modify our prediction framework slightly to use *sliding-windows* that monitor task resource usage online, instead of relying only on the first few minutes.

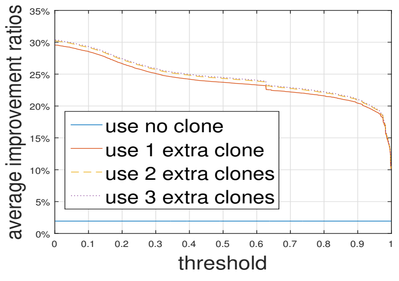
Our cloning algorithm is described in Algorithm 1. The input parameters include a cap R on the total number of retries being created and C, the number of extra clones that are



(a) Acceleration in failed multi-attempt tasks.



(b) Acceleration in successful multi-attempt tasks.



(c) Improvements for different thresholds and #clones.

Fig. 11. Task acceleration using proactive cloning.

Algorithm 1 Create task with clones based on failure prediction

```
1:  Start the first attempt of the original process;
2:  $r := 0$   // $r$ maintains number of created attempts
3:  $c := 0$   // $c$ maintains number of created clones
4:  **At any task attempt:**
5:  **upon event** *failure is predicted for the first time* **do**
6:      $k := \min\{1,\ R - r,\ C - c\}$
7:      start $k$ clones
8:      $c := c + k$
9:      $r := r + k$
10: **end event**
11: **upon event** *attempt failed* **do**
12:     **if** $r < R$ **then**
13:         start a new attempt
14:         $r := r + 1$
15:     **end if**
16: **end event**
```

allowed to be created when failure is predicted for a task. In the remainder of this section we first evaluate how effective this method is at reducing task completion times and then evaluate overheads it creates.

1) Evaluation of task acceleration: We use simulations based on the Google trace to evaluate by how much task completion times can be accelerated using our proactive cloning policy. After training our predictor on a randomly selected training set of 25,000 tasks, we evaluate Algorithm 1 on a different, also randomly selected set of 25,000 tasks. We do so by replaying tasks exactly as given in the trace, except that we modify the start time of retries based on the predictor (i.e. a retry gets started as soon as failure is predicted). We assume that the execution time and exit status of individual task attempts is the same as in the original trace, i.e. it is not affected by changing its start time. While in reality execution times can vary from one run to another, it is reasonable to assume that in expectation and on average it will be the same. Note that the tasks that can benefit from our policy include all tasks which experience at least one failed task attempt, independently of whether a task eventually succeeds (after one or more failed attempts) or permanently fails (after multiple failed attempts).

Figure 11(a) and (b) show the cumulative distribution function (CDF) of the completion time under Algorithm 1 as a fraction of the original completion time in the trace. For this experiment, we set R to 4 (i.e. a maximum of 4 retries), C to 1 (i.e. only 1 extra clone is started when a failure is predicted) and configure the failure prediction algorithm with 3 different thresholds, hence varying how aggressive the predictor is. Figure 11-(a) shows the results for jobs whose final exit status is failure and Figure 11-(b) shows the results for jobs that succeed (but after one or more failed attempts). We observe that even for a conservative predictor with threshold 0.9, the median failed job finishes 20% faster. More aggressive prediction does not significantly improve results. Figure 11-(b) shows the results for jobs who eventually succeed, but only after one or more failed attempts. Reduction in completion time is less in this case, but still significant.

TABLE V
THE OVERHEAD ON ENTIRE WORKLOAD W.R.T. DIFFERENT THRESHOLDS AND NUMBER OF EXTRA CLONES.

Threshold	1-clone	2-clones	3-clones
0.1	38.80%	74.84%	108.33%
0.5	10.82%	21.11%	30.32%
0.9	3.52%	6.92%	9.73%
0.95	2.89%	5.66%	7.97%
0.99	1.12%	2.09%	2.81%

We also experimented with different values of the number of clones C and show the results in Figure 11-(c). We find that aggressively starting more than one clone when a failure prediction is made has only little benefit when it comes to completion times. However, as we will see in the next subsection it can come with significant overheads.

2) Evaluation of overheads: Note that for a perfectly accurate task-failure predictor the amount of work in the simulated system would be the same as in the original system, since the same work is executed, just at potentially different times (retries take place earlier). The only source of overhead are false positives of the predictor, which trigger a retry to be started, although it later turns out that a previously started attempt succeeds. Hence, the overhead will depend on how aggressive the predictor is (its threshold) and the total number of clones that is started after each failure prediction (i.e. the parameter C). Table V reports the overhead (the increase in total work on the system) that comes with different configurations of algorithm 1. We observe that in order to keep overheads reasonable, thresholds need to be conservatively chosen, in the range of 0.9 or higher.

VII. Related work

Several papers have recently studied the properties of unsuccessful executions in a large-scale system [5], [7], [10], [13], [14] by analyzing the workload traces made available by Google for one of their clusters [17]. One key limitation of this previous work is that it focused on a single cluster only, without investigating if the observed characteristics of unsuccessful jobs can be generalized to other systems. Our work extends these studies by broadening our dataset to include, besides the Google trace, logs from two other large-scale systems at different organizations, and explores more factors that affect job and task reliability which previous work did not consider (e.g. resource-limit violation, job fault-tolerance configuration, machine outages, and more).

We find that some of the properties of unsuccessful jobs in the Google cluster that previous work reported do exist in other systems, such as relatively large resource requests and low chances of succeeding in task retries [13]. Other observed properties that show weaker trends in other clusters include the relatively high rates of job interruptions in multi-task jobs.

Another set of papers [6], [11], [12] focused primarily on job failure prediction, using neural networks [6], [11] or linear/quadratic discriminant analysis [12]. All these papers propose a failure mitigation policy based on terminating jobs that are expected to fail. We provide a different prediction technique that achieves significant improvements in prediction quality over previous work, and uses random forests which are easier to train and use in practice. Our work is also the first to propose a number of practical ways a predictor could be used and to present and evaluate in detail one particular use case that exploits features available in distributed frameworks.

VIII. Conclusion

The dependability of parallel clusters critically relies on their capability to deal effectively with failures. This paper uses a sizeable dataset of workload traces from different production systems to study job terminations in parallel clusters. We identify patterns that distinguish how unsuccessful jobs are configured and executed in the field.

We find that across multiple clusters, unsuccessful jobs ran for longer durations, requested more cluster resources, reported heavier I/O activity than completed jobs and more often have parameter configurations that deviate from a framework's default parameters. Our analysis points to several factors as important possible causes behind job failures (e.g. resource-limit violation and memory leaks), and identifies others as less significant (e.g. machine failures).

We design and evaluate an effective predictor of job and task terminations and identify parameters with strong predictive power of job or task termination. We find that knowledge of job config parameters known at launch time alone is sufficient to predict whether a job is going to be killed, and that a single task failure is a very accurate and early indicator of a job failure (despite built-in methods for recovering failed tasks, such as task retry). We are also able to predict accurately and early in its lifetime the exit status of individual task attempts, with the help of online resource usage.

Finally, we propose a number of methods for employing our results to mitigate failures in clusters and provide a detailed evaluation of one of them: we demonstrate how task cloning guided by failure prediction can be used effectively to limit the effect of unsuccessful executions.

Acknowledgment

We would like to thank Google, LANL, and the CMU Parallel Data Lab for making the job traces used in this paper publicly available. We would also like to thank Sotirios Damouras from the University of Toronto for his insightful answers to our statistics questions. Finally, we thank our anonymous reviewers for their feedback. This work has been funded by an NSERC discovery grant.

References

[1] Apache's Hadoop DFS. http://hadoop.apache.org.

[2] OpenCloud Hadoop cluster trace: format and schema. http://ftp.pdl.cmu.edu/pub/datasets/hla/dataset.html.

[3] Operational Data to Support and Enable Computer Science Research, Los Alamos National Laboratory. http://institute.lanl.gov/data/fdata/.

[4] L. Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, Oct. 2001.

[5] X. Chen, C. D. Lu, and K. Pattabiraman. Failure analysis of jobs in compute clouds: A google cluster case study. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 167–177, Nov 2014.

[6] X. Chen, C. D. Lu, and K. Pattabiraman. Failure prediction of jobs in compute clouds: A google cluster case study. In *Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on*, pages 341–346, Nov 2014.

[7] P. Garraghan, P. Townend, and J. Xu. An empirical failure-analysis of a large-scale cloud computing environment. In *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*, pages 113–120, Jan 2014.

[8] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 7:1–7:13, New York, NY, USA, 2012. ACM.

[9] C. Reiss, J. Wilkes, and J. L. Hellerstein. Google cluster-usage traces: format + schema. Technical report, Google Inc., Mountain View, CA, USA, Nov. 2011. Revised 2012.03.20. Posted at http://code.google.com/p/googleclusterdata/wiki/TraceVersion2.

[10] A. Rosà, L. Y. Chen, R. Birke, and W. Binder. Demystifying casualties of evictions in big data priority scheduling. *SIGMETRICS Perform. Eval. Rev.*, 42(4):12–21, June 2015.

[11] A. Ros, L. Y. Chen, and W. Binder. Catching failures of failures at big-data clusters: A two-level neural network approach. In *2015 IEEE 23rd International Symposium on Quality of Service (IWQoS)*, pages 231–236, June 2015.

[12] A. Ros, L. Y. Chen, and W. Binder. Predicting and mitigating jobs failures in big data clusters. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pages 221–230, May 2015.

[13] A. Ros, L. Y. Chen, and W. Binder. Understanding the dark side of big data clusters: An analysis beyond failures. In *DSN*, pages 207–218. IEEE Computer Society, 2015.

[14] A. Ros, L. Y. Chen, and W. Binder. Understanding unsuccessful executions in big-data systems. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pages 741–744, May 2015.

[15] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance computing systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '06, pages 249–258, Washington, DC, USA, 2006. IEEE Computer Society.

[16] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, A. A. Chien, P. Coteus, N. A. Debardeleben, P. Diniz, C. Engelmann, M. Erez, S. Fazzari, A. Geist, R. Gupta, F. Johnson, S. Krishnamoorthy, S. Leyffer, D. Liberty, S. Mitra, T. S. Munson, R. Schreiber, J. Stearley, and E. V. Hensbergen. Addressing failures in exascale computing. 2013.

[17] J. Wilkes. More Google cluster data. Google research blog, Nov. 2011. Posted at http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html.