Proactive Straggler Avoidance using Machine Learning

Neeraja J. Yadwadkar
AMP Lab, Department of Computer Science
University of California, Berkeley
neerajay@eecs.berkeley.edu

Wontae Choi
PAR Lab, Department of Computer Science
University of California, Berkeley
wtchoi@eecs.berkeley.edu

ABSTRACT

The MapReduce architecture provides self-managed parallelization with fault tolerance for large-scale data processing. *Stragglers*, the tasks running slower than other tasks of a job, could potentially degrade the overall cluster performance by increasing the job completion time. The original MapReduce paper [1] identified that Stragglers could arise due to various reasons including software mis-configurations, hardware degradation, overloaded nodes or resource contention. Straggler mitigation techniques are mainly concentrated on being agnostic to causes behind their occurrence and spawning speculative copies to mitigate them. A fundamental unanswered question though is "what really causes stragglers?". Answering this question will not only open up proactive techniques to avoid stragglers by smarter scheduling, but will also give insights into the design of clusters.

In this work, we analyze a production level Facebook Hadoop trace and perform regression using the node-level statistics in order to predict the task-execution times. We show that Machine Learning techniques are helpful in revealing the non-trivial correlation between task-execution times and node-level statistics such as CPU/memory utilization. We also present results of our analysis on MapReduce logs that we generated using the Berkeley cluster (icluster) with 11 nodes. Using decision tree like approach further, we show how to come up with interpretable rules that a cluster scheduler can easily use. These rules can guide the scheduler to a task assignment that avoids or minimizes the number of stragglers. Our evaluation shows that the proposed changes to scheduler cause a negligible overhead. Additionally, we also discuss observations and insights about the MapReduce framework, task execution times that we think are important.

The key contributions of this work include (a) Given that existing approaches assume the correlations between node-status and task execution time are either too hard to find or are non-existent, we show feasibility of finding correlations and detecting stragglers using Machine Learning and (b) We propose simple modifications to the Hadoop scheduler which are easy to accommodate and are low overhead.

1. INTRODUCTION

MapReduce is a data-intensive job processing framework widely used by giant companies such as Facebook, Google, and Yahoo!. The MapReduce model is to exploit inherent parallelism in the jobs by breaking them up into smaller tasks and by executing them in parallel on a massive cluster to gain performance benefits. The aim is to achieve the overall job execution time lower than it would be otherwise if the jobs are run sequentially. Given the need and scale of the big data applications (ex. Facebook hosts over 260 billion images), such simple and scalable large-scale data processing framework is a necessity.

1.1 Challenges

Though widely used and highly successful, unleashing full capacity of MapReduce environment still remains a challenge. There are multiple factors adding to this complexity including enormous cluster size (Facebook trace we are analyzing shows 5578 nodes). Stragglers are one of the potential causes behind cluster inefficiency. Straggler prediction is considered to be a hard problem due to the following major factors particularly challenging the prediction accuracy of task completion times:

Dependency between Tasks MapReduce framework divides a job into a sequence of time-ordered phases viz. map and reduce. This inherently renders job completion times dependent on corresponding task completion times. For example, reducers need to wait for all the corresponding mappers to finish execution and generate their output files. When some tasks take longer than the other tasks of the job, the overall job-completion time increases.

Dynamic changes in cluster environment Tasks of a job may run slowly for multiple reasons, such as software mis-configuration, hardware faults, imbalanced cluster, overloaded machines, contention over resources. Since the tasks continue to make progress unless they are faced with fatal errors, it is considered to be difficult to diagnose these failures.

Large variance in task execution time We observed that there is large variation in finishing time behavior of the same task. This is one of the biggest challenges in predicting stragglers! Determining if there are good

predictors for the long tail of finishing time is essential for this study.

We analyzed a production level trace from Facebook. We observed that the straggler-tasks are spread across the whole cluster. Figure 1 shows sample graphs of task execution times on randomly chosen jobs from approximately 27000 jobs running on the Facebook cluster. Note that majority of jobs finish in the initial few bins of duration but there is a long tail that indicates some tasks run slower than others and could potentially fault job's progress. We define such slow-running tasks to be *Stragglers*. Informally, a task that takes more than a certain threshold time (a multiple of the median-execution time taken by the other tasks of the job) is called a *Straggler*.

1.2 Existing Approaches

Two of the widely used existing straggler mitigation techniques, viz. 'Blacklisting' and 'Speculative execution' react in an 'after-the-fact' manner. **Blacklisting** Hadoop blacklists TaskTrackers (slave nodes) depending upon the number of task-failures experienced on that node. No further tasks are then scheduled on such nodes. Hadoop provides manual way of blacklisting a node (by modifying the mapred-site.xml conf file). This is intuitive, however, it is not trivial to decide when to blacklist a node. Node could just be temporarily overloaded and hence shouldn't actually be blacklisted. In such cases, manual blacklisting could result in wastage of resources. Other challenges to effective blacklisting are offered by Complex interactions involving network interactions, resource contentions. Blacklisting is hence considered inefficient as simple counting-based techniques or heuristics are incapable of finding the exact reasons behind slower task-executions.

Speculative Execution Instead of fixing the stragglers, Hadoop tries to detect such tasks and launches back-up copies. This is called *speculative execution*. Speculative execution is an optimization with a hope that these copies will finish faster. Though simple and elegant, speculative execution raises difficulty in certain contexts. Firstly, some tasks (e.g. tasks supporting interactive analytics) are so small that there is no room for straggler detection and speculation. If sequence of such light weighted jobs are cascaded and form a big analysis job as a whole, speculation has no way to protect it from stragglers. Secondly, there could be software bugs or hardware faults (such as disk read errors) due to which speculative execution may not be beneficial, rather it might increase load on the node and may cause further stragglers. Finally, speculative execution increases contention over the available resources resulting into higher latencies for new tasks.

Finding causes behind stragglers, could achieve overall efficient clusters with significant performance gain by avoiding speculative execution (i.e. redundant copies of already running tasks). Secondly, looking at the enormous scale of the cluster computing frameworks (tens of thousands of nodes clusters used in production level systems), slight improvement in efficiency could result in huge monetary gain. Thirdly, knowing the exact reasons behind slow task executions could enable better SLO-management. Predicting stragglers proactively instead of reacting after detection is thus highly desirable.

1.3 Proposal: Proactive Straggler Avoidance

We aim at proactive scheduling mechanism in order to avoid or minimize the impact of stragglers on the cluster. We analyze traces from two Hadoop clusters to understand the causes of stragglers and further, we propose modifications to Hadoop's scheduling mechanism according to our findings. Analysis of the Facebook's production cluster trace shows that Machine Learning brings out the non-trivial correlations between the task execution times and node-level statistics, such as resource utilization (CPU and/or memory). We observed that these correlations are node-specific; moreover on the same node, they change dynamically due to changes in workload and/or node-status. We confirm our observation by analyzing a 4 hour trace that we generated using the Berkeley EECS department's local Hadoop cluster (icluster). Finally, we propose modifications to the Hadoop scheduler to automatically incorporate our findings through Machine Learning techniques. The new system periodically learns correlations between node-level status and task-execution time in the form of decision-trees. This being easy-to-interprete allows scheduler to estimate task-execution time and, consequently, make a better decision. Our evaluation shows that the new design incurs only small overhead on scheduler.

To our knowledge, this is the first attempt to analyze production traces to figure out causes of stragglers and formulate interpretable rules which could be used in order to schedule tasks with the aim of preventing stragglers.

Paper Organization Section 2 discusses the relevant work. Section 3 explains Machine Learning techniques used. Section 4 presents analysis of the Facebook's trace. Section 5 explains the analysis on Berkeley EECS department's icluster in detail. In Section 6, we propose a modified hadoop scheduler and verify feasibility of the design. Section 7 concludes and presents the future direction.

2. RELATED WORK

Stragglers continue to exist even after blacklisting nodes. The MapReduce paper [1] noticed the problem of stragglers and suggested speculative execution as an optimization. Further improvements were provided including those by LATE [3], Mantri [4]. Mantri [4] points out the possible categories of root causes behind

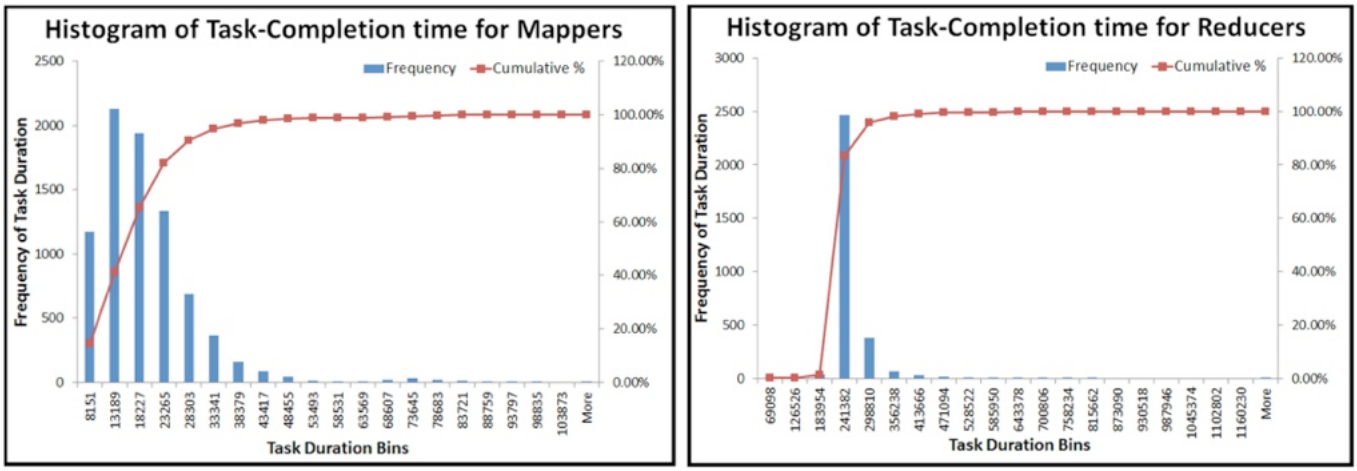

Figure 1: Stragglers do exist!.

stragglers: (a) machine characteristics that includes status aspects such as disk reliability and dynamic aspects such as resource (processor/memory etc.) contention. (b) network characteristics and (c) data skew. All the proposed approaches deal with stragglers after they occur. Whereas, in this work, we are attempting a proactive scheduling in order to avoid straggler tasks. We believe this would result in improved task completion times (as indicated by the initial results), higher overall system utilization and guarantee of meeting tighter Service Level Objectives (SLO) bounds.

In most of the related attempts, it is considered very difficult to model and find out correlations between task running times and dynamic node-level characteristics. Multiple papers [5, 6, 8, 7, 9] present that building models of running tasks and estimating task performance is non-trivial and inaccuracies in them could cause further degradation in performance. To our knowledge, this is the first attempt to analyze production level Hadoop traces to find out correlations between node statistics and task completion times.

Lack of locality could be an essential reason behind task's slow progress. This was addressed in 'delay-scheduling' [2] which aimed at achieving performance by locality-aware scheduling. We plan to accommodate our modifications to the scheduler in such a way that they go hand-in-hand with the decisions made by delay scheduling mechanism. We propose the following: using the rules (through regression tree) that we developed, select a set of nodes satisfying certain criterion for task-completion time. Then with delay scheduling mechanism, schedule a task on a node amongst the selected set of nodes that satisfies locality requirements for performance. If none of them satisfy locality, wait as per delay scheduling. As of now, we do not have experimental evaluation using this implementation, but we plan to verify this further.

Facebook cluster[10] employs Hadoop's fair-scheduler with modification to provide better resource isolation for jobs. Specifically, the modified algorithm reflects the status of each node on terms of CPU and physical/virtual memory utilization. Notice that other important factors defining node status, such as local disk I/O and network traffic, are pointed out to be less critical for the considered workload and hence are ignored. This fact supports our argument that node status is important in estimating task execution time, even though their main purpose, providing resource isolation, is different from our interest which is to improve overall job completion time by avoiding stragglers.

Number of attempts [18, 19, 20, 21] applied machine learning in order to predict resource contention in various contexts. Our work differs from these approaches in the sense that they estimate resource usage of applications whereas we aim at estimating execution time of applications (tasks).

3. SYSTEM DESIGN

In this work, we analyze a production level Facebook Hadoop trace and predict the task-execution times by regressing on the node-level statistics. We further build a regression tree to guide the scheduler. Figures 2 and 3 depicts the major components of our system which are explained in this section.

3.1 Task-execution time prediction

Estimating task-completion times is an important step for proactive scheduling. We use regression techniques as explained below.

3.1.1 Problem Formulation

From the Facebook trace, we collected (and estimated) a number of relevant parameters based on inputs from

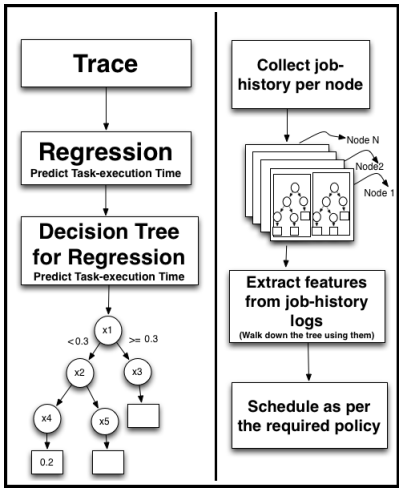

Figure 2: **System Workflow.**

experts in the MapReduce domain. Using these parameters, we posed the problem of predicting task-completion times as an instance of regression. Let N denote the total number of nodes and let n_k denote the total number of tasks executing on k^{th} node. Since, map and reduce tasks differ in the nature of their work, we need distinct models for them on each node. We basically have $2N$ regression instances each with n_k samples where $k = 1, \ldots, N$. Each of the N datasets is given by:

$$D_{map,k} = \{X_k, y_k\} \text{ and } D_{reduce,k} = \{X_k, y_k\}$$

where $X_k \in R^d$ is a d-dimensional feature vector[1], $y_k \in R$ denotes the task duration. Also $n_k = n_{mk} + n_{rk}$ where $n_{mk} = |(D_{map,k})|$ and $n_{rk} = |(D_{reduce,k})|$. $|x|$ denotes cardinality of x We want to learn the mapping $f : X \to y$ s.t. $f(x_i) = y_i \ \forall i$

3.1.2 Regression Techniques Used

We attempted well-known regression techniques including Linear Regression, Gaussian Processes, Support Vector Regression. For the sake of completeness, we briefly provide here some of the formulations we implemented.

Linear Regression In linear regression, the output variable Y is modeled as a linear function of real-valued independent variables X plus noise.

Objective: Find function $f : X \longrightarrow y$ that is a linear combination of input variables.

$$f(x) = a_0 + a_1 * x_1 + a_2 * x_2 + . + a_d * x_d$$

where $x_0 = 1, a_0, a_1, a_2, \ldots, a_d$ are the weights/parameters.

Gaussian Process Regression Due to lack of space, we briefly provide the GP regression formulation, for details, please refer [12, 13, 14]. In the simplest form of Gaussian Process regression, we assume that the output variable is given as:

$$y_i = f(x_i) + \epsilon_i \ \text{ where } \epsilon_i \sim N(0, \sigma^2_{noise}),$$

where σ^2_{noise} is the variance of the noise. If we have $f(x) = x^T w$, where w are the vector of weights of the features, the probability density of the observed data points from the training data set can be factored as,

$$P(y|X_k, w_k) = \prod_{i=1}^{n_k} p(y_i|x_i, w) = N(X_k^T w, \sigma^2_{noise} I).$$

In the Bayesian formalism,we need to specify prior probability distribution, lets put a zero mean Gaussian prior on weights w with covariance matrix Σ_p:

$$w \sim N(0, \Sigma_p).$$

Using the Bayes' rule, we can write the posterior:

$$p(w|X_k, y_k) \sim N(\hat{w} = \frac{1}{\sigma_n^2} A^{-1} X_k y_k, A^{-1}).$$

where $A = \sigma_n^{-2} X_k X_k^T + \Sigma_p^{-1}$. To make predictions, we average over all possible parameter values weighted by their posterior probability. Thus, the predictive distribution for $f_* \triangleq f(x_*)$ at x_* is obtained by averaging the output of all possible linear models with respect to Gaussian posterior,

$$\begin{aligned} p(f_*|x_*, X_k, y_k) &= \int p(f_*|x_*, w) p(w|X_k, y_k) dw \\ &= N(\frac{1}{\sigma_n^2} x_*^T A^{-1} X_k y_k, x_*^T A^{-1} x_*) \end{aligned} \quad (1)$$

We also evaluated Support Vector Machine Regression ($\nu - SVR$). The results are shown in Table 2. High accuracy using these techniques suggest that there exist predictability in the node statistics. However, interpretability is another major concern in this context and hence, we next used regression trees for regression.

Decision Trees for Regression Regression Tree was a natural choice for this work for the following reasons:

- Faster prediction facilitates inclusion of this code in the critical path of scheduler
- It leads to finding out variables that are most significant in prediction. Since this work focuses on estimating causes of stragglers, this proves to be important.

Decision trees are formed by a collection of rules based on variables in the modeling data set:

- Rules based on variables' values are selected to get the best split to differentiate observations based on the dependent variable. Various measure are used in order to decide which feature to split the dataset, such as entropy impurity, variance impurity, Gini Impurity and misclassification impurity.

[1]The features along with a brief definitive descriptions are presented in Table 2.

We used variance impurity in our implementation. We choose feature 'x' at tree-node 'L' that renders immediate child nodes as pure as possible. If i(L) denotes the impurity of node L, then the Gini/variance impurity is:

$$i(L) = \sum_{k \neq m} P(w_k)P(w_m)$$

where $P(w_k)$ denotes the fraction of nodes in category k.

The minimum impurity is sought after and such a feature is selected to split the data.

- Recursively, the same process is applied to each child node.
- Splitting stops when no further gain can be made, or some hard stopping criterion is met. Another way is to first build the complete tree and then prune it.

Each branch of the tree ends in a terminal node. Each observation falls into one and exactly one terminal node, and each terminal node is uniquely defined by a set of rules.

3.2 Interpretability using Regression Tree

This phase aims at building rules that are easily interpretable by the scheduler as explained in the section above. We use regression trees to this end in order to come up with a tree like structure on the node-statistics based on thresholds that are determined as part of the algorithm. Since Map and Reduce tasks are different by the nature of work they perform, we build two trees per node in the cluster. This task of building trees based on past logged node-statistics could also be distributed to individual nodes. These design alternatives are depicted in Figure 3. For simplicity and since we have a small 11-node cluster for testing, we have implemented design alternative I where master collects the node-statistics and learns regression trees. However, design alternative II is an easy and efficient extension for huge clusters.

3.3 Proactive scheduling mechanism

The estimation of task execution times is computed at regular time-intervals by walking both the trees generated in the previous phase, down as per the current values of features. If the trees are built in a distributed fashion on individual nodes, these estimates could then be piggybacked with the heartbeat messages sent from the TaskTrackers to the JobTrackers. In another option, in case of small sized clusters, the JobTracker could generate the tress and use them in order to take scheduling decisions. This also provides a ready framework for SLO-aware scheduling as every node is "advertising" task-completion times! These two possibilities of system architecture are shown in Figure 3. The figure also shows the design we are currently building based on

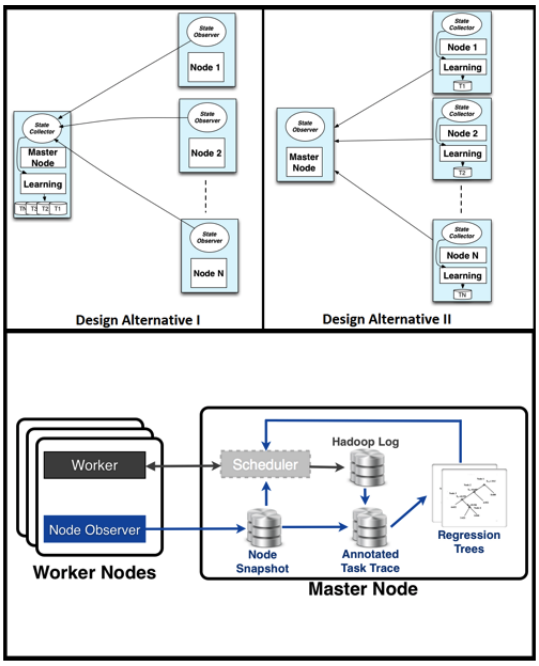

Figure 3: The task of learning regression trees can be performed at master (for smaller clusters) or it could be distributed to the slaves (for huge clusters). Second half of the figure shows the current implementation status.

design alternative I. Currently, we have the infrastructure for collecting logs and processing them ready with us. We also have preliminary implementation of modified scheduler that verifies feasibility of our approach in terms of overheads incurred.

4. FACEBOOK TRACE ANALYSIS

We first analyzes a trace generated in two days on Facebook's Hadoop cluster. The trace shows that there are 5578 nodes.It notes information about ∼27,000 jobs, and total of around 5.2 million tasks during the two days.

4.1 Per Job Analysis

Initially, we analyzed job-level statistics to see their impact on the task-execution times. Table 1 presents the list of such features available in the Facebook trace. Figure 4 shows the Pearson's correlation coefficient for each individual feature extracted from the trace. The plot shows the average correlation coefficient of each feature with the task execution time over all the 27000 jobs. The error bars indicate the standard deviation across jobs. Note that most of the features show very low correlation (less the 0.4) and very few which have a correlation greater than that, show a high value of standard deviation indicating highly variable correlations. This means that **simple correlations do not exist**

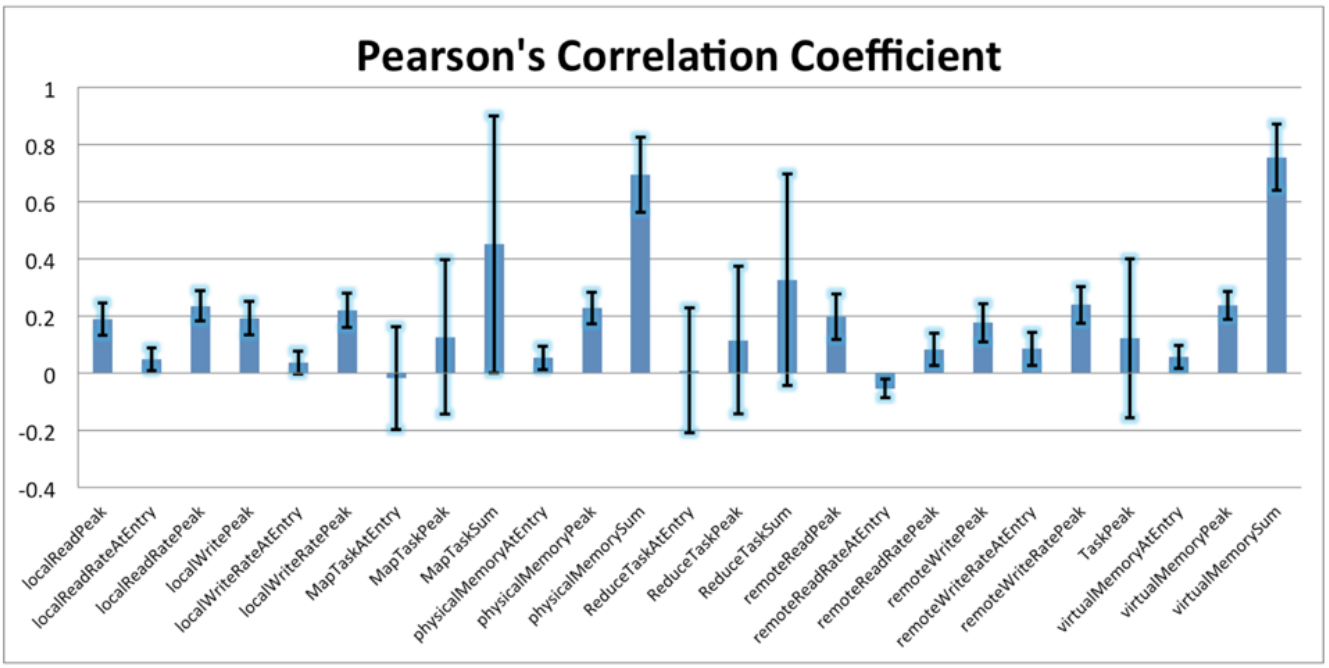

Figure 4: Pearson's correlation coefficient: The value of the correlation coefficient between various features and the task execution time. The columns plot the average value of the Pearson's correlation coefficients and the error bars indicate the standard deviation in the coefficients over approximately 27000 jobs.

Table 1: Features Directly Available from the Facebook Trace

Feature	Description
Task Id	Identifier of task.
Job Id	Corresponding Job Identifier.
Node ID	Corresponding Node Identifier.
Type	Map/Reduce
Status	Success/Failure
Start time	Starting time of the task
End time	Finishing time of the task
CPU time	Total cpu time consumed by the task
Physical Memory	Peak physical memory consumed by the task
Virtual Memory	Peak virtual memory consumed by the task
Local Read	Amount of data read from the local disk
Local Write	Amount of data written to the local disk
Remote Read	Amount of data read from HDFS cluster
Remote Write	Amount of data written to HDFS cluster

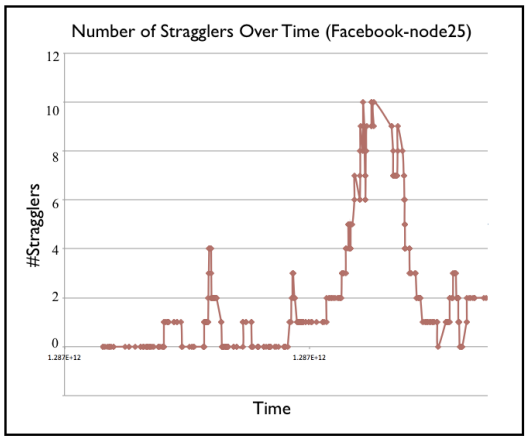

Figure 5: # of stragglers over time on a FB node.

and **job-level statistics do not matter**. Hence, we decided to focus more on the node-level statistics which is described in the following section.

4.2 Per Node Analysis

We observed from the Facebook trace that at some points in time, most of the tasks execute slowly as an effect of overall load on that node. Figure 5 shows such an example from Facebook trace. Limiting or restricting task assignment to such nodes (admission control), or preventing nodes from entering such situation will reduce straggler tasks. In order to confirm our conjecture that such situation arise due to contention over resources, we decided to seek correlation between the node status and execution times of the tasks.

Inferring Node Status Over Time

Since the original Facebook trace had per task information, we processed it further to find out the status of the node interim s of number of simultaneously executing tasks, total physical, virtual memory usage during the execution of each task. Please see Table 2 for the

Table 2: Features Inferred from the Facebook Trace

Category	Feature	Description
AtEntry(E)	CpuUsageAtEntry*	CPU usage at the moment the snapshot was taken.
	#MapTaskAtEntry	Number of Map tasks already executing at the moment the snapshot was taken.
	#ReduceTaskAtEntry	Number of Map tasks already executing at the moment the snapshot was taken.
	physicalMemoryAtEntry	Physical memory utilization at the moment the snapshot was taken.
	virtualMemoryAtEntry	Virtual memory utilization at the moment the snapshot was taken.
	localReadRateAtEntry	Rate of local data read
	localWriteRateAtEntry	Rate of local data written
	remoteReadRateAtEntry	Rate of remote data read
	remoteWriteRateAtEntry	Rate of remote data written
Peak)P	#MapTaskPeak	Max number of map tasks executing simultaneously at any point in time during considered task's execution.
	#ReduceTaskPeak	Max number of reduce tasks executing simultaneously at any point in time during considered task's execution.
	#TaskPeak	Max number of total tasks executing simultaneously at any point in time during considered task's execution.
	physicalMemoryPeak	Max physical memory utilization at any point in time during considered task's execution.
	virtualMemoryPeak	Max virtual memory utilization at any point in time during considered task's execution.
	localReadPeak	Max number of bytes read locally at any point in time during considered task's execution.
	localWritePeak	Max number of bytes written locally at any point in time during considered task's execution.
	remoteReadPeak	Max number of bytes read form remote nodes at any point in time during considered task's execution.
	remoteWritePeak	Max number of bytes written to remote nodes at any point in time during considered task's execution.
	localReadRatePeak	Maximum local read rate during task execution time window
	localWriteRatePeak	Maximum local write rate during task execution time window
	remoteReadRatePeak	Maximum remote read rate during task execution time window
	remoteWriteRatePeak	Maximum remote read rate during task execution time window
Sum(S)	#MapTaskSum	Total number of map tasks executing simultaneously.
	#ReduceTaskSum	Number of reduce tasks executing simultaneously.
	physicalMemorySum	Physical memory utilization during the task's execution.
	virtualMemorySum	Virtual memory utilization during the task's execution
RecentPeak(R)	CpuUsageRecentPeak*	Max CPU usage during past 50 seconds.
	#MapTaskRecentPeak*	Max number of Rap task during past 50 seconds.
	#ReduceTaskRecentPeak*	Max number of Reduce task during past 50 seconds.
	physicalMemoryRecentPeak*	Max physical memory utilization during past 50 seconds.
	virtualMemoryRecentPeak*	Max virtual memory utilization during past 50 seconds.
	localReadRateRecentPeak*	Max local read rate during past 50 seconds.
	localWriteRateRecentPeak*	Max local write rate during past 50 seconds.
	remoteReadRateRecentPeak*	Max remote read rate during past 50 seconds.
	remoteWriteRateRecentPeak*	Max remote write rate during past 50 seconds.

Feature with * mark are available only if NMON log are provided

Table 3: Evaluating Regression on Facebook Trace

Method	MAE.	RMSE.	Corr.
Linear Regression	0.014	0.0258	0.837
ν SVR (RBF Kernel)	0.011	0.030	0.849
ν SVR (Polynomial Kernel)	0.013	0.039	0.731
GP Regression	0.012	0.0346	0.663

complete list of features we extracted from the trace. Note that these represent peak and aggregate (sum) kind of features which is in fact an over-approximation of the situation.

Evaluation results on the Facebook Trace We evaluated different regression formulations with duration as the output real-valued variable. We evaluated linear regression, SVR (Regression using Support Vector Machines) and regression using Gaussian Processes. We also used different kernels such as RBF (Radial Basis Function) kernel and Polynomial kernel. Table 3 shows the results.

Though the correlation between predicted and actual task execution times was high, this analysis resulted in a high relative error. We believed that this is due to overly approximated features in the log that only specifies information at the JVM-level. We do not have more architectural details about the nodes in the Facebook cluster. Hence, we decided to systematically evaluate regression techniques on carefully chosen set of features on a local cluster to have more control to fetch finer details of the nodes in the cluster.

5. LOCAL CLUSTER ANALYSIS

In this section we describe the experimental set up for local cluster analysis along with the results. Our aim is to confirm the existence of correlations between node-status and the task execution times.

5.1 Experimental Setup

We used Berkeley EECS department's Hadoop cluster (icluster) which comprises of 11 nodes, each with 8 CPUs and 16GB RAM. All nodes have their own local disk and are also connected to a networked storage.

Data Generation Data defining he status of a node is collected using NMON system monitoring tool [17]. NMON periodically reads system status from \proc filesystem and records in a flat file. The information includes node's resource utilization such as CPU usage, memory usage, and I/O rate. We installed NMON on all the slave nodes and collected sampled status of each node every 10 second. We then process these logs generated by NMON and saved them on the shared networked storage. We extracted task level information from Hadoop logs. Hadoop maintains various category of logs, such as scheduler log, task-tracker log, and job history log, to help cluster maintenance. Using these, we were able to collect task information from job history logs. Finally, we combined the task level information with node-status information. This presents richer data as we have utilized NMON for the actual system counter in addition to the Hadoop logs.

Benchmark Workloads We implemented a randomized workload simulator in order to avoid a tightly coupling of results with a specific benchmark. As an input, our simulator takes running time, number of users, and range of input file sizes. In the simulation, each user submits a single job to actual Hadoop cluster, wait until

the job is finished and submits the next job after a wait for a random time interval. The benchmark workloads we used are from the Hadoop's example workloads:

- **Sort**: A program that sorts the data that is generated randomly. This represents an I/O bound (Rad-Write intensive) set of jobs.
- **WordCount**: A program that counts the number of words in the input files. Input files are created by invoking **TeraGen**, a random file generator job. This set represents Read intensive jobs.
- **MRBench**: MRBench represents jobs composed of lots of simple tasks. In our simulated workload, a single MRBench job forks 50 to 100 child tasks.
- **PI**: A program that estimates pi(π) using Monte-Carlo estimation method. This set represents a CPU-intensive or compute-heavy jobs.

We ran a simulation experiment for 3 hours with 11 simulated users and input file sizes ranging from 10M to 200M. (Workload running time was limited due to limited storage quota: only 800M of network storage space is allowed to student account.)

Table 4: Regression Result for Local Cluster

		Map			Reduce		
FeatureSet	Method	RAE	RRSE	Corr	RAE	RRSE	Corr
R	LR	0.24	4558.63	0.24	93.60	99.59	0.17
	GP	95.89	96.35	0.24	91.48	95.93	0.32
E	LR	98.09	96.57	0.26	101.35	127.01	0.36
	GP				76.58	81.07	0.60
S	LR	75.35	81.25	0.57	**23.59**	**26.91**	**0.86**
	GP	72.81	78.78	0.61	**29.76**	**32.85**	**0.85**
P	LR	70.15	74.00	0.67	56.53	67.92	0.72
	GP	64.07	68.79	0.72	46.92	52.47	0.79
R+E	LR	93.51	94.58	0.33	101.04	124.08	0.33
	GP	89.29	91.34	0.41	76.26	80.68	0.62
R+S	LR	69.70	75.37	0.65	31.58	37.27	0.81
	GP	65.16	72.00	0.69	38.82	42.84	0.80
R+P	LR	69.82	73.83	0.67	57.32	68.19	0.71
	GP	64.07	68.78	0.72	48.10	53.52	0.78
E+S	LR	68.27	73.52	0.67	45.07	54.29	0.77
	GP	63.06	70.01	0.72	55.08	59.30	0.80
E+P	LR	69.61	73.51	0.68	70.30	82.11	0.55
	GP	63.14	67.80	0.74	61.38	65.16	0.77
S+P	LR	57.18	61.18	0.79	29.97	36.50	0.82
	GP	46.73	52.31	0.85	42.52	46.80	0.82
R+E+S	LR	67.31	72.90	0.68	173.94	373.11	0.60
	GP	62.64	69.88	0.72	57.42	61.41	0.79
R+E+P	LR	69.49	73.49	0.68	174.63	272.21	0.48
	GP	63.13	67.79	0.74	61.94	65.60	0.76
R+S+P	LR	56.90	60.99	0.79	42.34	46.94	0.64
	GP	46.71	52.29	0.85	44.05	48.32	0.82
E+S+P	LR	**55.73**	**59.56**	**0.80**	119.24	232.70	0.65
	GP	**45.48**	**51.07**	**0.86**	59.05	62.49	0.80
All	LR	**54.77**	**57.54**	**0.82**	24.16	29.52	0.62
	GP	**42.08**	**45.15**	**0.90**	58.74	64.86	0.51

5.2 Regression

Table 4 shows evaluation of task execution time prediction using regression on the local cluster (icluster) trace. The first column shows used feature sets. E, R, P, S are acronyms for *AtEntry*, *RecentPeak*, *Peak*, and *Sum*. Please refer Table 2 for brief description of each feature set. TO carefully figure out the correlations between the distinct set of features (E, R, P, S), we planned set of experiments where we tried all the combination of the four feature sets. The second column shows the regression algorithms we used. LR denotes Linear Regression and GP denotes Gaussian Process. Rest of the columns show relative absolute error, root relative squared error, and correlation between features and task execution time for Map and Reduce tasks respectively.

Table 5: Regression Tree Evaluation on iCluster Trace

	Map			Reduce		
FeatureSet	RAE	RRSE	Corr	RAE	RRSE	Corr
S	69.59	76.09	0.64	20.27	23.59	0.87
E+S+P	42.33	48.78	0.87	26.04	32.94	0.86
All	42.32	48.8	0.87	21.97	29.25	0.9

Discussion We observe from these results that *Sum* and *Peak* are significantly correlated to task execution time. On the other hand, *AtEntry* set of features do not show any strong correlation. *RecentPeak* set also shows negligible correlations. In fact, *RecentPeak* degrades the prediction accuracy! Note that the set of *Sum* features provide the highest accuracy especially for reduce tasks. On the contrary, as we expected, for map tasks, the accuracy i higher when most of the features are used. In case of map tasks, even the set of *RecentPeak* features contribute to an improved prediction accuracy. We conjecture that this is due to the large variance in completion times of the same map task: the *Sum* feature set represent overall resource utilization of a node.Since reduce tasks often run for longer duration, at least longer than map tasks, they are more sensitive to overall node resource utilization rather then temporary statistics such as represented by *Peak* and AtEntry. Similarly, we observed that the variance in the completion time for reducers was small as compared to that of mappers, this also explains a better accuracy in predictions for reducers. Further, we believe that categorizing map tasks based on high-level job characteristic will improve accuracy of execution-time prediction for reduce tasks. We plan to explore this further.

Further, we used regression tree [15] to build interpretable rules to guide scheduler. Figure 8 shows examples of the regression trees built. Table 5 shows the results of using CART (Classification and Regression Tree) algorithm on the trace from the local cluster. For brevity, we only show the results using selected features as indicated by our features selection mechanism presented in this section above; i.e. we present results using S, E+S+P and All features (as these are the particular sets of features with the best accuracies of all the sets). Figures 6, 7 show results in pictorial manner. Note that the false positives are almost none meaning that we get reliable straggler predictions.

6. IMPLEMENTATION

We implemented preliminary version of modified sched-

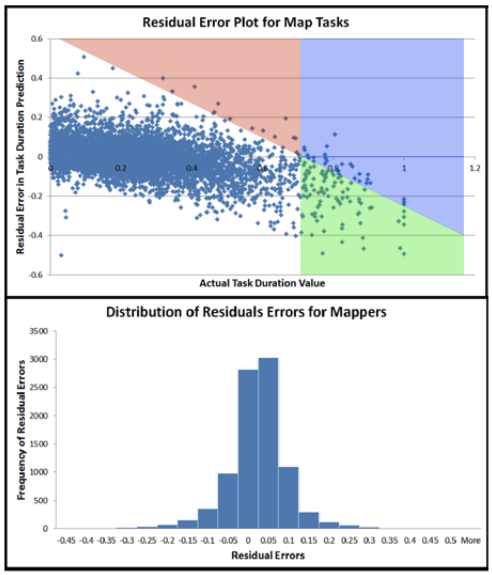

Figure 6: Results on icluster trace.

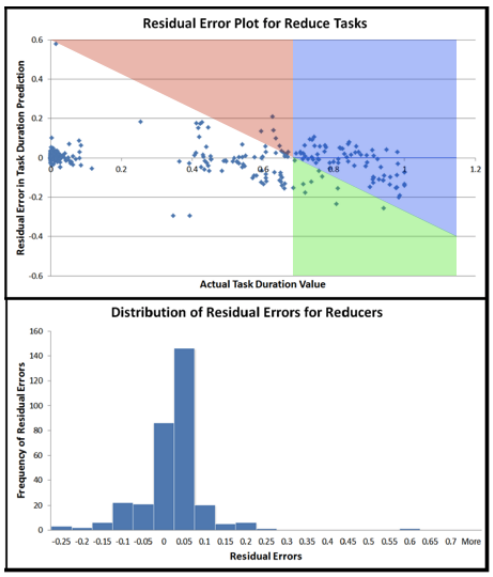

Figure 7: Results on icluster trace.

uler to verify feasibility of the proposed design. Implementation mainly has three components: Scheduler, Regression Tree Builder, and Data Collector. The critical aim of implementation is to minimize scheduling overhead. **Scheduler** Hadoop's fair-scheduler invokes *LoadManager* to check whether the upcoming task assignment could overload the target node before actually assigning the task. We modified *LoadManager* module to invoke additional check procedure. Algorithm 1 shows algorithm of the additional check procedure we implemented.

Algorithm 1 SchedulerAdviser Algorithm

```
function ADVISER(trees, snapshot, threshold, node, type)
    if outdated(trees) then
        trees ← reload()
    end if
    e ← estimate(trees[node][type], snapshot)
    return e ≤ threshold
end function
```

Regression Tree Builder Regression Tree Builder is comprises of two daemon processes. The first one periodically reconstructs regression trees from newly collected Hadoop logs and node's snapshots using Weka[**?**] Machine Learning library. The second daemon optimizes the process by walking down the tree using only a partial information of current node-status in order to get an early estimates of task execution time.

Data Collector Processed Hadoop logs and node snapshots are collected at the shared networked storage, as described in Section 5. Local log processing greatly reduces both network storage space consumption and the communication overhead.

7. EVALUATING FEASIBILITY OF DESIGN

We configured the scheduler to rebuild trees every second and executed simulated workload for 2 hours to see the amount of overhead our scheduler modification introduces. Following table shows brief summary of the results. Column *Sum* denotes total amount of time spent inside scheduler's code and *Average* denotes average time spent per scheduler function call.

	#Scheduler Call	Sum(ms)	Average(ms)
Original	23896	12458	0.52
Modified	22572	12880	0.57(+0.05)
Modified2	23023	16149	0.70(+0.18)

Row *Original* shows result without modification and row *Modified* shows result with modification. Row *Modified2* shows another experimental result where we force the scheduler to reload multiple trees to simulate the effect of huge number of nodes in the cluster. We see that in both the cases minimal overhead is incurred.

8. CONCLUSIONS

We revisited the fundamental assumption behind the existing straggler mitigation techniques that rely on detecting and executing slow running tasks speculatively. Machine Learning proves to be helpful in automatically finding out complex correlations between node/task characteristics and the task-completion times.

1. Simple correlations do not exist. We did not observe any correlation between individual task/node characteristics with the task-completion times. It agrees with previous observations by [3, 4].

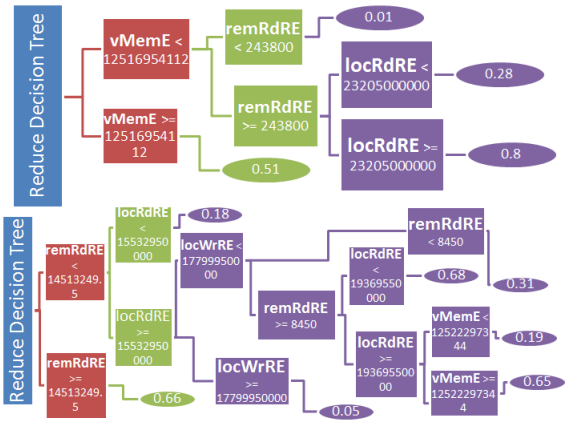

Figure 8: Sample Reduce Tree.

2. We need to consider task characteristics and node status together for better predictions.
3. We believe that considering higher level job clustering will improve prediction accuracy especially for map tasks.
   - In the current approach we are combining all the jobs on a node together without any distinction with respect to their inherent high level characteristics such as CPU-bound vs. memory-bound vs. IO-bound etc.
   - We plan to analysis jobs according to their above mentioned higher level characteristics as future work.

This Machine Learning approach enables self-adaptive cluster scheduling that is independent of node and workload characteristics.

9. FUTURE DIRECTIONS

Through this work, we have taken initial step towards identifying proactive ways for straggler prevention. Though initial results are only fairly good, the direction looks promising. On the data we generated in a controlled environment, we got encouraging results which imply that instrumenting Hadoop logging mechanism to extract more information could be very helpful. At the same time, we also point out that errors in predictions could harm job performance and cluster throughput badly. Hence, we plan to work on *feature selection* further. Importantly, since prediction errors could lead to large performance degradation, it is highly desirable to attach a *confidence measure* along with our predictions of task completion times.

Learning to predict task execution times is highly dependent on the cluster configuration, status of nodes, their age and workload-characteristics. Hence, it is required to update models learnt with new information as it becomes available. At the same time, the models need to be available quickly. This calls for online learning algorithms. We plan to look into it in order to build accurate models more efficiently.

10. REFERENCES

[1] Jeffrey Dean and Sanjay Ghemawat "MapReduce: Simplified Data Processing on Large Clusters," OSDI 2004

[2] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, Ion Stoica"Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling," EuroSys 2010

[3] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, Ion Stoica "Improving MapReduce Performance in Heterogeneous Environments," OSDI 2008

[4] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg Ion Stoica, Yi Lu Bikas Saha, Edward Harris "Reining in the Outliers in Map-Reduce Clusters using Mantri," OSDI 2010

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

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

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

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

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

[10] A. Thusoo, S. Anthony, N. Jain, R. Murthy, Z. Shao, D. Borthakur, J. S. Serma, and H.Liu "Data Warehousing and Analytics Infrastructure at Facebook", SIGMOD 2010

[11] D. Borthakur, J. S. Serma, J. Gray, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, R. Schmidt, and A. Aiyer "Apache Hadoop Goes Realtime at Facebook", SIGMOD 2011

[12] David J.C. Mackay, "Introduciton to Gaussian Processes"

[13] Joaquin Quiñonero-Candela, Carl Edward Rasmussen, "A Unifying View of Sparse Approximate Gaussian Process Regression", Journal of Machine Learning Research 2005

[14] Chirstopher K. I. Williams, "Regression with Gaussian Processes", July 1995

[15] L. Breiman, J. Friedman, CJ. Stone, and RA. Olshen, "Classification and regression trees", Wadsworth, Belmont, 1984

[16] Weka: 'http://www.cs.waikato.ac.nz/ml/weka/'

[17] NMON: 'http://nmon.sourceforge.net/pmwiki.php'

[18] M. Curtis-Maury, A. Snah, F. Blagojevic, DS. Nikolopoulos, BR. de Supiniski, and M. Schulz, "Prediction Models for Multi-dimensional Power-Performance Optimization on Many Cores", PACT 2008

[19] T. Moseley, JL. Kihm, DA. Connors, and D. Grunwald, "Methods for Modeling Resource Contention on Simultaneous Multithreading Processors", ICCD 2005

[20] BC. Lee, J. Collins, H. Wang, and D. Brooks, "CPR: Composable Performance Regression for Scalable Multiprocessor Models" MICRO 2008

[21] L. Tang, J. Mars, and ML. Soffa, "Compiling For Niceness:Mitigating Contention for QoS in Warehouse Scale Computers" CGO 2012

[22] Scribe: "https://github.com/facebook/scribe"