Wrangler: Predictable and Faster Jobs using Fewer Resources

Neeraja J. Yadwadkar
University of California, Berkeley
neerajay@eecs.berkeley.edu

Ganesh Ananthanarayanan
Microsoft Research
ga@microsoft.com

Randy Katz
University of California, Berkeley
randy@cs.berkeley.edu

Abstract

Straggler tasks continue to be a major hurdle in achieving faster completion of data intensive applications running on modern data-processing frameworks. Existing straggler mitigation techniques are inefficient due to their *reactive* and *replicative* nature – they rely on a wait-speculate-re-execute mechanism, thus leading to delayed straggler detection and inefficient resource utilization. Existing proactive techniques also over-utilize resources due to replication. Existing modeling-based approaches are hard to rely on for production-level adoption due to modeling errors. We present *Wrangler*, a system that *proactively* avoids situations that cause stragglers. *Wrangler* automatically learns to predict such situations using a statistical learning technique based on cluster resource utilization counters. Furthermore, *Wrangler* introduces a notion of a *confidence measure* with these predictions to overcome the modeling error problems; this confidence measure is then exploited to achieve a *reliable* task scheduling. In particular, by using these predictions to balance delay in task scheduling against the potential for idling of resources, *Wrangler* achieves a speed up in the overall job completion time. For production-level workloads from Facebook and Cloudera's customers, *Wrangler* improves the 99^{th} percentile job completion time by up to 61% as compared to speculative execution, a widely used straggler mitigation technique. Moreover, *Wrangler* achieves this speed-up while significantly improving the resource consumption (by up to 55%).

1. Introduction

Reducing the job completion time for data intensive applications running atop distributed processing frameworks [1, 2] has attracted a lot of attention recently [3–6]. A major challenge to achieving near-ideal job completion time is the slow running or *straggler* tasks – a recent study [6] shows that despite existing mitigation techniques, straggler tasks can be 6-8× slower than the median task in job on a production cluster. This leads to high job completion time, over-utilization of resources and increased user costs. Mitigating or even eliminating stragglers thus remains an important problem.

Existing approaches, whether based on replication or modeling, aren't enough to solve this problem. *Speculative execution* [1] is a replication-based reactive straggler mitigation technique that spawns redundant copies of the slow-running tasks, hoping a copy will reach completion before the original. This is the most prominently used technique today, including production clusters at Facebook and Microsoft Bing [5]. However, without any additional information, such reactive techniques can not differentiate between nodes that are inherently slow and nodes that are temporarily overloaded [7]. In the latter case, such techniques lead to unnecessary over-utilization of resources without necessarily improving the job completion times. Though proactive, Dolly [6] is still a replication-based approach that focusses only on interactive jobs and incurs extra resources. Being agnostic to the correlations between stragglers and nodes' status, replication-based approaches are wasteful.

Another proactive alternative is to model running tasks statistically to predict stragglers. However, realistic modeling of tasks in cluster environments is difficult due to complex and unpredictable interactions of various modules [5–11]. Black box approaches can learn these interactions automatically [12, 13]; however, these techniques are opaque and prone to errors that could lead to inefficient utilization and longer completion times [6].

To avoid such problems, a straggler mitigation approach should meet the following requirements:

- It should not wait until the tasks are already straggling.
- It should not waste resources for mitigating stragglers.

To this end, we introduce *Wrangler*, a system that predicts stragglers using an interpretable linear modeling technique based on cluster resource usage counters and uses these predictions to inform scheduling decisions. This allows it to avoid waiting until tasks are already running slow. *Wrangler* prevents wastage of resources by removing the need for

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SoCC '14, 3-5 Nov. 2014, Seattle, Washington, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3252-1/14/11…$15.00.
http://dx.doi.org/10.1145/2670979.2671005

replicating tasks. Furthermore, *Wrangler* introduces a notion of a *confidence measure* with these predictions to overcome the modeling error problems; this confidence measure is then exploited to achieve a *reliable* task scheduling. In particular, by using these predictions to balance delay in task scheduling against the potential for idling of resources, *Wrangler* achieves a speed up in the overall job completion time. A threshold on the confidence measure can be tuned to suit different workloads to avoid costly incorrect scheduling decisions. A prototype implementation of *Wrangler* demonstrates up to 61% improvement in overall job completion times while reducing the resource consumption by up to 55% for production-level workloads using a 50 node EC2 cluster.

2. Motivation

Existing approaches to straggler mitigation fall short in multiple ways: Reactive approaches act after tasks have already slowed down. Replication-based approaches, whether proactive or reactive, use extra resources. White or gray box approaches depend on causes that keep changing dynamically and hence are difficult to enumerate *a priori*. Finally, black box approaches are prone to modeling errors. We discuss each of these in detail below.

Reactive techniques rely on a *wait-and-speculate* re-execute mechanism. Speculative execution, a widely used approach for straggler mitigation, marks slow running tasks as stragglers and reacts by relaunching multiple copies of them. This is inefficient because a task is allowed to run for a significant amount of time before it can be identified as a straggler. By this time, other tasks of that job have made considerable progress already, increasing the possibility of extending the job's finishing time. SkewTune [14] avoids replicating tasks but is still a wait-and-speculate mechanism.

Replication-based approaches incur extra resources. For instance, speculative execution launches multiple redundant copies of the same task. As soon as one of these copies finishes execution, the rest are killed. This amounts to wastage of resources. LATE [4] improves over speculative execution using a notion of progress scores, but still results in resource waste. Cloning mechanisms [6], being replication-based, also incur extra resources.

Though there are no existing mechanisms that proactively avoid stragglers without replication, scheduling or load-balancing approaches [15–20] indirectly attempt to do so by reducing resource contention on a cluster. The delay scheduler [16], for example, focuses on reducing network bottlenecks. However, we show in Section 4.2.2 that despite these efforts, stragglers still occur. Being focused on load balancing in heterogeneous clusters, [18] ignores temporary overloading that occurs even in homogeneous clusters.

White or gray box approaches, whether reactive [3] or scheduling-based [15–17], attempt to identify stragglers based on whether local conditions exceed fixed thresholds defined over a set of resource usage statistics. However, in many cases, given resource usage statistic values sometimes result in a straggler, and other times do not. Further, In Section 4.2.2, we illustrate how *the contributors of straggler behavior vary across nodes and over time* in the Facebook and Cloudera real-world production traces. For example, we observed that two disk-intensive tasks scheduled on a node with slow disk were found to be straggling. The same tasks co-located on another node, however executed normally, as the other node had enough disk bandwidth. But when we avoided scheduling such tasks simultaneously on the node with slow disk, the tasks straggled due to other resource contention patterns, including network and memory. Even if we could easily distinguish conditions that always result in stragglers, it is infeasible to exhaust the space of all possible resource usage statistic settings to classify them as such. This limits the usefulness of white or gray box approaches.

An alternative is to use black box approaches to automatically learn a node's behavior. Previous work [12] has shown that high-quality predictions of stragglers can be made, although it did not attempt to incorporate such predictions into a scheduler. Incorporating such techniques into a scheduler could backfire during real-life deployment on production clusters [6]. To make these black box techniques more transparent, it is critical to be able to assess what the technique is learning from the data. Further, modeling errors might render the system's performance unstable. It is crucial to be robust against such errors for achieving predictable performance. Since off-the-shelf learning techniques do not include mechanisms for evaluating the quality of their output, these are not sufficient to tackle this problem. Despite these challenges, there is evidence to believe that machine learning techniques can be successfully incorporated into production systems: previous work [13, 21] have demonstrated the use of machine learning for selecting what resources to assign to a task. However, they did not perform straggler mitigation.

To address each of the issues identified above, Wrangler provides:

1. Interpretable models that can predict what conditions will lead to stragglers using readily available performance counters. These models automatically adapt to the dynamically changing resource usage patterns across nodes and across time. This capability of predicting if a node could cause a straggler opens up various avenues for avoiding stragglers.
2. A confidence measure that guards against modeling errors by telling us when the models are sure of their predictions. Modeling errors fundamentally limit real life applicability of previously proposed approaches [12, 13]. *Wrangler* addresses this by providing a configurable confidence measure. We show in Section 7.4 that the use of the confidence measure is crucial.
3. A task scheduling mechanism that incorporates these predictions to improve overall job finishing times. This

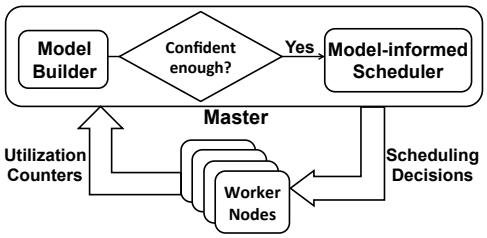


Figure 1: Architecture of *Wrangler*.

reduces resource consumption compared to replication-based mechanisms.

Although it serves as a straggler avoidance approach on its own, *Wrangler* can also be used in conjunction with existing mitigation approaches.

3. Our Proposal: *Wrangler*

Given resource utilization metrics of a node, *Wrangler* predicts if a newly assigned task on that node will turn out to be a straggler. It then uses these predictions to make scheduling decisions that proactively avoid straggler formation.

3.1 Architecture of *Wrangler*

Figure 1 shows *Wrangler's* system architecture, which extends Hadoop. Job scheduling in Hadoop is handled by a master, which controls the workers. The master assigns tasks to the worker nodes in response to the heartbeat message sent by them every few seconds. The assignments depend upon the number of available slots as well as locality. *Wrangler* has two basic components.

1. **Model Builder:** Using the job logs and snapshot of resource usage counters collected regularly from the worker nodes using a Ganglia-based [22] node-monitor, we build a model per node. These models predict if a task will straggle given its execution environment; they also attach a confidence measure to each of their predictions. Section 4 describes the Model builder in detail.
2. **Model-informed Scheduler:** Using the predictions from the models built earlier, a model-informed scheduler then selectively delays the start of task execution if that node is predicted to create a straggler. A task is delayed only if the confidence in the corresponding prediction exceeds the minimum required confidence. This avoids overloading of nodes, thus reducing their chances of creating stragglers. Section 5 details the Model-informed scheduler.

Our main tool is to defer a task's assignment until a node, that is likely to finish it in a timely manner, is found. Figure 2 shows an example job with three mappers and a reducer. Without *Wrangler*, Map_2 was a straggler, as its *normalized duration*, the ratio of its duration to the size of its input data, was much larger than the other mappers. When *Wrangler* predicted it to be a straggler on this node,

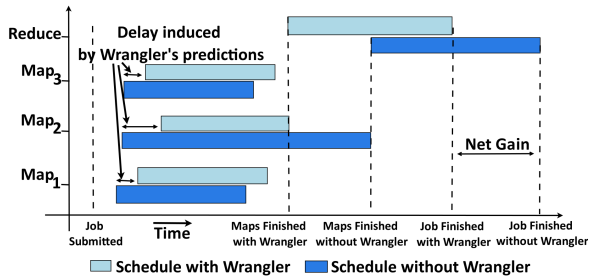
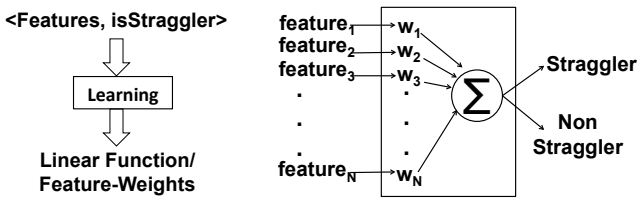


Figure 2: Lifetime of an example job observed with and without *Wrangler*. Note: there are other jobs (not shown here) executing simultaneously on the cluster. The careful assignment of Map_2 to a node, that is likely to finish it in a timely manner, accelerates the job completion despite the delay introduced in its launching.

this assignment was avoided. This decision introduced a delay in the task's start until the same node is no longer overcommitted or a different non-overcommitted node is found. Due to this assignment, Map_2 finished faster than it did without *Wrangler*. The reducer then started earlier and the scheduler achieved a net improvement in job duration.

3.2 Novelty of our Approach

Wrangler takes a radically different approach compared to previous straggler mitigation strategies by predicting straggler tasks before they are even launched and scheduling them well to avoid their occurrence in the first place. *Wrangler* achieves its goal by collecting extensive information and deriving useful correlations automatically. According to our observations, what causes stragglers varies across nodes and time. Being a learning-based approach, *Wrangler* is capable of adapting to various situations that cause stragglers. It can figure out for itself what factors are causing tasks to run slower than usual. Importantly, the straggler prediction models we build are interpretable; meaning that we can gain insights from what these models learn using the data (Section 4.2.2). This relieves us from having to explicitly diagnose each case manually, which as we argued earlier is infeasible. Note that even in the presence of more sophisticated schedulers, *Wrangler's* ability of adapting to dynamically changing cluster execution environments and changing resource patterns justifies its applicability. Additionally, the role of a *confidence measure* is crucial for handling modeling errors. This allows our probabilistic learning-based approach to be robust. Learning techniques do not compute confidence measure. Our work is the first to introduce the use of a confidence measure to ensure stability of straggler prediction models. Further, *Wrangler*'s model-informed scheduler induces delays in launching tasks on nodes that are predicted to create stragglers. In the worst case, due to possible prediction errors, our approach could lead to bounded sub-optimal performance. Our approach however, does not lead to incorrect execution, termination or replacement of tasks; thus maintaining liveness and correctness guarantees.



(a) Learning phase (b) Prediction phase

Figure 3: Linear modeling techniques for predicting stragglers: node's resource usage counters form the features. The Learning phase (a) learns the weights on features using the ground truth, i.e., a labeled dataset that consists of (1) features of a task and (2) whether it was an straggler or not. A linear model then predicts based on a score obtained by taking a linear combination of features with corresponding weights (b).

4. Building Straggler Prediction Models

The aim of *Wrangler's model builder* component is to build straggler prediction models such that they are *robust* with respect to possible modeling errors and are *interpretable* with respect to what they learn.

4.1 Linear Modeling for Predicting Stragglers

As we mentioned earlier, finding what actually causes stragglers is challenging due to complex task-to-node and task-to-task interactions. To capture these complex interactions, we collected numerous resource usage counters from the cluster using Ganglia [22]. Linear modeling techniques [23] from the machine learning domain are appropriate for probabilistic modeling of a node's behavior, which can be represented through the various resource usage counters. These techniques adapt to dynamically changing resource usage patterns on a node. This alleviates the pains of manual diagnosis of the source of individual straggler appearance. We learn the behavior of each node individually to be robust to heterogeneity in today's clusters.

As shown in Figure 3(a), during the learning phase, these techniques learn *weights* on the features using labeled data that represents the ground truth. In this context this data is the node's resource usage counters at the time of submission of a task and a label (isStraggler), indicating whether it was a straggler. Using these weights and the node's resource usage counters the model calculates a score for predicting if it will turn out to be a straggler. This prediction phase is depicted in Figure 3(b). Next, we provide a high-level intuitive understanding of one such linear modeling technique that we use, Support Vector Machines (SVM) with linear kernels. For mathematical details, see [24, 25].[1]

Support Vector Machines for Predicting Stragglers: SVM is a statistical tool that learns a linear function separating a given set of vectors (e.g., node's resource usage counters) into two classes (e.g., straggler class and non-straggler class). This linear function is called the *separating hyperplane*; each of the two half spaces defined by this hyperplane represents a class. In the model building phase, this hyperplane is computed such that it separates the vectors of node's resource usage counters belonging to one class (stragglers) from those of the other class (non-stragglers) with maximum distance (called margin) between them. Later, a new observed resource usage vector (i.e. a test vector) can be evaluated to see which side of the separating hyperplane it lies, along with a score to quantify the confidence in classification based on the distance from the hyperplane.

Features: We used node-level features spanning multiple broad categories as follows:

1. *CPU utilization:* CPU idle time, system and user time and speed of the CPU, etc.
2. *Network utilization:* Number of bytes sent and received, statistics of remote read and write, statistics of RPCs, etc.
3. *Disk utilization:* The local read and write statistics from the datanodes, amount of free space, etc.
4. *Memory utilization:* Amount of virtual, physical memory available, amount of buffer space, cache space, shared memory space available, etc.
5. *System-level features:* Number of threads in different states (waiting, running, terminated, blocked, etc.), memory statistics at the system level

Confidence Measure: Simply predicting a task to be a 'straggler' or a 'non-straggler' is not robust to modeling errors. To ensure reliable predictions, we introduce the notion of *confidence measure* along with the prediction of these linear models. We need a confidence measure to help decide if our predictions are accurate enough for preventing stragglers by influencing the scheduling decisions. The farther a node-counter vector is from the separating hyperplane, higher are the chances of it belonging to the the predicted class. To obtain a probability estimate of the prediction being correct, we can convert the distance from the separating hyperplane to a number in the range [0, 1]. We obtain these probabilities by fitting logistic regression models to this distance [26]. Next, we explain how to compute a confidence measure.

Snapshot of a node's resource usage counters represents the node at a given time instant. We denote this set of features as a vector x. The SVM outputs a linear hyperplane of the form $w^T x + b$, where w is a vector of weights learned by SVM corresponding to the features. Data points which have a positive score, i.e., $w^T x + b > 0$ are classified as stragglers and points which have a negative score are classified as non-stragglers. The SVM doesn't itself output probabilities, but the score ($w^T x + b$), which is also the distance of a point to the hyperplane, serves as a measure of the classifier's confidence. Points far away from the hyperplane (i.e with high positive scores or highly negative scores) are those

[1] Other classification techniques, such as decision trees could also be used, as demonstrated in [12]. We found their performance to be similar.

which the classifier is very confident about. We train a logistic regression classifier to convert this score into probabilities. The logistic regressor outputs a probability of the form $\frac{1}{1+exp(-\alpha s-\beta)}$ where s is the score and α and β are parameters that are estimated using logistic regression [24].

Imbalance in the dataset: Various modeling techniques are sensitive to imbalanced datasets. Non-straggler tasks outnumber the stragglers causing an imbalance in the dataset used for building models. Due to the way underlying optimization problems are formulated, the predictions favor the class with majority of instances. In this context, every task is predicted to be a non-straggler. Ideally, the best results are obtained when each class is represented equally in the learning dataset. We statistically oversample [27–30] the instances from the minority class (i.e. straggler class) which is a common technique for dealing with imbalanced datasets.

Interpretability: Our straggler prediction models are interpretable as they allow us to gain insights from what they learned using the data. These models automatically learn the contribution of a feature towards creation of a straggler, called *weight* of that feature. We bring out the causes behind stragglers using these weights assigned to the features.

For a given task, our models (SVM with linear kernel [24, 25]) predict based on a linear combination of features with their corresponding weights. In other words, their predictions are based on a score obtained by multiplying feature-values with their respective weights and adding these products. We then analyze the resource usage counters of actual straggler tasks. Given the node's resource usage counters at the launch time of such an actually observed straggler task, we want to find out the set of features that primarily caused it to be a straggler using the weights learned by our models. Since it is tedious to capture a holistic picture with over 100 feasible node resource utilization counters involved, we grouped them into five feature categories; CPU utilization, network utilization, disk utilization, memory utilization and other system-level features. We then selected a subset of these features that makes up at least 75% of the models' score for the given task. Since this subset of features has driven the model's decision to predict it to be a straggler, we deem this as the *cause* behind this straggler. We then find out the fraction of straggler tasks on that node that have the same cause. In Section 4.2.2, we present this analysis.

Next, we evaluate the prediction accuracy of SVM on production traces from Facebook and Cloudera. Section 5 describes how we used these predictions with associated confidence for affecting scheduling decisions to avoid stragglers.

4.2 Model-Builder Evaluation

We evaluate the straggler prediction models on real world production-level workloads from Facebook and Cloudera by replaying them on a 50 node EC2 cluster as explained below.

Trace	#Machines	Length	Date	#Jobs
FB2009	600	6 month	2009	1129193
FB2010	3000	1.5 months	2010	1169184
CC_a	100	1 month	2011	5759
CC_b	300	9 days	2011	22974
CC_d	400-500	2+ months	2011	13283
CC_e	100	9 days	2011	10790
Total	$\approx$ 4600	$\approx$ 11.5 months	-	2351183

Table 1: Dataset. *FB*: Facebook, *CC*: Cloudera Customer.

A prediction is correct if it matches with the actual label. We evaluate the models based on (1) how many times they predicted straggler tasks correctly: percentage true positives and (2) how many times they mis-predicted a non-straggler task to be a straggler: percentage false positives. High true positive and low false positive values indicate a good quality model.

4.2.1 Experimental set-up

Production-level Workloads: We learn to predict stragglers based on production level workload traces from multiple Hadoop deployments including those at Facebook and Cloudera's customers. Our dataset covers a wide set of workloads allowing for a better evaluation. Table 1 provides details about these workloads in terms of the number of machines, the length and date of data capture, total number of jobs in those workloads. Chen, et al., explain the data in further details in [31]. Together, the dataset consists of traces from over about 4600 machines captured over almost a year. For faithfully replaying these real-world production traces on our 50 node EC2 cluster, we used a statistical workload replay tool, SWIM [32] that synthesizes a workload with representative job submission rate and patterns, shuffle/input data size and output/shuffle data ratios (see [32] for details of replay methodology). SWIM scales the workload to the number of nodes in the experimental cluster.

Dataset: For building straggler prediction models, we need a labeled dataset that consists of a number of {features, label} pairs. In this context, features are the resource usage counters of a node at the time of submission of a task; and label is whether it was a straggler or not. To generate this dataset, we replayed the production-level traces (see Table 1), using SWIM [32] on Amazon EC2 cluster of 50 m1.xlarge instances. Using Ganglia [22] we captured the node's resource usage counters at regular intervals of 15 seconds; this forms the features in the dataset (see Section 4.1). We label the dataset by marking straggler tasks based on the following definition: Let normalized durations, $nd(t)$ be the ratio of task execution time to the amount of work done (bytes read/written) by task t.

Definition A task t_i of a job J is called a *straggler* if

$$nd(t_i) > \beta \times \underset{\forall t_j \in J}{\text{median}}\{nd(t_j)\} \quad (1)$$

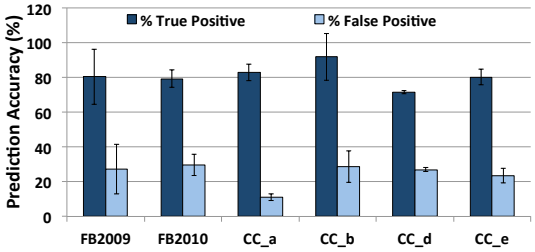


Figure 4: Classification accuracy of SVM for all workloads.

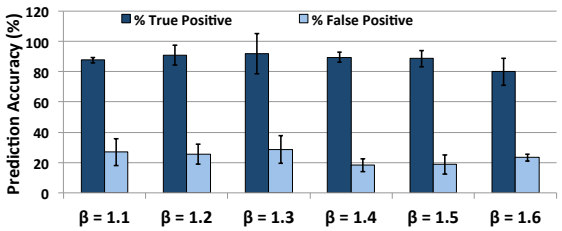


Figure 5: Sensitivity of classification using SVM with respect to values of β for *CC_b*. The error bars show the standard deviation across the percentage accuracy of models built for the 50 nodes in the cluster. Note: The valid range for β shown is $\beta > 1$ ($\beta = 1$ is simply the median) to $\beta < 1.7$ (lower enough to ensure minimum number of stragglers for oversampling and training).

In Section 4.2.2, we show that our models are agnostic to the value of β.

4.2.2 Results of Model-evaluation

Figure 4 shows the straggler prediction accuracy on all the workloads using SVM. We use the tasks executed over initial 2 hours to build models and test their accuracy on the tasks submitted over the next hour. Overall, using a linear kernel SVM with Sequential Minimal Optimization [33], we obtain about 80% true positive and about 30% false positive percentages. This means that we predicted 80% of total stragglers tasks as stragglers and mis-predicted 30% of non-stragglers to be stragglers. This completes the model building phase and then we deploy the models so that they provide hints to *Wrangler's* scheduler. In Section 7, for example, we show that we achieve 61% and 43% improvement in 99^{th} percentile of overall completion times for *FB2009* and *CC_b* respectively. This confirms that about 80% true positive percentage is good enough.

Sensitivity of the models to the definition of stragglers: Based on the value chosen for β, number of stragglers vary (see Definition 1). Intuitively, β indicates the extent to which a task is allowed to slow down before it is called a straggler. Our mechanism is agnostic to the value of β chosen for all the workloads. For brevity, we show a representative sensitivity analysis with respect to β for *CC_b* in Figure 5. We chose β from the range $(1, 1.7)$. $\beta = 1$ is simply the median and we did not see enough stragglers for β greater than 1.6 for CC_b to be able to oversample and build a model. In our experiments, we set β = 1.3.

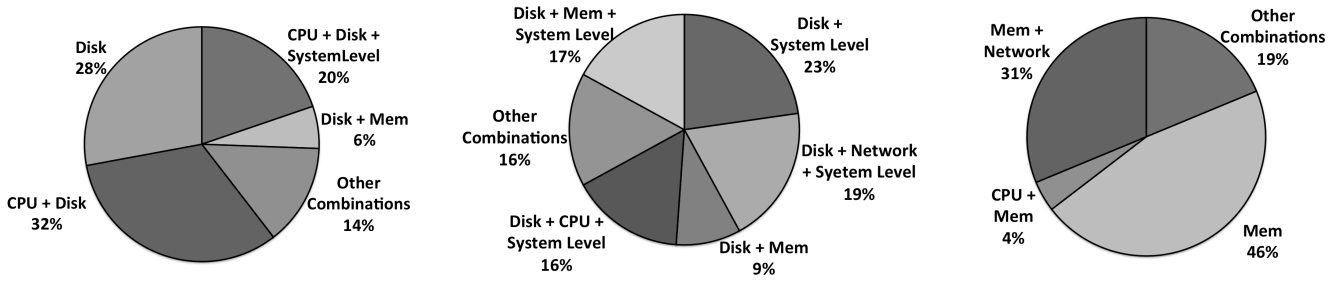
Insights Provided by the Models: We briefly describe the insights we obtained from the straggler prediction models about the causes behind them. However, we leave the detailed explanation of the causes for future work. In Section 7, we show that *Wrangler's* model-informed scheduler accelerates job completion by making careful task-to-node assignments; thus avoids such straggler-causing situations.

Figure 6 presents the percentage of stragglers created due to different causes. Figure 6(a) shows that for *FB2010*, disk utilization (I/O) was the primary bottleneck creating temporary hotspots along with interference with simultaneously executing tasks' memory, CPU usage patterns. Figure 6(b) shows the causes behind stragglers on another node in the cluster executing the same workload (*FB2010*). Although the disk (I/O) usage still dominates, other features also contribute considerably to the creation of stragglers on this node. On a node executing *CC_b*, as shown in Figure 6(c), memory contention contributed the most in creating stragglers.

From our analysis of causes behind stragglers indicated by the models, we note the following:

- Causes behind stragglers vary across nodes – this is true even for the clusters of the same instance types on Amazon EC2. This justifies our approach of building a straggler prediction model per node. This decision also makes our approach robust to heterogeneity.
- Causes vary across workloads – we see that for *FB2010*, the dominating contributor was disk usage whereas the tasks of *CC_b* contend over memory. In [31], Chen et al., explain that *FB2010* is I/O intensive, supporting this insight obtained from our models.
- Network utilization features were not seen to be the prime contributors in our experiments for any of the workloads we evaluated on. We believe network utilization was not the bottleneck as we had enabled the locality-aware delay scheduling mechanism [16]. This indicates that since none of the existing schedulers are straggler-aware, they cannot eliminate stragglers.
- Complex task-to-task interactions on an underlying node tend to create temporary hotspots. Lack of this information can cause scheduling decisions to go wrong.

Note that it is hard to know the contributors to the straggling behavior of tasks apriori without the help from the models. This justifies our use of learning-based models that automatically adapt to various causes behind stragglers. Using the straggler prediction models, we can proactively inform the schedulers of such straggler causing situations. In Section 5, we propose such a scheduler that exploits these predictions to avoid creating stragglers in the first place.



(a) *FB2010, Node*$_1$ (b) *FB2010, Node*$_2$ (c) *CC_b, Node*$_1$

Figure 6: Causes behind stragglers

5. Model-informed Scheduling

In this section, we describe our scheduling algorithm that uses the straggler predictions to selectively delay task assignment to nodes. We then explain the significance of tuning parameters of this algorithm and describe how we learn their values. Finally, we conclude with a theoretical analysis that bounds the delays our algorithm can introduce.

Algorithm 1 Model-informed scheduling algorithm

Let N=$\{n_i : i$=$1,\ldots,$# workers$\}$ be the set of worker nodes
Let $willStraggle_i \in \{$yes, no$\}$ be the prediction using a snapshot of resource usage counters of i^{th} worker node using its model
Let $p \in [0,1]$ be the minimum acceptable confidence of predictions.

```
1:  /* PREDICT runs every Δ interval in background */
2:  procedure PREDICT
3:    for all the workers in N
4:      collect a snapshot of node's resource usage counters
5:      willStraggle_i = prediction if worker n_i will create a straggler
6:      confidence_i = confidence in the above prediction

7:  procedure SCHEDULE
8:    for a task chosen as per the preferred scheduling policy
9:      when heartbeat is received from a worker indicating free slot(s)
10:       if willStraggle_i == yes with confidence_i > p
11:         reject the task from being assigned to n_i
12:       else
13:         proceed as per the configured scheduling policy
```

5.1 *Wrangler's* Scheduling Algorithm

In Hadoop, the master manages job scheduling by assigning tasks to workers in response to *heartbeats* sent by them every few seconds. When the scheduler receives a heartbeat indicating that a map or reduce slot is free, the configured scheduling policy, such as Fair-scheduling, selects tasks to be assigned.

Wrangler's scheduling algorithm proposes to extend any of the existing schedulers. Before launching a task, our model-informed scheduler predicts if a worker will finish it in a timely manner. If the worker is predicted to create a straggler at that time, the task is not assigned to it. When we find a worker that is not predicted to create a straggler, the task is then assigned to it.

Algorithm 1 details this scheduling policy. The *predict* procedure (lines 2-6) is executed in background every Δ time interval to predict if the workers will create stragglers. All the predictions also have a confidence measure (line 6) attached to them. The *schedule* procedure (lines 7-13) is the hook we embed in the default scheduler code. We modified the Fair-scheduler code for our prototype (see Section 6). When a heartbeat is received, our scheduler delays a tasks's assignment to a worker if it is predicted to create a straggler with confidence higher than a configured threshold p in Algorithm 1. Otherwise, we let the default scheduling policy make the assignment decision (lines 12-13). Note that *Wrangler* processes the predictions in background and keeps them ready for the scheduler to use (see Section 6). This allows us to be off the critical path that makes scheduling decisions.

Note that *Wrangler* acts as a system that provides hints to the default/configured scheduler. This means that the decision of which task to launch next is left to the underlying scheduler. *Wrangler* only informs the scheduler whether or not a newly available node is likely to finish a task in timely manner. Inclusion of task-level features could enable further improvements in overall job completion times, we left this extension to future work.

5.2 Learning the Parameters: p and Δ

Algorithm 1 has two tunable parameters: p is the minimum acceptable confidence in predictions needed for them to influence the scheduling decisions and Δ, that decides how frequently a snapshot of node's resource usage counters is collected and predictions are computed using it. Next we explain how we tune these parameters.

5.2.1 Learning p: Threshold on Confidence Measure

Parameter p is the minimum acceptable confidence of predictions. p takes a value in range [0,1]. If p is too low, many tasks will get delayed, adversely affecting job completion and underutilizing resources. On the other hand, if p is too high, many long running tasks will get scheduled. We must set it to balance good resource utilization without increasing the chances of tasks straggling. We observed that the value of p needed for maximum improvement in overall job completion times varies from one workload to another.

Our approach is to learn p automatically during the model building phase so as to avoid the need for manual tuning. As

we explained in Section 4.2.2, we use the data generated by tasks executed over initial 2 hours to build models and test their accuracy on the tasks submitted over the next hour. The set of tasks submitted in this one hour are unseen by the learning algorithm, referred to as a validation set. To decide the value of p, we use the confidence (see Section 4.1) on predictions on tasks in the validation set. For every node, we extracted a range of confidence values producing the most accurate predictions. We set p equal to the median of a range that is agreed upon by a majority of the nodes (forming a quorum). In Section 7.5, we show the sensitivity of *Wrangler* with respect to multiple values of p. In that section, we present the experimental validatation over the next 10 hours that the chosen values for p achieve maximum gain in job completion times for the workloads listed in Table 1. A different p value per node could also be used if desired (we do not evaluate this experimentally in this paper).

5.2.2 Learning Δ: Interval between Predictions

Recall that to be off the critical path of making scheduling decisions, *Wrangler* keeps the predictions ready for the scheduler to use (Section 6 provides implementation details). To ensure that the predictions reflect current state of a node, *Wrangler* regularly collects nodes' resource usage counters and predicts using their respective models if a node can create a straggler at the time. Parameter Δ determines how frequently this background process should be invoked. If Δ is too high, the predictions may not be fresh enough to reflect dynamic changes in the node's status. Extracting predictions out of already built models is of the order of submiliseconds [34]; it is not too expensive. Hence, small Δ is a safe choice. We suggest setting Δ to a smaller value than the minimum inter-task submission times.

The value we chose for our experimental set up was decided based on the time spent collecting the node resource usage counters and predicting based on them. With our current centralized implementation of *Wrangler's* prototype, where the resource usage counters from all the nodes are collected at the master using Ganglia, Δ is set to 15 seconds. On a distributed implementation of *Wrangler* (see Section 7.8), it is feasible to have every node collect its resource usage counters, predict using it and send this prediction to the master along with the heartbeat. This implementation makes Δ independent of the number of nodes in the cluster.

5.3 Bound on Delays Introduced by *Wrangler*

Since *Wrangler* delays tasks that may straggle, it is important to bound such a delay. Wrangler drives the cluster through the following three states (see Figure 7), with respect to the predictions on the constituent nodes:

- N: No node is predicted to create a straggler
- S: Some nodes are predicted to create stragglers and
- A: All the nodes are predicted to create stragglers.

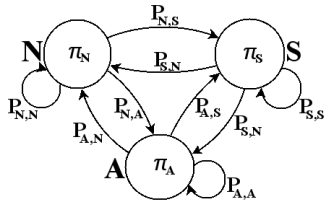


Figure 7: A simplified State-Machine that captures the behavior of a cluster executing parallel data intensive workloads.

For this analysis to be tractable, we make a simplifying assumption that the cluster behavior could be modeled as an irreducible ergodic Markov chain [35, 36] in which the cluster's future state depends on the currently observed state.

Let P be a 3×3 matrix describing the transition probabilities for the states in Figure 7. Let π be a 3×1 vector, $[\pi_N, \pi_S, \pi_A]^T$ comprising the steady state distributions of the 3 states. To ensure that *Wrangler* does not delay tasks indefinitely, we do not want the cluster to end up in state A at steady state. For this, we need to show that π_A is very close to 0. We do this analysis as follows. When the system attains steady state, the transition matrix P has no effect on the vector π. Using the elementary properties of stationary distributions, we have that $\pi P = \pi$ (for details, see [35]). This equation can be solved through eigen-analysis to find the eigenvector (π) of the transition matrix P corresponding to the eigenvalue of 1. Using the log of the state transitions of the cluster executing various workloads, we estimated π and found that π_A was indeed 0 as desired. In Section 7.7 we describe an empirical analysis of the delays induced by *Wrangler*. Here, we provided a theoretical guarantee that a task, as seen in real workloads, will never be delayed indefinitely. One way of bounding delays in case of under-provisioned systems or workloads with a high rate of job submission could be to provide a tunable parameter that limits the number of times a task's assignment gets rejected. We leave this for future work.

6. Implementation

We implemented *Wrangler* (see Figure 1) by embedding it in the Fair scheduler's code. It consists of about 200 lines of code: about 10 lines embedded in Fair scheduler, and the rest for building models using SVM, capturing nodes' resource usage counters and extracting predictions from the models. We use Weka [37] for building SVM and logistic regression models. In our prototype, the Hadoop logs, node's resource usage counters are collected centrally at the master node for further processing and building models. However, *Wrangler* could be implemented in a distributed manner where all the worker nodes collect and process their statistics, build and use their models independently (see Section 7.8).

***Wrangler's* Training and Usage Workflow:** When a workload starts executing on the cluster, *Wrangler* collects the job logs and node-level statistics from all the nodes and pro-

cesses them to generate the dataset to build models. It builds one straggler prediction model for each node in the cluster taking less then a second per node using Weka [37]. In our experiments, we captured training data for about 2 hours[2]. The jobs do not need to wait until the training period is over as the default scheduling policy will be in effect during this time. Once training data was collected, it typically took a few seconds to build a model per node. Once the models are built, *Wrangler's* background process frequently captures the resource usage counters from all worker nodes. It predicts if a new task might run slower if assigned to a particular node using that node's model. *Wrangler* also reports the confidence it has in its prediction. Time taken by each prediction was the order of sub-milliseconds. This prediction for every worker node happens at a regular interval of Δ and is read by the scheduler for making assignment decisions. This way, the prediction process does not come in the critical path of making scheduling decisions.

Models could be updated or re-built regularly once enough jobs have finished execution and new data has been collected. This data collection and model building phase could overlap with job executions. We show the improvements achieved on the workloads from Facebook and Cloudera's customers, by building models only once and testing for the next 10 hours. We leave analyzing the usefulness of the training for continuously changing workloads as well as across different workloads for future work.

7. Evaluation

We demonstrate *Wrangler's* effectiveness by answering the following questions through experimental evaluation:

1. Does *Wrangler* improve job completion times?
2. Does *Wrangler* reduce resource consumption?
3. Is *Wrangler* reliable in presence of modeling errors?
4. How sensitive is *Wrangler* with respect to parameters?
5. *How* does *Wrangler* improve job completion times?
6. What if *Wrangler* mis-predicts?
7. Does *Wrangler* scale well?

7.1 Setup

Real world production workloads: We evaluate using two workloads from Facebook (*FB2009* and *FB2010*) and two from Cloudera (*CC_b* and *CC_e*). We replay the production logs to synthesize representative workloads using SWIM [32] that faithfully reproduces the job submission patterns, data sizes, and data ratios on our 50-node cluster of m1.xlarge instances on Amazon EC2.

Baseline: Although *Wrangler* serves as a straggler avoidance approach on its own, it can also be used in conjunction with existing mitigation approaches to accrue further reduction in job completion times and resources consumed. To show its effectiveness, we compare *Wrangler* against speculative execution, a widely used straggler mitigation technique in Hadoop production clusters.

Metrics: We look at the % reduction in job completion times as our primary metric. Let T_w be the job execution time with *Wrangler* and T be without *Wrangler*, then

$$\%Reduction = \frac{T - T_w}{T} * 100 \quad (2)$$

A positive value indicates reduction in job completion times whereas negative values indicate increase.
Highlights of our results are:

- For Facebook 2009 production Hadoop trace, *Wrangler* improves the overall job completion times by 61% at the 99^{th} percentile and by 20% at the 99.9^{th} percentile over Hadoop's speculative execution.
- For Cloudera customer's production Hadoop trace, *CC_b*, *Wrangler* improves the overall job completion times by 43% at the 99^{th} percentile and by 22% at the 99.9^{th} percentile over speculative execution.
- Being proactive, *Wrangler* consumed 55% and 40% lesser resources for Facebook 2009 and *CC_b* respectively compared to the reactive speculative execution mechanism.

7.2 Does *Wrangler* improve job completion times?

We evaluated the gains on the tasks of previously unseen jobs arriving after the models are built. Figure 8 shows improvement in average, 75^{th} and higher percentile job completion time statistics for the same set of jobs executed with *Wrangler* and without *Wrangler* (i.e., with speculative execution). For *FB2009*, *Wrangler* achieves an improvement of 61% at the 99^{th} percentile and 57% in the average job completion times. For CC_b, we see the improvements of 43% at the 99^{th} percentile and 44% in the average job completion times. Note that, for *CC_e*, *Wrangler* slowed down the 75^{th} to the 90^{th} percentiles. We found that *CC_e* has a bursty job-submission pattern. This means that, in a time period shorter that Δ, many tasks were submitted. However, the resource usage counters from all the nodes were collected only once in this Δ time interval. The predictions were based on these resource usage counters and hence most likely were not timely representative of the load on the nodes. This could be avoided by setting Δ to a value smaller than the minimum inter-task submission times. Figure 8 summarizes the maximum gains achieved for each workload after tuning for the right value of p (Section 7.5).

For *FB2010*, we achieved lower gains compared to those achieved for *FB2009*. The number of stragglers found per hour in *FB2010* is comparable to those in *FB2009*. However, [31] notes that Facebook's workload has changed significantly from 2009 to 2010. *FB2010* has a higher job sub-

[2] We divided the data collected in these 2 hours in 3 parts; we used 2 of them for training and the remaining part for testing the model's correctness.

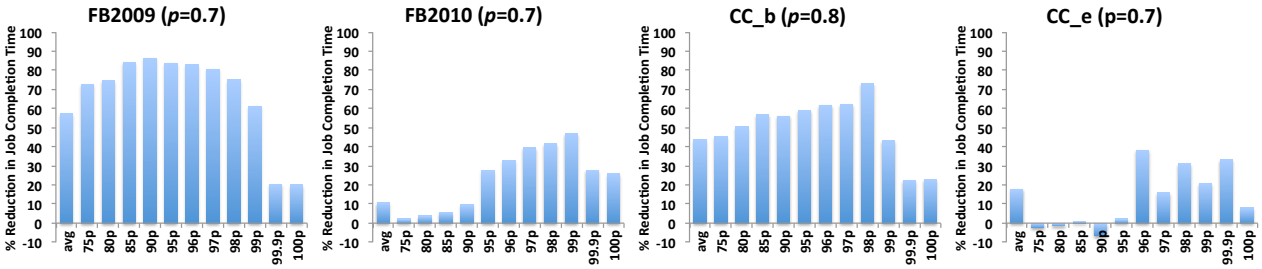


Figure 8: Summary of *Wrangler's* improvements in job completion times for all the workloads with the tuned vale of p (see Sections 5.2.1 and 7.5): This plot shows that *Wrangler* successfully reduces the completion time by carefully scheduling potential stragglers.

Workload	Total Task-Seconds		% Reduction in
	w/o *Wrangler*	With *Wrangler*	total task-seconds
FB-2009	903980	405953	55.09
FB-2010	296893	223339	24.77
CC_b	201444	120559	40.15
CC_e	694564	637319	8.24

Table 2: Resource utilization with and without *Wrangler* in terms of total task execution times (in seconds) across all the jobs during our replay. Being proactive, *Wrangler* achieves the improved completion times while consuming lesser resources compared to the wait-and-speculate re-execution mechanisms.

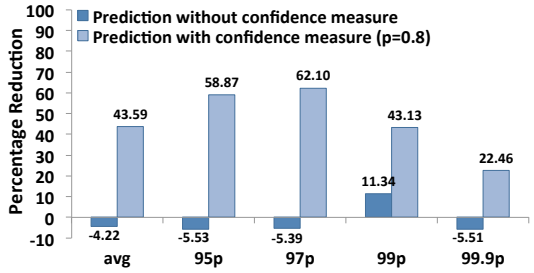


Figure 9: The confidence measure attached with each of the predictions is crucial. This plot compares the reduction in overall job completion time achieved by *Wrangler* with and without using the confidence measure attached with its predictions for *CC_b* workload.

mission rate, higher I/O rate and is more compute intensive. This does not affect the prediction accuracy, but the model-informed scheduling mechanism needs comparatively less-occupied nodes to achieve faster job completions. *Wrangler's* key idea is to avoid overloading a few nodes and instead distribute that load evenly. However, if load is consistently high on the cluster, *Wrangler's* gains are limited.

7.3 Does *Wrangler* reduce resource consumption?

We showed that *Wrangler* significantly improves the job completion time when used in conjunction with speculative execution. The reactive relaunch-based mechanism of speculative execution consumes extra resources for the redundantly launched copies of straggler tasks. On the contrary, being proactive *Wrangler* achieves overall faster job completions by smarter task to node assignments. Table 2 shows that *Wrangler* consumes lesser resources as compared to speculative execution resulting in reduced costs by freeing up the resources sooner. As we noted earlier, *CC_e* has a bursty job submission pattern and even in this case, the delay based mechanism of *Wrangler* speeds up the jobs while using lesser total resources.

7.4 Is *Wrangler* reliable in presence of modeling errors?

Wrangler's achieves reliability in presence of prediction errors by attaching confidence with the straggler predictions. We have observed that this confidence measure plays a crucial role in improving the completion times for all the workloads by allowing only confident predictions to influence scheduling decisions. Figure 9 shows the percentage reduction in job completion times achieved by *Wrangler* with and without the confidence measure for *CC_b* as an example. In the absence of a confidence measure, the modeling errors drive the scheduling decisions to change too frequently. This makes the costs incurred by delaying the tasks to weigh more than the reduction achieved in job completion time. Thus, *Wrangler* achieves its goal of being robust to modeling errors by the novel use of a confidence measure.

7.5 How sensitive is *Wrangler* with respect to p?

We described how to learn a value of p during training in Section 5.2.1. In this section, we evaluate the sensitivity of *Wrangler* with respect to p. We calculated the percentage reduction in job completion times *Wrangler* achieves with different values of p in a range of $[0,1]$. In Figure 10, we present a subset of them. Our experimental validation shows that for both the Facebook workloads (*FB2009* and *FB2010* in Figure 10 (a) and (b)) a value of 0.7 attains the maximum gains in terms of improved job completion times and resource utilization. The right value of p turned out to be 0.8 for *CC_b* and 0.7 for *CC_e* (see Figure 10 (c) and (d) respectively.

From this analysis, we note that choosing the right value for p is important for improving job completion times.

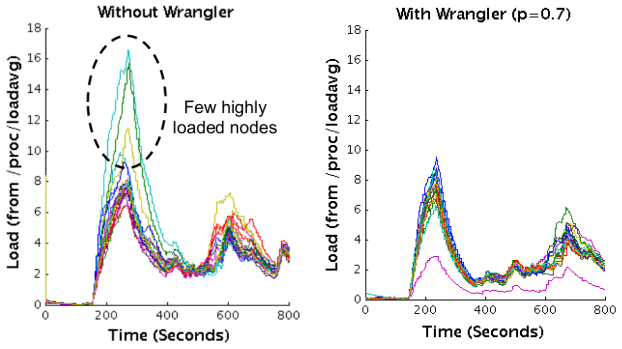


Figure 11: Load/resource utilization without and with *Wrangler* (p=0.7) for *FB2010*: Even with highly intensive *FB2010* workload, *Wrangler* speeds up the 99^{th} percentile of job completions by 47% by avoiding overloading of a few nodes and distributing the load more evenly (see Section 7.6).

We also note that this value is workload dependent. Small changes in its value could change the improvements drastically for some workloads (for example, see CC_b in Figure 10 (c)). However, in Section 5.2.1 we described an automatic way to choose a starting value for p during the model building phase itself without waiting for jobs to execute. Additionally, to tune to its right value, we need to observe the performance of the jobs for relatively small amount of time (we verified on a window of 30 minutes). *Wrangler's* architecture is flexible and allows changes in value of p on the fly until it converges to a right value.

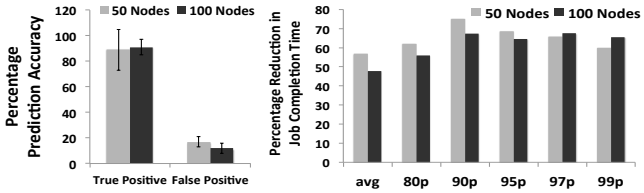
7.6 *How* does *Wrangler* improve job completion times?

Wrangler avoids stragglers by implicitly doing a better job of load balancing than the scheduling approaches that use only statically available information. Figure 11 shows the load on the cluster executing *FB2010* without *Wrangler* and with *Wrangler* (p=0.7). Without *Wrangler*, a few nodes enter a heavily loaded state and tend to remain loaded as new tasks are then assigned to them in addition. For a right value of p, with *Wrangler*, we observed that the load is now distributed evenly and most of the nodes are almost equally utilized. So even in this limiting case of *FB2010*, where the cluster is heavily loaded, *Wrangler* improves job completions significantly (47.06% at the 99^{th} percentile, see Figure 10 (b)).

7.7 What if *Wrangler* mis-predicts?

The main impact of wrong predictions is on the delay experienced by tasks. In Section 5.3, we statistically proved that *Wrangler* will not delay tasks indefinitely. Also, since our method only delays potential stragglers, there is no risk of terminating or replacing them. Thus, *Wrangler* does not impact the correctness or liveness properties of the system.

In this section, we analyze empirically observed delays for the Facebook and Cloudera workloads in our dataset.



(a) Prediction Accuracy (b) *Improved Job Completion*

Figure 13: Scalability of Wrangler's centralized prototype: The prediction accuracies shown in (a) and percentage reduction in job completion times shown in (b) for a larger (100 nodes) cluster are significant and comparable with those for a 50 nodes cluster.

Figure 12 shows the percentage delay with respect to the task's original execution time (i.e., without *Wrangler*) and the percentage reduction in *task* completion times achieved by *Wrangler* for these workloads. Note that Figure 12 presents speed-up at *task*-level whereas all the other results presented earlier are the *job*-level speed-ups. We see that for *FB2009*, the delay introduced by *Wrangler* is less than 2% for 75% of the tasks and less than 11% for 90% of the tasks. Notice that the delays introduced for *FB2010* are higher than *FB2009* and the corresponding improvements in task completion times are also lower. This is due to the high load on all the nodes of the cluster executing *FB2010* as we described in Section 7.2. This explains *Wrangler's* decision to delay the tasks more while seeking for situations where the nodes are not over committed.

7.8 Does *Wrangler* scale well?

Our current prototype of *Wrangler* is centralized: the resource usage counters from all the nodes are collected, processed, node-wise models are built and used centrally from a location accessible to the scheduler. Most actual users use cluster size of about 100 nodes [38–40]. We executed *FB*2009 on a larger cluster with 100 nodes. Note that in our experiments, we maintain fixed amount of work per node by re-scaling the real-life workload appropriately.

Figure 13 (a) shows that we get comparable prediction accuracies of the models for the small and larger clusters. Moreover, Figure 13 (b) shows that the percentage reduction in job completion time achieved by *Wrangler's* centrally implemented prototype is comparable across cluster sizes.

Results from Figures 13 (a) and (b) show that *Wrangler* scales well to larger cluster sizes. The central implementation might stress the master with increase in the size of a cluster further. We believe that *Wrangler* could be extended for even larger clusters by distributing these responsibilities to individual nodes. This allows for most of the computations to take place locally, thereby reducing the network traffic. In this scenario, each node would build its own model using locally collected resource usage counters. At a regular time interval Δ (see Section 5.2.2), the straggler predictions would be computed using locally available recent re-

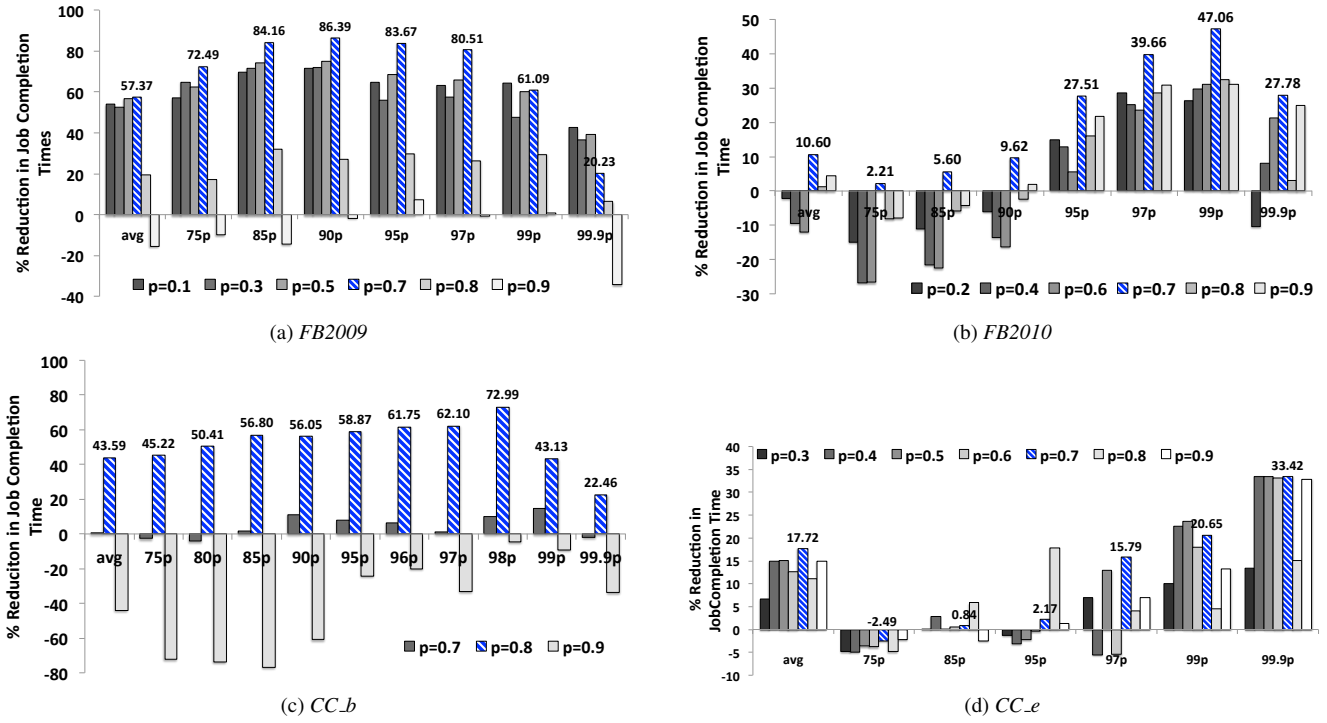


Figure 10: Reduction in job completion times achieved by *Wrangler* with various values of p for all the workloads. Data labels are shown for the p value that achieves the highest gain in completion times for each of the four workloads.

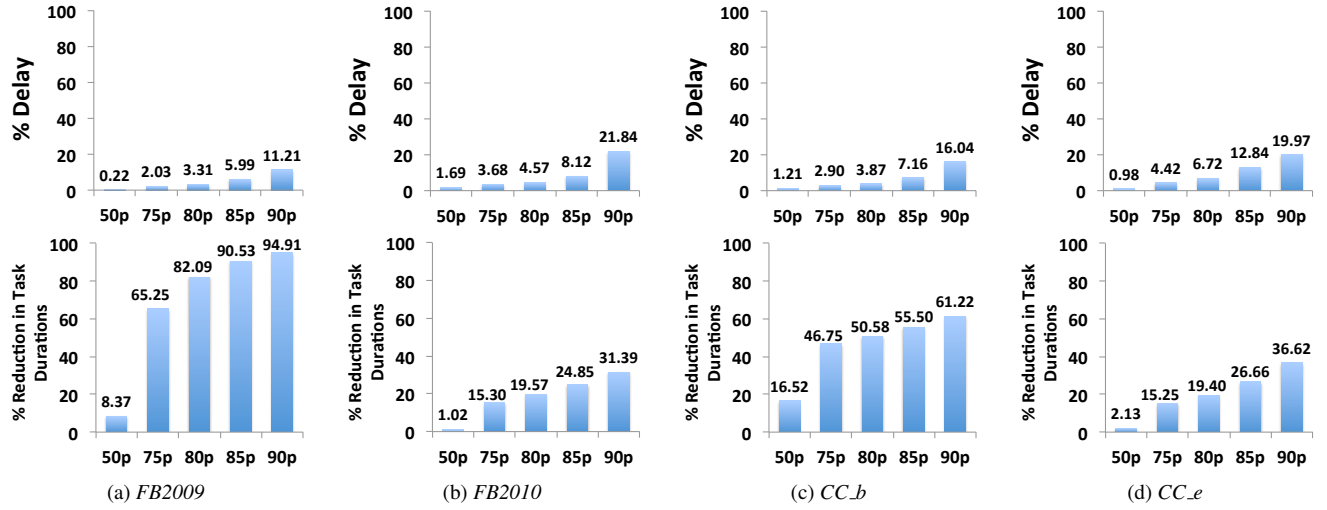


Figure 12: Empirical analysis of the amount of delay introduced by *Wrangler* and the speed up achieved in task completions: With marginal delays, *Wrangler* achieves significant reduction in task durations. *FB2010* being a resource intensive workload [31], *Wrangler* induces slightly higher delays seeking for nodes that are not over-committed. This results in reduced improvements since the cluster is mostly loaded. Since *CC_e* contains lesser number of stragglers, *Wrangler* has limited opportunity to improve task completions (see Section 7.7).

source usage counters and model at each node. These model predictions would then be piggybacked with the heartbeat messages sent to the master by each node. Finally, the master would use these predictions to influence scheduling decisions as described in our current workflow. We leave this as an avenue for future research.

8. Conclusion

Wrangler proactively avoids stragglers to achieve faster job completions while using fewer resources. Rather than allowing tasks to execute and detecting them as stragglers when they run slow, *Wrangler* predicts stragglers before they are

launched. *Wrangler's* notion of a *confidence measure* allows it to overcome modeling errors. Further, *Wrangler* leverages this confidence measure to achieve a reliable task scheduling; thus eliminating the need for replicating them. Prototype on Hadoop using an EC2 cluster of 50 nodes showed that *Wrangler* speeds up the 99^{th} percentile job execution times by up to 61% and consumes up to 55% lesser resources as compared to the speculative execution for production workloads at Facebook and Cloudera's customers. Although it serves as a straggler avoidance approach on its own, *Wrangler* can also be used in conjunction with existing mitigation approaches. In the future, we aim to speed up the training process by (1) reducing the time spent for capturing training data per node in a cluster and (2) training straggler prediction models across workloads.

9. Acknowledgments

We are indebted to Vivek Chawda, Tathagata Das, Bharath Hariharan, John Kubiatowicz, Anthony Joseph and Ion Stoica for helpful discussions regarding the project; Chiranjib Bhattacharyya, Aurojit Panda, Sara Alspaugh, Yanpei Chen, Rachit Agarwal, Anurag Khandelwal, Joseph Gonzalez and other members of the AMPLab for their insightful comments on various drafts of this paper. We thank the anonymous reviewers of HotCloud 2013, OSDI 2014 and SoCC 2014 for help in improving this work with their extremely helpful comments. We also thank our shepherd, Fred Douglis, for help in shaping the final version of the paper.

This research is supported in part by NSF CISE Expeditions Award CCF-1139158, LBNL Award 7076018, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, SAP, The Thomas and Stacey Siebel Foundation, Adobe, Apple, Inc., Bosch, C3Energy, Cisco, Cloudera, EMC, Ericsson, Facebook, GameOnTalis, Guavus, HP, Huawei, Intel, Microsoft, NetApp, Pivotal, Splunk, Virdata, VMware, and Yahoo!.

References

[1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[2] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[3] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[4] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy H. Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI*, 2008.

[5] Ganesh Ananthanarayanan, Michael Chien-Chun Hung, Xiaoqi Ren, Ion Stoica, Adam Wierman, and Minlan Yu. Grass: Trimming stragglers in approximation analytics. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*, pages 289–302, Seattle, WA, April 2014. USENIX Association.

[6] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective straggler mitigation: Attack of the clones. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, pages 185–198, Berkeley, CA, USA, 2013. USENIX Association.

[7] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, February 2013.

[8] Barroso L. Dean, Jeff. *Achieving Rapid Response Times in Large Online Services*. http://research.google.com/people/jeff/latency.html, 2012.

[9] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@home: an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, November 2002.

[10] Gaurav D. Ghare and Scott T. Leutenegger. Improving speedup and response times by replicating parallel programs on a snow. In *Proceedings of the 10th International Conference on Job Scheduling Strategies for Parallel Processing*, JSSPP'04, pages 264–287, Berlin, Heidelberg, 2005. Springer-Verlag.

[11] Walfredo Cirne, Francisco Brasileiro, Daniel Paranhos, Luís Fabrício W. Góes, and William Voorsluys. On the efficacy, efficiency and emergent behavior of task replication in large distributed systems. *Parallel Comput.*, 33(3), April 2007.

[12] Edward Bortnikov, Ari Frank, Eshcar Hillel, and Sriram Rao. Predicting execution bottlenecks in map-reduce clusters. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pages 18–18, Berkeley, CA, USA, 2012. USENIX Association.

[13] Shekhar Gupta, Christian Fritz, Bob Price, Roger Hoover, Johan Dekleer, and Cees Witteveen. Throughputscheduler: Learning to schedule on heterogeneous hadoop clusters. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC'13)*, pages 159–165, San Jose, CA, 2013. USENIX.

[14] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skewtune: Mitigating skew in mapreduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 25–36, New York, NY, USA, 2012. ACM.

[15] Matei Zaharia. The Hadoop Fair Scheduler. http://developer.yahoo.net/blogs /hadoop/FairSharePres.ppt.

[16] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fair-

ness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, 2010.

[17] Hadoop's Capacity Scheduler. http://hadoop.apache.org/core/docs/current/capacity_scheduler.html.

[18] Faraz Ahmad, Srimat T. Chakradhar, Anand Raghunathan, and T. N. Vijaykumar. Tarazu: Optimizing mapreduce on heterogeneous clusters. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 61–74, New York, NY, USA, 2012. ACM.

[19] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 261–276, New York, NY, USA, 2009. ACM.

[20] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Wang, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. Pacman: Coordinated memory caching for parallel jobs. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 20–20, Berkeley, CA, USA, 2012. USENIX Association.

[21] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 127–144, New York, NY, USA, 2014. ACM.

[22] Matthew L. Massie, Brent N. Chun, and David E. Culler. The ganglia distributed monitoring system: Design, implementation and experience. *Parallel Computing*, 30:2004, 2003.

[23] Nello Cristianini and John Shawe-Taylor. *An Introduction to Support Vector Machines: And Other Kernel-based Learning Methods*. Cambridge University Press, New York, NY, USA, 2000.

[24] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[25] Christopher J. C. Burges. A tutorial on support vector machines for pattern recognition. *Data Min. Knowl. Discov.*, 2(2):121–167, June 1998.

[26] John C. Platt. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. In *ADVANCES IN LARGE MARGIN CLASSIFIERS*, pages 61–74. MIT Press, 1999.

[27] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. Smote: Synthetic minority oversampling technique. *Journal of Artificial Intelligence Research*, 16:321–357, 2002.

[28] Nitesh V. Chawla, Nathalie Japkowicz, and Aleksander Kotcz. Editorial: Special issue on learning from imbalanced data sets. *SIGKDD Explor. Newsl.*, 6(1):1–6, June 2004.

[29] Yanmin Sun, Mohamed S. Kamel, Andrew K. C. Wong, and Yang Wang. Cost-sensitive boosting for classification of imbalanced data. *Pattern Recogn.*, 40(12):3358–3378, December 2007.

[30] Zhi-Hua Zhou and Xu-Ying Liu. Training cost-sensitive neural networks with methods addressing the class imbalance problem. *IEEE Trans. on Knowl. and Data Eng.*, 18(1):63–77, January 2006.

[31] Yanpei Chen, Sara Alspaugh, and Randy H. Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *PVLDB*, 5(12):1802–1813, 2012.

[32] Yanpei Chen, Archana Ganapathi, Rean Griffith, and Randy Katz. The case for evaluating mapreduce performance using workload suites. In *Proceedings of the 2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '11, Washington, DC, USA, 2011. IEEE Computer Society, https://github.com/SWIMProjectUCB/SWIM/wiki/.

[33] John C. Platt. Fast training of support vector machines using sequential minimal optimization. In Bernhard Schölkopf, Christopher J. C. Burges, and Alexander J. Smola, editors, *Advances in Kernel Methods*, pages 185–208. MIT Press, Cambridge, MA, USA, 1999.

[34] Antoine Bordes, Seyda Ertekin, Jason Weston, and Léon Bottou. Fast kernel classifiers with online and active learning. *J. Mach. Learn. Res.*, 6:1579–1619, December 2005.

[35] Paul G. Hoel, Sidney C. Port, and Charles J. Stone. *Introduction to Stochastic Processes*. Houghton Mifflin Company, Boston, MA, 1972.

[36] Sheldon M. Ross. *Introduction to Probability Models, Eighth Edition*. Academic Press, 8 edition, January 2003.

[37] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: an update. *SIGKDD Explor. Newsl.*, 11(1), November 2009.

[38] Gunho Lee, Niraj Tolia, Parthasarathy Ranganathan, and Randy H. Katz. Topology-aware resource allocation for data-intensive workloads. *SIGCOMM Comput. Commun. Rev.*, 41(1):120–124, January 2011.

[39] Timothy Prickett Morgan. Cluster sizes reveal hadoop maturity curve. http://www.enterprisetech.com/2013/11/08/cluster-sizes-reveal-hadoop-maturity-curve/.

[40] Amazon redshift. http://aws.amazon.com/redshift/faqs/.