Autoscaling for Hadoop Clusters

Anshul Gandhi, Sidhartha Thota
Stony Brook University
{*anshul,sthota*}*@cs.stonybrook.edu*

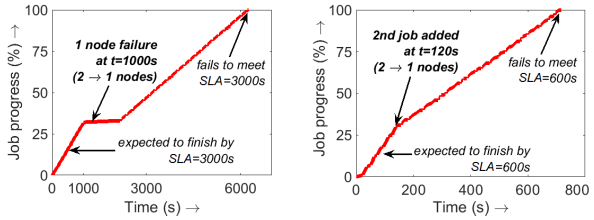
Parijat Dube, Andrzej Kochut, Li Zhang
IBM Research
{*pdube,akochut,zhangli*}*@us.ibm.com*

***Abstract*—Unforeseen events such as node failures and resource contention can have a severe impact on the performance of data processing frameworks, such as Hadoop, especially in cloud environments where such incidents are common. SLA compliance in the presence of such events requires the ability to quickly and dynamically resize infrastructure resources. Unfortunately, the distributed and stateful nature of data processing frameworks makes it challenging to accurately scale the system at run-time. In this paper, we present the design and implementation of a model-driven autoscaling solution for Hadoop clusters. We first develop novel gray-box performance models for Hadoop workloads that specifically relate job execution times to resource allocation and workload parameters. We then employ these models to dynamically determine the resources required to successfully complete the Hadoop jobs as per the user-specified SLA under various scenarios including node failures and multi-job executions. Our experimental results on three different Hadoop cloud clusters and across different workloads demonstrate the efficacy of our models and highlight their autoscaling capabilities.**

I. Introduction

The growing need for data processing and analytics has resulted in the development of a broad set of tools specifically tailored for data processing, such as Hadoop [4] and Spark [5]. Cloud service providers, recognizing this emerging trend, have started incorporating analytics as one of the key capabilities within their Cloud offerings (such as Elastic MapReduce [3] from Amazon [2] and the Sahara project [9] from OpenStack [7]). Cloud users are typically interested in completing their analysis in a timely manner (execution time SLA) while minimizing their resource rental cost. Unfortunately, this is a difficult task for cloud-deployed data processing systems for several reasons:

1. The *dynamic, shared nature of cloud computing* often results in unpredictable application performance. For example, job progress can be (severely) affected by node failures or resource contention due to colocated applications [10], [15], as we illustrate in Figure 1 (see Section III for our experimental setup). Subsequent *dynamic* performance recovery and execution time SLA compliance are non-trivial.
2. Data processing jobs are often *composed of multiple stages*, each requiring possibly different resource allocations.
3. The performance of such applications *depends on many internal and external parameters* (for example, Ganglia provides 200+ metrics for monitoring Hadoop), and these relationships are often complex. For example, job execution time depends on resource allocation, input data set size, job configuration parameters, etc., in addition to being workload-dependent (see Section IV). Of these, resource allocation and job configuration parameters typically affect execution time non-linearly [16], [26].



(a) Node failures (b) Resource contention

Figure 1. Performance variation in the cloud due to, for example, node failures and resource contention can lead to SLA violations for Hadoop applications.

The above challenges make it *difficult to meet SLA* requirements while efficiently utilizing resources. On the other hand, *over-provisioning* can help meet SLAs, but leads to wastage of resources, and is thus expensive. Prior work on performance management of data processing systems has largely focussed on optimal *static* resource allocation, such as ARIA [26] and Elastisizer [20]. The few approaches that focus on dynamic allocation rely on expensive, complex simulations (as in Jockey [14]) or optimizations (as in Rayon [12]), or manually tuned heuristics (as in MR-Runner [17]), to derive the required scaling rules; such approaches are not well suited for dynamic environments, such as clouds, with strict execution time SLAs.

In this paper, we focus specifically on enabling *autoscaling of data processing applications*. Given its popularity, we choose Hadoop [4] as our case study of a data processing framework. Autoscaling of resources in Hadoop is non-trivial because of the need to maintain data locality. However, Hadoop is a distributed system that is also fault tolerant. We exploit the fault tolerant design features in Hadoop to realize run-time elasticity, enabling *autoscaling of Hadoop clusters while jobs are in progress, to meet user-specified SLA targets*.

We employ a model-driven approach to autoscaling. Our approach involves developing *approximate performance models* for Hadoop jobs that can then be tuned based

978-1-5090-1961-8/16 $31.00 © 2016 IEEE
DOI 10.1109/IC2E.2016.11

on the workload profile and cluster configuration, rather than attempting to build exact workload-agnostic performance models. In particular, we first derive generic white-box performance models for Hadoop that take resource allocation into account. We then employ (regression-based) black-box modeling to tune our generic models for specific workloads, including iterative machine learning algorithms. The resulting *gray-box* models only focus on important system parameters, providing very good accuracy (typically less than 5% error) without being overly complex. Importantly, our gray-box models can be easily adapted to other cluster configurations, as we show in our results. Finally, we leverage these models to decide the scaling actions needed to meet the execution time SLAs for data processing workloads under a range of scenarios including autoscaling while the job is in progress, autoscaling to recover from failures, autoscaling to minimize resource rental costs, and autoscaling to overcome resource contention. Our implementation results on two OpenStack-deployed private clouds and an AWS EC2-based public cloud under various Hadoop workloads (compute-heavy, I/O-heavy, and iterative multi-job workloads) demonstrate the efficacy of our autoscaling.

The rest of the paper is organized as follows. Section II provides necessary background on Hadoop. We describe our experimental setup, including clusters and workloads, in Section III. We discuss our modeling efforts in Section IV and our evaluation results in Section V. We present related work in Section VI and conclude in Section VII.

II. Hadoop Overview

We use Apache Hadoop [4] as our data processing framework for this paper. Hadoop enables distributed data processing on large data sets. The Hadoop cluster typically consists of several nodes that together host the data on HDFS [23] and provide computational resources. One node is designated as the Master node, and the rest are designated as Slave nodes. Hadoop applications submit jobs directly to the Master. Each job is typically composed of a Map stage and a Reduce stage. These stages are further split into several tasks (for more details on Map/Reduce, we refer the reader to the seminal MapReduce paper [13]). A task can be considered as the smallest unit of work in Hadoop.

The Map/Reduce computations are managed by the JobTracker service, hosted by the Master. Individual Map/Reduce tasks are managed and executed by the TaskTracker service, hosted on each Slave node. The JobTracker maintains queues for incoming tasks, and these tasks are then assigned to TaskTrackers on Slave nodes. Depending on the availability of resources, a job is typically processed in several successive "waves" of tasks. Once all waves (and thus, all tasks) are processed, the job completes. We use **job execution time** as the performance metric in this paper. Note that multiple jobs can be simultaneously executed on a shared cluster.

The HDFS is managed by the NameNode service, typically hosted by the Master node. Individual data blocks are managed by DataNode services, typically hosted on each Slave node. Hadoop is designed to be fault tolerant. The failure of a single task or node does not result in job failure; the failed task(s) is simply restarted on another node (although, in reality, multiple task failures can be triggered, as discussed in Section V).

III. Experimental Setup

We experiment with three different cloud deployments:

OpenStack Havana (private cloud): is composed of multiple SoftLayer [24] hypervisors. Each hypervisor consists of 8 CPU cores and 64GB of memory. We set up Hadoop with 1 Master node VM and multiple Slave node VMs on this cloud. Our default Slave node is configured with 1 core, 4GB memory, and 80GB of disk space. We typically configure 1 slot per node, and run the TaskTracker and DataNode on each Slave node. We use the default replication factor of 3 and the default block size of 64MB for HDFS [23]. Unless specified otherwise, we use the OpenStack Havana deployment.

OpenStack Icehouse (private cloud): is similar to the Havana deployment, except that we employ the CapacityScheduler [6] to execute multiple jobs simultaneously in a shared Hadoop cluster, as opposed to the default JobQueueTaskScheduler.

AWS EC2 (public cloud): is composed of multiple t2.medium instances hosted in the N.Virginia region [2] as our Master and Slave nodes with 2 vCPUs and 4GB memory each. Again, we employ the CapacityScheduler for this deployment.

A. Workloads

We employ three different Hadoop workloads for our experiments: WordCount (included in Hadoop base installation), TeraSort (part of HiBench [21]), and Kmeans (part of HiBench). WordCount is typically dominated by the Map stage, and is CPU-bound. TeraSort is made up of a CPU-bound Map stage and an I/O-bound Reduce stage. Kmeans is a multi-job workload, made up of multiple CPU-bound Iteration jobs and a single I/O-bound Classification job [19].

B. Autoscaling

In order to dynamically add and remove nodes from the Hadoop cluster, we create a customized VM image preloaded with Hadoop. A new Slave node can be dynamically added by booting a new VM via OpenStack Nova using the customized image, and then starting the TaskTracker and DataNode services on it. The new node announces itself to the Master using heartbeat messages. Note that the transfer of existing HDFS data to the new Slave nodes should delay task execution. While we did notice a transient performance degradation while autoscaling for our EC2 cluster, we did not notice any significant delays for our OpenStack clusters;

this is likely because of the high network bandwidth between VMs on the same physical cluster as opposed to the low bandwidth between distributed public cloud instances.

To dynamically remove a Slave node, we update the exclude list on the Master and dynamically refresh the node configuration. The Master then migrates the HDFS data, if any, from this node before excluding it. Once excluded, the node can be turned off via OpenStack Nova commands.

C. Controller

We employ a simple reactive controller to ensure execution time SLA compliance for Hadoop jobs. The execution time SLA is provided as input to the controller, along with job- and cluster-specific configuration parameters. We modified Hadoop to periodically log the fraction of input data that has yet to be processed. We use this information, along with the workload-specific models described in Section IV, to periodically predict the remaining execution time of the job. If the predicted execution time exceeds the SLA target, we invoke autoscaling. To determine the amount of scaling that is required to meet the SLA target, we again leverage our workload-specific models. Our controller executes the autoscaling by issuing commands to the Hadoop and OpenStack APIs. Experiments reported in Section V employ this controller.

IV. Modeling

We now present our modeling results for various Hadoop workloads. Our modeling methodology involves first developing generic performance models for Hadoop (Section IV-B) based on its characteristics (Section IV-A). A list of variables used in our modeling is shown in Table I for reference. The resulting white-box models are then tuned, via regression, based on the profiles of specific workloads (Section IV-C). Validation tests (Section IV-D) illustrate the high accuracy of our gray-box models. Our models are very versatile and can be easily adapted to different cluster configurations and resource sharing models (Section IV-E). We employ our models for autoscaling in Section V.

A. Characterizing Hadoop Job Execution Time

A Hadoop job consists of a Map stage and a Reduce stage. In the Map stage, all Map tasks are executed in possibly multiple Map waves. Since each Map task processes roughly similar sized input data (based on the default 64MB block size), the execution times of different Map tasks, for a given workload and cluster configuration, are similar. This is confirmed by our preliminary experiments for different WordCount Map tasks obtained using various input text files. The Reduce stage can also have multiple Reduce waves, with each Reduce wave consisting of a shuffle phase, a sort phase, and a reduce phase. Depending on the configuration setting (`slowstart`), the shuffle phase of the first Reduce wave may overlap with the Map stage. For this reason, the duration of the first Reduce wave is different from that of subsequent Reduce waves; this was also observed in prior work [26].

T_{job}	Total job execution time
T_{ms} (T_{rs})	Execution time of Map (Reduce) stage
T_{mt} (T_{rt})	Execution time of Map (Reduce) task
T_{frw}	Execution time of first Reduce wave
T_{srw}	Execution time of subsequent Reduce waves
M (R)	Number of Map (Reduce) tasks
N_{mw} (N_{rw})	Number of Map (Reduce) waves
N_{mc} (N_{rc})	Number of Map (Reduce) configured cores
n_{ms} (n_{rs})	Number of Map (Reduce) slots/cores
D	Size of input data
$f(.)$	Relationship between Map task execution time on a core and input data per Map task
$g_f(.)$ ($g_s(.)$)	Relationship between first (subsequent) Reduce wave task execution time on a core and input data per Reduce task

Table I
List of variables used in modeling

The scheduling of tasks (where a task can be either Map or Reduce) on Slave nodes depends on the number of cores per node and the number of slots per core. Note that the number of slots per node is at least equal to the number of cores per node. On a core, the slots are executed in a processor-sharing manner. If only one slot is occupied, that slot gets all the processor cycles. However, when multiple slots are concurrently executing tasks on the same core, the processor cycles are shared equally among these tasks.

Let T_{ms} be the execution time of the Map stage (time taken from the start of the first Map task until the end of the last Map task). Let T_{rs} be the non-overlapping execution time of the Reduce stage (time taken from the end of the last Map task until the end of the last Reduce task). Then, the total job execution time, T_{job}, is:

$$T_{job} = T_{ms} + T_{rs}. \quad (1)$$

A Map stage consists of one or more Map waves. Since each Map wave involves concurrent execution of multiple Map tasks across different Slave nodes, the execution time of a typical Map wave is the same as that of a single Map task, T_{mt}. Thus, if N_{mw} denotes the number of Map waves, we have:

$$T_{ms} = N_{mw} \times T_{mt}. \quad (2)$$

1) Map Task Execution Time (T_{mt}): The input data is equally partitioned (based on the block size) among different Map tasks of a job and each Map task (for a given workload) applies the same operation on its input data. Thus, the execution time of a Map task depends on the size of its input data (block size), the type of Map operation (workload), and the core configuration (e.g., processor speed). Assuming homogeneous cores in the cluster, we can write T_{mt} as some function of the input data partition. Let D be the total input data for the job and M be the number of Map tasks. Then each Map task processes D/M amount of data. We also need to account for the number of processor cycles allocated to the task slot. Let N_{mc} be the number of cores in the cluster

configured with Map slots and let n_{ms} be the number of Map slots per core. Since the core (and its cycles) is equally shared by all resident slots, we can write:

$$T_{mt} \quad = \quad f(D/M) \times \min\left(\left\lceil \frac{M}{N_{mc}} \right\rceil, n_{ms}\right), \qquad (3)$$

where the form of $f(.)$ as a function of input data per Map task depends on the workload. Note that $f(D/M)$ represents the normalized Map task execution time on a core *if* the task gets all its processor cycles. However, given n_{ms} Map slots per core, each Map task only gets $1/n_{ms}$ fraction of the processor cycles, thus amplifying its execution time (by n_{ms}). The min term in Eq. (3) accounts for the corner cases where M is less than N_{mc}. To illustrate the significance of the min term, consider an example cluster with $M = 3$, $N_{mc} = 4$, and $n_{ms} = 2$. Since the number of Map tasks, M, is less than the number of cores, N_{mc}, each core will execute one task, with all the resources on the core devoted to that task. For this case, $\min\left(\left\lceil \frac{M}{N_{mc}} \right\rceil, n_{ms}\right) = \min(1,2) = 1$, thus $T_{mt} = f(D/M)$. Now consider the case where $M = 5$. Since the number of Map slots in the cluster is $N_{mc} \times n_{ms} = 8$, and the number of Map tasks is greater than N_{mc} but less than the total number of map slots in the cluster, one core will execute two tasks in parallel (sharing the processor cycles) while the other three cores will execute one task each. Since T_{mt} depends on the completion time of the last task(s) in a given cluster, $T_{mt} = f(D/M) \times 2$.

2) Map Stage Execution Time (T_{ms}): The Map stage execution time, T_{ms}, depends on the total number of Map waves, N_{mw}. N_{mw}, in turn, depends on the *total* number of Map slots available, $N_{mc} \times n_{ms}$, as:

$$N_{mw} \quad = \quad \left\lceil \frac{M}{(N_{mc} \times n_{ms})} \right\rceil. \qquad (4)$$

From Eqs. (3), (4), and (2), we get:

$$T_{ms} = \left\lceil \frac{M}{(N_{mc} \times n_{ms})} \right\rceil \times f(D/M) \times \min\left(\left\lceil \frac{M}{N_{mc}} \right\rceil, n_{ms}\right). \qquad (5)$$

Observe that in Eq. (5) we have expressed T_{ms} as a function of the input data size (D), number of Map tasks (M), and cluster configuration parameters (N_{mc} and n_{ms}).

3) Reduce Stage Execution Time (T_{rs}): Since a Reduce wave involves concurrent execution of multiple Reduce tasks, the execution time of a Reduce wave is (almost) the same as the execution time of a Reduce task of the wave. As discussed earlier, due to the overlap of the first Reduce wave with the Map stage, the execution time of the Reduce tasks belonging to the first Reduce wave is different from that of subsequent Reduce waves. We only consider the non-overlapping execution time of the first Reduce wave in computing the total execution time of the job. Let T_{frw} and T_{srw} be the execution time of first and subsequent Reduce waves, respectively. To account for the discrepancy between first and subsequent Reduce waves execution, we model the dependency of T_{frw} and T_{srw} on Reduce task input data using different functions, as follows:

$$T_{frw} = g_f(D/R) \times \min\left(\left\lceil \frac{R}{N_{rc}} \right\rceil, n_{rs}\right), \qquad (6)$$

$$T_{srw} = g_s(D/R) \times min\left(\left\lceil \frac{R}{N_{rc}} \right\rceil, n_{rs}\right), \qquad (7)$$

where the min term on the right hand side (RHS) of Eqs. (6) and (7) is due to the sharing of the processor on a Reduce node by concurrent Reduce slots, with N_{rc} and n_{rs} being the number of Reduce cores and the number of Reduce slots per core in the cluster, respectively. The functions g_f and g_s represent the workload-dependent relationships between input data and execution time for Reduce tasks of the first and subsequent waves, respectively. Note that the input data per Reduce task is *not* (D/R) since the Map stage modifies the input data. Prior work [26] has shown that the Map stage output data is typically proportional to the Map stage input data, for a given workload. We use the g functions to model this relation.

Lastly, to account for the overhead of data movement in the Shuffle phase, we introduce a linear M/R term. The numerator (M) denotes the number of Shuffle sources, and the denominator (R) accounts for the fraction of data transmitted to each Reduce task. With N_{rw} being the total number of Reduce waves, we can write:

$$T_{rs} \quad = \quad T_{frw} + (N_{rw} - 1) \cdot T_{srw} + \lambda \frac{M}{R}, \qquad (8)$$

where λ is some (workload-dependent) constant. Using expressions for T_{frw} and T_{srw} from Eqs. (6) and (7) in Eq. (8), and setting $N_{rw} = \lceil R/(N_{rc} \times n_{rs}) \rceil$, we get:

$$T_{rs} = \left[g_f(D/R) + g_s(D/R) \left(\left\lceil \frac{R}{(N_{rc} \times n_{rs})} \right\rceil - 1 \right) \right] \times \min\left(\left\lceil \frac{R}{N_{rc}} \right\rceil, n_{rs} \right) + \lambda \frac{M}{R}. \qquad (9)$$

B. Generic Model for Hadoop Job Execution Time

We now develop generic performance models for Hadoop based on the above analysis. We will then tune our generic models based on the workload in Section IV-C.

From Eqs. (5) and (9) we get the Map stage and Reduce stage execution time in terms of the cluster parameters (N_{mc}, N_{rc}, n_{ms}, n_{rs}) and workload parameters (D, M, R). However, f, g_f, and g_s are still unknown functions.

We approximate f, g_f, and g_s as polynomials with unknown coefficients. In order to capture the possibly non-linear dependence on input parameters, we approximate these functions using second-order polynomials: $f(D/M) \approx \alpha_0 + \alpha_1(D/M) + \alpha_2(D/M)^2$, $g_f(D/R) \approx \beta_0 + \beta_1(D/R) + \beta_2(D/R)^2$, and $g_s(D/R) \approx \gamma_0 + \gamma_1(D/R) + \gamma_2(D/R)^2$, where the α, β, and γ coefficients are workload-dependent parameters and represent the specificity of a workload. The generic performance models for Hadoop workloads can now be fully

expressed, with unknown coefficients, as:

$$\left.\begin{aligned} T_{ms} &= \left(\alpha_0+\alpha_1(D/M)+\alpha_2(D/M)^2\right) \\ &\quad \times \left\lceil \frac{M}{(N_{mc}\times n_{ms})} \right\rceil \cdot \min\left(\left\lceil \frac{M}{N_{mc}} \right\rceil, n_{ms} \right), \\ T_{rs} &= \left[\left(\beta_0+\beta_1(D/R)+\beta_2(D/R)^2\right) \right. \\ &\quad \left. + \left(\gamma_0+\gamma_1(D/R)+\gamma_2(D/R)^2\right)\left(\left\lceil \frac{R}{(N_{rc}\times n_{rs})} \right\rceil - 1\right) \right] \\ &\quad \times \min\left(\left\lceil \frac{R}{N_{rc}} \right\rceil, n_{rs} \right) + \lambda(M/R), \\ T_{job} &= T_{ms} + T_{rs}. \end{aligned}\right\} \quad (10)$$

Note that coefficient values can be zero. In fact, when calibrating the models for different workloads, we find that $\alpha_2 \approx 0$, thus reducing f to a first-order polynomial.

C. Workload-based Model Tuning

We now tune our models via regression for specific workloads. The training data for model calibration is collected by running the workload with different data sizes and hardware (number of nodes and the number of cores per node) and software (number of Map/Reduce tasks and the number of Map/Reduce slots per core) parameters. After experimentation, we parse the Hadoop log files to obtain observed values of different execution times, $\hat{T}_{ms}$ and $\hat{T}_{rs}$. The training data can now be used to determine the values of the coefficients $(\alpha, \beta, \gamma, \lambda)$ in equation set (10) for each workload.

To affirm the predictive power of our models, we obtain the corresponding goodness of fit coefficient, the R^2 value. R^2 is a statistical measure for regression and takes values between 0 (low accuracy) and 1 (high accuracy). We also define the training error, Δ, as the relative distance, measured as a percentage, between the observed data and the regression hyperplane; smaller the percentage value, better is the fit.

$$\Delta = \frac{1}{N}\sum_{i=1}^{N} \frac{|\text{Measured Value(n) - Model Value (n)}|}{\text{Model Value(n)}},$$

where N is the number of data points.

While the profiling of workloads is an overhead and requires time, it is necessary. Given the fundamental algorithmic differences in various data processing workloads (such as WordCount and Kmeans), it is not possible to estimate execution times of a workload by profiling other workloads. While run-time (online) model calibration of workloads during their execution time is possible, it makes it very difficult to meet application SLAs, especially for short jobs that we consider in Section V which complete in a few minutes. Finally, note that it is typically not possible to estimate completion times by simply extrapolating job progress because of the differences in Map and Reduce stages (and the Reduce sub-phases such as Shuffle), which lead to very different progress rates during the lifetime of a job (see, for instance, Fig. 5(b)). This is further evidenced by the non-linear terms in equation set (10).

Metric	Model	R^2	Δ
T_{ms}	$(0.4(D/M)+6)\cdot\lceil M/(N_{mc}\times n_{ms})\rceil \cdot \min(\lceil \frac{M}{N_{mc}} \rceil, n_{ms})$	1	1.7%
T_{rs}	$(4.9+(5\times10^{-4})(D/R)+(6.1+(7.3\times10^{-3})(D/R))$ $\cdot(\lceil R/(N_{rc}\times n_{rs})\rceil - 1))\cdot\min(\lceil \frac{R}{N_{rc}} \rceil, n_{rs})+0.1(M/R)$	.97	8.2%

Table II
EXECUTION TIME MODEL FOR WORDCOUNT

Metric	Model	R^2	Δ
T_{ms}	$(0.6(D/M)-29)\cdot\lceil M/(N_{mc}\times n_{ms})\rceil \cdot \min(\lceil \frac{M}{N_{mc}} \rceil, n_{ms})$	1	1.1%
T_{rs}	$(16.3+(3.9\times10^{-4})(D/R)$ $+((6.2\times10^{-2})(D/R)+(5\times10^{-6})(D/R)^2)$ $\cdot(\lceil R/(N_{rc}\times n_{rs})\rceil - 1))\cdot\min(\lceil \frac{R}{N_{rc}} \rceil, n_{rs})+3.5(M/R)$	.99	6.9%

Table III
EXECUTION TIME MODEL FOR TERASORT

1) WordCount and TeraSort: Tables II and III show the estimated functions for T_{ms} and T_{rs} for WordCount and TeraSort. The job execution time, T_{job}, can then be calculated using Eq. (1). The R^2 and Δ values suggest that equation set (10) is a good fit for WordCount and TeraSort.

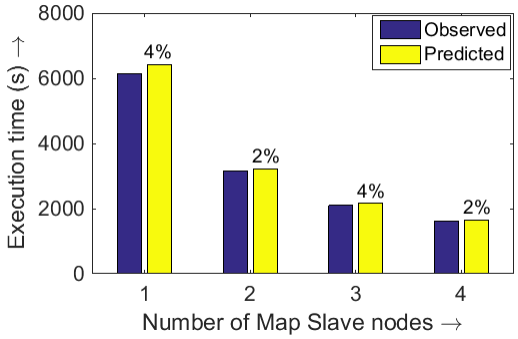
2) Kmeans: Kmeans is a complex multi-job workload consisting of multiple CPU-bound Iteration jobs and a single I/O-bound Classification job. We model its job execution time in the two phases (Iteration and Classification) separately. The modeling for Kmeans proceeds along similar lines as for WordCount and TeraSort, and is thus omitted.

D. Model Validation

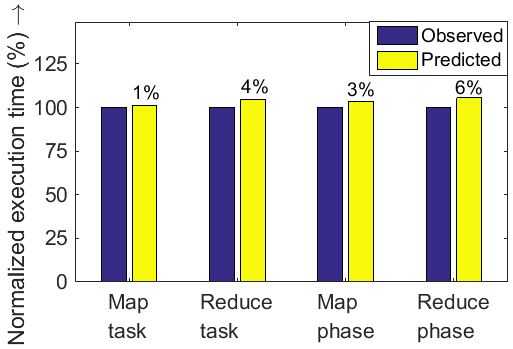
We now validate the workload-specific models developed above using test data. Note that the test data is different (in terms of job and/or cluster configurations) from the training data used to build the above models.

1) WordCount: Figure 2 shows our validation results for WordCount using a 12GB input data set (resulting in 190 Map tasks and 1 Reduce task). We configure our cluster to have 1 Reduce Slave node with 1 Reduce slot, and multiple Map Slave nodes with 1 Map slot each. Figure 2(a) shows our experimentally observed and model predicted execution times for the above configuration as a function of the number of Map Slave nodes employed (recall, from Section III-A, that WordCount is dominated by the Map stage). As expected, execution time decreases with an increase in the number of Map Slave nodes. Prediction accuracy for our model is very good, with less than 5% error. Our comprehensive model can also be used to estimate the execution time of individual tasks and stages. We illustrate this capability in Figure 2(b) which shows the normalized execution time of Map and Reduce tasks and stages. Again, prediction accuracy is high.

2) TeraSort: Figure 3 shows our validation results for TeraSort. We configure our cluster to have an equal number of Map and Reduce Slave nodes, but we vary the number of



(a) Job execution times for WordCount as a function of number of Map Slave nodes.



(b) Normalized execution times for WordCount for different tasks and stages.

Figure 2. Observed versus Predicted execution times for WordCount using a 12GB input data set. Modeling error is typically less than 5%.

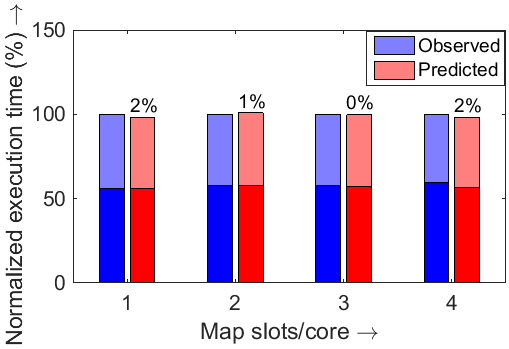


Figure 3. Observed versus Predicted normalized job execution times for TeraSort. The lower (darker) stacked bars indicate the Map stage time, and the upper (lighter) stacked bars indicate the (non-overlapping portion of) Reduce stage time. The total height of each bar represents the job execution time. Modeling error is at most 2%.

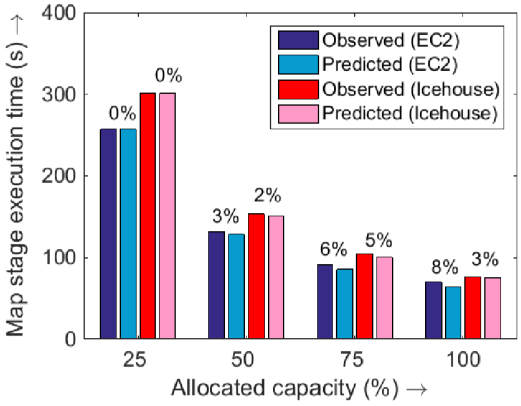


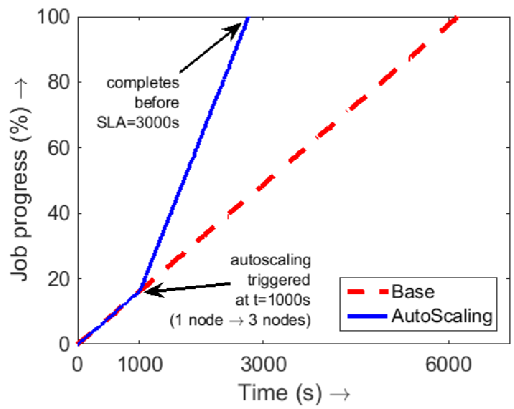
Figure 4. Map stage execution times for WordCount as a function of capacity for our EC2 and Icehouse clusters (using CapacityScheduler).

slots per Slave node. We use a 20GB input data set (with 298 Map tasks and 1 Reduce task), and we configure 1 Reduce slot per Reduce Slave node, and vary the number of Map slots per Map Slave node. The experimentally observed and model predicted normalized job execution times highlight our modeling accuracy (at most 2% error). Note that, using the equations in Table III, we are able to provide accurate execution times for individual Map (lower bars) and Reduce (upper bars) stages as well. We obtain similar results (at most 4% error) when varying the number of Reduce slots per Reduce Slave node. Our experiments reveal that the number of slots per node does not significantly affect execution time (except for a small change in the number of Map and Reduce waves due to rounding).

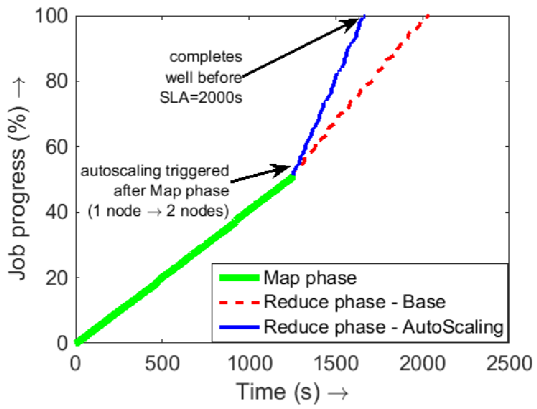
3) Kmeans: For Kmeans, we use a 1.5GB (75,000 records) input data set, resulting in 100 Map tasks. We configure our cluster to have 1 Reduce Slave node with 1 Reduce slot, and multiple Map Slave nodes with 1 Map slot each. The number of Reduce tasks is 1 for Iteration and 0 (Map-only job) for Classification. Our model predicted execution times as a function of the number of Map Slave nodes for the Iteration phase and the Classification phase are within 5% of the experimentally observed execution times. As expected, execution time decreases with the number of Map nodes; detailed results are omitted due to lack of space.

E. Model Versatility

Our models are very versatile and can be easily *applied to different clusters*. The values of the coefficients $(\alpha, \beta, \gamma, \lambda)$ in our model, that are determined via profiling, capture the properties of the cluster configuration. Likewise, our model also applies to clusters with *different resource sharing policies*, as dictated by the scheduler. While the default scheduler in the Havana deployment that we have considered thus far only allows the execution of one job at a time, the CapacityScheduler [6] allows multiple jobs to execute simultaneously on the same cluster. Our resource-aware models can be applied to such settings by treating the resource allocation as job-specific. For example, if a core has two Map slots and two jobs are simultaneously executing on this core with a 50% capacity allocation each, then each job will be entitled to $N_{mc} = 0.5$ Map cores (half a core). Figure 4 shows our model validation results for WordCount using a 768MB input file on our EC2 cluster (with 2 2-core slave nodes with 1 Map slot and 1 Reduce slot per core) and our Icehouse cluster (with 4 1-core slave nodes with 1 Map slot and 1 Reduce slot per core). We see that our models



(a) Scaling up capacity for WordCount using our approach. Autoscaling is triggered at 1000s, resulting in the addition of 2 Map Slave nodes, thereby allowing the job to finish before the 3000s SLA.



(b) Autoscaling for TeraSort. Our approach scales up capacity for the Reduce stage to ensure that the job completes before the 2000s SLA.

Figure 5. Dynamically scaling up capacity to meet execution time SLA using our model-driven autoscaling approach for (a) WordCount and (b) TeraSort.

work very well even in the case of multi-job executions. Importantly, Figure 4 shows that our models can be easily extended to other cluster configurations.

V. Evaluation

In this section, we leverage the above workload-specific models to determine the amount of resources needed by a given job to meet its execution time SLA. We employ the controller described in Section III-C along with the models in Section IV-C to execute autoscaling for *various scenarios* (scaling capacity up/down, failures, resource contention) across *three different Hadoop cloud deployments* (discussed in Section III) using *three different Hadoop workloads* (discussed in Section III-A). In all cases, we successfully meet SLA targets by dynamically autoscaling the Hadoop cluster. We now present a subset of our results in more detail. In particular, we use our default OpenStack Havana deployment to evaluate our approach for scaling up capacity (Section V-A), scaling down capacity (Section V-B), and scaling for performance recovery following node failures (Section V-C). We then use our OpenStack Icehouse and EC2 deployments (with the CapacityScheduler) to evaluate our approach for scaling capacity in the presence of resource contention due to multiple jobs (Section V-D).

For each experiment, we first illustrate results using the default approach, referred to as **Base**, that does not employ dynamic resizing. We then repeat the experiment and, to highlight our controller, dynamically invoke our approach, referred to as **AutoScaling**, *while* the job is in progress.

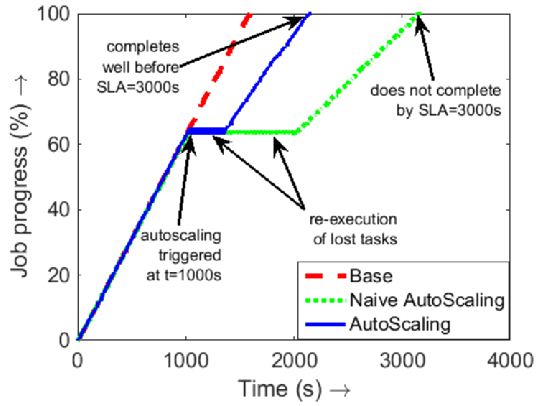
A. Scaling up capacity

Figure 5 demonstrates our ability to dynamically scale up capacity as needed to meet execution time SLAs. Recall, from Section III-B, that scaling up requires booting a new VM, and then starting the TaskTracker and DataNode services on it. In Figure 5(a), we run WordCount on a D=12GB text file on our OpenStack cluster described in Section III. The execution time SLA is set to 3000s. We initially start the job with 1 Map Slave node and 1 Reduce Slave node. Recall that WordCount is dominated by the Map stage. At the 1000s mark, we notice that less than 20% of the input data has been processed (via Hadoop log files), suggesting a possible SLA violation. We thus invoke our autoscaling controller. The controller leverages the model equations in Table II for WordCount, using the job progress value to adjust the input data size (D) accordingly; as 20% of 12GB has been processed, we set $D = 12 \times (1 - 0.2) = 9.6$GB. Our model suggests adding 2 new Map Slave nodes to meet the 3000s SLA; the controller then dynamically triggers this addition (by making API calls to OpenStack) *while* the job is executing. As seen in Figure 5(a), we successfully meet our 3000s SLA with autoscaling. Observe that without autoscaling we do not meet our SLA, as shown by our experimental results for Base in Figure 5(a). The difference in slopes between Base and AutoScaling after the 1000s mark reflects the additional throughput afforded by the two additional Map Slave nodes.

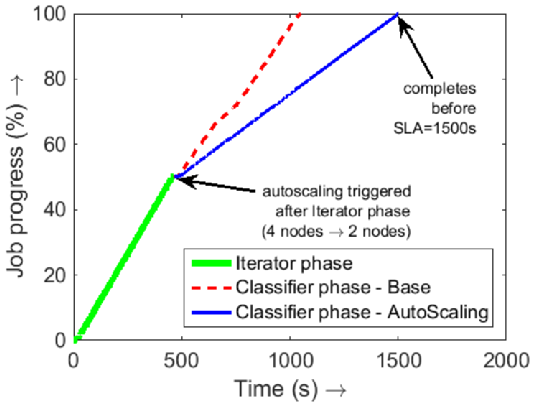
In Figure 5(b), we run TeraSort on a 10GB input data set configured with 12 Reduce tasks. We set the execution time SLA to 2000s. Since TeraSort is dominated by the Reduce stage, we trigger autoscaling after the execution of the Map stage. The controller leverages the model equations for TeraSort (Table III), and determines that an additional Reduce Slave node is required to meet the SLA. Accordingly, a second Reduce node is dynamically added, allowing the job to complete before 2000s. Observe that, without autoscaling, the job would miss the execution time SLA, albeit barely, as shown in our experimental results for Base in Figure 5(b).

B. Scaling down capacity

Scaling down capacity while still meeting the SLA helps reduce resource wastage and associated resource rental costs. Scaling down capacity in Hadoop is trickier than scale-out because of *intermediate data* that might be stored at the



(a) Scaling down capacity for WordCount using our approach. The naive autoscaling approach does not account for re-execution of lost tasks, resulting in SLA violation. Our approach takes re-execution into account, allowing the job to complete before the 3000s SLA.



(b) Autoscaling for Kmeans using our approach. We scale down capacity between the Iteration phase and the Classification phase while ensuring that the 1500s SLA is met.

Figure 6. Dynamically scaling down capacity while still meeting execution time SLA using our autoscaling approach for (a) WordCount and (b) Kmeans.

retiring node. In particular, when a node is dynamically removed from a Hadoop cluster *while* a job is active, the current tasks executing on that node, as well as *completed* tasks of the active job on that node will be reinserted into the JobTracker queue [27, Chapter 6]. By default, Hadoop will re-execute these "lost" tasks, even if the Shuffle phase has copied the data over to the Reduce nodes. This is because the retiring node could also contain Reduce tasks, whose data is now lost. Thus, there will be a significant delay in the execution time of the job as a result of node removal.

Figure 6(a) illustrates this delay. Here, we start WordCount on a 12GB text file with 4 Map nodes and 1 Reduce node. The SLA is set to 3000s. At the 1000s mark, we invoke autoscaling. Given that the job has already processed 60% of the input data, our controller scales down capacity. Using the model equations in Table II for WordCount, the controller removes two Map nodes. This immediately results in re-execution of lost Map tasks on the existing two Map nodes. During this re-execution, job progress stalls, as indicated by the flat horizontal line (dotted green line) in Figure 6(a). Once the lost tasks are processed, progress continues. Unfortunately, in this case, we miss the 3000s SLA. Fortunately, we can improve our model to account for delays due to re-execution of lost tasks. In particular, we can compute the amount of additional data that must now be processed due to node removals (which is simply the amount of data processed thus far by the retiring nodes), and add that to the unprocessed data size (D). The controller now leverages the model equations for WordCount, using the new unprocessed data size as the input parameter, and computes the required scaling. Our model suggests that we remove only one node to meet the 3000s SLA. As shown in Figure 6(a) (solid blue line), we successfully meet the SLA by removing one Map Slave node at the 1000s mark.

When scaling down capacity between jobs (as opposed to during the execution of a job), lost tasks can be avoided.

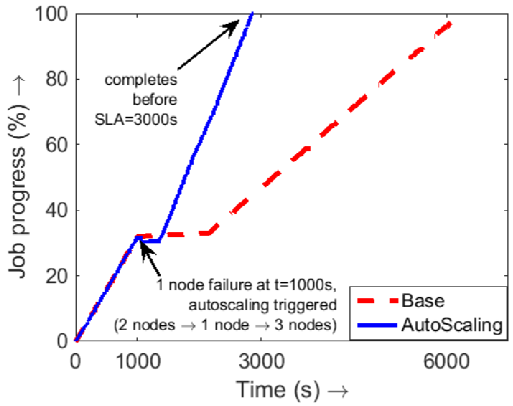
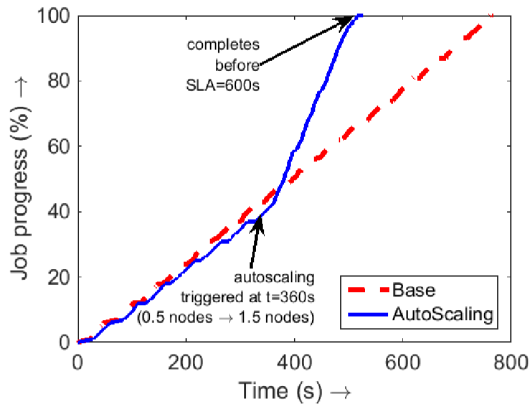


Figure 7. Autoscaling for WordCount in the event of a failure. One node fails at the 1000s mark. Autoscaling is then invoked, which results in the addition of 2 Map nodes, allowing the job to finish by the 3000s SLA.

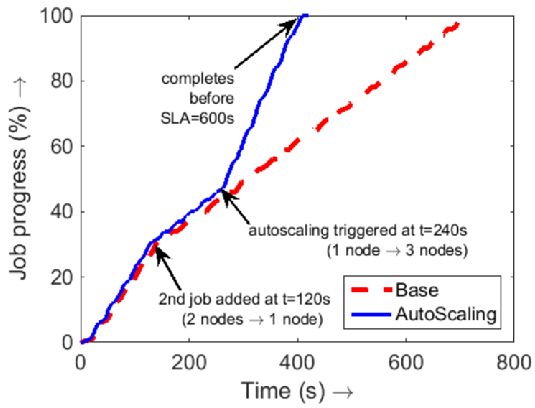
This is illustrated in Figure 6(b), where we run Kmeans on a 75,000 records data set using 4 Map Slave nodes and 1 Reduce Slave node. The execution time SLA is set to 1500s. We invoke autoscaling after the Iteration phase, and before the Classification phase. Recall, from Section III-A, that the Iteration and Classification are individual MapReduce jobs. Thus, by triggering autoscaling between these jobs, we avoid lost tasks, as illustrated by the continually progressing plot (solid blue line) in Figure 6(b). Again, we successfully meet the SLA (albeit barely) while scaling down capacity.

C. Scaling for failure recovery

Autoscaling can also be employed to recover from performance loss following a node failure. Since a node failure is similar to removal of a node, we use our updated model that accounts for lost tasks. Figure 7 illustrates our failure recovery scenario where we start with WordCount running on a 12GB input data set with 2 Map Slave nodes and 1 Reduce Slave node. The SLA is set to 3000s. At the 1000s mark, we trigger a Map Slave node failure. The controller detects



(a) Two jobs are started at time 0. Autoscaling is invoked at the 360s mark, resulting in the addition of one node which is fully allocated to our job, allowing the job to finish by the 600s SLA.



(b) One job is started at time 0. A second job joins the cluster at the 120s mark, slowing down our job. Autoscaling detects this slowdown and adds two new nodes, allowing the job to finish by the 600s SLA.

Figure 8. Successfully autoscaling capacity using our approach for multi-job (resource contention) scenarios using 2 WordCount jobs with 2GB data.

the resulting loss in performance, via Hadoop log files, and invokes autoscaling using the WordCount model equations. Autoscaling results in the addition of two Map Slave nodes, allowing the job to complete by 3000s. Without autoscaling, the baseline approach (red dashed line) completes the job after 6000s, significantly violating the SLA.

D. Scaling in the presence of multiple jobs

We now focus on the scenario where multiple Hadoop jobs are simultaneously executed, as is common in shared clusters. Recall, from Section IV-E, that our models easily adapt to multi-job settings. Autoscaling in multi-job scenarios is complicated by the fact that other jobs can *steal* capacity. Figure 8(a) illustrates this scenario on our EC2 cluster. We initially configure 1 2-core node with 1 Map slot and 1 Reduce slot per core and start our job. The SLA is set to 600s. Another job is started at the same instant, and results in a 50% capacity allocation each. At the 360s mark, we invoke autoscaling. Our controller detects the slow progress and autoscales by adding one new slave node to our job. This results in our job speeding up its execution and finishing before 600s. Note the fractional nodes in Figure 8(a); this is because of capacity sharing between jobs.

A more challenging scenario is depicted in Figure 8(b) for our Icehouse cluster where a new job arrives *during* the execution of our job, thus dynamically stealing capacity. Without a dynamic controller, SLA compliance for this scenario is infeasible. Fortunately, our approach can handle such complex scenarios. We initially configure 2 1-core nodes with 1 Map slot and 1 Reduce slot per core, and set the SLA to 600s. At the 2-minute mark, we start a second WordCount job (not shown) which is allocated 50% cluster capacity. This results in a slowdown for our job as shown for Base. However, our AutoScaling approach notices the slowdown and autoscales by adding 2 additional 1-core nodes for our job. This results in our job speeding up its execution and finishing before 600s.

VI. Related Work

Most of the existing work on allocating resources for data processing jobs, such as ARIA [26], Elastisizer [20], and CRESP [11], focus on *optimal static allocation* for capacity planning of future jobs that are not yet deployed. Our focus in this paper is on autoscaling the resources allocated to a *currently executing* job so as to meet user-specified SLA targets *in the presence of unexpected events such as failures and resource contention*, which are all too common in clouds.

There are a few recent works that address dynamic resource allocation for data processing workloads. Jockey [14] is an SLA-compliant controller for data processing systems. Jockey leverages past executions of a job to build a detailed simulator capable of determining dynamic resource allocations. However, the past executions must be on a similar data set, an assumption that we relax in our work as we allow arbitrary input sizes (parameter D in our model). Further, the authors acknowledge the long run time of the simulator, thus proposing to use it in an offline manner, similar to our modeling-based approach. Rayon [12] is a reservation-based scheduler that allows for dynamic redistribution of resources in YARN via preemption. The authors use an an optimization framework to schedule jobs based on their deadlines. Changes to the cluster capacity (due to, for example, failures) trigger redistribution of resources and may result in the rejection (and preemption) of jobs that are unlikely to meet their deadline. Our approach goes beyond redistribution and dynamically adds more VMs as required to ensure SLA-compliance, thus allowing the cluster to expand beyond capacity and avoid rejecting jobs. ParaTimer [22] provides estimates of remaining time for data processing jobs in the event of failures or contention. Our approach can leverage the complementary work of ParaTimer to provide more accurate estimates of remaining data processing (the parameter D in our models).

MR-Runner [17] enables the sharing of an elastic MapRe-

duce cluster by redistributing resources based on user specifications. Fawkes [18] proposes a sharing mechanism for dynamically balancing the resource allocation across multiple MapReduce clusters. DynMR [25] makes better use of existing resources by interleaving tasks during idle periods. While the above approaches enable dynamic reconfiguration of resources, *they do not support SLA-driven resource allocation*, thus placing the burden of SLA compliance on the users. Likewise, Amazon's Elastic MapReduce [3] allows users to add or remove MapReduce nodes online; however, the user is responsible for deciding the dynamic resource allocation, just as in the case of AWS Auto Scaling [1] and OpenStack Heat [8]. By contrast, we specifically focus on *SLA-compliant resource autoscaling*.

VII. Conclusion

Data processing is an especially appealing class of applications for cloud delivery because of its transient high resource requirement for large data sets. Unfortunately, exploiting the elastic nature of cloud resources for data processing applications is challenging. In this paper, we present our solution for agile autoscaling of cloud-deployed data processing clusters. Our solution relies on our resource-aware performance models that are further tuned for specific workloads. With the help of these models, we accurately estimate the dynamic resource requirements of a Hadoop job for a given execution time SLA. Our experimental results on OpenStack and EC2 clusters demonstrate the efficacy of our solution under various use cases including resource contention and node failures.

ACKNOWLEDGMENT

This research was supported by an NSF CRII CSR Grant 1464151 and a 2015 IBM Faculty Award.

References

[1] Amazon Auto Scaling. http://aws.amazon.com/autoscaling.
[2] Amazon EC2. http://aws.amazon.com/ec2.
[3] Amazon Elastic MapReduce. https://aws.amazon.com/elasticmapreduce.
[4] Apache Hadoop. http://hadoop.apache.org/.
[5] Apache Spark. http://spark.apache.org/.
[6] CapacityScheduler Guide. https://hadoop.apache.org/docs/r1.2.1/capacity_scheduler.html.
[7] OpenStack. http://www.openstack.org.
[8] OpenStack Heat. https://wiki.openstack.org/wiki/Heat.
[9] OpenStack Sahara. http://wiki.openstack.org/wiki/Sahara.
[10] Sean Kenneth Barker and Prashant Shenoy. Empirical Evaluation of Latency-sensitive Application Performance in the Cloud. In *Proceedings of the 1st ACM Conference on Multimedia Systems*, pages 35–46, Phoenix, AZ, USA, 2010.
[11] Keke Chen, J. Powers, Shumin Guo, and Fengguang Tian. CRESP: Towards Optimal Resource Provisioning for MapReduce Computing in Public Clouds. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1403–1412, 2014.
[12] Carlo Curino, Djellel E. Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. Reservation-based Scheduling: If You're Late Don't Blame Us! In *Proceedings of the ACM Symposium on Cloud Computing*, pages 2:1–2:14, Seattle, WA, USA, 2014.
[13] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Opearting Systems Design & Implementation*, pages 137–150, 2004.
[14] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems*, pages 99–112, Bern, Switzerland, 2012.
[15] Anshul Gandhi, Parijat Dube, Alexei Karve, Andrzej Kochut, and Harsha Ellanti. The Unobservability Problem in Clouds. In *Proceedings of the 3rd International Conference on Cloud and Autonomic Computing*, Cambridge, MA, USA, 2015.
[16] Anshul Gandhi, Parijat Dube, Alexei Karve, Andrzej Kochut, and Li Zhang. Adaptive, Model-driven Autoscaling for Cloud Applications. In *Proceedings of the 11th International Conference on Autonomic Computing*, pages 57–64, Philadelphia, PA, USA, 2014.
[17] Bogdan Ghit, Nezih Yigitbasi, and Dick Epema. Resource Management for Dynamic MapReduce Clusters in Multicluster Systems. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 1252–1259, Salt Lake City, UT, USA, 2012.
[18] Bogdan Ghit, Nezih Yigitbasi, Alexandru Iosup, and Dick Epema. Balanced Resource Allocations Across Multiple Dynamic MapReduce Clusters. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pages 329–341, Austin, TX, USA, 2014.
[19] Bhaskar Gowda. HiBench: A Representative and Comprehensive Hadoop Benchmark Suite. In *Proceedings of the 2012 Workshop on Big Data Benchmarking*, San Diego, CA, USA.
[20] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A Self-tuning System for Big Data Analytics. In *Proceedings of the 5th Conference on Innovative Data Systems Research*, pages 261–272, Asilomar, CA, USA, 2011.
[21] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In *Proceedings of the 22nd International Conference on Data Engineering Workshops*, pages 41–51, Los Alamitos, CA, USA, 2010.
[22] Kristi Morton, Magdalena Balazinska, and Dan Grossman. ParaTimer: A Progress Indicator for MapReduce DAGs. In *Proceedings of the 2010 ACM International Conference on Management of Data*, pages 507–518, Indianapolis, IN, USA.
[23] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Proceedings of the 26th Symposium on Mass Storage Systems and Technologies*, pages 1–10, Lake Tahoe, NV, USA, 2010.
[24] SoftLayer Technologies, Inc. http://www.softlayer.com.
[25] Jian Tan, Alicia Chin, Zane Zhenhua Hu, Yonggang Hu, Shicong Meng, Xiaoqiao Meng, and Li Zhang. DynMR: Dynamic MapReduce with ReduceTask Interleaving and MapTask Backfilling. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 2:1–2:14, Amsterdam, The Netherlands, 2014.
[26] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. ARIA: Automatic Resource Inference and Allocation for Mapreduce Environments. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, pages 235–244, Karlsruhe, Germany, 2011.
[27] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2009.