ARIA: Automatic Resource Inference and Allocation for MapReduce Environments

Abhishek Verma, Ludmila Cherkasova, Roy H. Campbell

HP Laboratories
HPL-2011-58

Keyword(s):

Map Reduce, Modeling, Resource Allocation, Scheduling.

Abstract:

MapReduce and Hadoop represent an economically compelling alternative for efficient large scale data processing and advanced analytics in the enterprise. A key challenge in shared MapReduce clusters is the ability to automatically tailor and control resource allocations to different applications for achieving their performance goals. Currently, there is no job scheduler for MapReduce environments that given a job completion deadline, could allocate the appropriate amount of resources to the job so that it meets the required Service Level Objective (SLO). In this work, we propose a framework, called ARIA, to address this problem. It comprises of three inter-related components. First, for a production job that is routinely executed on a new dataset, we build a job profile that compactly summarizes critical performance characteristics of the underlying application during the map and reduce stages. Second, we design a MapReduce performance model, that for a given job (with a known profile) and its SLO (soft deadline), estimates the amount of resources required for job completion within the deadline. Finally, we implement a novel SLO-based scheduler in Hadoop that determines job ordering and the amount of resources to allocate for meeting the job deadlines. We validate our approach using a set of realistic applications. The new scheduler effectively meets the jobs' SLOs until the job demands exceed the cluster resources. The results of the extensive simulation study are validated through detailed experiments on a 66-node Hadoop cluster.

External Posting Date: May 6, 2011 [Fulltext] Approved for External Publication
Internal Posting Date: May 6, 2011 [Fulltext]
To be published in Proceedings of the 8th IEEE International Conference on Autonomic Computing (ICAC'2011), June 14-18, 2011, Karlsruhe, Germany.

ARIA: Automatic Resource Inference and Allocation for MapReduce Environments[*]

Abhishek Verma
University of Illinois at
Urbana-Champaign, IL, US
verma7@illinois.edu

Ludmila Cherkasova
Hewlett-Packard Labs
Palo Alto, CA, US
lucy.cherkasova@hp.com

Roy H. Campbell
University of Illinois at
Urbana-Champaign, IL, US
rhc@illinois.edu

ABSTRACT

MapReduce and Hadoop represent an economically compelling alternative for efficient large scale data processing and advanced analytics in the enterprise. A key challenge in shared MapReduce clusters is the ability to automatically tailor and control resource allocations to different applications for achieving their performance goals. Currently, there is no job scheduler for MapReduce environments that given a job completion deadline, could allocate the appropriate amount of resources to the job so that it meets the required Service Level Objective (SLO). In this work, we propose a framework, called *ARIA*, to address this problem. It comprises of three inter-related components. First, for a production job that is routinely executed on a new dataset, we build a *job profile* that compactly summarizes critical performance characteristics of the underlying application during the map and reduce stages. Second, we design a *MapReduce performance model*, that for a given job (with a known profile) and its SLO (soft deadline), estimates the amount of resources required for job completion within the deadline. Finally, we implement a novel *SLO-based scheduler* in Hadoop that determines job ordering and the amount of resources to allocate for meeting the job deadlines.

We validate our approach using a set of realistic applications. The new scheduler effectively meets the jobs' SLOs until the job demands exceed the cluster resources. The results of the extensive simulation study are validated through detailed experiments on a 66-node Hadoop cluster.

Categories and Subject Descriptors: C.4 [Computer System Organization] Performance of Systems, D.2.6.[Software] Programming Environments.

General Terms: Algorithms, Design, Performance, Measurement, Management

Keywords: MapReduce, Modeling, Resource Allocation, Scheduling

[*]This work was originated and largely completed while A. Verma worked at HP Labs during the summer 2010. R. Campbell and A. Verma are supported in part by NSF CCF grant #0964471.

1. INTRODUCTION

Many enterprises, financial institutions and government organizations are experiencing a paradigm shift towards large-scale data intensive computing. Analyzing large amount of unstructured data is a high priority task for many companies. The steep increase in volume of information being produced often exceeds the capabilities of existing commercial databases. Moreover, the performance of physical storage is not keeping up with improvements in network speeds. All these factors are driving interest in alternatives that can propose a better paradigm for dealing with these requirements. MapReduce [3] and its open-source implementation Hadoop present a new, popular alternative: it offers an efficient distributed computing platform for handling large volumes of data and mining petabytes of unstructured information. To exploit and tame the power of information explosion, many companies[1] are piloting the use of Hadoop for large scale data processing. It is increasingly being used across the enterprise for advanced data analytics and enabling new applications associated with data retention, regulatory compliance, e-discovery and litigation issues.

In the enterprise setting, users would benefit from sharing Hadoop clusters and consolidating diverse applications over the same datasets. Originally, Hadoop employed a simple FIFO scheduling policy. Designed with a primary goal of minimizing the makespan of large, routinely executed batch workloads, the simple *FIFO* scheduler is quite efficient. However, job management using this policy is very inflexible: once long, production jobs are scheduled in the MapReduce cluster the later submitted short, interactive adhoc queries must wait until the earlier jobs finish, which can make their outcomes less relevant. The Hadoop Fair Scheduler (HFS) [21] solves this problem by enforcing some fairness among the jobs and guaranteeing that each job at least gets a predefined minimum of allocated slots. While this approach allows sharing the cluster among multiple users and their applications, HFS does not provide any support or control of allocated resources in order to achieve the application performance goals and service level objectives (SLOs).

In MapReduce environments, many production jobs are run periodically on new data. For example, Facebook, Yahoo!, and eBay process terabytes of data and event logs per day on their Hadoop clusters for spam detection, business intelligence and different types of optimization. For users who require service guarantees, a performance question to be answered is the following: given a MapReduce job J with input dataset D, how many map/reduce slots need to be allocated to this job over time so that it finishes within (soft) deadline T?

[1]"Powered by" Hadoop, `http://wiki.apache.org/hadoop/PoweredBy`

Currently, there is no job scheduler for MapReduce environments that given a job completion deadline, could estimate and allocate the appropriate number of map and reduce slots to the job so that it meets the required deadline. In this work, we design a framework, called **ARIA** (**A**utomated **R**esource **I**nference and **A**llocation), to address this problem. It is based on three inter-related components.

- For a production job that is routinely executed on a new dataset, we build a *job profile* that reflects critical performance characteristics of the underlying application during map, shuffle, sort, and reduce phases.
- We design a *MapReduce performance model*, that for a given job (with a known profile), the amount of input data for processing and a specified soft deadline (job's SLO), estimates the amount of map and reduce slots required for the job completion within the deadline.
- We implement a novel *SLO-scheduler* in Hadoop that determines job ordering and the amount of resources that need to be allocated for meeting the job's SLOs. The job ordering is based on the EDF policy (*Earliest Deadline First*). For resource allocation, the new scheduler relies on the designed performance model to suggest the appropriate number of map and reduce slots for meeting the job deadlines. The resource allocations are dynamically recomputed during the job's execution and adjusted if necessary.

We validate our approach using a diverse set of realistic applications. The application profiles are stable and the predicted completion times are within 15% of the measured times in the testbed. The new scheduler effectively meets the jobs' SLOs until the job demands exceed the cluster resources. The results of our simulation study are validated through experiments on a 66-node Hadoop cluster[2].

This paper is organized as follows. Section 2 introduces profiling of MapReduce jobs. Section 3 establishes performance bounds on job completion time. The initial evaluation of introduced concepts is done in Section 4. We design an SLO-based performance model for MapReduce jobs in Section 5. Section 6 outlines the ARIA implementation. We evaluate the efficiency of the new scheduler in Section 7. Section 8 describes the related work. Finally, we summarize the results and outline future work.

2. JOB EXECUTIONS AND JOB PROFILE

The amount of allocated resources may drastically impact the job progress over time. In this section, we discuss different executions of the same MapReduce job as a function of the allocated map and reduce slots. Our goal is to extract a single *job profile* that uniquely captures critical performance characteristics of the job execution in different stages.

2.1 Job Executions

MapReduce jobs are distributed and executed across multiple machines: the map stage is partitioned into *map tasks* and the reduce stage is partitioned into *reduce tasks*.

Each map task processes a logical split of the input data that generally resides on a distributed file system. The map task applies the user-defined map function on each record and buffers the resulting output. This intermediate data is hash-partitioned for the different reduce tasks and written to the local hard disk of the worker executing the map task.

The reduce stage consists of three phases: *shuffle*, *sort* and *reduce* phase. In the *shuffle phase*, the reduce tasks fetch the intermediate data files from map tasks, thus following the "pull" model. In the *sort phase*, the intermediate files from all the map tasks are sorted. An external merge sort is used in case the intermediate data does not fit in memory. After all the intermediate data is shuffled, a final pass is made to merge all these sorted files. Thus, the shuffle and sort phases are interleaved. Finally, in the *reduce phase*, the sorted intermediate data (in the form of a key and all its corresponding values) is passed to the user-defined reduce function. The output from the reduce function is generally written back to the distributed file system.

Job scheduling in Hadoop is performed by the job master, which manages a number of worker nodes in the cluster. Each worker has a fixed number of *map* and *reduce slots*, which can run tasks. The number of map and reduce slots is statically configured (typically to one or two per core). The workers periodically send heartbeats to the master to reporting the number of free slots and the progress of the tasks that they are currently running. Based on the availability of free slots and the rules of the scheduling policy, the master assigns map and reduce tasks to slots in the cluster.

Let us demonstrate different executions of the same job using the *sort benchmark* [13], which involves the use of identity map/reduce function (i.e., the entire input of map tasks is shuffled to reduce tasks and written as output). First, we run the sort benchmark with 8GB input on 64 machines[3], each configured with a single map and a single reduce slot, i.e., with 64 map and 64 reduce slots overall.

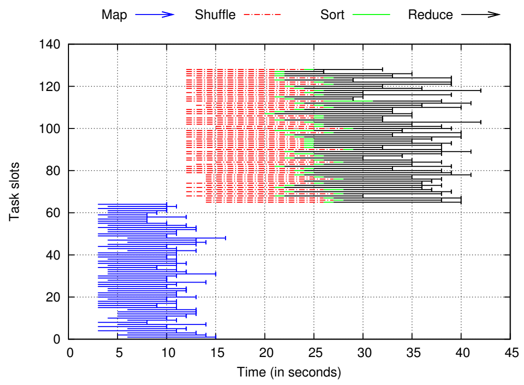


Figure 1: Sorting with 64 map and 64 reduce slots.

Figure 1 shows the progress of the map and reduce tasks over time (on the x-axis) vs the 64 map slots and 64 reduce slots (on the y-axis). Since the file blocksize is 128MB, there are 8GB/128MB = 64 input splits. As each split is processed by a different map task, the job consists of 64 map tasks. This job execution results in single map and reduce *wave*. We split each reduce task into its constituent shuffle, sort and reduce phases (we show the sort phase duration that is complementary to the shuffle phase). As seen in the figure, a part of the shuffle phase overlaps with the map stage.

Next, we run the sort benchmark with 8GB input on the same testbed, except this time, we provide it with a fewer resources: 16 map slots and 22 reduce slots. As shown in Figure 2, since the number of map tasks is greater than the number of provided map slots, the map stage proceeds in multiple rounds of slot assignment, viz. 4 waves ($\lceil 64/16 \rceil$) and the reduce stage proceeds in 3 waves ($\lceil 64/22 \rceil$).

As observed from Figures 1 and 2, it is difficult to predict the completion time of the same job when different amount of resources are given to the job. In the next section, we

[2]It is a representative cluster size for many enterprises. The Hadoop World 2010 survey reported the average cluster size as 66 nodes.

[3]Details of our testbed can be found in Section 4.1.

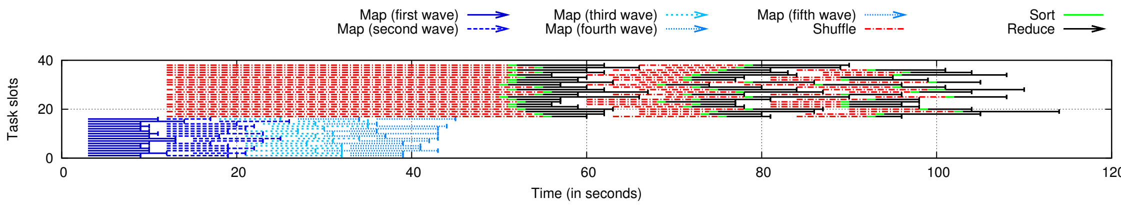


Figure 2: Sorting with 16 map and 22 reduce slots.

introduce a job profile that can be used for prediction of the job completion time as a function of assigned resources.

2.2 Job Profile

Our goal is to create a compact job profile that is comprised of performance *invariants* which are independent on the amount of resources assigned to the job over time and that reflects all phases of a given job: map, shuffle, sort, and reduce phases. This information can be obtained from the counters at the job master during the job's execution or parsed from the logs. More details can be found in Section 6.

The **map stage** consists of a number of map tasks. To compactly characterize the task duration distribution and other invariant properties, we extract the following metrics: $(M_{min}, M_{avg}, M_{max}, AvgSize_M^{input}, Selectivity_M)$, where

- M_{min} – the minimum map task duration. M_{min} serves as an estimate for the beginning of the shuffle phase since it starts when the first map task completes.
- M_{avg} – the average duration of map tasks to summarize the duration of a map wave.
- M_{max} – the maximum duration of map tasks. It is used as a worst time estimate for a map wave completion.
- $AvgSize_M^{input}$ - the average amount of input data per map task. We use it to estimate the number of map tasks to be spawned for processing a new dataset.
- $Selectivity_M$ – the ratio of the map output size to the map input size. It is used to estimate the amount of intermediate data produced by the map stage.

As described earlier, the **reduce stage** consists of the shuffle/sort and reduce phases.

The **shuffle/sort phase** begins only after the first map task has completed. The shuffle phase (of any reduce wave) completes when the entire map stage is complete and all the intermediate data generated by the map tasks has been shuffled to the reduce tasks and has been sorted. Since the shuffle and sort phases are interleaved, we do not consider the sort phase separately and include it in the shuffle phase. After shuffle/sort completes, the reduce phase is performed. Thus the profiles of shuffle and reduce phases are represented by the *average* and *maximum* of their tasks durations. Note, that the measured shuffle durations include networking latencies for the data transfers that reflect typical networking delays and contention specific to the cluster.

The shuffle phase of the first reduce wave may be significantly different from the shuffle phase that belongs to the next reduce waves (illustrated in Figure 2). This happens because the shuffle phase of the first reduce wave overlaps with the entire map stage and depends on the number of map waves and their durations. Therefore, we collect two sets of measurements: (Sh_{avg}^1, Sh_{max}^1) for shuffle phase of the first reduce wave (called, *first shuffle*) and $(Sh_{avg}^{typ}, Sh_{max}^{typ})$ for shuffle phase of the other waves (called, *typical shuffle*). Since we are looking for the performance invariants that are independent of the amount of allocated resources to the job, we characterize a shuffle phase of the first reduce wave in a special way and include only the non-overlapping portions of the first shuffle in (Sh_{avg}^1 and Sh_{max}^1). Thus the job profile in the shuffle phase is characterized by two pairs of measurements: $(Sh_{avg}^1, Sh_{max}^1, Sh_{avg}^{typ}, Sh_{max}^{typ})$.

The **reduce phase** begins only after the shuffle phase is complete. The profile of the reduce phase is represented by: $(R_{avg}, R_{max}, Selectivity_R)$: the *average* and *maximum* of the reduce tasks durations and the *reduce selectivity*, denoted as $Selectivity_R$, which is defined as the ratio of the reduce output size to its input.

3. ESTIMATING JOB COMPLETION TIME

In this section, we design a MapReduce performance model that is based on *i)* the job profile and *ii)* the performance bounds of completion time of different job phases. This model can be used for predicting the job completion time as a function of the input dataset size and allocated resources.

3.1 Theoretical Bounds

First, we establish the performance bounds for a makespan (a completion time) of a given set of n tasks that is processed by k servers (or by k slots in MapReduce environments).

Let $T_1, T_2, \ldots, T_n$ be the duration of n tasks of a given job. Let k be the number of slots that can each execute one task at a time. The assignment of tasks to slots is done using a simple, online, *greedy* algorithm, i.e., assign each task to the slot with the earliest finishing time.

Let $\mu = (\sum_{i=1}^{n} T_i)/n$ and $\lambda = \max_i \{T_i\}$ be the *mean* and *maximum duration* of the n tasks respectively.

Makespan Theorem: The makespan of the greedy task assignment is at least $n \cdot \mu/k$ and at most $(n-1)\cdot\mu/k + \lambda$. [4]

The lower bound is trivial, as the best case is when all n tasks are equally distributed among the k slots (or the overall amount of work $n \cdot \mu$ is processed as fast as possible by k slots). Thus, the overall makespan is at least $n \cdot \mu/k$.

For the upper bound, let us consider the worst case scenario, i.e., the longest task $\hat{T} \in \{T_1, T_2, \ldots, T_n\}$ with duration λ is the last processed task. In this case, the time elapsed before the final task $\hat{T}$ is scheduled is at most the following: $(\sum_{i=1}^{n-1} T_i)/k \le (n-1)\cdot\mu/k$. Thus, the makespan of the overall assignment is at most $(n-1)\cdot\mu/k + \lambda$. [5]$\blacksquare$

The difference between lower and upper bounds represents the range of possible job completion times due to non-determinism and scheduling. These bounds are particularly useful when $\lambda \ll n \cdot \mu/k$, i.e., when the duration of the longest task is small as compared to the total makespan.

3.2 Completion Time Estimates of a MapReduce Job

Let us consider job J with a given profile either built from executing this job in a staging environment or extracted

[4]Tighter lower and upper bounds can be defined for some special cases, e.g., if $n \le k$ then lower and upper bounds are equal to λ, or lower bound can be defined as $max(n \cdot \mu/k, \lambda)$. However, this would complicate the general computation. Typically, for multiple waves, the proposed bounds are tight. Since our MapReduce model actively uses Makespan Theorem, we chose to use a simpler version of the lower and upper bounds.

[5]Similar ideas were explored in the classic papers on scheduling, e.g., to characterize makespan bounds [5].

from past job executions. Let J be executed with a new dataset that is partitioned into N_M^J map tasks and N_R^J reduce tasks. Let S_M^J and S_R^J be the number of map and reduce slots allocated to job J respectively.

Let M_{avg} and M_{max} be the average and maximum durations of map tasks (defined by the job J profile). Then, by Makespan Theorem, the lower and upper bounds on the duration of the entire map stage (denoted as T_M^{low} and T_M^{up} respectively) are estimated as follows:

$$T_M^{low} = N_M^J \cdot M_{avg} / S_M^J \tag{1}$$

$$T_M^{up} = (N_M^J - 1) \cdot M_{avg} / S_M^J + M_{max} \tag{2}$$

The reduce stage consists of shuffle (which includes the interleaved sort phase) and reduce phases. Similarly, Makespan Theorem can be directly applied to compute the lower and upper bounds of completion times for reduce phase (T_R^{low}, T_R^{up}) since we have measurements for average and maximum task durations in the reduce phase, the numbers of reduce tasks N_R^J and allocated reduce slots S_R^J. [6]

The subtlety lies in estimating the duration of the shuffle phase. We distinguish the non-overlapping portion of the *first shuffle* and the task durations in the *typical shuffle* (see Section 2 for definitions). The portion of the typical shuffle phase in the remaining reduce waves is computed as follows:

$$T_{Sh}^{low} = \left(\frac{N_R^J}{S_R^J} - 1 \right) \cdot Sh_{avg}^{typ} \tag{3}$$

$$T_{Sh}^{up} = \left(\frac{N_R^J - 1}{S_R^J} - 1 \right) \cdot Sh_{avg}^{typ} + Sh_{max}^{typ} \tag{4}$$

Finally, we can put together the formulae for the lower and upper bounds of the overall completion time of job J:

$$T_J^{low} = T_M^{low} + Sh_{avg}^1 + T_{Sh}^{low} + T_R^{low} \tag{5}$$

$$T_J^{up} = T_M^{up} + Sh_{max}^1 + T_{Sh}^{up} + T_R^{up} \tag{6}$$

T_J^{low} and T_J^{up} represent optimistic and pessimistic predictions of the job J completion time. In Section 4, we compare whether the prediction that is based on the average value between the lower and upper bounds tends to be closer to the measured duration. Therefore, we define:

$$T_J^{avg} = (T_M^{up} + T_J^{low})/2. \tag{7}$$

Note that we can re-write Eq. 5 for T_J^{low} by replacing its parts with more detailed Eq. 1 and Eq. 3 and similar equations for sort and reduce phases as it is shown below:

$$T_J^{low} = \frac{N_M^J \cdot M_{avg}}{S_M^J} + \frac{N_R^J \cdot (Sh_{avg}^{typ} + R_{avg})}{S_R^J} + Sh_{avg}^1 - Sh_{avg}^{typ} \tag{8}$$

This presentation allows us to express the estimates for completion time in a simplified form shown below:

$$T_J^{low} = A_J^{low} \cdot \frac{N_M^J}{S_M^J} + B_J^{low} \cdot \frac{N_R^J}{S_R^J} + C_J^{low}, \tag{9}$$

where $A_J^{low} = M_{avg}$, $B_J^{low} = (Sh_{avg}^{typ} + R_{avg})$, and $C_J^{low} = Sh_{avg}^1 - Sh_{avg}^{typ}$. Eq. 9 provides an explicit expression of a job completion time as a function of map and reduce slots allocated to job J for processing its map and reduce tasks, i.e., as a function of (N_M^J, N_R^J) and (S_M^J, S_R^J). The equations for T_J^{up} and T_J^{avg} can be written similarly.

[6] For simplicity of explanation, we omit the normalization step of measured durations in job profile with respect to $AvgSize_M^{input}$ and $Selectivity_M$.

4. INITIAL EVALUATION OF APPROACH

In this section, we perform a set of initial performance experiments to justify and validate the proposed modeling approach based on application profiling. We use a motivating example WikiTrends for these experiments and later evaluate five other applications in Section 7.

4.1 Experimental Testbed

We perform our experiments on 66 HP DL145 GL3 machines. Each machine has four AMD 2.39MHz cores, 8 GB RAM and two 160GB hard disks. The machines are set up in two racks and interconnected with gigabit Ethernet. This is a well-balanced configuration with disk I/O being a primary bottleneck. We used Hadoop 0.20.2 with two machines as the JobTracker and the NameNode, and remaining 64 machines as workers. Each worker is configured with four map and four reduce slots (unless explicitly specified otherwise). The file system blocksize is set to 64MB. The replication level is set to 3. Speculative execution is disabled as it did not lead to significant improvements in our experiments.

In order to validate our model, we use the data from the Trending Topics (TT)[7]: Wikipedia article traffic logs that were collected (and compressed) every hour in September and October 2010. We group these hourly logs according to the month. Our MapReduce application, called *WikiTrends*, counts the number of times each article has been visited according to the given input dataset, which is very similar to the job that is run periodically by TT.

4.2 Stability of Job Profiles

In our first set of experiments, we investigate whether the job profiles for a given job are stable across different input datasets and across different executions on the same dataset. To this end, we execute our MapReduce job on the September and October datasets and with variable number of map and reduce slots. The job profiles for the map and reduce stage are summarized in Table 1. The job "Month$_{x,y}$" denotes the MapReduce job run on the logs of the given month with x number of map slots and y number of reduce slots allocated to it. Table 1 shows that the map stage of the job profile is stable across different job executions and different datasets used as input data.

Job	**Map Task duration**			**Avg Input Size in MB**	**Selectivity**
	Min	Avg	Max		
Sept$_{256,256}$	94	144	186	59.85	10.07
Oct$_{256,256}$	86	142	193	58.44	9.98
Sept$_{64,128}$	94	133	170	59.85	10.07
Oct$_{64,128}$	71	132	171	58.44	9.98

Job	**Shuffle/Sort**		**Reduce**		
	Avg	Max	Avg	Max	Selectivity
Sept$_{256,256}$	12	20	16	33	0.37
	121	152			
Oct$_{256,256}$	13	34	17	35	0.36
	122	156			
Sept$_{64,128}$	10	25	15	139	0.37
	122	152			
Oct$_{64,128}$	11	26	15	109	0.36
	123	153			

Table 1: Map and Reduce profiles of four jobs.

Table 1 also shows the job profiles for tasks in shuffle/sort and reduce phases. The shuffle statistics include two sets of data: the average and maximum duration of the non-overlapping portion of the first and typical shuffle waves. We performed the experiment 10 times and observed less than

[7] `http://trendingtopics.org`

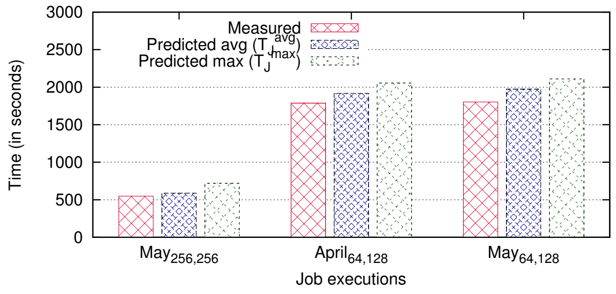


Figure 3: Comparison of predicted and measured job completion times across different job executions and datasets.

5% variation. The average metric values are very consistent across different job instances (these values are most critical for our model). The maximum values show more variance. To avoid the outliers and to improve the robustness of the measured maximum durations we can use instead the mean of a few top values. From these measurements, we conclude that job profiles across different input datasets and across different executions of the same job are indeed similar.

4.3 Prediction of Job Completion Times

In our second set of experiments, we try to predict the job completion times when the job is executed on a different dataset and with different numbers of map and reduce slots.

We build a job profile from the job executed on the September logs with 256 map and 256 reduce slots. Using this job profile and applying formulae described in Section 3, we predict job completion times of the following job configurations:

- September logs: 64 map and 128 reduce slots;
- October logs: 256 map and 256 reduce slots;
- October logs: 64 map and 128 reduce slots.

The results are summarized in Figure 3. We observe that the relative error between the predicted average time T_J^{avg} and the measured job completion time is less than **10%** in all these cases, and hence, the predictions based on T_J^{avg} are well suited for ensuring the job SLOs.

5. ESTIMATING RESOURCES FOR A GIVEN DEADLINE

In this section, we design an efficient procedure to estimate the minimum number of map and reduce slots that need to allocated to a job so that it completes within a given (soft) deadline.

5.1 SLO-based Performance Model

When users plan the execution of their MapReduce applications, they often have some *service level objectives* (SLOs) that the job should complete within time T. In order to support the job SLOs, we need to be able to answer a complementary performance question: given a MapReduce job J with input dataset D, what is the minimum number of map and reduce slots that need to be allocated to this job that it finishes within T?

There are a few design choices for answering this question:

- T is targeted as a *lower bound* of the job completion time. Typically, this leads to the least amount of resources that are allocated to the job for finishing within deadline T. The lower bound corresponds to "ideal" computation under allocated resources and is rarely achievable in real environments.
- T is targeted as an *upper bound* of the job completion time. This would lead to a more aggressive resource allocations and might result in a job completion time that is much smaller (better) than T because worst case scenarios are also rare in production settings.
- T is targeted as the *average* between lower and upper bounds on the job completion time. This solution might provide a balanced resource allocation that is closer for achieving the job completion time T.

The allocations of map and reduce slots to job J (with a known profile) for meeting soft deadline T are found using a variation of Eq. 9 introduced in Section 3, where A_J^{low}, B_J^{low}, and C_J^{low} are defined.

$$A_J^{low} \cdot \frac{N_M^J}{S_M^J} + B_J^{low} \cdot \frac{N_R^J}{S_R^J} = T - C_J^{low} \qquad (10)$$

Let us use a simplified form of this equation shown below:

$$\frac{a}{m} + \frac{b}{r} = D \qquad (11)$$

where m is the number of map slots, r is the number of reduce slots allocated to the job J, and a, b and D represent the corresponding constants (expressions) from Eq. 10. This

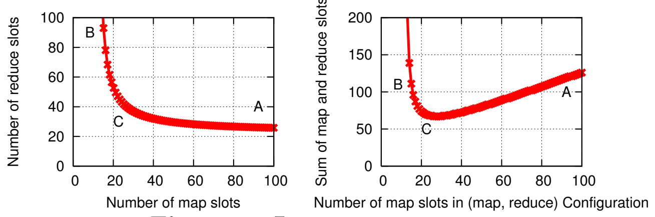


Figure 4: Lagrange curve

equation yields a hyperbola if m and r are the variables. All integral points on this hyperbola are possible allocations of map and reduce slots which result in meeting the same SLO deadline T. As shown in Figure 4 (left), the allocation could use the maximum number of map slots and very few reduce slots (shown as point A) or very few map slots and the maximum number of reduce slots (shown as point B). These different resource allocations lead to different amount of resources used (as a combined sum of allocated map and reduce slots) shown Figure 4 (right). There is a point where the sum of the map and reduce slots is minimized (shown as point C). We will show how to calculate this minima on the curve using Lagrange's multipliers, since we would like to conserve the map and reduce slots allocated to job J.

We wish to minimize $f(m, r) = m + r$ over $\frac{a}{m} + \frac{b}{r} = D$. We set $\Lambda = m + r + \lambda\frac{a}{m} + \lambda\frac{b}{r} - D$. Differentiating Λ partially with respect to m, r and λ and equating to zero, we get

$$\frac{\partial \Lambda}{\partial m} = 1 - \lambda\frac{a}{m^2} = 0 \qquad (12)$$

$$\frac{\partial \Lambda}{\partial r} = 1 - \lambda\frac{b}{r^2} = 0 \qquad (13)$$

$$\frac{\partial \Lambda}{\partial \lambda} = \frac{a}{m} + \frac{b}{r} - D = 0 \qquad (14)$$

Solving these equations simultaneously, we get

$$m = \frac{\sqrt{a}(\sqrt{a} + \sqrt{b})}{D}, \quad r = \frac{\sqrt{b}(\sqrt{a} + \sqrt{b})}{D} \qquad (15)$$

SLO (s)	Allocated (map, reduce) slots based on		
	Lower bound	Average bound	Upper bound
240	(41, 21)	(58, 30)	(64, 64)
300	(33, 17)	(43, 22)	(63, 32)
360	(27, 14)	(34, 18)	(45, 23)
420	(24, 12)	(28, 15)	(35, 18)

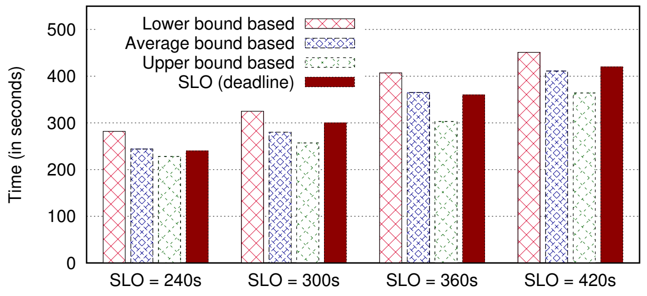


Figure 5: Slot allocations and job completion times based on minimum resource allocation based on different bounds.

These values are the optimal values of map and reduce slots such that the number of slots used is minimized while meeting the deadline. In practice, these values have to be integral. Hence, we round up the values found by these equations and use them as an approximation.

5.2 Initial Evaluation of SLO-based Model

In this section, we perform an initial set of performance experiments to validate the SLO-based model introduced in Section 5.1. We aim to evaluate the accuracy of resource allocations recommended by the model for completing the job within a given deadline T. For validation we use the *WikiTrends* application (see Section 4.1 for more details).

The WikiTrends application consists of 71 map and 64 reduce tasks. We configure one map and reduce slot on each machine. We vary the SLO (deadline) for the job through 4, 5, 6 and 7 minutes. Using the lower, average and upper bound as the target SLO, we compute the minimum number of map and reduce slot allocations as shown in the table in Figure 5. Using these map and reduce slot allocations, we execute the WikiTrends application and measure the job completion times as shown in Figure 5. The model based on the lower bound suggests insufficient resource allocations: the job executions with these allocations missed their deadlines. The model based on the upper bound aims to over provision resources (since it aims to "match" the worst case scenario). While all the job executions meet the deadlines, the measured job completion times are quite lower than the target SLO. We observe that the average bound based allocations result in job completion times which are closest to the given deadlines: within 7% of the SLO.

6. ARIA IMPLEMENTATION

Our goal is to propose a novel SLO-scheduler for MapReduce environments that supports a new API: a job can be submitted with a desirable job completion deadline. The scheduler will then estimate and allocate the appropriate number of map and reduce slots to the job so that it meets the required deadline. To accomplish this goal we designed and implemented a framework, called *ARIA*, to address this problem. The implementation consists of the following five interacting components shown in Figure 6:

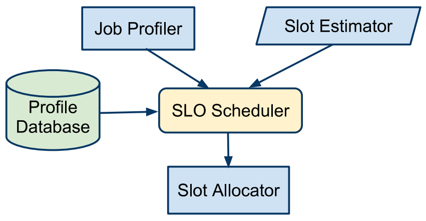


Figure 6: ARIA implementation.

1. **Job Profiler:** It collects the job profile information for the currently running or finished jobs. We use the Hadoop counters which are sent from the workers to the master along with each heartbeat to build the profile. This profile information can also be gleaned from the logs in the HDFS output directory or on the job master after the job is completed. The job profile is then stored persistently in the profile database.
2. **Profile Database:** We use a MySQL database to store the past profiles of the jobs. The profiles are identified by the (user, job name) which can be specified by the application.
3. **Slot Estimator:** Given the past profile of the job and the deadline, the slot estimator calculates the minimum number of map and reduce slots that need to be allocated to the job in order to meet its SLO. Essentially, it uses the Lagrange's method to find the minima on the allocation curve introduced in Section 5.1.
4. **Slot Allocator:** Using the slots calculated from the slot estimator, the slot allocator assigns tasks to jobs such that the job is always below the allocated thresholds by keeping track of the number of running map and reduce tasks. In case there are spare slots, they can be allocated based on the additional policy. There could be different classes of jobs: jobs with/without deadlines. We envision that jobs with deadlines will have higher priorities for cluster resources than jobs without deadlines. However, once jobs with deadlines are allocated their required minimums for meeting the SLOs, the remaining slots can be distributed to the other job classes.
5. **SLO-Scheduler:** This is the central component that co-ordinates events between all the other components. Hadoop provides support for a pluggable scheduler. The scheduler makes global decisions of ordering the jobs and allocating the slots across the jobs. The scheduler listens for events like job submissions, worker heartbeats, etc. When a heartbeat containing the number of free slots is received from the workers, the scheduler returns a list of tasks to be assigned to it.

The SLO-scheduler has to answer two inter-related questions: which job should the slots be allocated and how many slots should be allocated to the job? The scheduler executes the *Earliest Deadline First* algorithm (EDF) for job ordering to maximize the utility function of all the users. The second (more challenging) question is answered using the Lagrange computation discussed in Section 5. The detailed slot allocation schema is shown in Algorithm 1.

As shown in Algorithm 1, it consists of two parts: 1) when a job is added, and 2) when a heartbeat is received from a worker. Whenever a job is added, we fetch its profile from the database and compute the minimum number of map and

Algorithm 1 Earliest deadline first algorithm

```
1: When job $j$ is added:
2: Fetch $Profile_j$ from database
3: Compute minimum number of map and reduce slots
   $(m_j, r_j)$ using Lagrange's multiplier method
4: When a heartbeat is received from node $n$:
5: Sort $jobs$ in order of earliest deadline
6: for each slot $s$ in free map/reduce slots on node $n$ do
7:   for each job $j$ in $jobs$ do
8:     if $RunningMaps_j < m_j$ and $s$ is map slot then
9:       if job $j$ has unlaunched map task $t$ with data on
         node $n$ then
10:        Launch map task $t$ with local data on node $n$
11:      else if $j$ has unlaunched map task $t$ then
12:        Launch map task $t$ on node $n$
13:      end if
14:    end if
15:    if $FinishedMaps_j > 0$ and $s$ is reduce slot and
       $RunningReduces_j < r_j$ then
16:      if job $j$ has unlaunched reduce task $t$ then
17:        Launch reduce task $t$ on node $n$
18:      end if
19:    end if
20:  end for
21: end for
22: for each task $T_j$ finished slots by node $n$ do
23:   Recompute $(m_j, r_j)$ based on the current time, cur-
      rent progress and deadline of job $j$
24: end for
```

reduce slots required to complete the job within its specified deadline using the Lagrange's multiplier method discussed earlier in Section 5.1.

Workers periodically send a heartbeat to the master reporting their health, the progress of their running tasks and the number of free map and reduce slots. In response, the master returns a list of tasks to be assigned to the worker. The master tracks the number of running and finished map and reduce tasks for each job. For each free slot and each job, if the number of running maps is lesser than the number of map slots we want to assign it, a new task is launched. As shown in Lines 9 - 13, preference is given to tasks that have data local to the worker node. Finally, if at least one map has finished, reduce tasks are launched as required.

In some cases, the amount of slots available for allocation is less than required minima for job J and then J is allocated only a fraction of required resources. As time progresses, the resource allocations are recomputed during the job's execution and adjusted if necessary as shown in Lines 22-24 (this is a very powerful feature of the scheduler that can increase resource allocation if the job execution progress is behind the targeted and expected one). Whenever a worker reports a completed task, we decrement N_M^J or N_R^J in the SLO-based model and recompute the minimum number of slots.

7. EVALUATION

In this section, we evaluate the efficiency of the new SLO-scheduler using a set of realistic workloads. First, we motivate our evaluation approach by a detailed analysis of the simulation results. Then we validate the simulation results by performing similar experiments in the 66-node Hadoop cluster.

7.1 Workload

Our experimental workload consists of a set of representative applications that are run concurrently. We can run the same application with different input datasets of varying sizes. A particular application reading from a particular set of inputs is called an application instance. Each application instance can be allocated varying number of map and reduce slots resulting in different job executions. The applications used in our experiments are as follows:

1. **Word count:** This application computes the occurrence frequency of each word in the Wikipedia article history dataset. We use three datasets of sizes: 32GB, 40GB and 43GB.

2. **Sort:** This applications sorts a set of records that is randomly generated. The application uses identity map and identity reduce functions as the MapReduce framework does the sorting. We consider three instances of Sort: 8GB, 64GB and 96GB.

3. **Bayesian classification:** We use a step from the example of Bayesian classification trainer in Mahout[8]. The mapper that extracts features from the input corpus and outputs the labels along with a normalized count of the labels. The reduce performs a simple addition of the counts and is also used as the combiner. The input dataset is the same Wikipedia article history dataset, except the chunks split at page boundaries.

4. **TF-IDF:** The Term Frequency - Inverse Document Frequency application is often used in information retrieval and text mining. It is a statistical measure to evaluate how important a word is to a document. We used the TF-IDF example from Mahout and used the same Wikipedia articles history dataset.

5. **WikiTrends:** This application is described in detail in Section 4.1, since it is used in the initial evaluation.

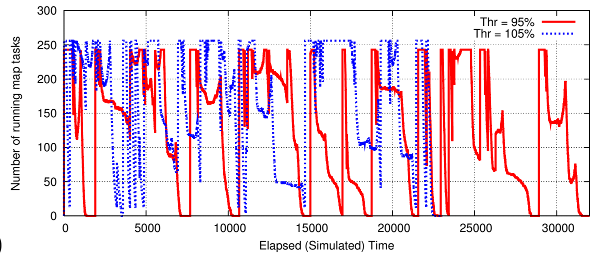
6. **Twitter:** This application uses the 25GB twitter dataset created by [12] containing an edge-list of twitter userids. Each edge (i, j) means that user i follows user j. The Twitter application counts the number of asymmetric links in the dataset, that is, $(i, j) \in E$, but $(j, i) \notin E$. We use three instance processing 15GB, 20GB and 25GB respectively.

7.2 Simulation

We implement a discrete event simulator in order to understand the efficacy of our scheduling and resource allocation algorithm. We do not simulate details of worker nodes (their hard disks or network packet transfers) as it is done in MRPerf [19], because we use job profiles to represent job latencies during different phases of MapReduce processing in the cluster. We concentrate on simulating the job master decisions and the task/slot allocations across multiple jobs.

We maintain data structures similar to the Hadoop job master such as the *job queue*: a priority queue of jobs sorted by the earliest deadline first. Since the slot allocation algorithm makes a new decision when a map or reduce task completes, we simulate the jobs at the task level. The simulator maintains priority queues for *seven event types*: job arrivals and departures, map and reduce task arrivals and departures, and a timer event (used for accounting purposes).

[8] `http://http://mahout.apache.org/`

a) 

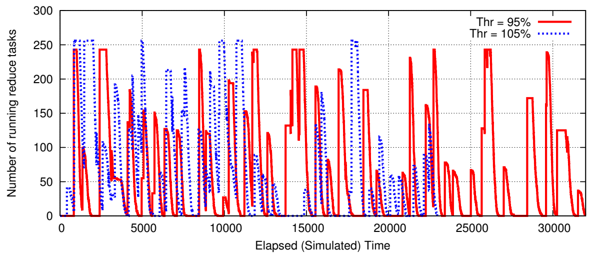
b) 

Figure 7: Cluster load over time in the simulations with *Yahoo!* workload: a) map slots, b) reduce slots.

In the simulations (and later in the testbed evaluations), we will assess the quality of scheduling and resource allocations decisions by observing the following metric. Let the execution consist of a given set of n jobs $J_1, J_2, \ldots, J_n$ with corresponding deadlines $D_1, D_2, \ldots, D_n$, and known job profiles. Let these jobs be completed at times $T_1, T_2, \ldots, T_n$, and let Θ be the set of all jobs whose deadline has been exceeded. Then we compute the following utility function:

$$\sum_{J \in \Theta} \frac{T_J - D_J}{D_J}$$

This function denotes the the sum of the *relative deadlines exceeded*. We are interested in minimizing this value.

We perform simulations with two different workloads denoted as W_1 and W_2.

Workload W_1 represents a mix of the six realistic applications with different input dataset sizes as introduced and described in Section 7.1. We refer to W_1 as the *testbed workload*. It aims to simulate the workload that we use for SLO-scheduler evaluation in our 66-node Hadoop testbed.

Workload W_2 (called *Yahoo!* workload) represents a mix of MapReduce jobs that is based on the analysis of the M45 Yahoo! cluster [10], and that is generated as follows:

- The job consists of the number of map and reduce tasks defined by the distributions $N(154, 558)$ and $N(19, 145)$ respectively, where $N(\mu, \sigma)$ is the *normal distribution* with mean μ and standard deviation σ.
- Map task durations are defined by $N(100, 20)$ and reduce task durations are from $N(300, 30)$, because reduce tasks are usually longer than map tasks since they perform shuffle, sort and reduce. (The study in [10] did not report the statistics for individual map and reduce task durations, so we chose them as described above).
- The job deadline (which is relative to the job completion time) is set to be uniformly distributed in the interval $[T_J, 3 \cdot T_J]$, where T_J is the completion time of job J given all the cluster resources.

The job deadlines in W_1 and W_2 are generated similarly. Table 2 provides the summary of W_1 and W_2.

Set	App	Number of map tasks	Map task duration	Reduce task duration
W1	Bayes	54, 68, 72	436s	33s
	Sort	256, 512, 1024	9s	53s
	TF-IDF	768	11s	66s
	Twitter	294, 192, 390	59s	65s
	Wikitrends	71, 720, 740	179s	79s
	WordCount	507, 640, 676	56s	21s
W2	Yahoo!	$N(154, 558)$	$N(100, 20)$	$N(300, 30)$

Table 2: Simulation workloads W_1 and W_2.

To understand the effectiveness of resource allocation by the SLO-scheduler and the quality of its decisions we aim to create varying load conditions. However, instead of spawning jobs with an inter-arrival distribution, we spawn a new job so as to keep the load in the cluster below a certain threshold. Since our goal is to evaluate the efficiency of the SLO-scheduler decisions, we would like to generate the arrival of jobs that have realistic chances of completing in time. If the cluster does not have enough resources for processing of the newly arrived job then the scheduler might fail to allocate sufficient resources, and the job might not be able to meet its deadline. However, in this case, it is not the scheduler's fault. Moreover, as we will demonstrate later, there is a drastic difference between the peak and average load measured in the simulated cluster over time, which provides an additional motivation to design a job arrival process driven by the load threshold. We define the load as the sum of the percentage of running map and reduce tasks compared to the total cluster capacity. We spawn a new job whenever its minimum pair computed by the Lagrange method combined with the current cluster load is below the threshold.

The simulator was very helpful in designing the ARIA evaluation approach. We were able to generate, execute, and analyze many different job arrival scenarios and their outcomes before deciding on the job arrivals driven by a threshold. Simulations take a few minutes compared to multi-hour executions of similar workloads in the real testbed.

Table 3 summarizes the results of our simulations for *Yahoo!* and *testbed* workloads with 100 jobs. Each experiment was repeated 100 times. In the table, we report the simulation results averaged across these 100 runs.

Load threshold for arrival (%)	SLO exceeded utility (%)		# of jobs with missed SLO		Average Load (%)	
Workload	W1	W2	W1	W2	W1	W2
105	3.31	12.81	0.54	5.21	28.15	34.31
100	1.41	4.65	0.43	3.54	26.51	33.30
95	0	0	0	0	25.21	30.77
90	0	0	0	0	24.70	29.43
85	0	0	0	0	23.52	28.47

Table 3: Simulation results with *testbed* ($W1$) and *Yahoo!* ($W2$) workloads.

We observe that allocation decisions of the SLO-scheduler enables the job to efficiently meet the job deadlines when the load threshold is less than 100%. Moreover, even with a higher load threshold (105%) we see that only a few jobs are missing their deadlines. The last column reports the average utilization of the cluster during the runs measured as the percentage of running map and reduce tasks compared to the total cluster capacity. It is significantly lower than the load threshold used for job arrivals. Figure 7 shows the number of running map and reduce tasks in the cluster over time when processing the *Yahoo!* workload for two different simulations: with load threshold of 95% and 105%. It shows that the average load can be a misleading metric to observe: the individual map and reduce slots' utilization might be quite high, but since the reduce tasks do not start till the map tasks are completed for a given job this can lead to a low average utilization in the cluster. A similar situation is observed for simulations with *testbed* workload (W_1). We omit the figure due to lack of space.

We observe the similarity of simulation results for two quite different workload sets, that makes us to believe in the generality of presented conclusions.

7.3 Testbed Evaluation of ARIA

For evaluating ARIA and validating the efficiency of resource allocation decisions of the new SLO-scheduler in our 66-node Hadoop cluster, we used applications described in Section 7.1 that constitute the *testbed workload* W_1 as summarized in Table 2.

First, we executed each of the applications in isolation with their different datasets and three different map and reduce slot allocations. This set of experiments was run three times and the job profiles and the variation in the averages was less than 10% (up to 20% variation was seen in the maxima and minima). These experiments and the results are similar to ones we performed in our initial performance evaluation study presented in Section 4. Then, we also executed these applications along with each other, and the extracted job profiles show a slightly higher variation while still being very close to the earlier extracted job profiles.

Using the same evaluation approach and technique designed during our simulation experiments and described in detail in Section 7.2, we maintain the load on the testbed cluster below a certain threshold for generating varying load conditions. To assess the quality of scheduling and resource allocation decisions, we observe the number of jobs exceeding deadlines and measure the relative deadlines exceeded. The results are summarized in Table 4.

Load threshold for arrival (%)	SLO exceeded utility (%)	# of jobs with missed SLOs	Average Load (%)
105	7.62	1	29.27
100	4.58	1	27.34
95	0	0	26.46
90	0	0	25.63
85	0	0	24.39

Table 4: Results of *testbed workload* execution.

We observe that a very few jobs miss their deadlines for load threshold above 100% with relatively low numbers for exceeded deadlines. We also measured the accuracy of job completions with respect to the given jobs deadlines and observe that they range in between 5%-15%, which is slightly worse than the simulation results, but close to our initial evaluations presented in Figure 5. Average utilization measured in the testbed is very close to simulation results, that further validates the accuracy of our simulator.

The performance overhead of the scheduling in ARIA is negligible: it takes less than 1 second for scheduling 500 jobs on our 66 node cluster. Likewise, the logging/accounting infrastructure is enabled by default on production clusters and can be used for generating job profiles.

We do not compare our new scheduler with any other existing schedulers for Hadoop or proposed in literature, because all these schedulers have very different objective functions for making scheduling decisions. For example, it would not be useful to compare our scheduler with the Hadoop Fair Scheduler (HFS) [21] because HFS aims to support a fair resource allocation across the running jobs and does not provide the targeted resource allocations for meeting the jobs SLOs. As a result, the HFS scheduler might have a high number of jobs with missed deadlines and arbitrarily high SLO-exceeded utility function.

8. RELATED WORK

Job scheduling and workload management in MapReduce environments is a new topic, but it has already received much attention. Originally, MapReduce (and its open source implementation Hadoop) was designed for periodically running large batch workloads. With a primary goal of minimizing the makespan of these jobs the simple *FIFO* scheduler (initially used in these frameworks) is very efficient. As the number of users sharing the same MapReduce cluster increased, a new *Capacity* scheduler [2] was introduced to support more efficient cluster sharing. Capacity scheduler partitions the resources into pools and provides separate queues and priorities for each pool. However, within the pools, there are no additional capabilities for performance management of the jobs.

In order to maintain fairness between different users, the *Hadoop Fair Scheduler* (HFS) [21] was proposed. It allocates equal shares to each of the users running the MapReduce jobs, and also tries to maximize data locality by delaying the scheduling of the task, if no local data is available. Similar fairness and data locality goals are pursued in *Quincy* scheduler [8] proposed for the Dryad environment [7]. The authors design a novel technique that maps the fair-scheduling problem to the classic problem of min-cost flow in a directed graph to generate a schedule. While both HFS and Quincy allow fair sharing of the cluster among multiple users and their applications, these schedulers do not provide any special support for achieving the application performance goals and the service level objectives (SLOs).

FLEX [20] extends HFS by proposing a special slot allocation schema that aims to optimize explicitly some given scheduling metric. FLEX relies on the speedup function of the job (for map and reduce stages) that produces the job execution time as a function of the allocated slots. This function aims to represent the application model, but it is not clear how to derive this function for different applications and for different sizes of input datasets. FLEX does not provide a technique for job profiling and detailed MapReduce performance model, but instead uses a set of simplifying assumptions about the job execution, tasks durations and job progress over time. The authors do not offer a case study to evaluate the accuracy of the proposed approach and models in achieving the targeted job deadlines.

Dynamic proportional share scheduler [17] allows users to bid for map and reduce slots by adjusting their spending over time. While this approach enables dynamically controlled resource allocation, it is driven by economic mechanisms rather than a performance model and/or application profiling. Polo et al. [16] introduce an online job completion time estimator which can be used for adjusting the resource allocations of different jobs. However, their estimator tracks the progress of the map stage alone and has no information or control over the reduce stage. Ganapathi et al. [4] use Kernel Canonical Correlation Analysis to predict the performance of MapReduce workloads. However, they concentrate on Hive queries and do not attempt to model the actual execution of the MapReduce job. The authors discover the feature vectors through statistical correlation.

Morton et al. [14] propose *ParaTimer* for estimating the progress of parallel queries expressed as Pig scripts that can translate into directed acyclic graphs (DAGs) of MapReduce jobs. Instead of detailed job profiling that is designed in our work, the authors rely on earlier debug runs of the query for estimating throughput of map and reduce stages on the user input data samples. The approach relies on a simplistic assumption that map (reduce) tasks of the same job have the same duration. It is not clear how the authors measure the duration of reduce tasks (what phases of the reduce task are included in the measured duration), especially since the reduce task durations of the first wave and later waves are very different. Usage of the FIFO scheduler limits the approach applicability for progress estimation of multiple jobs running in the cluster with a different Hadoop scheduler.

Phan et al. [15] aim to build an off-line optimal schedule for a set of MapReduce jobs with given deadlines by detailed task ordering of these jobs. The scheduling problem is formulated as a constraint satisfaction problem (CSP). The authors assume that every (map or reduce) task duration of every job is known in advance. MapReduce jobs with a single map and reduce waves are considered. There are some other simplifications in MapReduce job processing where the data transfer (shuffle and sort) is considered as a separate (intermediate) phase between map and reduce tasks while in reality the shuffle phase overlaps significantly with map stage. All these assumptions and the CSP complexity issues make it difficult to generalize the proposed approach.

Originally, Hadoop was designed for homogeneous environment. There has been recent interest [22] in heterogeneous MapReduce environments. Our approach and the proposed SLO-scheduler will efficiently work in heterogeneous MapReduce environments. In a heterogeneous cluster, the slower nodes would be reflected in the longer tasks durations, and they all would contribute to the average and maximum task durations in the job profile. While we do not explicitly consider different types of nodes, their performance is reflected in the job profile and used in the future prediction. As the job progresses, the resource allocations are recomputed during the job's execution and adjusted if necessary. So if occasionally, the job was assigned a higher percentage of slots residing on the slower nodes then the scheduler compensates for the bad performance and slower job progress by adding extra slots for processing. This is a very powerful feature of our scheduler that can increase resource allocation if the job execution progress is behind the targeted and expected one. So, while we did not explicitly target the heterogeneous environment, our approach will efficiently work in heterogeneous Hadoop clusters as well.

Much of the recent work also focuses on anomaly detection, stragglers and outliers control in MapReduce environments [11, 18, 1] as well as on optimization and tuning cluster parameters and testbed configuration [6, 9]. While this work is orthogonal to our research, these results are important for performance modeling in MapReduce environments. Providing more reliable, well performing, balanced environment enables reproducible results, consistent job executions and supports more accurate performance modeling.

9. CONCLUSION

In the enterprise setting, sharing a MapReduce cluster among multiple applications is a common practice. Many of these applications need to achieve performance goals and SLOs, that are formulated as the completion time guarantees. In this work, we propose a novel framework *ARIA* to address this problem. It is based on the observation that we can profile a job that runs routinely and then use its profile in the designed MapReduce performance model to estimate the amount of resources required for meeting the deadline. These resource requirements are enforced by the SLO-scheduler. Our current job ordering is inspired by the EDF scheduling which is a "champion" policy for real-time processing. However, the deadline scheduling in MapReduce environments has a few significant and interesting distinctions as compared to the traditional assumptions. MapReduce job execution time depends on the amount of map and reduce slots allocated to the job over time. One can "speed-up" the job completion by increasing the resource allocation and it can be done during both map and reduce stages. This creates new opportunities for different scheduling policies and their performance comparison.

The proposed performance model is designed for the case without node failures. The next step is to extend this performance model for incorporating different failure scenarios and estimating their impact on the application performance and achievable "degraded" SLOs. We intend for apply designed models for solving a broad set of problems related to capacity planning of MapReduce applications and the analysis of various resource allocation trade-offs in the SLOs support.

10. REFERENCES

[1] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *Proc. of OSDI'2010.*

[2] Apache. Capacity Scheduler Guide, 2010. URL `http://hadoop.apache.org/common/docs/r0.20.1/capacity_scheduler.html`.

[3] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51 (1):107–113, 2008.

[4] A. Ganapathi, Y. Chen, A. Fox, R. Katz, and D. Patterson. Statistics-driven workload modeling for the cloud. In *Proc. of 5th Intl. Workshop on Self Managing Database Systems*, 2010.

[5] R.L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Tech. Journal*, 45(9):1563–1581, 1966.

[6] Intel. Optimizing Hadoop* Deployments, 2010. URL `http://communities.intel.com/docs/DOC-4218`.

[7] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS OS Review*, 41(3):72, 2007.

[8] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proc. of SOSP'2009.*

[9] K. Kambatla, A. Pathak, and H. Pucha. Towards optimizing hadoop provisioning in the cloud. In *Proc. of the First Workshop on Hot Topics in Cloud Computing*, 2009.

[10] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan. An Analysis of Traces from a Production MapReduce Cluster. In *Proc. of CCGrid'2010.*

[11] A. Konwinski, M. Zaharia, R. Katz, and I. Stoica. X-tracing Hadoop. *Hadoop Summit*, 2008.

[12] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *Proc. of WWW'2010.*

[13] O. O'Malley and A.C. Murthy. Winning a 60 second dash with a yellow elephant, 2009.

[14] K. Morton, M. Balazinska, D. Grossman.ParaTimer: a progress indicator for MapReduce DAGs. In *Proc. of SIGMOD'2010.*

[15] L. Phan, Z. Zhang, B. Loo, and I. Lee. Real-time MapReduce Scheduling. *Tech. Report No. MS-CIS-10-32, UPenn*, 2010.

[16] J. Polo, D. Carrera, Y. Becerra, J. Torres, E. Ayguadé, M. Steinder, and I. Whalley. Performance-driven task co-scheduling for MapReduce environments. In *12th IEEE/IFIP Network Operations and Management Symposium.* ACM, 2010.

[17] T. Sandholm and K. Lai. Dynamic Proportional Share Scheduling in Hadoop. *LNCS: Proc. of the 15th Workshop on Job Scheduling Strategies for Parallel Processing*, 2010.

[18] J. Tan, X. Pan, S. Kavulya, E. Marinelli, R. Gandhi, and P. Narasimhan. Kahuna: Problem Diagnosis for MapReduce-based Cloud Computing Environments. In *12th IEEE/IFIP NOMS*, 2010.

[19] G. Wang, A.R. Butt, P. Pandey, and K. Gupta. A simulation approach to evaluating design decisions in MapReduce setups. In *Proc of MASCOTS'2009.*

[20] J. Wolf, et al. FLEX: A Slot Allocation Scheduling Optimizer for MapReduce Workloads. In *Proc.of Middleware'2010.*

[21] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proc. of EuroSys*, pages 265–278. ACM, 2010.

[22] M. Zaharia, A. Konwinski, A.D. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI*, 2008.