Deadline-based Workload Management for MapReduce Environments: Pieces of the Performance Puzzle

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

Vijay S. Kumar
Hewlett-Packard Labs Palo Alto, CA, US.
vijay.s.kumar@hp.com

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

***Abstract*—Hadoop and the associated MapReduce paradigm, has become the de facto platform for cost-effective analytics over "Big Data". There is an increasing number of MapReduce applications associated with live business intelligence that require completion time guarantees. There is a lack of performance models and workload analysis tools for automated performance management of such MapReduce jobs. In this work, we introduce and analyze a set of complementary mechanisms that enhance workload management decisions for processing MapReduce jobs with deadlines. The three mechanisms we consider are the following: *1)* a policy for job ordering in the processing queue; *2)* a mechanism for allocating a tailored number of map and reduce slots to each job with a completion time requirement; *3)* a mechanism for allocating and deallocating (if necessary) spare resources in the system among the active jobs. We analyze the functionality and performance benefits of each mechanism via an extensive set of simulations over diverse workload sets. The proposed mechanisms form the integral pieces in the performance puzzle of automated workload management in MapReduce environments. We implement a novel deadline-based Hadoop scheduler that integrates all these three mechanisms and provides an efficient support for serving MapReduce jobs with deadlines. The results of our simulation study are validated through experiments on a 66-node Hadoop cluster.**

***Keywords*-MapReduce; Performance; Resource Allocation.**

I. Introduction

Hadoop, and the associated MapReduce paradigm, has become the compelling choice for performing advanced analytics over unstructured information and enabling efficient "Big Data" processing. There is an increasing number of MapReduce applications, e.g., personalized advertising, sentiment analysis, spam detection, real-time event log analysis, etc., that require completion time guarantees and are deadline-driven. In an enterprise setting, users share Hadoop clusters and benefit from processing a diverse variety of applications over the same or different datasets. None of the existing Hadoop schedulers support completion time guarantees. There are a few research efforts that suggest different approaches for addressing this goal [1]–[3].

In this work, we continue a direction that is initiated in ARIA [1]. First of all, many production jobs are run periodically on new data. We take advantage of this observation, and for a job that is routinely executed on a new dataset, we automatically build its job profile that reflects critical performance characteristics of the underlying application during all the execution phases: map, shuffle, sort, and reduce phases. Our profiling technique does not require any modifications or instrumentation of either the application or of the underlying Hadoop execution engine. All this information can be obtained from the counters at the job master during the job's execution or alternatively parsed from the logs.

Second, using the knowledge about the job profiles, we design a set of MapReduce performance models with complementary functionality: *i)* for a given job, we can estimate the job completion time as a function of allocated resources, and *ii)* for a given job with a specified soft deadline (job's SLO), we can estimate the amount of map and reduce slots required for completing the job within the deadline.

In this paper, we introduce and analyze a set of complementary mechanisms that enhance workload management decisions for processing MapReduce jobs with deadlines. The three mechanisms we consider are the following:

1) *An ordering policy for the jobs in the processing queue.* For example, even if no profiling information is available about the arriving MapReduce jobs, one can utilize the job deadlines for ordering. The job ordering based on the EDF policy (*Earliest Deadline First*) was successfully used in real-time processing. The EDF job ordering might be used with a default resource allocation policy in Hadoop, where the maximum number of available map (or reduce) slots is allocated to each job at the head of the queue. The possible drawback of this scheme is that in many cases, it is impossible to preempt/reassign the already allocated resources to a newly arrived job with an "earlier" deadline without killing the running tasks.

2) *A mechanism for allocating a tailored number of map and reduce slots* to each job for supporting the job completion goals. If the job profiling information is available, then our resource allocation policy can be much more precise and intelligent: for each job with a specified deadline, we can estimate and allocate the appropriate number of map and reduce slots required for completing the job within the deadline. The interesting feature of this mechanism is that as the time progresses and the job deadline gets closer, the introduced mechanism can recompute (and adjust) the amount of resources needed by each job to meet its deadline.

3) *A mechanism for allocating and deallocating (if necessary) spare resources* in the system among the active jobs. Assume that a cluster has spare resources, i.e., unallocated map and reduce slots left after each job was assigned its minimum resource quota for meeting a given deadline. It would be beneficial to design a mechanism that allocates these spare resources among the running jobs to improve the Hadoop cluster utilization and its performance. The main challenge in designing such a mechanism is accurate decision making on how the slots in the cluster should be

re-allocated or de-allocated to the newly-arrived job with an earlier deadline. The naïve, straightforward approach could de-allocate the spare resources by cancelling their running tasks, and then by re-allocating these slots to the new job. However, it may lead to undesirable churn in resource allocation and wasted, unproductive usage of cluster resources. In this paper, we introduce a novel mechanism that enables a scheduler to accurately predict whether the cluster will have a sufficient amount of released resources over time for the new job to be completed within its deadline. The mechanism exploits the job profile information for making the prediction. It uses a novel modeling technique to avoid cancelling the currently running tasks if possible. The mechanism de-allocates the spare slots (i.e., cancels the execution of extra tasks above the minimum resource quota for each job) only when the amount of released resources over time does not guarantee a timely completion of the newly arrived job.

We implement a novel deadline-based Hadoop scheduler that integrates all the three mechanisms. We analyze the functionality and performance benefits of each mechanism via an extensive set of simulations over diverse workload sets. The analysis presents a set of performance metrics that reflect the quality of job scheduling and slot allocation decisions provided by these different mechanisms. The solution that integrates all the three mechanisms is a clear winner in providing the most efficient support for serving MapReduce jobs with deadlines. We observe a similarity in simulation results for two different workload sets, that leads us to believe in the generality of presented conclusions. The results of our simulation study are validated through experiments on a 66-node Hadoop cluster. The remainder of the paper presents our results in more detail.

II. Background

This section provides a basic background on the MapReduce framework and its open source implementation Hadoop. We also briefly outline the performance modeling approach introduced in ARIA [1] and used in this work.

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 (default size is 64 MB). 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 *shuffle/sort* and *reduce* phases. In the *shuffle/sort phase*, the reduce tasks fetch the intermediate data files from map tasks. 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. Note, that the reduce tasks can process their data only after all the map tasks are completed. 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 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 scheduling policy, the master assigns map and reduce tasks to slots in the cluster. Currently, none of the existing Hadoop schedulers (e.g., FIFO, *Capacity* scheduler [4], or *Hadoop Fair Scheduler* [5]) are designed to support MapReduce jobs with completion time goals. There were a few research efforts that suggest different approaches for addressing this goal [1]–[3].

In this work, we continue a direction that is initiated in ARIA [1]. The proposed MapReduce performance model [1] evaluates lower and upper bounds on the job completion time. It is based on a general model for computing performance bounds on makespan of a given set of n tasks that are processed by k servers (e.g., n map tasks are processed by k slots in MapReduce environment). Let $T_1, T_2, \ldots, T_n$ be the duration of n tasks in a given set. 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 an online, *greedy* algorithm: assign each task to the slot which finished its running task the earliest. Let avg and max be the *average* and *maximum* duration of the n tasks respectively. Then the makespan of a greedy task assignment is at least $(n \cdot avg)/k$ and at most $(n-1) \cdot avg/k + max$. These lower and upper bounds on the completion time can be easily computed if we know the average and maximum durations of the set of tasks and the number of allocated slots.

As motivated by the above model, in order to approximate the overall completion time of a MapReduce job, we need to estimate the *average* and *maximum* task durations during different execution phases of the job, i.e., map, shuffle/sort, and reduce phases. Measurements such as M_{avg}^j (R_{avg}^j), the average map (reduce) task duration for a job j can be obtained from the execution logs that record past job executions. In our earlier paper [1], we describe the automated profiling tool that extracts a compact MapReduce job profile from the past job executions. By applying the outlined bounds model, we can express the estimates for job completion time (lower bound T_J^{low} and upper bound T_J^{up}) as a function of map/reduce tasks (N_M^J, N_R^J) and the allocated map/reduce slots (S_M^J, S_R^J) using the following equation form:

$$T_J^{low} = A_J^{low} \cdot \frac{N_M^J}{S_M^J} + B_J^{low} \cdot \frac{N_R^J}{S_R^J} + C_J^{low} \qquad (1)$$

The equation for T_J^{up} can be written in a similar form (for details, see [1]). Typically, the average of lower and upper bounds (T_J^{avg}) is a good approximation of the job completion time.

Note that once we have a technique for predicting the job completion time, it also can be used for solving the inverse problem: finding the appropriate number of map and reduce slots that could support a given job deadline. Equation 1 yields a hyperbola if S_M^J and S_R^J are the variables. All integral points on this hyperbola are possible allocations of map and reduce slots which result in meeting the same deadline. There is a point where the sum of the required map and reduce slots is minimized. We calculate this minima on the curve using Lagrange's multipliers [1], since we would like to conserve (minimize) the number of map and reduce slots required for the adequate resource allocation per job.

III. Three Pieces of the Puzzle in Deadline-Based Workload Management

In this section, we introduce a set of complementary mechanisms that enhance the scheduling and resource allocation decisions for processing MapReduce jobs with deadlines. The three mechanisms considered are the following:

1) the job ordering in the processing queue;
2) the amount of resources (the number of map and reduce slots) that are allocated for each job processing;
3) the policy for allocating and deallocating the spare resources in the system among the active jobs.

In this section, we discuss the intuition and logic behind these mechanisms as well as their implementation details.

A. *Job Ordering Policy*

The job ordering in workload management emphasizes solely the ordering of jobs to achieve performance enhancements. For example, real-time operating systems employ a dynamic scheduling policy called Earliest Deadline First (EDF) which is one of traditional (textbook) scheduling policies for jobs with deadlines. EDF is an optimal scheduling algorithm on preemptive uniprocessors [6] in the following sense: if a collection of independent jobs, each characterized by an arrival time, an execution requirement, and a deadline, can be scheduled (by any algorithm) such that all the jobs complete by their deadlines, then EDF will schedule this set of jobs such that they all complete by their deadlines.

The nature of MapReduce job processing differs significantly from the traditional EDF assumptions. None of the known classic results are directly applicable to job/task scheduling with deadlines in MapReduce environments, where the job completion time is a function of allocated resources (as was demonstrated in Section II). Therefore, just using the EDF job ordering as a basic mechanism for deadline-based scheduling in MapReduce environments will not alone be sufficient to support the job completion time guarantees. The next section discusses the additional mechanism that aims to enhance the EDF job ordering by enabling a tailored resource allocation for a MapReduce job with a given deadline and a known job profile.

B. *Resource Allocation Policy*

Job scheduling in Hadoop is performed by a master node, which manages a number of worker nodes in the cluster. As we discussed in Section II, the job ordering defines which job should be processed next by the master. In addition, the scheduling policy of the job master should decide how many map/reduce slots should be allocated to a current job.

A simple default resource allocation policy in Hadoop assigns the maximum number of map (or reduce) slots for each job in the queue. For example, the original FIFO scheduler in Hadoop follows this policy. Thus, each job is allocated the maximum amount of resources defined either by the maximum number of slots available in the cluster or by the maximum number of map/reduce tasks comprising the job (if it is smaller than the number of available slots in the cluster). Let us denote the policy with Earliest Deadline First job ordering and the default resource allocation as just *EDF*. This policy reflects the performance that can be achieved when there is no additional knowledge about performance characteristics of the arriving MapReduce jobs. However, the possible drawback of this default policy is that it always allocates the maximum resources to each job, and does not try to tailor the appropriate amount of resources that is necessary for completing the job within its deadline. Therefore, in many cases, it is impossible to preempt/reassign the already allocated resources (without killing the running tasks) to provide resources for a newly arrived job with an earlier deadline.

In the case where the job profiles are known, we can use this additional knowledge in performance modeling for the accurate estimates of map and reduce slots required for completing the job within the deadline. We call the mechanism that allocates the minimal resource quota required for meeting a given job deadline as *MinEDF*. The interesting and powerful feature of this mechanism is that as the time progresses and the job deadline gets closer to the current time, the introduced mechanism can recompute and adjust the amount of resources needed to each job to meet its deadline. So, in essence, *MinEDF* aims to dynamically allocate the minimum sufficient resources to the job for completing within the deadline while leaving the remaining, spare resources to the future arriving jobs.

C. *Allocating and De-allocating Spare Cluster Resources Among the Active Jobs*

When there is a large number of jobs competing for cluster resources the mechanism that allocates only the minimal quota of map and reduce slots for meeting job deadlines is appealing and may seem like the right approach. However, assume that a cluster has spare resources, i.e., unallocated map and reduce slots left after each job has been assigned its minimum resource quota. Then, the question is whether we could design a mechanism that allocates these spare resources among the currently active jobs to improve the Hadoop cluster utilization and its performance, but in case of a new job arrival with an earlier deadline, these slots can be dynamically de-allocated (if necessary) to service the newly-arrived job with an earlier deadline.

Extracted job profiles can be used for two complementary goals. First, they can be used for performance modeling of the job completion time and required resources. Second, since the job profiles provide information about map/reduce task durations, these metrics can be used to estimate when the allocated map/reduce slots are going to be released back to the job master for re-assignment. This way, we have a powerful modeling mechanism that enables a scheduler to predict whether the newly arrived job (with a deadline earlier than deadlines of some actively running jobs) can be completed in time by simply waiting while some of the allocated slots finish processing their current tasks before being re-allocated to the newly arrived job. If the prediction returns a negative answer, i.e., the amount of released resources over time does not guarantee a completion of the newly arrived job with a given deadline, then the scheduler makes a decision of how many of the slots (as well as which ones) should cancel processing their tasks, and be re-allocated to the new job immediately. For this cancellation procedure, the scheduler only considers the so-called spare slots, i.e., the slots that do not belong to the minimum quota of slots allocated to the currently running jobs for meeting their deadlines.

This designed mechanism further enhances the *MinEDF* functionality to efficiently utilize spare cluster resources. It resembles work-conserving scheduling, so we refer to it as *MinEDF-WC*. We have implemented *MinEDF-WC* as a novel deadline-based Hadoop scheduler that integrates all the three mechanisms and provides an efficient support for serving MapReduce jobs with deadlines. Algorithm 1 reflects how these three mechanisms work together as integral parts of the intelligent workload management in MapReduce environments. The pseudo-code shown in Algorithm 1 presents job scheduling and detailed slot allocation scheme.

IV. Evaluation

In this section, we analyze performance benefits of the three mechanisms introduced in Section III by comparing performance of *EDF*, *MinEDF*, and *MinEDF-WC*. Note, that the *EDF* scheduler uses only one out of three mechanisms: it applies EDF job ordering with a default resource allocation policy. The *MinEDF* scheduler uses two out of three introduced mechanisms: in addition to EDF job ordering it utilizes the mechanism for allocating a tailored amount of map and reduce slots to each job for meeting its deadline. Finally, *MinEDF-WC* represents a scheduler that integrates all the three mechanisms for workload management of jobs with deadlines.

First, we analyze these schedulers and their performance with our simulation environment SimMR [7]. SimMR is comprised of three components: *i)* Trace Generator that creates a replayable MapReduce workload; *ii)* Simulator Engine that accurately emulates the job master functionality in Hadoop; and *iii)* a pluggable scheduling policy that dictates the scheduler decisions on job ordering and the amount of resources allocated to different jobs over time. We perform simulation experiments with two workload sets:

1) A synthetic trace generated with statistical distributions that characterize the Facebook workload, and

Algorithm 1 Min Earliest Deadline First - Work Conserving (MinEDF-WC) Algorithm

Input: New Job $\hat{J}$ with deadline $D_{\hat{J}}$, Priority Queue of currently executing *jobs*, Number of free map slots F_M, Number of free reduce slots F_R, Number N_M^j of map tasks and N_R^j of reduce tasks in Job j

```
1:  ON THE ARRIVAL OF NEW JOB $\hat{J}$ :
2:  (MinMaps$_{\hat{J}}$, MinReduces$_{\hat{J}}$) $\leftarrow$ ComputeMinResources($\hat{J}$, $D_{\hat{J}}$)
3:  // Do we have enough resources to meet this job's deadline right now?
4:  if MinMaps$_{\hat{J}} < F_M$ and MinReduces$_{\hat{J}} < F_R$ then return
5:  // Will we have enough resources in the future?
6:  Sort jobs by increasing task durations
7:  for each job $j$ in $jobs$ do
8:      if CompletedMaps$_j < N_M^j$ and MinMaps$_{\hat{J}} > F_M$ then
9:          // Job $j$ is in the Map stage and $\hat{J}$ is short on map slots
10:         ExtraMaps$_j \leftarrow$ RunningMaps$_j -$ MinMaps$_j$
11:         $F_M \leftarrow F_M +$ ExtraMaps$_j$
12:         (MinMaps$_{\hat{J}}$, MinReduces$_{\hat{J}}$) $\leftarrow$ ComputeMinResources($\hat{J}$,
            $D_{\hat{J}}$ - $M_{avg}^j$)
13:         if MinMaps$_{\hat{J}} < F_M$ and MinReduces$_{\hat{J}} < F_R$ then return
14:     else if CompletedMaps$_j = N_M^j$ and MinReduces$_j > F_R$ then
15:         // Job $j$ is in the Reduce stage and $\hat{J}$ is short on reduce slots
16:         ExtraReduces$_j \leftarrow$ RunningReduces$_j -$ MinReduces$_j$
17:         $F_R \leftarrow F_R +$ ExtraReduces$_j$
18:         (MinMaps$_{\hat{J}}$, MinReduces$_{\hat{J}}$) $\leftarrow$ ComputeMinResources($\hat{J}$,
            $D_{\hat{J}}$ - $R_{avg}^j$)
19:         if MinMaps$_{\hat{J}} < F_M$ and MinReduces$_{\hat{J}} < F_R$ then return
20:     end if
21: end for
22: // Not enough resources to meet deadline in future, need to kill tasks
23: for each job $j$ in $jobs$ do
24:     if RunningMaps$_j >$ MinMaps$_j$ then
25:         $F_M \leftarrow F_M +$ RunningMaps$_j -$ MinMaps$_j$
26:         KillMapTasks($j$, RunningMaps$_j -$ MinMaps$_j$)
27:         if MinMaps$_{\hat{J}} < F_M$ then return
28:     end if
29: end for
30: ON RELEASE OF A MAP SLOT:
31: Find job $j$ among $jobs$ with earliest deadline and CompletedMaps$_j <$
    $N_M^j$ and RunningMaps$_j <$ MinMaps$_j$ return $j$
32: if such job $j$ is not found, then return job $j$ with the earliest deadline
    with CompletedMaps$_j < N_M^j$
33: ON RELEASE OF A REDUCE SLOT:
34: Find job $j$ among $jobs$ with earliest deadline and CompletedMaps$_j =$
    $N_M^j$ and CompletedReduces$_j < N_R^j$ and RunningReduces$_j <$
    MinReduces$_j$ return $j$
35: if such job $j$ is not found, then return job $j$ with the earliest deadline
    with CompletedMaps$_j = N_M^j$ and CompletedReduces$_j < N_R^j$
```

2) A real testbed trace comprised of multiple job runs in our 66-node cluster.

Second, we validate the simulation results through experiments on a 66-node Hadoop cluster with the following configuration. Each node is an HP DL145 GL3 machine with four AMD 2.39GHz cores, 8 GB RAM and two 160GB hard disks. The machines are set up in two racks and interconnected with gigabit Ethernet. We use Hadoop 0.20.2 with two machines for JobTracker and NameNode, and remaining 64 machines as worker nodes. Each slave is configured with a single map and reduce slot. The default blocksize of the file system is set to 64MB and the replication level is set to 3. We disable speculation as it did not lead to any significant improvements. Our simulated cluster mimics this 66-node testbed configuration.

The next section describes the set of collected metrics used for schedulers' analysis and evaluation.

A. Metrics

Let the execution consist of a given set of n jobs $(J_1, T_1^{ar}, D_1), (J_2, T_2^{ar}, D_2), \ldots, (J_n, T_n^{ar}, D_n)$, where T_i^{ar} denotes the arrival time of job J_i and D_i denotes the corresponding deadline (relative to the arrival) of job J_i. We assume that the profiles of the jobs $J_1, \ldots, J_n$ are known. Let these jobs be completed at times $T_1^{compl}, T_2^{compl}, \ldots, T_n^{compl}$, and let Θ be the set of all jobs whose deadline has been exceeded. We evaluate the scheduling policies described in the previous section based on the following metrics:

1) **Relative deadline exceeded:** This metric denotes the the sum of the *relative* deadlines exceeded.

$$\sum_{J \in \Theta} (T_J^{compl} - T_J^{ar} - D_J)/D_J$$

2) **Missed-deadline jobs:** It measures a fraction of jobs that have exceeded their deadlines: Θ/n.
3) **Average job completion time:**

$$\sum_{1 \le i \le n} (T_i^{compl} - T_i^{ar})/n$$

4) **Number of spare slot allocations:** This metric only applies to *MinEDF-WC* policy, where a job J might be allocated extra slots in addition to the minimum resource quota computed by the performance model. This metric presents the aggregate number of extra slots allocated across all the jobs over time.
5) **Number of spare slot processing cancellations:** This metric only applies to *MinEDF-WC*. It presents the aggregate number of extra slots allocated and then cancelled during their processing (i.e., with tasks being killed) across all the jobs over time.

B. Simulation Results with Synthetic Facebook Workload

For the Facebook workload, we use the detailed description of MapReduce jobs in production at Facebook in October 2009 provided by Zaharia et. al. [5]. Table I summarizes the number of map and reduce tasks and the number of jobs in the Facebook workload. We use the plot of map and reduce durations in Fig. 1 of [5], and try to identify the statistical distributions which best fits the provided plot. We fit more than 60 distributions such as Weibull, LogNormal, Pearson, Exponential, Gamma, etc. using StatAssist[1]. Our analysis shows that the LogNormal distribution fits best the provided Facebook task duration distributions. $LN(9.9511, 1.6764)$ fits the map task duration distribution with a Kolmogorov-Smirnov value of 0.1056, where $LN(x; \mu, \sigma) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-\frac{(\ln x - \mu)^2}{2\sigma^2}}$ is the LogNormal distribution with mean μ and variance σ. For the reduce task duration, $LN(12.375, 1.6262)$ fits with a Kolmogorov-Smirnov value of 0.0451.

[1]http://www.mathwave.com/help/easyfit/html/tools/assist.html

Bin	Map Tasks	Reduce Tasks	# Jobs Run
1	1	NA	380
2	2	NA	160
3	10	3	140
4	50	NA	80
5	100	NA	60
6	200	50	60
7	400	NA	40
8	800	180	40
9	2400	360	20
10	4800	NA	20

Table I
Job sizes for each bin in Facebook workload (from Table 3 in [5]).

We use these respective LogNormal distributions to generate a synthetic workload (with 1000 jobs) that is similar to a reported Facebook workload. The job deadline (which is relative to the job completion time) is set to be uniformly distributed in the interval $[1 \cdot T_J, 2 \cdot T_J]$, where T_J is the completion time of job J given all the cluster resources (i.e., maximum number of map/reduce slots that job can utilize).

Figures 1 and 2 show the simulation results of processing the generated synthetic Facebook workload with *EDF*, *MinEDF*, and *MinEDF-WC* schedulers while varying the mean of the exponential inter arrival times. The simulation results are averaged over 100 runs.

Note, that performance of the *EDF* scheduler reflects the very basic mechanism performance, i.e., deadline job ordering only (with no additional job profiling available or used). Figure 1(left) shows that *MinEDF* and *MinEDF-WC* schedulers result in a significantly smaller relative deadline exceeded metric compared to the *EDF* scheduler. By adding the mechanism that utilizes job profiling and performance modeling for allocating a tailored amount of resources per job, the scheduler is able to dramatically improve the resource allocation decisions that lead to a much smaller deadline violation even under high job arrival rates. Figure 1(center) shows that *MinEDF-WC* scheduler is able to support the smallest fraction of jobs missing their deadlines. By adding the mechanism that intelligently allocates and deallocates spare resources in the cluster, the scheduler is able to significantly improve the quality of resource allocation decisions and to minimize the number of jobs missing their deadlines. The impact of this mechanism can be also observed in Figure 1(right) that reflects the most improved average completion time under the work-conserving nature of *MinEDF-WC* scheduler. Since *MinEDF* does not utilize spare resources and only allocates the minimum resource quota required for job completion within the deadline, the average job completion time does not improve under *MinEDF* even as the load decreases.

Figure 2 shows the number of extra resources allocated

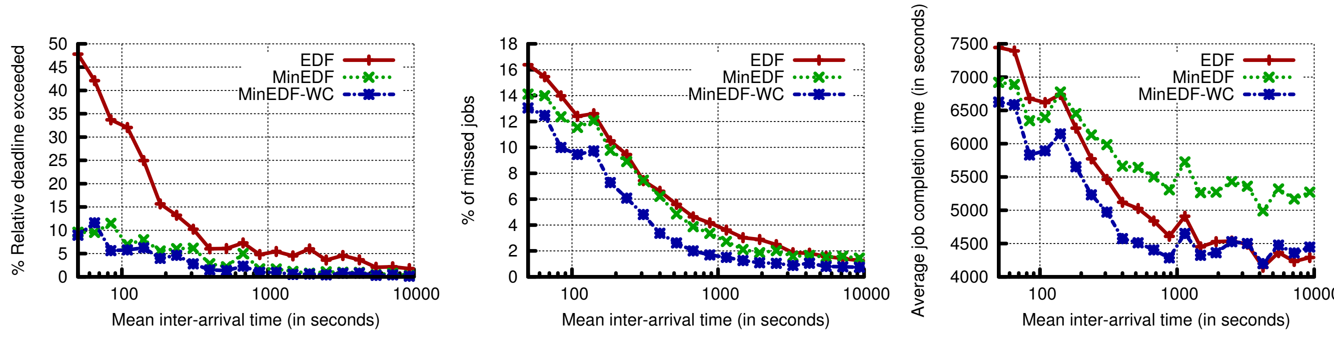


Figure 1. Comparing different metrics for Facebook workload: 1000 jobs averaged over 100 runs with deadline = $[1 \cdot T_J, 2 \cdot T_J]$

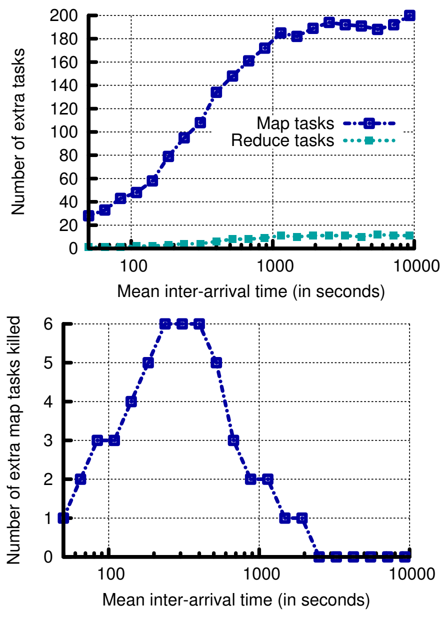


Figure 2. Comparing different metrics for Facebook workload: 1000 jobs averaged over 100 runs with deadline = $[1 \cdot T_J, 2 \cdot T_J]$

and de-allocated by the *MinEDF-WC* scheduler. Under high load (inter-arrival time $\leq$ 100), there are very few spare resources. As the load decreases, the number of spare slot allocations increase. The number of killed tasks is very low at any point: under high load it is small because very few spare resources are available. At medium load some of the allocated extra tasks have to be killed in order to re-assign spare resources to more urgent jobs. At low loads, the scheduler does not need to resort to killing tasks. Indeed, the *MinEDF-WC* scheduler handles spare resources very efficiently and avoids cancelling the currently running tasks whenever possible.

C. *Simulation Results with with Replayed Testbed Workload*

Our testbed workload consists of a set of representative applications executed on three different datasets as follows:

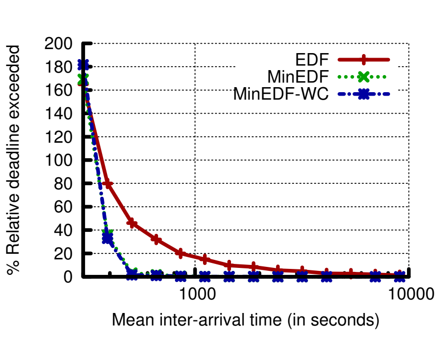
1) **Word count** application computes the occurrence frequency of each word in 32GB, 40GB and 43GB Wikipedia article history dataset.
2) **Sort** application sorts 16GB, 32GB and 64GB of random generated text data.
3) **Bayesian classification** application from Mahout[2] over three sizes Wikipedia article history datasets.
4) **TF-IDF:** the Term Frequency- Inverse Document Frequency application from Mahout used over the same Wikipedia articles history datasets.
5) **WikiTrends** application counts the number of times each article has been visited. Wikipedia article traffic logs were collected (and compressed) every hour in April, May and June 2010.
6) **Twitter:** This application uses the 12GB, 18GB and 25GB twitter dataset created by Kwak et. al. [8] containing an edgelist 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 $(j, i) \notin E$.

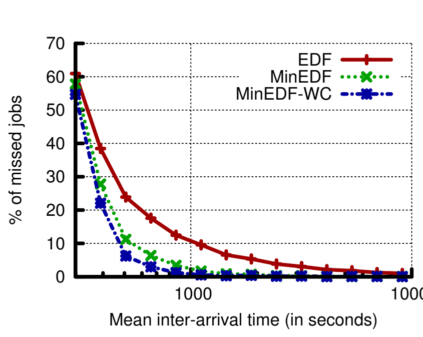
[2]http://mahout.apache.org

For the real workload trace, we use a mix of six described applications with three different input dataset sizes. We run these 18 jobs in our 66-nodes Hadoop testbed, and create the replayable job traces for SimMR. We generate an equally probable random permutation of arrival of these jobs and assume that the inter-arrival time of the jobs is exponential.

Figures 3 and 4 show the simulation results for processing 1000 jobs (consisting of multiple permutations of the 18 jobs described above) under the *EDF*, *MinEDF*, and *MinEDF-WC* schedulers and the deadlines being generated using the interval $[2 \cdot T_J, 4 \cdot T_J]$. The results are averaged over 100 runs.

These simulation results reflect our scheduler's performance over a broader range of loads in the cluster. At very high loads, the cluster is under significant overload, most of the jobs are missing their deadline and the performance of the three schedulers becomes very similar. The jobs that are missing their deadlines start requiring the maximum number of map/reduce slots from the scheduler, and therefore all the three schedulers start behaving more and more like the *EDF* scheduler. At medium loads (inter-arrival times $\geq$ 300s), we see significantly better results under *MinEDF* and *MinEDF-WC* schedulers as compared to *EDF* with respect to two major metrics: lower relative deadline exceeded and very low percentages of jobs with missed deadlines (see Figure 3). The *MinEDF-WC* scheduler is able to support the smallest fraction of jobs missing their deadlines: it accurately tailors required resources per job and intelligently allocates and de-allocates spare resources in the cluster to significantly improve the quality of resource allocation decisions and to maximize the number of jobs meeting





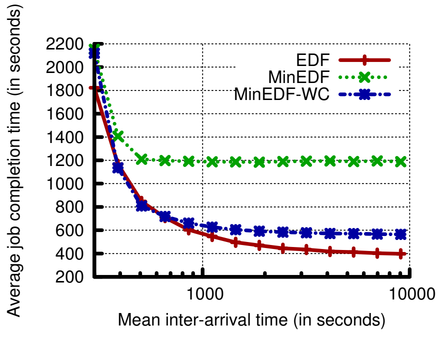
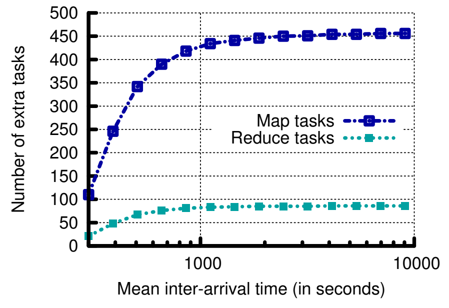


Figure 3. Comparing different metrics for 1000 jobs of Testbed workload, averaged over 100 runs. Deadline = $[2 \cdot T_J, 4 \cdot T_J]$.



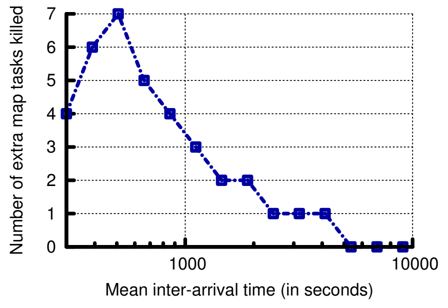


Figure 4. Comparing different metrics for 1000 jobs of Testbed workload, averaged over 100 runs. Deadline = $[2 \cdot T_J, 4 \cdot T_J]$.

Metrics	**EDF**	**MinEDF**	**MinEDF-WC**
Rel. Deadline Exceeded (%)	82.54 (83.79)	63.77 (69.73)	42.38 (46.87)
Missed-deadline jobs(%)	17 (20)	17 (17)	10 (10)
Avg job completion time (s)	1290 (1360)	1090 (1110)	1200 (1240)
# of extra map tasks	NA	NA	324 (336)
# of extra reduce tasks	NA	NA	81 (89)
# of map tasks killed	NA	NA	3.78 (3.94)

Table II
Metrics collected from running applications on Testbed. Values in parentheses show the corresponding metric measured in the simulation.

their deadlines. Figures 3 and 4 with simulated testbed workload show similar trends to the Facebook workload. In summary, the *MinEDF-WC* scheduler (that integrates all three mechanisms) shows superior performance compared to *EDF* and *MinEDF* schedulers.

D. Validating Simulation Results with Testbed Execution

We validate the testbed workload simulation results by executing this workload in our 66-node cluster under three different schedulers (with a chosen inter arrival time for this validation). Table II reports the metrics collected from the testbed executions and corresponding simulations (see values in parentheses). We observe that all the simulation metrics are close to metrics collected by running these applications on the 66-nodes Hadoop cluster (metrics from testbed measurements and metrics from simulations are within 10% of each other). The *MinEDF-WC* scheduler that integrates all three mechanisms for workload management of jobs with deadlines shows the best performance.

V. Related Work

Scheduling of incoming jobs and the assignment of processors to the scheduled jobs has been an important factor for optimizing the performance of parallel and distributed systems. It has been studied extensively in scheduling theory (see a variety of papers and textbooks on the topic [6], [9]–[16]. Designing an efficient distributed server system often assumes choosing the "best" task assignment policy for a given model and user requirements. However, the question of "best" job scheduling or task assignment policy is still open for many models. Typically, the choice of the algorithm is driven by performance objectives.

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 *FIFO* scheduler. As the number of users sharing the same MapReduce cluster increased, a new *Capacity* scheduler [4] was introduced to support more efficient cluster sharing. Capacity scheduler partitions the resources into pools and provides separate queues and priorities for each pool. In order to maintain fairness between different users, the *Hadoop Fair Scheduler* (HFS) [5] 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. While both HFS and Capacity scheduler allow 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 [2] 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.

Morton et al. [17] 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.

Moseley et al. [18] formalize MapReduce scheduling as a generalized version of the classical two-stage flexible flow-shop problem with identical machines. They provide approximate algorithms for minimizing the makespan of a set of MapReduce jobs in the offline and online scheduling scenarios.

Polo et al. [3] 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.

All earlier approaches do not support deadline-based objectives for MapReduce environments, and are quite different from the framework proposed and evaluated in our work.

VI. Conclusion

Design of new schedulers and scheduling policies for MapReduce environments has been an active research topic in industry and academia during the last years. Most of these efforts were driven by the user specific goals and resource utilization management. In this work, we introduce three complementary mechanisms that enhance workload management decisions for processing MapReduce jobs with deadlines. Two of them utilize the novel modeling technique that is based on accurate job profiling and new performance models for tailored resource allocations in MapReduce environments. We analyze the functionality and performance benefits of each mechanism and implement a novel deadline-based Hadoop scheduler that integrates all the three mechanisms. In our extensive simulation study and testbed experiments, we demonstrate significant improvements in quality of job scheduling decisions and completion time guarantees provided by the new scheduler.

In the study, we only consider MapReduce jobs with completion time goals. One may ask, how to incorporate "regular" job processing that does not impose job deadlines, or how to enforce other useful policies such as job priorities, or special policies for allocated extra resources among the particular jobs, etc. We believe that the proposed framework is easily extensible to handle different classes of MapReduce jobs with logically partitioned (or prioritized) cluster resources among them.

References

[1] A. Verma, L. Cherkasova, and R. H. Campbell, "ARIA: Automatic Resource Inference and Allocation for MapReduce Environments," in *Proc. of Intl. Conference on Autonomic Computing (ICAC)*. ACM/IEEE, 2011.

[2] J. Wolf, D. Rajan, K. Hildrum, R. Khandekar, V. Kumar, S. Parekh, K.-L. Wu, and A. Balmin, "FLEX: A Slot Allocation Scheduling Optimizer for MapReduce Workloads," *To appear in Middleware*, 2010.

[3] 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*, 2010.

[4] Apache, "Capacity Scheduler Guide," 2010. [Online]. Available: http://hadoop.apache.org/common/docs/r0.20.1/capacity_scheduler.html

[5] 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*. ACM, 2010, pp. 265–278.

[6] M. Pinedo, *Scheduling: theory, algorithms, and systems*. Springer Verlag, 2008.

[7] A. Verma, L. Cherkasova, and R. H. Campbell, "Play It Again, SimMR!" in *Proc. of Intl. IEEE Cluster 2011 (Cluster'2011)*. Austin, TX, USA, Sept, 2011.

[8] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *Proc. of intl. Conference on World Wide Web*. ACM, 2010, pp. 591–600.

[9] J. Blazewicz, *Scheduling in computer and manufacturing systems*. Springer-Verlag, New York, NJ, USA, 1996.

[10] G. Blelloch, P. Gibbons, and Y. Matias, "Provably efficient scheduling for languages with fine-grained parallelism," *Journal of the ACM (JACM)*, vol. 46, no. 2, pp. 281–321, 1999.

[11] R. Blumofe and C. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM (JACM)*, vol. 46, no. 5, pp. 720–748, 1999.

[12] C. Chekuri and M. Bender, "An efficient approximation algorithm for minimizing makespan on uniformly related machines," *Integer Programming and Combinatorial Optimization*, pp. 383–393, 1998.

[13] C. Chekuri and S. Khanna, "Approximation algorithms for minimizing average weighted completion time," *Handbook of Scheduling: Algorithms, Models, and Performance Analysis. CRC Press, Inc., Boca Raton, FL, USA*, 2004.

[14] B. Lampson, "A scheduling philosophy for multiprocessing systems," *Communications of the ACM*, vol. 11, no. 5, 1968.

[15] J. Leung, *Handbook of scheduling: algorithms, models, and performance analysis*. CRC Press, 2004.

[16] L. Tan and Z. Tari, "Dynamic task assignment in server farms: Better performance by task grouping," in *Computers and Communications, 2002. Proceedings. ISCC 2002. Seventh International Symposium on*. IEEE, 2002, pp. 175–180.

[17] K. Morton, M. Balazinska, and D. Grossman, "ParaTimer: a progress indicator for MapReduce DAGs," in *Proc. of SIGMOD*. ACM, 2010, pp. 507–518.

[18] B. Moseley, A. Dasgupta, R. Kumar, and T. Sarlós, "On scheduling in map-reduce and flow-shops," in *Proc. of SPAA*, 2011.