View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by UPCommons. Portal del coneixement obert de la UPC

2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

Adaptive MapReduce Scheduling in Shared Environments

Jordà Polo, Yolanda Becerra, David Carrera,
Jordi Torres and Eduard Ayguadé
Barcelona Supercomputing Center (BSC) -
Technical University of Catalonia (UPC)

Malgorzata Steinder
IBM T.J. Watson Research Center
Yorktown, NY

***Abstract*—In this paper we present a MapReduce task scheduler for shared environments in which MapReduce is executed along with other resource-consuming workloads, such as transactional applications. All workloads may potentially share the same data store, some of them consuming data for analytics purposes while others acting as data generators. This kind of scenario is becoming increasingly important in data centers where improved resource utilization can be achieved through workload consolidation, and is specially challenging due to the interaction between workloads of different nature that compete for limited resources. The proposed scheduler aims to improve resource utilization across machines while observing completion time goals. Unlike other MapReduce schedulers, our approach also takes into account the resource demands for non-MapReduce workloads, and assumes that the amount of resources made available to the MapReduce applications is variable over time. As shown in our experiments, our proposal improves the management of MapReduce jobs in the presence of variable resource availability, increasing the accuracy of the estimations made by the scheduler, thus improving completion time goals without an impact on the fairness of the scheduler.**

***Keywords*—MapReduce, Scheduling, Distributed, Analytics, Transactional, Adaptive, Availability, Shared Environments**

I. Introduction

In recent years, the industry and research community have witnessed an extraordinary growth in research and development of data-related technologies. In addition to distributed, large-scale data processing workloads such as MapReduce [1], other distributed systems have been introduced to deal with the management of huge amounts of data [2] [3] providing at the same time support for both data-analytics and transactional workloads.

Instead of running these services in completely dedicated environments, which may lead to underutilized resources, it is becoming more common to multiplex different and complementary workloads in the same machines. This is turning clusters and data centers into shared environments in which each one of the machines may be running different applications simultaneously at any point in time: from database servers to MapReduce jobs to other kinds of applications [4]. This constant change is challenging since it introduces higher variability and thus makes performance of these systems less predictable.

In particular, in this paper we consider an environment in which data analytics jobs, such as MapReduce applications, are collocated with transactional workloads. In this scenario, deep coordination between management components is critical, and single applications can not be considered in isolation but in the full context of mixed workloads in which they are deployed. Integrated management of resources in presence of MapReduce and transactional applications is challenging since the demand for transactional workloads is known to be bursty and varying over time, while MapReduce schedulers usually expect that available resources are unaltered over time. Transactional workloads are usually of higher priority than analytics jobs because they are directly linked to the QoS perceived by the users. As such, in our approach transactional workloads are considered as critical and we assume that only resources not needed for transactional applications can be committed to MapReduce jobs.

In this work we present a novel scheduler, the Reverse-Adaptive Scheduler, that allows the integrated management of data processing frameworks such as MapReduce along with other kinds of workloads that can be used for both, transactional and analytics workloads. The scheduler expects that each job is associated with a completion time goal that is provided by users at job submission time. These goals are treated as soft deadlines as opposed to the strict deadlines familiar in real-time environments: they simply guide workload management. We also assume that the changes in workload intensity over time for transactional workloads can be well characterised, as has been previously stated in the literature [5].

Existing previous work on MapReduce scheduling involved estimating the resources that needed to be allocated to each job in order to meet its completion goals [6], [7], [8]. This naive estimation worked fine under the assumption that the total amount of resources remained stable over time. However, in a scenario with consolidated workloads we are targeting a more dynamic environment in which resources are shared with other frameworks and availability changes depending on external and a priori unknown factors. The scheduler proposed in this paper proactively deals with dynamic resource availability while still being guided by completion time goals.

While resource management has been widely studied in MapReduce environments, to our knowledge no previous work has focused on shared scenarios with transactional workloads.

The remaining sections of the paper are organized as follows: We first present a motivating example to illustrate the problem that the proposed scheduler aims to address in

978-1-4799-2784-5/14 $31.00 © 2014 IEEE
DOI 10.1109/CCGrid.2014.65

IEEE computer society

Section II. After that, we provide an overview of the problem in Section III, and then describe our scheduler in Section IV. An evaluation of our proposal is studied in Section V. Finally, we discuss related work in Section VI and conclude in Section VII.

II. Motivating example

Consider a system running two major distributed frameworks: a MapReduce deployment used to run background jobs, and a distributed data-store that handles transactional operations and serves data to a front-end. Both workloads share the same machines, but since the usage of the front-end changes significantly over time depending on the activity of external entities, so does the availability of resources left for the MapReduce jobs. Notice that the demand of resources over time for the front-end activities is supposed to be well characterized [5], and therefore it can be predicted in the form of a given function $f(t)$ known in advance.

In the proposed system, the MapReduce workload consists of 3 identical jobs: J1, J2, and J3. All jobs are submitted at time 0, but have different deadlines: D1 (6.5h), D2 (15h), and D3 (23.1h). Colocated with the MapReduce jobs, we have a fron-end driven transactional workload that consumes available resources over time. The amount of resources committed to the critical transactional workload is defined by the function $f(t)$.

Figure 1 shows the expected outcome of an execution using a MapReduce scheduler that is not aware of the dynamic availability of resources and thus assumes resources remain stable over time. Figure 2 shows the behaviour of a scheduler aware of changes in availability and capable of leveraging the characteristics of other workloads to scheduler MapReduce jobs. In both Figures, the solid thick line represents $f(t)$. Resources allocated to the transactional workload are shown as the white region on top of $f(t)$, while resources allocated to the MapReduce workload are shown below $f(t)$, being each job represented by a different pattern. X-axis shows time, while Y-axis represents compute nodes allocated to the workloads.

Figure 1 represents the expected behavior of a scheduler that is not aware of the presence of other workloads. As it is not able to predict a future reduction in avaialable resources, it is not able to cope with dynamic availability and misses the deadline of the first two jobs because it unnecessarily assigns tasks from all jobs (e.g. from time 0 to 5, and from time 7 to 11 approximately). On the other hand, Figure 2 shows the behaviour of the scheduler proposed in this paper, the Reverse-Adaptive Scheduler, which distributes nodes across jobs considering future availability of resources. From time 0 to D1, most of the tasktrackers are assigned tasks from J1, and the remaining to J2 since it also needs those resources to reach its goal on time. From time D1 to D2, most of the resources go to J2 in order to meet a tight goal. However, as soon as J2 is estimated to reach its deadline, a few tasks from J3 are assigned as well starting around time 4. Finally, from time D2 until the end only tasks from J3 remain to be executed.

III. Problem Statement

We are given a cluster of machines, formed by a set of nodes $\mathcal{N} = \{1, \ldots, N\}$ in which we need to run different workloads. We use n to index the set of nodes. We are also

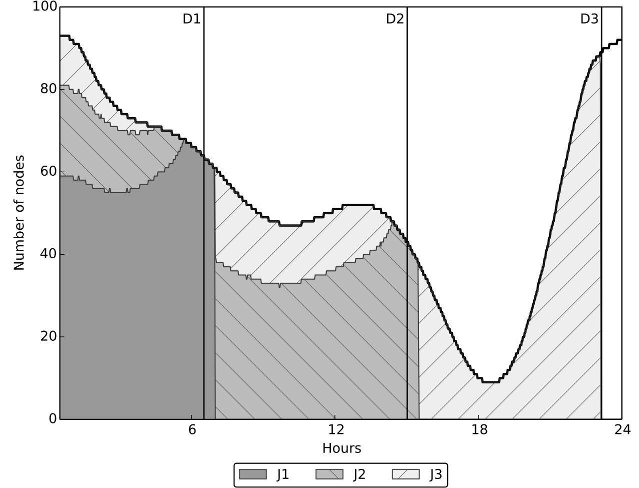

Fig. 1. Distribution of assigned resources over time running the sample workload using the Adaptive Scheduler [6] without dynamic resource availability

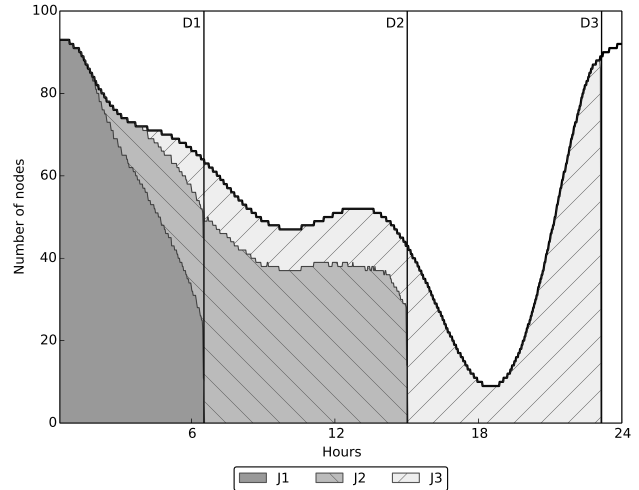

Fig. 2. Distribution of assigned resources over time running the sample workload using the Reverse-Adaptive Scheduler

given a set of MapReduce jobs $\mathcal{J} = \{1, \ldots, J\}$, that has to be run in $\mathcal{N}$. We use j to index the set of MapReduce jobs.

Each node n hosts two main processes: a MapReduce slave and a non-MapReduce process that represents another kind of workload. While MapReduce usually consists of a tasktracker and a datanode in Hadoop terminology, we summarize both of them for simplicity and refer to them as the *tasktracker* process hereafter. Similarly, The non-MapReduce process could represent any kind of workload but we identify it as *data-store* in this paper.

We refer to the set of MapReduce processes, or tasktrackers, as $\mathcal{TT} = \{1, \ldots, N\}$ and the set of data-store processes committed to the front-end activity as $\mathcal{DS} = \{1, \ldots, N\}$, and we use tt and ds respectively to index these sets.

With each node n we associate a series of resources, $\mathcal{R} = \{1, \ldots, R\}$. Each resource of node n has an associated capacity

$\Omega_{n,1}, \ldots, \Omega_{n,r}$, which is shared between the capacity allocated to the data-store and to the tasktracker so that $\Omega_{n,1} = (\Omega^{ds}_{n,1} + \Omega^{tt}_{n,1}), \ldots, \Omega_{n,r} = (\Omega^{ds}_{n,r} + \Omega^{tt}_{n,r})$, where the capacity of the data-store takes preference over the tasktracker.

The usage of each data-store ds changes over time depending on the demand imposed by its users, represented as a function $f(t)$. This function is a prediction of expected workload intensity over time, and affects the resource capacity reserved for each data-store, $\Omega^{ds}_{n,1}, \ldots, \Omega^{ds}_{n,r}$. In turn, since the capacity of each node remains the same, the available resources for each tasktracker tt, $\Omega^{tt}_{n,1}, \ldots, \Omega^{tt}_{n,r}$, also changes to adapt to the remaining capacity left by the data-store.

A MapReduce job (j) is composed of a set of tasks, already known at submission time, that can be divided into map tasks and reduce tasks. Each tasktracker tt provides to the cluster a set of job-slots in which tasks can run. Each job-slot is specific for a particular job, and the scheduler will be responsible for deciding the number of job-slots to create on each tasktracker for each job in the system.

Each job j can be associated with a completion time goal, T^{j}_{goal}, the time at which the job should be completed. When no completion time goal is provided, the assumption is that the job needs to be completed at the earliest possible time.

Additionally, with each job we associate a resource consumption profile. The resource usage profile for a job j consists of a set of average resource demands $\mathcal{D}_j = \{\Gamma_{j,1}, \ldots, \Gamma_{j,r}\}$. Each resource demand consists of a tuple of values. That is, there is one value associated for each task type and phase (map, reduce in shuffle phase, and reduce in reduce phase, including the final sort).

We use symbol P to denote a placement matrix with the assignment of tasks to tasktrackers, where cell $P_{j,tt}$ represents the number of tasks of job j placed on tasktracker tt. For simplicity, we analogously define P^M and P^R, as the placement matrix of Map and Reduce tasks. Notice that $P = P^M + P^R$. Recall that each task running in a tasktracker requires a corresponding slot to be created before the task execution begins, so hereafter we assume that placing a task in a tasktracker implies the creation of an execution slot in that tasktracker.

IV. Reverse-Adaptive Scheduler

The driving principles of the scheduler are resource availability awareness and continuous job performance management. The former is used to decide task placement on tasktrackers over time, while the latter is used to estimate the number of tasks to be run in parallel for each job in order to meet performance objectives, expressed in the form of completion time goals. Job performance management has been extensively evaluated and validated in our previous work, presented as the Adaptive Scheduler [6] [7]. In this paper we extend the resource availability awareness of the scheduler when the MapReduce jobs are collocated with other time-varying workloads.

One key element of our proposal in this paper is the variable S_{fit}, which is an estimator of the minimal number of tasks that should be allocated in parallel to a MapReduce job to keep its chances to reach its deadline, assuming that the available resources will change over time as predicted by $f(t)$. Notice that the novelty of this estimator is the fact that it also considers the variable demand of resources introduced by other external workloads. Thus, the main components of the Reverse-Adaptive Scheduler, as described in the following sections, are:

- S_{fit} estimator. Described in Section IV-B.
- Utility function that leverages S_{fit} used as a per-job performance model. Described in Section IV-C.
- Placement algorithm that leverages the previous two components. Described in Section IV-D.

A. Intuition

The intuition behind the reverse scheduling approach is that it divides time into stationary periods, in which no job completions are expected. One period ends and starts in instants in which a job completion time goal is expected. When a job is expected to complete at the end of a period, the scheduler calculates the amount of resource to be allocated during the period for the job to make its completion goal. If the available resources are not enough, the amount of pending work is pushed back to the immediately preceding period. Notice that the amount of the available resources for the period is determined by the function $f(t)$, that estimates the resources that will have to be committed to the non-MapReduce workloads. When more than one job co-exists in the same period, they compete for the available resources, and they are allocated following a fairness criteria that will try to make all jobs obtain the same utility from the decided schedule.

For the sake of clarity, Figure 3 retakes the example presented in Section II and shows how the placement decision is made step by step. Starting at the desired completion time, which is represented by the deadline of the last job, we assign as many tasks as possible from the jobs that are supposed to be running within that timeframe, compressed between that deadline and the previous one. In this case only J3 is running and we are able to assign most of its tasks, as shown in Figure 3(a). Next we estimate the timeframe between time 7 and 15 as shown in Figure 3(b), in which we would like to run all the tasks from J2 and the remaining ones from J3. The scheduler is able to run the remaining tasks from J3, but since there aren't enough resources to run all the tasks from J2, the remaining ones are carried to the last timeframe. Similarly, in the final step of the estimation as shown in Figure 3(c), the scheduler evaluates the timeframe between 0 and 7, in which it is supposed to execute J1 and the remaining tasks from J2.

Once the estimation of expected availability is completed, the scheduler is aware of all the steps needed to reach its desired state from the current state, and therefore proceeds to create the next placement of jobs that will satisfy its final goal.

B. Estimation of the resources to allocate to each job

We consider a scenario in which jobs are dynamically submitted by users. Each submission includes both the job's completion time goal (if one is provided) and its resource consumption profile. This information is provided via the job configuration file. The scheduler maintains a list of active

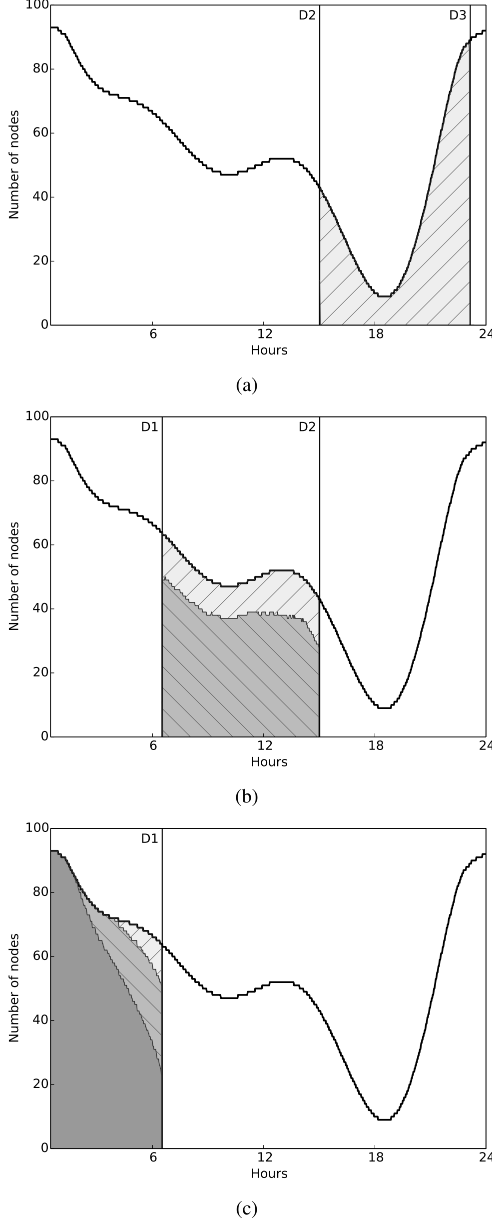

Fig. 3. Step by step estimation with the Reverse-Adaptive Scheduler

jobs and a list of tasktrackers. For each active job it stores a descriptor that contains the information provided when the job was submitted, in addition to state information such as number of pending tasks. For each tasktracker tt, the scheduler also knows its resource capacity at any point in time, $\Omega_{tt,1},\ldots,\Omega_{tt,r}$ since it can be derived from the function that describes the transactional workload pattern, $f(t)$. Note that predicting long-term usage patterns of such workloads has been studied before [9]. While some workloads are inherently unpredictable, the prediction error is small compared to overall workload variability, and there are also well known ways of dealing with it such as adding a buffer to cover for the error.

For any job j in the system, let s^j_{pend} be the number of map tasks pending of execution. The scheduler estimates the minimum number of map tasks that should be allocated concurrently during the next placement cycle, s^j_{fit}, by *reversing* the expected execution assuming all jobs meet their completion time goal T^j_{goal}, and relying on the observed task length (μ^j) and the availability of resources over time (Ω_{tt}).

Algorithm 1 Reverse fitting algorithm to estimate s_{fit}

Require: J: List of Jobs in the system; s^j_{pend}: Number of pending map tasks for each job; Γ_j and Ω_{tt}: Resource demand and capacity for each job and tasktracker correspondingly, as used by the auxiliary function fit

1: **for** j in J **do**
2: $\quad s^j_{fit} = s^j_{pend}$
3: **end for**
4: P = []
5: Sort J by completion time goal
6: **for** j in J **do**
7: $\quad a = T^{next(J)}_{goal}$ // deadline for the next job in J
8: $\quad b = T^j_{goal}$ // deadline for j
9: $\quad$ **for** p in P **do**
10: $\quad\quad$ **if** $s^p_{fit} > 0$ **then**
11: $\quad\quad\quad s^p_{fit} = s^p_{fit} - fit(p,a,b)$
12: $\quad\quad$ **end if**
13: $\quad$ **end for**
14: $\quad$ **if** $s^j_{fit} > 0$ **then**
15: $\quad\quad s^j_{fit} = s^j_{fit} - fit(j,a,b)$
16: $\quad$ **end if**
17: $\quad$ Add j to P
18: **end for**
19: **return** s^j_{fit} for each job in J

Algorithm 1 shows how this estimation takes place. We first start assuming that for each job j, s^j_{fit} equals the number of pending tasks s^j_{pend} (lines 1-3), and then proceed to subtract as many tasks as possible beginning from the job with the last deadline to the job with the earliest deadline (lines 5-8), and as long as they *fit* within the availble amount of resources (lines 9-17). The algorithm uses the $fit()$ function, which given a job j and two points in time a and b returns the amount of tasks from job j that can be assigned between time a and b, taking into consideration the profile and resource requirements of said job. Notice also how on every iteration we try to *fit* tasks between the two last deadlines (lines 7-8), and try to assign tasks from jobs with the latest deadlines first as long as they still have remaining tasks left (lines 9-13).

In addition to the main estimator s^j_{fit}, which estimates the minimum number of tasks to be allocated for each job during the next placement cycle, we also calculate the average number of tasks that should be allocated over time considering a fixed availability of resources equal to the average amount of resources from current time to its deadline, s^j_{req}. The latter is used to assign remaining the resources left after allocating the minimum number of tasks with the former, if any.

C. Performance Model

To measure the performance of a job given a placement matrix, we define a utility function that combines the number of map and reduce slots allocated to the job with its com-

pletion time goal and job characteristics. Below we provide a description of this function.

Given placement matrices P^M and P^R, we can define the number of map and reduce slots allocated to a job j as $s^j_{alloc} = \sum_{tt \in \mathcal{TT}} P^M_{j,tt}$ and $r^j_{alloc} = \sum_{tt \in \mathcal{TT}} P^R_{j,tt}$ correspondingly.

Based on these parameters and the previous definitions of s^j_{pend} and r^j_{pend}, we define the utility of a job j given a placement P as:

$$u_j(P) = u^M_j(P^M) + u^R_j(P^R), \quad \text{where } P = P^M + P^R \quad (1)$$

and where u^M_j is a utility function that denotes increasing satisfaction of a job given a placement of map tasks, and u^R_j is a utility function that shows satisfaction of a job given a placement of reduce tasks. The definition of both is as follows:

$$u^M_j(P^M) = \begin{cases} \frac{log(s^j_{alloc})}{log(s^j_{fit})} - 1 & s^j_{alloc} < s^j_{fit} \\ \frac{s^j_{alloc} - s^j_{fit}}{2 \times (s^j_{req} - s^j_{fit})} & s^j_{fit} < s^j_{alloc} < s^j_{req} \\ \frac{s^j_{alloc} - s^j_{req}}{2 \times (s^j_{pend} - s^j_{req})} + \frac{1}{2} & s^j_{fit} < s^j_{req} < s^j_{alloc} \\ \frac{s^j_{alloc} - s^j_{fit}}{s^j_{pend} - s^j_{fit}} & s^j_{req} \leq s^j_{fit} < s^j_{alloc} \end{cases} \quad (2)$$

$$u^R_j(P^R) = \frac{log(r^j_{alloc})}{log(r^j_{pend})} - 1 \quad (3)$$

Notice that in practice a job will never get more tasks allocated to it than it has remaining: to reflect this in theory we cap the utility at $u_j(P) = 1$ for those cases.

The definition of u differentiates between three cases: (1) the satisfaction of the job grows logarithmically from $-\infty$ to 0 if the job has fewer map slots allocated to it than it requires to meet its completion time goal; (2) the function grows linearly between 0 and 0.5, when $s^j_{alloc} = s^j_{req}$ and thus in addition to the absolute minimum required for the next control cycle, the job is also allocated the estimated number of slots required over time to meet the completion time goal; and (3) the function grows linearly between 0.5 and 1.0, when $s^j_{alloc} = s^j_{pend}$ and thus all pending map tasks for this job are allocated a slot in the current control cycle.

Notice that u^M_j is a monotonically increasing utility function, with values in the range $(-\infty, 1]$. The intuition behind this function is that a job is unsatisfied ($u^M_j < 0$) when the number of slots allocated to map tasks is less than the minimum number required to meet the completion time goal of the job. Furthermore, the logarithmic shape of the function stresses the fact that it is critical for a job to make progress and therefore at least one slot must be allocated. A job is no longer unsatisfied ($u^M_j = 0$) when the allocation equals the requirement ($s^j_{alloc} = s^j_{req}$), and its satisfaction is positive ($u^M_j > 0$) and grows linearly when it gets more slots allocated than required. The maximum satisfaction occurs when all the pending tasks are allocated within the current control cycle ($s^j_{alloc} = s^j_{pend}$). The intuition behind u^R_j is that reduce tasks should start at the earliest possible time, so the shuffle sub-phase of the job (reducers pulling data produced by map tasks) can be fully pipelined with execution of map tasks. The logarithmic shape of this function indicates that any placement that does not run all reducers for a running job is unsatisfactory. The range of this function is $[-1, 0]$ and, therefore, it is used to subtract satisfaction of a job that, independently of the placement of map tasks, has unsatisfied demand for reduce tasks. If all the reduce tasks for a job are allocated, this function gets value 0 and thus, $u_j(P) = u^M_j(P^M)$.

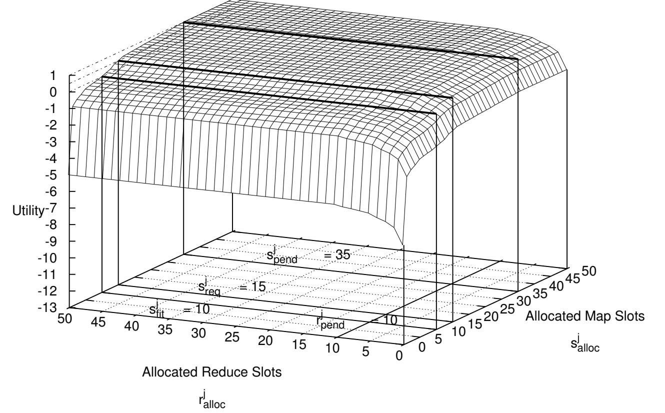

Fig. 4. Shape of the Utility Function when $s^j_{fit} = 10$, $s^j_{req} = 15$ $s^j_{pend} = 35$, and $r^j_{pend} = 10$

Figure 4 shows the generic shape of the utility function for a job that requires at least 10 map tasks allocated during the next cycle ($s^j_{fit} = 10$), 15 map tasks concurrently over time ($s^j_{req} = 15$) to meet its completion time goal, has 35 map tasks ($s^j_{pend} = 35$) pending to be executed, and has been configured to run 10 reduce tasks ($r^j_{pend} = 10$), none of which have been started yet. On the X axis, a variable number of allocated slots for reduce tasks (r^j_{alloc}) is shown. On the Y axis, a variable number of allocated slots for map tasks (s^j_{alloc}) is shown. Finally, the Z axis shows the resulting utility value.

D. Job Placement Algorithm

Given an application placement matrix P, a utility value can be calculated for each job in the system. The performance of the system can then be measured as an ordered vector of job utility values, U. The objective of the scheduler is to find a new placement P of jobs on tasktrackers that maximizes the global objective of the system, $U(P)$, which is expressed as follows:

$$\max \quad \min_j u_j(P) \quad (4)$$

$$\min \quad \Omega_{tt,r} - \sum_{tt} (\sum_j P_{j,tt}) * \Gamma_{j,r} \quad \forall_r \quad (5)$$

such that

$$\forall_{tt} \forall_r \quad (\sum_j P_{j,tt}) * \Gamma_{j,r} \leq \Omega_{tt,r} \quad (6)$$

$$and \quad \Omega_{n,r} = \Omega_{tt,r} + \Omega_{ds,r} \quad (7)$$

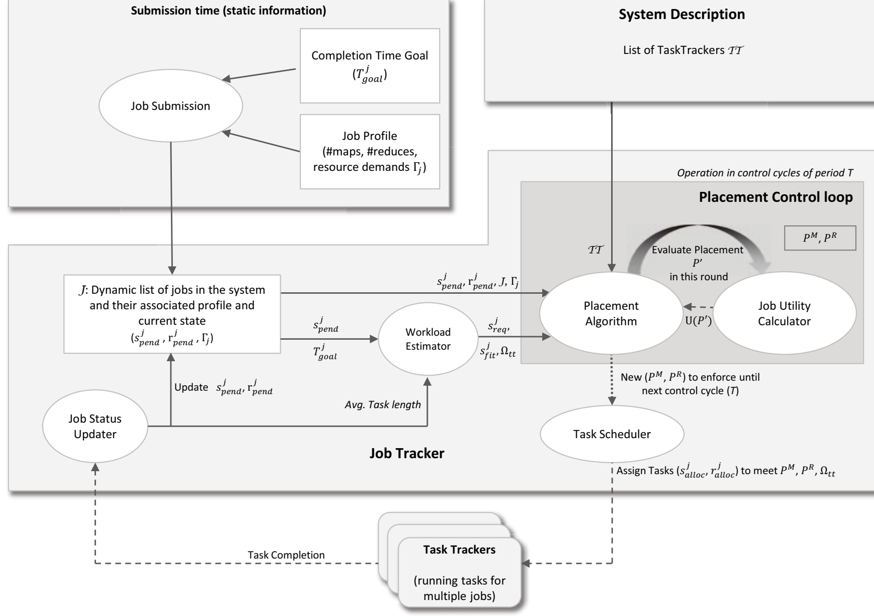

Fig. 5. System architecture of the Reverse-Adaptive Scheduler

This optimization problem is a variant of the Class Constrained Multiple-Knapsack Problem. Since this problem is NP-hard, the scheduler adopts a heuristic inspired by [10]. While not described in this paper, the Placement Algorithm itself is the same as that described in [7], but using the proposed utility function described in Section IV-C.

E. Scheduler Architecture

Figure 5 illustrates the architecture and operation the scheduler. The system consists of five components: Placement Algorithm, Job Utility Calculator, Task Scheduler, Job Status Updater and Workload Estimator. Each submission includes both the job's completion time goal (if one is provided) and its resource consumption profile.

Most of the logic behind the scheduler resides the utility-driven Placement Control Loop and the Task Scheduler. The former is responsible for producing placement decisions, while the latter is responsible for enforcing the decisions made by the former. The Placement Control Loop operates in control cycles of period T. Its output is a new placement matrix P that will be active until the next control cycle is reached (current time + T). The Task Scheduler is responsible for enforcing the placement decisions. The Job Utility Calculator calculates a utility value for an input placement matrix which is then used by the Placement Algorithm to choose the best placement choice available. Upon completion of a task, the TaskTracker notifies the Job Status Updater, which for any job j in the system, triggers an update of s^j_{pend} and r^j_{pend} in the job descriptor. The Job Status Updater also keeps track of the average task length observed for every job in the system, which is later used to estimate the completion time for each job. The Workload Estimator estimates the number of map tasks that should be allocated concurrently (s^j_{req}) to meet the completion time goal of each job, as well as the parameter S^j_{fit}.

In this work we concentrate on the estimation of the parameter S^j_{fit} that feeds the Placement Algorithm, as well as the performance model used by the Job Utility Calculator. The major change in this architecture compared to the scheduler presented in [7] is the introduction of the Workload Estimator, that not only estimates the demand for MapReduce tasks as did in previous work, but also provides estimates for the data-store resource consumption, derived from the calculation of $f(t)$.

V. Evaluation

This section includes the description of the experimental environment, including the simulation platform we have built, and the results from the experiments that explore the improvements of our scheduler compared to previous existing schedulers: the default FIFO scheduler, the Adaptive Scheduler described in [7], and the Reverse-Adaptive scheduler proposed in this paper.

In Experiment 1 (Section V-B) we consider the standard scenario in which MapReduce is the only workload running in the system and thus the performance of the scheduler should be similar to previous approaches. In Experiment 2 (Section V-C) we introduce an additional workload in order to gain insight on how does the proposed scheduler perform in this kind of shared environment. And finally, Experiment 3 (Section V-D) shows the impact that the burstiness of transactional workloads may have on the scheduler.

A. Simulation Platform

In order to simulate a shared environment, we built a system with two components. First, a workload generator to model the behaviour of multiple clients submitting jobs to the MapReduce cluster. And second, a server simulator to handle the workloads' submissions and schedule jobs depending on different policies.

The workload generator that describes the behaviour of clients takes the cluster configuration information as well as the desired workload parameters, and instantiates a number of

TABLE I. MAIN WORKLOAD CONFIGURATION PARAMETERS.

Parameter	Value	Description
Cluster size	100	Total number of nodes in the system.
Node availability	$f(t)$	Function that represents the available number of nodes over time.
System load	$0.2 - 1.0$	Utilization of the MapReduce workload during the simulation. Determines the number of jobs.
Arrival distribution	Poisson: $n \approx 200 - 2500, \lambda \approx 1.5 - 15$	How arrivals are distributed over time. Depends on system load.
Job length distribution	Lognorm: $\mu = 62.0$, $\sigma = 15.5$	Determines the number of tasks of each job.
Deadline distribution	Uniform: $1.5x - [4, 8, 12]x$	Factor relative to completion time of jobs when executed in isolation.

jobs to meet those requirements. Table I describes the main workload configuration parameters used for the experiments. The dynamic availability of resources of the transactional workload ($f(t)$) is based on a real trace obtained from Twitter's frontend during an entire day, and has the same shape as that shown in Figures 1 and 2, with peak transactional utilization around hour 18. The distribution of MapReduce job lengths, which determines the number of tasks of each job, follows a lognormal that resembles the job sizes observed in known traces from Yahoo! and Facebook [11], but scaled down to a smaller number of jobs to fit into the 100-node cluster used during the simulations. For the distribution of deadlines factors we use a 3 different categories: tight (between 1.5x and 4x), regular (from 1.5x to 8x), and relaxed (from 1.5x to 12x).

In the experiments we simulate a total of 7 days, and in order to make sure the simulation is in a steady state we study and generate all the statistics for the 5 days in the middle, considering only jobs that either start or finish within that time window. For each experiment we obtain the averages and standard deviations of runing 10 different simulated workloads generated with the same configuration parameters.

The simulation platform is written in Python using the NumPy and SciPy packages, and the Reverse-Adaptive implementation in particular is based on *splines* for fast, approximate curve fitting, interpolation and integration. While our proposal hasn't been optimized and is slower than the other schedulers we are simulating, it doesn't represent a performance issue for the amount of concurrent jobs that are usually executed in this kind of environment, specially considering MapReduce clusters run the scheduler on a dedicated machine and decisions are only made once per placement cycle, which is in the order of tens of seconds. In our experiments with hundreds to few thousands of jobs the scheduler is able to generate placement matrices in a time that always remains in the order of milliseconds. Our current implementation easily scales up to thousands of jobs and nodes.

B. Experiment 1: No Transactional Workload

The goal of this experiment is to evaluate the scenario in which there is no additional workload other than MapReduce itself, and to assess that the scheduler doesn't introduce any flaw even in the worst-case scenario in which there is no transactional workload. It also represents the standard scenario considered by most MapReduce schedulers, which are only concerned with assigning tasks to a fixed number of nodes in the cluster.

To this end we disable the transactional workload on the simulator and make all resources available to the MapReduce workload. We then run the same experiments using the default FIFO scheduler, the Adaptive scheduler, and our proposed scheduler, the Reverse-Adaptive scheduler.

Figure 6 shows the percentage of missed deadlines for each scheduler under different configurations. On the first row the distribution of deadline factors assigned to jobs (meaning the time each job is given to complete) is uniformly distributed and ranges from a minimum of 1.5x to a maximum of 4x. On the second and third rows, the maximum deadline factor is increased to 8x and 12x respectively. Each row shows different load factors as well, which represent how busy is the cluster: from 0.2 (very small load) to 1.0 (fully loaded). As it can be observed, there is a significant difference between the default FIFO scheduler, which always misses more deadlines, and the other deadline-aware schedulers. Also, as expected, increasing the maximum deadline factor also has an impact on the number of missed deadlines on all schedulers, but even more so on the Adaptive and Reverse-Adaptive schedulers since that gives them more flexibility and a higher chance of distributing the execution of jobs.

On the other hand, in this scenario the Reverse-Adaptive scheduler performs exactly like the Adaptive scheduler under all configurations since it isn't able to leverage the information about the characteristics of other non-MapReduce workloads in order to improve its performance. But it also shows that under no circumstances will the Reverse-Adaptive scheduler perform worse than previous deadline-aware schedulers in terms of missed deadlines.

C. Experiment 2: Transactional Workload

In this experiment we evaluate the Reverse-Adaptive scheduler in the presence of transactional workloads, and compare it to other schedulers, showing also additional metrics that help understand the behaviour of our algorithm. In particular, we study the same workload under different load levels: from 0.2 (low load) to 0.8 (high load); and also with different deadline factor distributions, ranging from 1.5x–4x to 1.5–12x. The transactional workload changes the availability of resources over time, and is based on a real trace as described in V-A.

Figure 7 shows the results for each deadline factor distribution: 1.5x – 4x (Figure 7(a)), 1.5x – 8x (Figure 7(b)), 1.5x – 12x (Figure 7(c)). And each figure shows the number of jobs that miss their deadline (1st row), time beyond deadline for jobs that miss their deadline (2nd row), and distance to deadline for jobs that meet their deadline (3rd row). For this experiment we also run a fourth execution of the simulator with a different optimization goal that only takes into account minimizing the number of missed deadlines, and doesn't

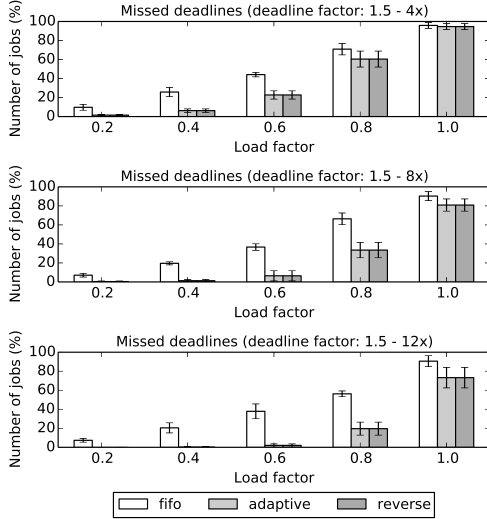

Fig. 6. Experiment 1: No transactional workload.

consider any fairness goals found in other schedulers. It is shown in the first row of Figures 7(a) to 7(c) as a horizontal line on the Reverse-Adaptive scheduler bars. We use these as a reference to distinguish why are schedulers missing deadlines, as it marks the minimum amount of jobs that will miss their deadline independently of the policies of the scheduler.

As it can be observed in the three figures, introducing a dynamic transactional workload allows the scheduler to improve the number of missed deadlines without a significant impact on other metrics. As shown in Figure 7(a), which represents executions when running with a tight deadline factor distribution between 1.5x and 4x, the number of deadlines missed by the Reverse-Adaptive scheduler is always noticeable lower than that of the Adaptive and FIFO schedulers (1st row), while the time beyond deadline is only slightly lower (2nd row), and distance to deadline remains mostly the same with very small variations (3rd row). These results remain the same with more relaxed deadline factors as shown in Figure 7(b) and 7(c). Notice that the improvement in terms of percentage of missed deadlines with the Reverse-Adaptive scheduler compared to other schedulers is similar despite the different deadline factors. This is basically because in these three scenarios the actual chance of improving is similar, as shown by the horizontal lines marking the percentage of jobs that will be missed for certain.

D. Experiment 3: Burstiness of Transactional Workload

This experiment explores the impact of transactional workload burstiness on the scheduler. While the previous experiment shows that the scheduler is able to leverage the characteristics of the transactional workload to improve the performance of the scheduler, in this experiment we show how the shape of the availability function affects the chances of improving the overall results. In particular, burstiness in this scenario means variability between the highest and lowest points of the availability function. Figure 8 shows the different burstiness *levels* evaluated in this experiment: from high burstiness (level 3) to low burstiness (level 1). As a reference to

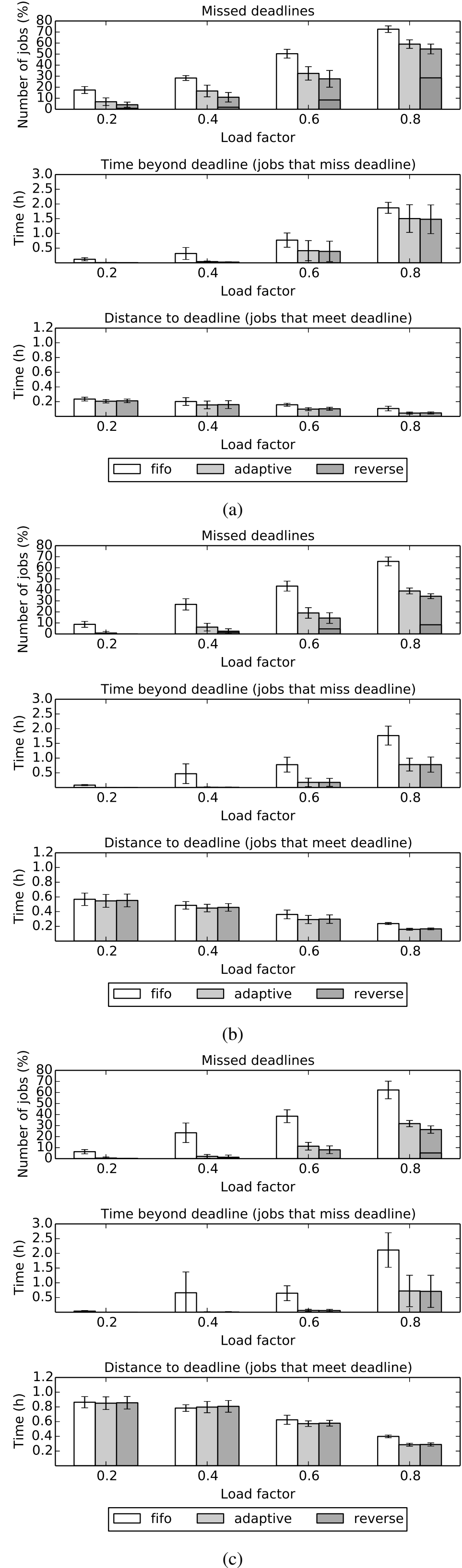

Fig. 7. Experiment 2: Scheduling with transactional workload. Deadline factors: 1.5x – 4x (a), 1.5x – 8x (b), 1.5x – 12x (c).

compare to previous executions, note that Experiment 1 has no burstiness at all, while Experiment 2 represents high burstiness (equivalent to level 3).

Figure 9 shows the number of jobs that miss their deadline when running with different burstiness levels. To simplify, only simulations with a medium deadline factor (1.5x–8x) are shown; other factors behave similarly. As it can be observed, the more bursty the availability function, the more likely it is that the scheduler improves its performance, lowering the amount of missed deadlines relative to other schedulers. The higher variability of high burstiness levels makes available resources less predictable, and this has a significant impact on other schedulers because their estimations become less accurate. However, it also provides provides more means for the Reverse-Adaptive scheduler to plan in advance how to schedule present tasks.

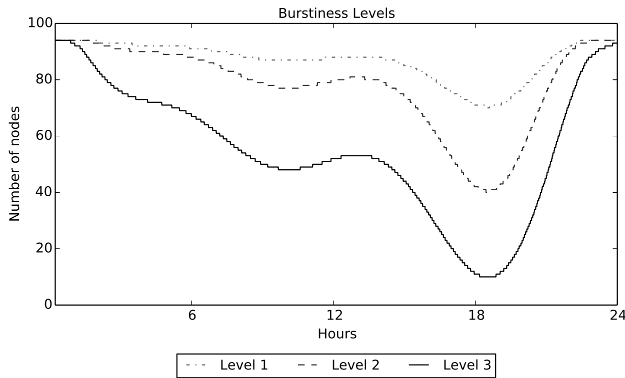

Fig. 8. Experiment 3: Burstiness level classification.

VI. Related Work

Much work has been done in the space of scheduling for MapReduce. Since the number of resources and slots in a Hadoop cluster is fixed through out the lifetime of the cluster, most of the proposed solutions can be reduced to a variant of the *task-assignment* or *slot-assignment* problem. The Capacity Scheduler [12] is a pluggable scheduler developed by Yahoo! which partitions resources into pools and provides priorities for each pool. Hadoop's Fair Scheduler [13] allocates equal shares to each tenant in the cluster. All these schedulers are built on top of the same resource model and do not support high-level goals nor dynamic availability in shared environments.

The performance of MapReduce jobs has attracted much interest in the Hadoop community. Recently, there has been increasing interest in user-centric data analytics. The Adaptive Scheduler [6] enables soft-deadline support for MapReduce jobs. It differs from this paper's proposal in that it does not take into consideration neither the resources of the system nor other workloads. Similarly, Flex [14] is a scheduler proposed as an add-on to the Fair Scheduler to provide Service-Level-Agreement (SLA) guarantees. More recently, Aria [8] introduces a novel resource management framework that consists of a job profiler, a model for MapReduce jobs and a SLO-scheduler based on the Earliest Deadline First scheduling strategy. Flex and Aria are both slot-based and therefore suffer from the same aforementioned limitations.

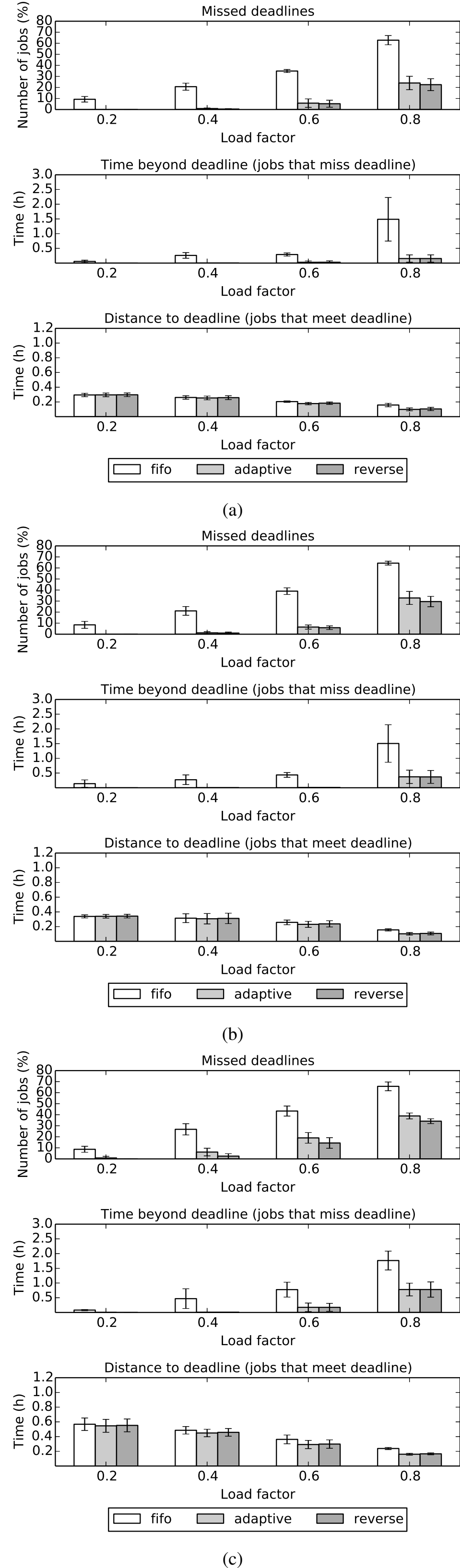

Fig. 9. Experiment 3: Execution with different burstiness: level 1 (a), level 2 (b), and level 3 (c); deadline factor from 1.5x to 8x.

New platforms have been proposed to mix MapReduce frameworks like Hadoop with other kinds of workloads. Mesos [15] intends to improve cluster utilization on shared environments, but is focused on batch-like and HPC instead of transactional workloads. Finally, the Hadoop community has also recognized the importance of developing a resource-aware scheduling for MapReduce, and proposed a framework [16] that introduces a resource model consisting of a 'resource container' which is fungible across jobs. We think that our resource management techniques can be leveraged within this framework to enable better resource management.

VII. Conclusions

In this paper we have presented the Reverse-Adaptive Scheduler, which introduces a novel resource management and job scheduling scheme for MapReduce when executed in shared environments along with other kinds of workloads. Our scheduler is capable of improving resource utilization and job performance. The model we introduce allows for the formulation of a placement problem which is solved by means of a utility-driven algorithm. This algorithm in turn provides our scheduler with the adaptability needed to respond to changing conditions in resource demand and availability of resources.

The scheduler works by estimating the need of resources that should be allocated to each job, but in a more proactive way than previously existing work, since the estimation takes into account the expected availability of resources. In particular, the proposed algorithm consists of two major steps: reversing the execution of the workload and generating the current placement of tasks. Reversing the execution of the workload involves creating an estimated placement of the full workload over time, assigning tasks in the opposite direction: starting at the desired end state and finishing at the current state. The *reversed* placement is used as an estimation to know how many tasks are left at the current state, which allows the scheduler to determine what's the need of tasks for each job and how should they share the available resources. The presented scheduler relies on existing profiling information based on previous executions of jobs to make scheduling and placement decisions.

The goal of the scheduler is to determine the best possible placement of tasks across the tasktrackers so as to maximize resource utilization in the cluster while observing the completion time goal for each job. To achieve this objective, the system dynamically manages the number of slots each tasktracker will provision for each job, and controls the execution of their tasks. Our experiments in a simulated environment driven by representative MapReduce workloads demonstrate the effectiveness of our proposal. To the best of our knowledge this is the first scheduling framework to take into account other non-MapReduce workloads, such as transactional workloads, in addition to leveraging resource information to improve the utilization of resources in the system and meet completion time goals on behalf of users.

Acknowledgements

This work is partially supported by the Ministry of Science and Technology of Spain and the European Union's FEDER funds (TIN2012-34557), by the Generalitat de Catalunya (2009-SGR-980), by the BSC-CNS Severo Ochoa program (SEV-2011-00067) and by the by the European Commissionś IST activity of the 7th Framework Program under contract number 317862 (COMPOSE).

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *OSDI'04: Proceedings of the 6th Symposium on Operating Systems Design and Implementation*. San Francisco, CA: USENIX Association, December 2004, pp. 137–150.

[2] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 4:1–4:26, Jun. 2008.

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, ser. SOSP '07. NY, USA: ACM, 2007, pp. 205–220.

[4] L. A. Barroso, J. Clidaras, and U. Hölzle, "The datacenter as a computer: An introduction to the design of warehouse-scale machines, second edition," *Synthesis Lectures on Computer Architecture*, vol. 8, no. 3, pp. 1–154, 2013.

[5] G. Pacifici, W. Segmuller, M. Spreitzer, and A. Tantawi, "Dynamic estimation of cpu demand of web traffic," in *Proceedings of the 1st international conference on Performance evaluation methodolgies and tools*, ser. valuetools '06. New York, NY, USA: ACM, 2006.

[6] J. Polo, D. Carrera, Y. Becerra, M. Steinder, and I. Whalley, "Performance-driven task co-scheduling for MapReduce environments," in *Network Operations and Management Symposium, NOMS*, Osaka, Japan, 2010, pp. 373–380.

[7] J. Polo, C. Castillo, D. Carrera, Y. Becerra, I. Whalley, M. Steinder, J. Torres, and E. Ayguadé, "Resource-aware adaptive scheduling for mapreduce clusters," in *Middleware 2011*, ser. Lecture Notes in Computer Science, vol. 7049. Springer Berlin Heidelberg, 2011, pp. 187–207.

[8] A. Verma, L. Cherkasova, and R. H. Campbell, "ARIA: Automatic Resource Inference and Allocation for MapReduce Environments," in *8th IEEE International Conference on Autonomic Computing*, Karlsruhe, Germany., June 2011.

[9] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper, "Workload analysis and demand prediction of enterprise data center applications," in *Workload Characterization, 2007. IISWC 2007. IEEE 10th International Symposium on*, Sept 2007, pp. 171–180.

[10] C. Tang, M. Steinder, M. Spreitzer, and G. Pacifici, "A scalable application placement controller for enterprise data centers," in *Procs. of the 16th intl. conference on World Wide Web*, ser. WWW '07. NY, USA: ACM, 2007, pp. 331–340.

[11] Y. Chen, A. S. Ganapathi, R. Griffith, and R. H. Katz, "A methodology for understanding mapreduce performance under diverse workloads," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-135, Nov 2010.

[12] Yahoo! Inc. Capacity Scheduler. http://developer.yahoo.com/blogs/hadoop/posts/2011/02/capacity-scheduler/.

[13] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *OSDI'08*. Berkeley, USA: USENIX Association, 2008, pp. 29–42.

[14] 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," in *Middleware 2010*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, vol. 6452, pp. 1–20.

[15] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *8th USENIX conference on Networked systems design and implementation*. USENIX Association, 2011, pp. 22–22.

[16] Arun Murthy. Next Generation Hadoop. [Online]. Available: http://developer.yahoo.com/blogs/hadoop/posts/2011/03/mapreduce-nextgen-scheduler/