Resource-Aware Adaptive Scheduling for MapReduce Clusters

Jordà Polo[1], Claris Castillo[2], David Carrera[1], Yolanda Becerra[1], Ian Whalley[2], Malgorzata Steinder[2], Jordi Torres[1], and Eduard Ayguadé[1]

[1] Barcelona Supercomputing Center (BSC) and
Technical University of Catalonia (UPC)
{jordap,dcarrera,yolandab,torres,eduard}@ac.upc.edu
[2] IBM T.J. Watson Research Center
{claris,inw,steinder}@us.ibm.com

Abstract. We present a resource-aware scheduling technique for MapReduce multi-job workloads that aims at improving resource utilization across machines while observing completion time goals. Existing MapReduce schedulers define a static number of slots to represent the capacity of a cluster, creating a fixed number of execution slots per machine. This abstraction works for homogeneous workloads, but fails to capture the different resource requirements of individual jobs in multi-user environments. Our technique leverages job profiling information to dynamically adjust the number of slots on each machine, as well as workload placement across them, to maximize the resource utilization of the cluster. In addition, our technique is guided by user-provided completion time goals for each job. Source code of our prototype is available at [1].

Keywords: MapReduce, scheduling, resource-awareness, performance management.

1 Introduction

In recent years, the industry and research community have witnessed an extraordinary growth in research and development of data-analytic technologies. Pivotal to this phenomenon is the adoption of the MapReduce programming paradigm [2] and its open-source implementation Hadoop [3].

Pioneer implementations of MapReduce [3] have been designed to provide overall system goals (e.g., job throughput). Thus, support for user-specified goals and resource utilization management have been left as secondary considerations at best. We believe that both capabilities are crucial for the further development and adoption of large-scale data processing. On one hand, more users wish for ad-hoc processing in order to perform short-term tasks [4]. Furthermore, in a Cloud environment users pay for resources used. Therefore, providing consistency between price and the quality of service obtained is key to the business model of the Cloud. Resource management, on the other hand, is also important as Cloud providers are motivated by profit and hence require both high levels of automation and resource utilization while avoiding bottlenecks.

F. Kon and A.-M. Kermarrec (Eds.): Middleware 2011, LNCS 7049, pp. 187–207, 2011.

The main challenge in enabling resource management in Hadoop clusters stems from the resource model adopted in MapReduce. Hadoop expresses capacity as a function of the number of tasks that can run concurrently in the system. To enable this model the concept of typed-'slot' was introduced as the schedulable unit in the system. 'Slots' are bound to a particular type of task, either reduce or map, and one task of the appropriate type is executed in each slot. The main drawback of this approach is that slots are fungible across jobs: a task (of the appropriate type) can execute in any slot, regardless of the job of which that task forms a part. This loose coupling between scheduling and resource management limits the opportunity to efficiently control the utilization of resources in the system. Providing support for user-specified 'goals' in MapReduce clusters is also challenging, due to high variability induced by the presence of outlier tasks (tasks that take much longer than other tasks) [5–8]. Solutions to mitigate the detrimental impact of such outliers typically rely on scheduling techniques such as speculative scheduling [9], and killing and restarting of tasks [5]. These approaches, however, may result in wasted resources and reduced throughput. More importantly, all existing techniques are based on the typed-slot model and therefore suffer from the aforementioned limitations.

In this work we present RAS [1], a Resource-aware Adaptive Scheduler for MapReduce capable of improving resource utilization and which is guided by completion time goals. In addition, RAS addresses the system administration issue of configuring the number of slots for each machine, which—as we will demonstrate—has no single, homogeneous, and static solution for a multi-job MapReduce cluster.

While existing work focuses on the current typed-slot model—wherein the number of tasks per worker is fixed throughout the lifetime of the cluster, and slots can host tasks from any job—our approach offers a novel resource-aware scheduling technique which advances the state of the art in several ways:

- Extends the abstraction of 'task slot' to 'job slot'. A 'job slot' is job specific, and has an associated resource demand profile for map and reduce tasks.
- Leverages resource profiling information to obtain better utilization of resources and improve application performance.
- Adapts to changes in resource demand by dynamically allocating resources to jobs.
- Seeks to meet soft-deadlines via a utility-based approach.
- Differentiates between map and reduce tasks when making resource-aware scheduling decisions.

The structure of the paper is as follows. We give an overview of Hadoop in Section 2. The scheduler's design and implementation is described in detail in Section 3. An evaluation of our prototype in a real cluster is is presented in Section 4. Finally, we discuss related work in Section 5 and conclude in Section 6.

2 MapReduce and Hadoop

The execution of a MapReduce job is divided into a Map phase and a Reduce phase. In the Map phase, the map tasks of the job are run. Each map task comprises the execution of the job's *map()* function as well as some supporting actions (for example, data sorting). The data output by each map task is written into a circular memory buffer—when this buffer reaches a threshold, its content is sorted by key and flushed to a temporary file. These files are then served via HTTP to machines running reduce tasks. Reduce tasks are divided into three sub-phases: shuffle, sort and reduce. The shuffle sub-phase is responsible for copying the map output from the machines hosting maps to the reducer's machine. The sort sub-phase sorts the intermediate data by key. Finally, the reduce sub-phase, which runs the job's *reduce()* function, starts after the keys destined for the particular reducer have been copied and sorted, and the final result is then written to the distributed file system.

Hadoop [3] is an open source implementation of MapReduce provided by the Apache Software Foundation. The Hadoop architecture follows the master/slave paradigm. It consists of a master machine responsible for coordinating the distribution of work and execution of jobs, and a set of worker machine responsible for performing work assigned by the master. The master and slaves roles are performed by the 'JobTracker' and 'TaskTracker' processes, respectively. The singleton JobTracker partitions the input data into 'input splits' using a splitting method defined by the programmer, populates a local task-queue based on the number of obtained input splits, and distributes work to the TaskTrackers that in turn process individual splits. Work units are represented by 'tasks' in this framework. There is one map task for every input split generated by the JobTracker. The number of reduce tasks is defined by the user. Each TaskTracker controls the execution of the tasks assigned to its hosting machine.

In Hadoop resources are abstracted into typed slots. A slot is bound to a particular type of task (map or reduce), but is fungible across jobs. The slot is the schedulable unit, and as such is the finest granularity at which resources are managed in the system. The number of slots per TaskTracker determines the maximum number of concurrent tasks that are allowed to run in the worker machine. Since the number of slots per machine is fixed for the lifetime of the machine, existing schedulers implement a task-assignment solver. The default scheduler in Hadoop, for example, implements the FIFO (First-In First-Out) scheduling strategy. More recently, the Fair Scheduler [9] assigns jobs to 'pools', and then guarantees a certain minimum number of slots to each pool.

3 Resource-Aware Adaptive Scheduler

The driving principles of RAS are resource awareness and continuous job performance management. The former is used to decide task placement on TaskTrackers over time, and is the main object of study of this paper. The latter is used to estimate the number of tasks to be run in parallel for each job in order to meet some performance objectives, expressed in RAS in the form of completion time goals, and was extensively evaluated and validated in [6].

In order to enable this resource awareness, we introduce the concept of 'job slot'. A job slot is an execution slot that is bound to a particular job, and a particular task type (reduce or map) within that job. This is in contrast to the traditional approach, wherein a slot is bound only to a task type regardless of the job. In the rest of the paper we will use the terms 'job slot' and 'slot' interchangeably. This extension allows for a finer-grained resource model for MapReduce jobs. Additionally, RAS determines the number of job slots, and their placement in the cluster, dynamically at run-time. This contrasts sharply with the traditional approach of requiring the system administrator to statically and homogeneously configure the slot count and type on a cluster. This eases the configuration burden and improves the behavior of the MapReduce cluster.

Completion time goals are provided by users at job submission time. These goals are treated as soft deadlines in RAS as opposed to the strict deadlines familiar in real-time environments: they simply guide workload management.

3.1 Problem Statement

We are given a set of MapReduce jobs $\mathcal{J} = \{1, \ldots, J\}$, and a set of TaskTrackers $\mathcal{TT} = \{1, \ldots, TT\}$. We use j and tt to index into the sets of jobs and TaskTrackers, respectively. With each TaskTracker tt we associate a series of resources, $\mathcal{R} = \{1, \ldots, R\}$. Each resource of TaskTracker tt has an associated capacity $\Omega_{tt,1}, \ldots, \Omega_{tt,r}$. In our work we consider disk bandwidth, memory, and CPU capacities for each TaskTracker. Note that extending the algorithm to accommodate for other resources, e.g., storage capacity, is straightforward.

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 of tasks on 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.

Based on the parameters described above, the goal of the scheduler presented in this paper is to determine the best possible placement of tasks across the TaskTrackers as to maximize resource utilization in the cluster while observing

the completion time goal for each job. To achieve this objective, the system will dynamically manage the number of job-slots each TaskTracker will provision for each job, and will control the execution of their tasks in each job-slot.

3.2 Architecture

Figure 1 illustrates the architecture and operation of RAS. The system consists of five components: Placement Algorithm, Job Utility Calculator, Task Scheduler, Job Status Updater and Job Completion Time Estimator.

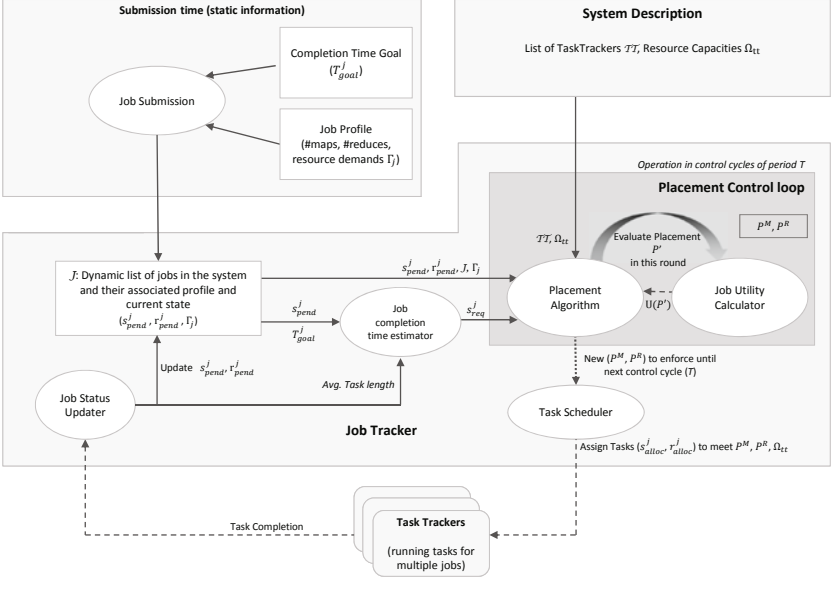


Fig. 1. System architecture

Most of the logic behind RAS resides in the JobTracker. 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 XML file. The JobTracker maintains a list of active 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) it stores that TaskTracker's resource capacity (Ω_{tt}).

For any job j in the system, let s^j_{pend} and r^j_{pend} be the number of map and reduce tasks pending execution, respectively. Upon completion of a task, the TaskTracker notifies the **Job Status Updater**, which 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 **Job Completion Time Estimator** estimates the number of *map* tasks that should be allocated concurrently (s^j_{req}) to meet the completion time goal of each job. To perform this calculation it relies on the completion time goal T^j_{goal}, the number of pending *map* tasks (s^j_{pend}), and the observed average task length. Notice that the scenario we focus on is very dynamic, with jobs entering and leaving the system unpredictably, so the goal of this component is to provide estimates of s^j_{req} that guide resource allocation. This component leverages the techniques already described in [6] and therefore we will not provide further details in this paper.

The core of RAS is the Placement Control loop, which is composed of the **Placement Algorithm** and the **Job Utility Calculator**. They operate in control cycles of period T, which is of the order of tens of seconds. The output of their operation is a new placement matrix P that will be active until the next control cycle is reached (current time + T). A short control cycle is necessary to allow the system to react quickly to new job submissions and changes in the task length observed for running jobs. In each cycle, the Placement Algorithm component examines the placement of tasks on TaskTrackers and their resource allocations, evaluates different candidate placement matrices and proposes the final output placement to be enforced until next control cycle. 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.

The **Task Scheduler** is responsible for enforcing the placement decisions, and for moving the system smoothly between a placement decision made in the last cycle to a new decision produced in the most recent cycle. The Task Scheduler schedules tasks according to the placement decision made by the Placement Controller. Whenever a task completes, it is the responsibility of the Task Scheduler to select a new task to execute in the freed slot, by providing a task of the appropriate type from the appropriate job to the given TaskTracker.

In the following sections we will concentrate on the problem solved by the Placement Algorithm component in a single control cycle.

3.3 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 completion 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:

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

where u_j^M is a utility function that denotes increasing satisfaction of a job given a placement of map tasks, and u_j^R is a utility function that shows satisfaction of a job given a placement of reduce tasks. The definition of both functions is:

$$u_j^M(P^M) = \begin{cases} \frac{s^j_{alloc} - s^j_{req}}{s^j_{pend} - s^j_{req}} & s^j_{alloc} \geq s^j_{req} \\ \frac{log(s^j_{alloc})}{log(s^j_{req})} - 1 & s^j_{alloc} < s^j_{req} \end{cases} \tag{2}$$

$$u_j^R(P^R) = \frac{log(r^j_{alloc})}{log(r^j_{pend})} - 1 \tag{3}$$

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.

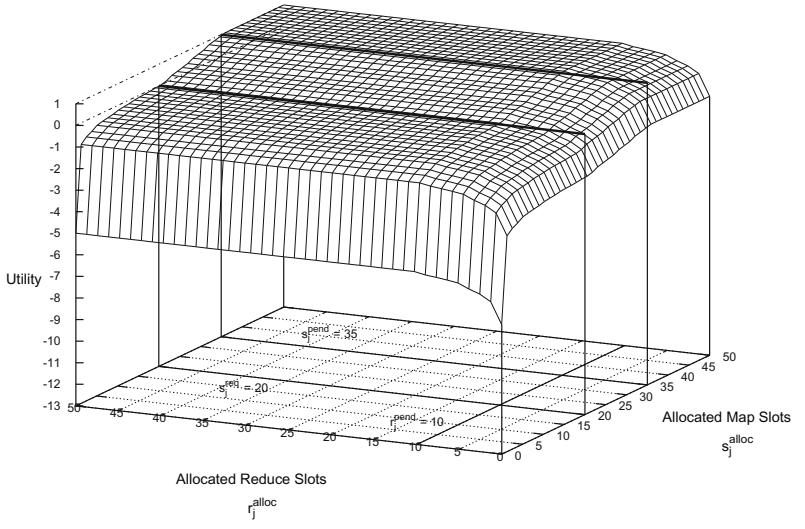


Fig. 2. Shape of the Utility Function when $s^j_{req} = 20$, $s^j_{pend} = 35$, and $r^j_{pend} = 10$

The definition of u differentiates between two 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; and (2) the function grows linearly between 0 and 1, 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_j^M is a monotonically increasing utility function, with values in the range $(-\infty, 1]$. The intuition behind this function is that a job is unsatisfied ($u_j^M < 0$) 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

($u_j^M = 0$) when the allocation equals the requirement ($s^j_{alloc} = s^j_{req}$), and its satisfaction is positive ($u_j^M > 0$) 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_j^R 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, $u_j(P) = u_j^M(P^M)$.

Figure 2 shows the generic shape of the utility function for a job that requires at least 20 map tasks to be allocated concurrently ($s^j_{req} = 20$) 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.

3.4 Placement Algorithm and Optimization Objective

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 RAS 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:

$$\max \quad \min_j u_j(P) \tag{4}$$

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

such that

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

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], and which is outlined in Algorithm 1. The proposed algorithm consists of two major steps: placing reduce tasks and placing map tasks.

Algorithm 1. Placement Algorithm run at each Control Cycle

Inputs $P^M(job,tt)$: Placement Matrix of Map tasks, $P^R(job,tt)$: Placement Matrix of Reduce tasks, J: List of Jobs in the System, D: Resource demand profile for each job, TT: List of TaskTrackers in the System
Γ_j and Ω_{tt}: Resource demand and capacity for each Job each TaskTracker correspondingly, as used by the auxiliary function $room\_for\_new\_job\_slot$

{——————— **Place Reducers** ———————}

for job in J **do**
  Sort TT in increasing order of overall number of reduce tasks placed (first criteria), and increasing order of number of reducers job placed (second criteria)
  for tt in TT **do**
    if $room\_for\_new\_job\_slot(job, tt)$ & $r^{job}_{pend} > 0$ **then**
      $P^R(job, tt) = P^R(job, tt) + 1$
    end if
  end for
end for

{——————— **Place Mappers** ———————}

for $round = 1 \ldots rounds$ **do**
  for tt in TT **do**
    $job_{in} \leftarrow min\ U(job_{in}, P)$, $room\_for\_new\_job\_slot(job_{in}, tt)$,
    $job_{out} \leftarrow max\ U(job_{out}, P)$, $P^M(job_{out}, tt) > 0$
    repeat
      $P_{old} \leftarrow P$
      $job_{out} \leftarrow max\ U(job_{out}, P)$, $P(job_{out}, tt) > 0$
      $P^M(job_{out}, tt) = P^M(job_{out}, tt) - 1$
      $job_{in} \leftarrow min\ U(job_{in}, P)$, $room\_for\_new\_job\_slot(job_{in}, tt)$
    until $U(job_{out}, P) < U(job_{in}, P_{old})$
    $P \leftarrow P_{old}$
    repeat
      $job_{in} \leftarrow min\ U(job_{in}, P)$, $room\_for\_new\_job\_slot(job_{in}, tt)$
      $P^M(job_{in}, tt) = P^M(job_{in}, tt) + 1$
    until $\nexists job$ such that $room\_for\_new\_job\_slot(job, tt)$
  end for
end for

if map phase of a job is about to complete in this control cycle **then**
  switch profile of placed reducers from shuffle to reduce and wait for Task Scheduler to drive the transition.
end if

Reduce tasks are placed first to allow them to be evenly distributed across TaskTrackers. By doing this we allow reduce tasks to better multiplex network resources when pulling intermediate data and also enable better storage usage. The placement algorithm distributes reduce tasks evenly across TaskTrackers while avoiding collocating any two reduce tasks. If this is not feasible—due to the total number of tasks—it then gives preference to avoiding collocating reduce tasks from the same job. Recall that in contrast to other existing schedulers, RAS dynamically adjusts the number of map and reduce tasks allocated per TaskTracker while respecting its resource constraints. Notice also that when

reduce tasks are placed first, they start running in shuffle phase, so that their demand of resources is directly proportional to the number of map tasks placed for the same job. Therefore, in the absence of map tasks for the same job, a reduce task in shuffle phase only consumes memory. It therefore follows that the system is unlikely to be fully booked by reduce tasks[1].

The second step is placing map tasks. This stage of the algorithm is utility-driven and seeks to produce a placement matrix that balances satisfaction across jobs while treating all jobs fairly. This is achieved by maximizing the lowest utility value in the system. This part of the algorithm executes a series of rounds, each of which tries to improve the lowest utility of the system. In each round, the algorithm removes allocated tasks from jobs with the highest utility, and allocates more tasks to the jobs with the lowest utility. For the sake of fairness, a task gets de-allocated only if the utility of its corresponding job remains higher than the lowest utility of any other job in the system. This results in increasing the lowest utility value across jobs in every round. The loop stops after a maximum number of rounds has reached, or until the system utility no longer improves. This process allows for satisfying the optimization objective introduced in Equation 4.

Recall that RAS is resource-aware and hence all decisions to remove and place tasks are made considering the resource constraints and demands in the system. Furthermore, in order to improve system utilization it greedily places as many tasks as resources allow. This management technique is novel and allows for satisfying the optimization objective introduced in Equation 5.

The final step of the algorithm is to identify if any running jobs will complete their map phase during the current control cycle. This transition is important because it implies that reduce tasks for those jobs will start the reduce phase. Therefore, the algorithm has to switch the resource demand profile for the reduce tasks from 'shuffle' to 'reduce'. Notice that this change could overload some TaskTrackers in the event that the 'reduce' phase of the reduce tasks uses more resources than the 'shuffle' phase. RAS handles this by having the Task Scheduler drive the placement transition between control cycles, and provides overload protection to the TaskTrackers.

3.5 Task Scheduler

The Task Scheduler drives transitions between placements while ensuring that the actual demand of resources for the set of tasks running in a TaskTracker does not exceed its capacity. The placement algorithm generates new placements, but these are not immediately enforced as they may overload the system due to tasks still running from the previous control cycle. The Task Scheduler component takes care of transitioning without overloading any TaskTrackers in the system by picking jobs to assign to the TaskTracker that do not exceed its current capacity, sorted by lowest utility first. For instance, a TaskTracker

[1] We present our thoughts on how to handle the pathological case wherein the number of reduce tasks is so large that there is not enough memory for deploying more tasks in Section 6.

that is running 2 map tasks of job A may have a different assignment for the next cycle, say, 4 map tasks of job B. Instead of starting the new tasks right away while the previous ones are still running, new tasks will only start running as previous tasks complete and enough resources are freed. Recall that the scheduler is adaptive as it continuously monitors the progress of jobs and their average task length, so that any divergence between the placement matrix produced by the algorithm and the actual placement of tasks enforced by the Task Scheduler component is noticed and considered in the following control cycle. The Task Scheduler component is responsible for enforcing the optimization objective shown in Equation 6.

3.6 Job Profiles

The proposed job scheduling technique relies on the use of job profiles containing information about the resource consumption for each job. Profiling is one technique that has been successfully used in the past for MapReduce clusters. Its suitability in these clusters stems from the fact that in most production environments jobs are ran periodically on data corresponding to different time windows [4]. Hence, profiles remains fairly stable across runs [8].

Our profiling technique works offline. To build a job profile we run a job in a sandbox environment with the same characteristics of the production environment. We run the job in isolation multiple times in the sandbox using different configurations for the number of map task slots per node (1 map, 2 maps, ..., up to N). The number of reduce tasks is set to the desired level by the user submitting the job. In the case of multiple reduce tasks, they execute on different nodes.

From the multiple configurations, we select that one in which the job completed fastest, and use that execution to build our profile. We monitor CPU, I/O and memory usage in each node for this configuration using `vmstat`. The reasoning behind this choice is that we want to monitor the execution of a configuration in which competition for resources occurs and some system bottlenecks are hit, but in which severe performance degradation is not yet observed.

Note that obtaining CPU and memory is straight forward for the various phases. For example, if the bottleneck is CPU (that is to say, the node experiences 100% CPU utilization) and there are 4 map tasks running, each map task consumes 25% CPU. Profiling I/O in the shuffle phase is less trivial. Each reduce task has a set of threads responsible for pulling map outputs (intermediate data generated by the map tasks): the number of these threads is a configurable parameter in Hadoop (hereafter `parallelCopies`). These threads are informed about the availability and location of a new map output whenever a map task completes. Consequently, independent of the number of map outputs available, the reduce tasks will never fetch more than `parallelCopies` map outputs concurrently. During profiling we ensure that there are at least `parallelCopies` map outputs available for retrieval and we measure the I/O utilization in the reduce task while shuffling. It can therefore be seen that our disk I/O measurement is effectively an upper bound on the I/O utilization of the shuffle phase.

In RAS we consider jobs that run periodically on data with uniform characteristics but different sizes. Since the map phase processes a single input split of fixed size and the shuffle phase retrieves `parallelCopies` map outputs concurrently (independently of the input data size) their resource profile remain similar. Following these observations, the completion time of the map tasks remains the same while the completion time of the shuffle phase may vary depending on the progress rate of the map phase. The case of the reduce phase is more complicated. The reducer phase processes all the intermediate data at once and this one tends to increase (for most jobs we know of) as the input data size increases. In most of the jobs that we consider we observe that the completion time of the reduce phase scales linearly. However, this is not always the case. Indeed, if the job has no reduce function and simply relies on the shuffle phase to sort, we observe that the completion time scales super-linearly ($n.log(n)$). Having said that, our approach can be improved, for example by using historical information.

Profile accuracy plays a role in the performance of RAS. Inaccurate profiles lead to resource under- or overcommitment. This dependency exists in a slot-based system too, as it also requires some form of profiling to determine the optimal number of slots. The optimal slot number measures a job-specific capacity of a physical node determined by a bottleneck resource for the job, and it can be easily converted into an approximate resource profile for the job (by dividing bottleneck resource capacity by the slot number). Provided with these profiles, RAS allows jobs with different optimal slot numbers to be co-scheduled, which is a clear improvement over classical slot-based systems. The profiling technique used in this paper allows multi-resource profiles to be built, which helps improve utilization when scheduling jobs with different resource bottlenecks. Since the sandbox-based method of profiling assumes that resource utilization remains stable among different runs of the same job on different data, it may fail to identify a correct profile for jobs that do not meet this criterion. For those jobs, an online profiler or a hybrid solutions with reinforcement learning may be more appropriate since RAS is able to work with profiles that change dynamically and allows different profiling technologies to be used for different jobs. While we are not addressing them in this paper, such techniques have been studied in [11–13].

4 Evaluation

In this section we include results from two experiments that explore the two objectives of RAS: improving resource utilization in the cluster (Experiment 1) and meeting jobs' completion time goals (Experiment 2). In Experiment 1, we consider resource utilization only, and compare RAS with a state-of-the-art non-resource-aware Hadoop scheduler. In order to gain insight on how RAS improves resource utilization, we set a relaxed completion time goal with each job. This allow us to isolate both objectives and reduce the effect of completion time goals in the algorithm. In Experiment 2, we consider completion time goals on the same workload. Thus effectively evaluating all capabilities of RAS.

4.1 Experimental Environment and Workload

We perform all our experiments on a Hadoop cluster consisting of 22 2-way 64-bit 2.8GHz Intel Xeon machines. Each machine has 2GB of RAM and runs a 2.6.17 Linux kernel. All machines in the cluster are connected via a Gigabit Ethernet network. The version of Hadoop used is 0.23. The cluster is configured such that one machine runs the JobTracker, another machine hosts the NameNode, and the remaining 20 machines each host a DataNode and a TaskTracker.

Table 1. Workload: 3 Applications, 3 Job instances each (Big, Medium, and Small)

	Sort	**Combine**	**Select**
Instance Label	J1/J8/J9	J2/J6/J7	J3/J4/J5
Input size	90GB/19GB/6GB	5GB/13GB/50GB	10GB/25GB/5GB
Submission time	0s/2,500s/3,750s	100s/600s/1,100s	200s/350s/500s
Length in isolation	2,500s/350s/250s	500s/750s/2,500s	400s/280s/50s
	Experiment 1		
Completion time	3,113s/3,670s/4,100s	648s/3,406s/4,536s	1,252s/608s/623s
	Experiment 2		
Completion time	3,018s/3,365s/4,141s	896s/2,589s/4,614s	802s/550s/560s
Completion time goal	3,000s/3,400s/4,250s	850s/2,600s/6,000s	1,250s/1,100s/950s

To evaluate RAS we consider a representative set of applications included in the Gridmix benchmark, which is part of the Hadoop distribution. These applications are Sort, Combine and Select. For each application we submit 3 different instances with different input sizes, for a total of 9 jobs in each experiment. A summary of the workload can be seen in Table 1, including the label used for each instance in the experiments, the size of its associated input data set, the submission time, and the time taken by each job to complete if the entire experimental cluster is dedicated to it. Additionally, we include the actual completion times observed for each instance in Experiment 1 and 2. Finally, for Experiment 2, we include also the completion time goal associated to each instance.

The resource consumption profiles provided to the scheduler are shown in Table 2. They were obtained following the description provided in Section 3.6. The values are the percentage of each TaskTracker's capacity that is used by a single execution of the sub-phase in question.

Table 2. Job profiles (shuffle: consumed I/O per map placed, upper bound set by `parallelCopies`, the number of threads that pull map output data)

	Sort	**Combine**	**Select**
	Map/Shuffle/Reduce	Map/Shuffle/Reduce	Map/Shuffle/Reduce
CPU	30%/-/20%	25%/-/10%	15%/-/10%
I/O	45%/0.15%/50%	10%/0.015%/10%	20%/0.015%/10%
Memory	25%/-/60%	10%/-/25%	10%/-/25%

4.2 Experiment 1: Execution with Relaxed Completion Time Goals

The goal of this experiment is to evaluate how RAS improves resource utilization compared to the Fair Scheduler when completion time goals are so relaxed that the main optimization objective of the algorithm is to maximize resource utilization (see Equation 5). To this end we associate with each job instance an highly relaxed completion time goal. We run the same workload using both the Fair Scheduler and RAS and compare different aspects of the results.

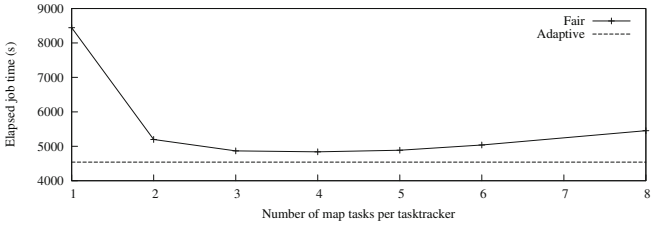
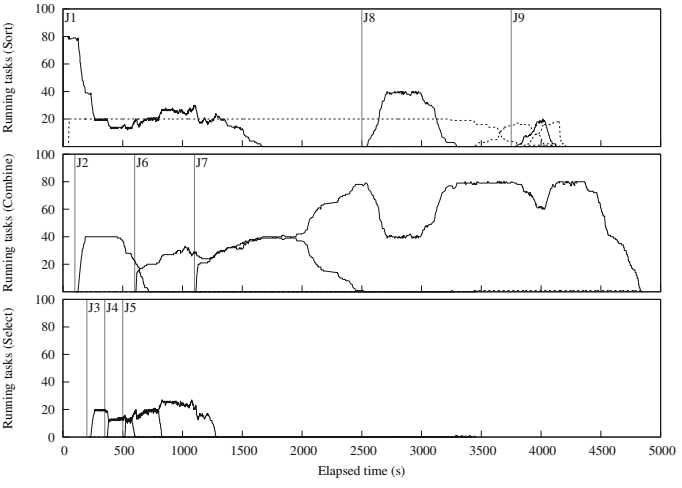


Fig. 3. Experiment 1: Workload makespan with different Fair Scheduler configurations (Y-axis starts at 4000s)

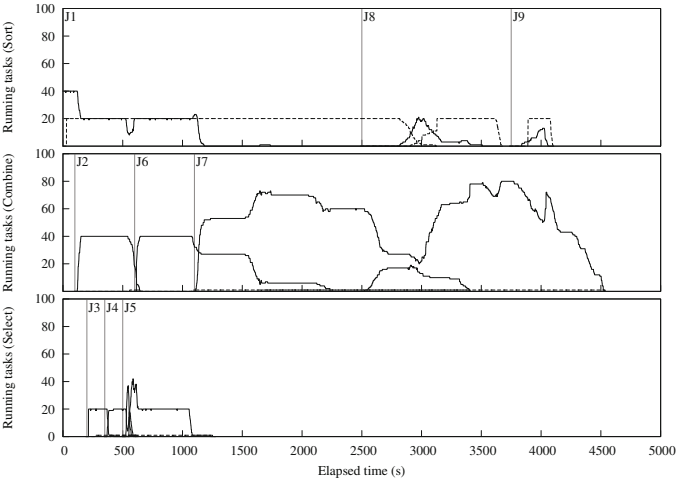
Dynamic task concurrency level per TaskTracker. Our first objective in this experiment is to study how the dynamic management of the level of task concurrency per-TaskTracker improves workload performance. To this end, we run the same workload using the Fair Scheduler with different concurrency level configurations: specifically, we vary the maximum number of map slots per TaskTracker from 1 to 8, and compare the results with the execution using RAS. Results are shown in Figure 3. As can be seen, the best static configuration uses 4 concurrent map tasks per TaskTracker (80 concurrent tasks across 20 TaskTrackers). Configurations that result in low and high concurrency produce worse makespan due to resources being underutilized and overcommitted, respectively.

We observe that RAS outperforms the Fair Scheduler for all configurations, showing an improvement that varies between 5% and 100%. Our traces show that the average task concurrency level in RAS was 3.4 tasks per TaskTracker. Recall that the best static configuration of per-TaskTracker task concurrency depends on the workload characteristics. As workloads change over time in real systems, even higher differences between static and dynamic management would be observed. RAS overcomes this problem by dynamically adapting task concurrency level based on to the resource usage in the system.

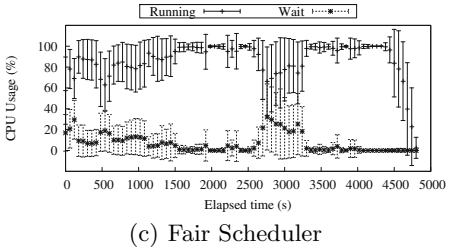
Resource allocation and Resource utilization. Now we look in more detail at the execution of the workload using RAS as compared to the Fair Scheduler running a static concurrency level of 4. Figures 4(a) and 4(b) show the task assignment resulting from both schedulers. For the sake of clarity, we group jobs corresponding to the same application (Sort, Combine or Select) into different rows. Each row contains solid and dotted lines, representing the number of running map and reduce tasks respectively. The submission time for each job is shown by a (labeled) vertical line, following the convention presented in Table 1.



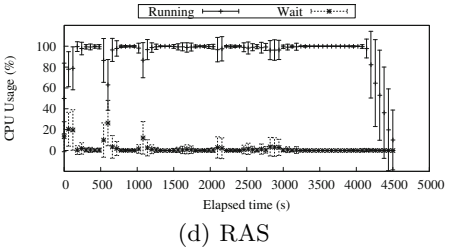
(a) Fair Scheduler



(b) RAS



(c) Fair Scheduler



(d) RAS

Fig. 4. Experiment 1: Workload execution and CPU utilization: (a) and (c) correspond to Fair Scheduler using 4 slots per TaskTracker; (b) and (d) correspond to RAS using a variable number slots per TaskTracker

Combine and Select are configured to run one single reduce task per job since there is no benefit from running them with more reduce tasks on our testing environment; the dotted line representing the reduce is at the bottom of the chart. As it can be observed, RAS does not allocate more concurrent map slots than the Fair Scheduler during most of the execution. Moreover, the sum of reduce and map tasks remains lower than the sum of reduce and map tasks allocated by the Fair Scheduler except for a small time interval ($\sim$ 100s) immediately after the submission of Job 6 (J6). RAS is able to improve the makespan while maintaining a lower level of concurrency because it does better at utilizing resources which ultimately results in better job performance. To get a better insight on how RAS utilizes resources as compared to the Fair Scheduler we plot the CPU utilization for both schedulers in Figures 4(c) and 4(d). These figures show the percentage of CPU time that TaskTrackers spent running tasks (either in system or user space), and the time that the CPU was waiting. For each metric we show the mean value for the cluster, and the standard deviation across TaskTrackers. Wait time represents the time that the CPU remains idle because all threads in the system are either idle or waiting for I/O operations to complete. Therefore, it is a measure of resource wastage, as the CPU remains inactive. While wait time is impossible to avoid entirely, it can be reduced by improving the overlapping of tasks that stress different resources in the TaskTracker. It is noticeable that in the case of the Fair Scheduler the CPU spends more time waiting for I/O operations to complete than RAS. Further, modifying the number of concurrent slots used by the Fair Scheduler does not improve this result. The reason behind this observation is key to our work: other schedulers do not consider the resource consumption of applications when making task assignment decisions, and therefore are not able to achieve good overlap between I/O and CPU activity.

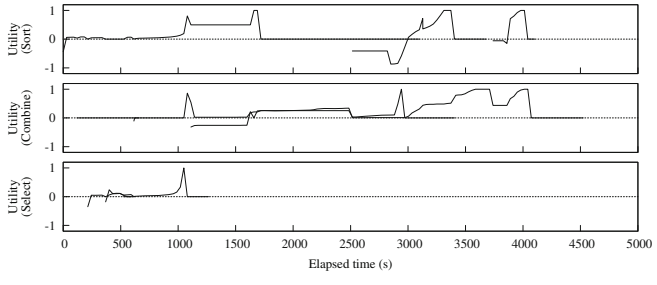


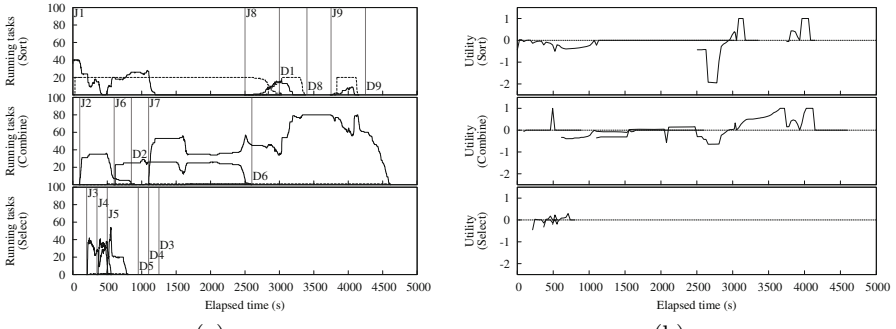
Fig. 5. Experiment 1: Job Utility

Utility guidance. Finally, to illustrate the role of the utility function in RAS, Figure 5 shows the utility value associated with each job during the execution of the workload. Since the jobs have extremely lax completion time goals, they are assigned a utility value above 0 immediately after one task for each job is placed. As can be seen, the allocation algorithm balances utility values across jobs for most of the execution time. In some cases, though, a job may get higher utility than the others: this is explained by the fact that as jobs get closer to

completion, the same resource allocation results in higher utility. This is seen in our experiments: for all jobs, the utility increases until all their remaining tasks are placed. In this experiment we can also see that Job 7 has a very low utility right after it is launched (1,100s) in contrast with the relatively high utility of Job 1, even though most resources are actually assigned to Job 7. This is because while Job 1 has very few remaining tasks, no tasks from Job 7 have been completed and thus its resource demand estimation is not yet accurate. This state persists until approximately time 1,650s).

4.3 Experiment 2: Execution with Tight Completion Time Goals

In this experiment we evaluate the behavior of RAS when the applications have stringent completion time goals. To do this we associate a tight completion time goal with the workload described for our previous experiment.



(a) (b)

Fig. 6. Experiment 2: Workload execution and Job utility

Figure 6(a) shows the number of concurrent tasks allocated to each job during the experiment. We use vertical lines and labels to indicate submission times (labeled J1 to J9) and completion time goals (labeled D1 to D9) for each of the nine jobs in the workload. To illustrate how RAS manages the tradeoff between meeting completion time goals and maximizing resource utilization, we look at the particular case of Job 1 (Sort), Job 7 (Combine) and Job 8 (Sort), submitted at times J1 (0s), J7 (1,100s) and J8 (2,500s) respectively. In Experiment 1 their actual completion times were 3,113s, 4,536s and 3,670s, while in Experiment 2 they completed at times 3,018s, 4,614s and 3,365s respectively. Because their completion time goals in Experiment 2 are 3,000s, 6,000s and 3,400s (a factor of 1.2X, 1.9X and 2.5X compared to their length observed in isolation), the algorithm allocates more tasks to Job 1 and Job 8 at the expense of Job 7, which sees its actual completion time delayed with respect to Experiment 1 but still makes its more relaxed goal. It is important to remark again that completion time goals in our scheduler are soft deadlines used to guide the workload management as opposed to strict deadlines in which missing a deadline is associated with

strong penalties. Finally, notice that Job 1 and Job 8 would have clearly missed their goals in Experiment 1: here, however, RAS adaptively moves away from the optimal placement in terms of resource allocation to adjust the actual completion times of jobs. Recall that RAS is still able to leverage a resource model while aiming at meeting deadlines, and still outperforms the best configuration of Fair Scheduler by 167 seconds, 4,781s compared to 4,614s.

To illustrate how utility is driving placement decisions, we include Figure 6(b), which shows the utility of the jobs during the workload execution and gives a better intuition of how the utility function drives the scheduling decisions. When a job is not expected to reach its completion time goal with the current placement, its utility value goes negative. For instance, starting from time 2,500s when J8 is launched and the job still has very few running tasks, the algorithm places new tasks to J8 at the expense of J7. However, as soon as J8 is running the right amount of tasks to reach the deadline, around time 3,000s, both jobs are balanced again and the algorithm assigns more tasks to J7.

5 Related Work

Much work have been done in the space of scheduling for MapReduce. Since the number of 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 [14] is a pluggable scheduler developed by Yahoo! which partition resources into pools and provides priorities for each pool. Hadoop's Fair Scheduler [9] allocates equal shares to each tenant in the cluster. Quincy scheduler [15] proposed for the Dryad environment [16] also shares similar fairness goals. All these schedulers are built on top of the slot model and do not support user-level goals.

The performance of MapReduce jobs has attracted much interest in the Hadoop community. Stragglers, tasks that take an unusually long time to complete, have been shown to be the most common reason why the total time to execute a job increases [2]. Speculative scheduling has been widely adopted to counteract the impact of stragglers [2, 9]. Under this scheduling strategy, when the scheduler detects that a task is taking longer than expected it spawns multiple instances of the task and takes the results of the first completed instance, killing the others [9]. In Mantri [5] the effect of stragglers is mitigated via the 'kill and restart' of tasks which have been identified as potential stragglers. The main disadvantage of these techniques is that killing and duplicating tasks results in wasted resources [5, 9]. In RAS we take a more proactive approach, in that we prevent stragglers resulting from resource contention. Furthermore, stragglers caused by skewed data cannot be avoided at run-time [5] by any existing technique. In RAS the slow-down effect that these stragglers have on the end-to-end completion time of their corresponding jobs is mitigated by allocating more resources to the job so that it can still complete in a timely manner.

Recently, there has been increasing interest in user-centric data analytics. One of the seminal works in this space is [6]. In this work, the authors propose a scheduling scheme that enables soft-deadline support for MapReduce jobs. It

differs from RAS in that it does not take into consideration the resources in the system. Flex [7] 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 suffers from the same limitations we mentioned earlier. One of the first works in considering resource awareness in MapReduce clusters is [17]. In this work the scheduler classifies tasks into good and bad tasks depending on the load they impose in the worker machines. More recently, the Hadoop community has also recognized the importance of developing a resource-aware scheduling for MapReduce. [18] outlines the vision behind the Hadoop scheduler of the future. The framework proposed introduces a resource model consisting of a 'resource container' which is—like our 'job slot'—fungible across job tasks. We think that our proposed resource management techniques can be leveraged within this framework to enable better resource management.

6 Conclusions

In this paper we have presented the Resource-aware Adaptive Scheduler, RAS, which introduces a novel resource management and job scheduling scheme for MapReduce. RAS is capable of improving resource utilization and job performance. The cornerstone of our scheduler is a resource model] based on a new resource abstraction, namely 'job slot'. This model allows for the formulation of a placement problem which RAS solves 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.

The presented scheduler relies on existing profiling information based on previous executions of jobs to make scheduling and placement decisions. Profiling of MapReduce jobs that run periodically on data with similar characteristics is an easy task, which has been used by many others in the community in the past. RAS pioneers a novel technique for scheduling reduce tasks by incorporating them into the utility function driving the scheduling algorithm. It works in most circumstances, while in some others it may need to rely on preempting reduce tasks (not implemented in the current prototype) to release resources for jobs with higher priority. Managing reduce tasks in this way is not possible due to limitations in Hadoop and hence it affects all existing schedulers. In RAS we consider three resource capacities: CPU, memory and I/O. It can be extended easily to incorporate network infrastructure bandwidth and storage capacity of the TaskTrackers. Nevertheless, network bottlenecks resulting from poor placement of reduce tasks [5] can not be addressed by RAS without additional monitoring and prediction capabilities.

Our experiments, in a real cluster driven by representative MapReduce workloads, demonstrate the effectiveness of our proposal. To the best of our knowledge

RAS is the first scheduling framework to use a new resource model in MapReduce and leverage 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 (FEDER funds) under contract TIN2007-60625, by IBM through the 2010 IBM Faculty Award program, and by the HiPEAC European Network of Excellence (IST-004408).

References

1. Adaptive Scheduler, `https://issues.apache.org/jira/browse/MAPREDUCE-1380`
2. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: OSDI 2004, San Francisco, CA, pp. 137–150 (December 2004)
3. Hadoop MapReduce, `http://hadoop.apache.org/mapreduce/`
4. Thusoo, A., Shao, Z., Anthony, S., Borthakur, D., Jain, N., Sen Sarma, J., Murthy, R., Liu, H.: Data warehousing and analytics infrastructure at facebook. In: Proceedings of the 2010 International Conference on Management of Data, SIGMOD 2010, pp. 1013–1020. ACM, New York (2010)
5. Ananthanarayanan, G., Kandula, S., Greenberg, A., Stoica, I., Lu, Y., Saha, B., Harris, E.: Reining in the outliers in map-reduce clusters using mantri. In: OSDI 2010, pp. 1–16. USENIX Asoc., Berkeley (2010)
6. Polo, J., Carrera, D., Becerra, Y., Steinder, M., Whalley, I.: Performance-driven task co-scheduling for MapReduce environments. In: Network Operations and Management Symposium, NOMS, pp. 373–380. IEEE, Osaka (2010)
7. Wolf, J., Rajan, D., Hildrum, K., Khandekar, R., Kumar, V., Parekh, S., Wu, K.-L., Balmin, A.: Flex: A Slot Allocation Scheduling Optimizer for Mapreduce Workloads. In: Gupta, I., Mascolo, C. (eds.) Middleware 2010. LNCS, vol. 6452, pp. 1–20. Springer, Heidelberg (2010)
8. Verma, A., Cherkasova, L., Campbell, R.H.: ARIA: Automatic Resource Inference and Allocation for MapReduce Environments. In: 8th IEEE International Conference on Autonomic Computing, Karlsruhe, Germany (June 2011)
9. Zaharia, M., Konwinski, A., Joseph, A.D., Katz, R., Stoica, I.: Improving mapreduce performance in heterogeneous environments. In: OSDI 2008, pp. 29–42. USENIX Association, Berkeley (2008)
10. Tang, C., Steinder, M., Spreitzer, M., Pacifici, G.: A scalable application placement controller for enterprise data centers. In: Procs. of the 16th Intl. Conference on World Wide Web, pp. 331–340. ACM, NY (2007)
11. Herodotou, H., Babu, S.: Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs. In: VLDB (2010)
12. Pacifici, G., Segmuller, W., Spreitzer, M., Tantawi, A.N.: Dynamic estimation of cpu demand of web traffic. In: Lenzini, L., Cruz, R.L. (eds.) VALUETOOLS. ACM International Conference Proceeding Series, vol. 180, p. 26. ACM (2006)
13. Tesauro, G., Jong, N.K., Das, R., Bennani, M.N.: A hybrid reinforcement learning approach to autonomic resource allocation. In: Proceedings of the 2006 IEEE International Conference on Autonomic Computing, pp. 65–73. IEEE Computer Society, Washington, DC (2006)
14. Yahoo! Inc. Capacity scheduler, `http://developer.yahoo.com/blogs/hadoop/posts/2011/02/capacity-scheduler/`

15. Isard, M., Prabhakaran, V., Jon Currey, U.W., Talwar, K., Goldberg, A.: Quincy: fair scheduling for distributed computing clusters. In: SOSP 2009 (2009)
16. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: distributed data-parallel programs from sequential building blocks. In: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems, EuroSys 2007, pp. 59–72. ACM, New York (2007)
17. Dhok, J., Varma, V.: Using pattern classification for task assignment, http://researchweb.iiit.ac.in/~jaideep/jd-thesis.pdf
18. Murthy, A.: Next Generation Hadoop, http://developer.yahoo.com/blogs/hadoop/posts/2011/03/mapreduce-nextgen-scheduler/