An Adaptive Scheduling Algorithm for Dynamic Heterogeneous Hadoop Systems

Aysan Rasooli , Douglas G. Down

Department of Computing and Software
McMaster University
{*rasooa, downd*}*@mcmaster.ca*

Abstract

The MapReduce and Hadoop frameworks were designed to support efficient large scale computations. There has been growing interest in employing Hadoop clusters for various diverse applications. A large number of (heterogeneous) clients, using the same Hadoop cluster, can result in tensions between the various performance metrics by which such systems are measured. On the one hand, from the service provider side, the utilization of the Hadoop cluster will increase. On the other hand, from the client perspective the parallelism in the system may decrease (with a corresponding degradation in metrics such as mean completion time). An efficient scheduling algorithm should strike a balance between utilization and parallelism in the cluster to address performance metrics such as fairness and mean completion time. In this paper, we propose a new Hadoop cluster scheduling algorithm, which uses system information such as estimated job arrival rates and mean job execution times to make scheduling decisions. The objective of our algorithm is to improve mean completion time of submitted jobs. In addition to addressing this concern, our algorithm provides competitive performance under fairness and locality metrics (with respect to other well-known Hadoop scheduling algorithms - Fair Sharing and FIFO). This approach can be efficiently applied in heterogeneous clusters, in contrast to most Hadoop cluster scheduling algorithm work, which assumes homogeneous clusters. Using simulation, we demonstrate that our algorithm is a very promising candidate for deployment in real systems.

Copyright © 2011 Aysan Rasooli and Dr. Douglas G. Down. Permission to copy is hereby granted provided the original copyright notice is reproduced in copies made.

1 Introduction

Cloud computing provides massive clusters for efficient large scale computation and data analysis. MapReduce [5] is a well-known programming model which was first designed for improving the performance of large batch jobs on cloud computing systems. However, there is growing interest in employing MapReduce and its open-source implementation, called Hadoop, for various types of jobs. This leads to sharing a single Hadoop cluster between multiple users, which run a mix of long batch jobs and short interactive queries on a shared data set.

Sharing a Hadoop cluster between multiple users has several advantages, such as statistical multiplexing (lowering costs), data locality (running computation where the data is), and increasing the utilization of the resources. Dur-

ing the past years companies have used Hadoop clusters for executing specific applications; for instance Facebook is using a Hadoop cluster for analyzing usage patterns to improve site design, spam detection, data mining and ad optimization [10].

Assigning a new type of job to the current workload mix on a Hadoop cluster, may severely degrade the performance of the system, which may not be tolerable for certain applications. Moreover, heterogeneity is a neglected issue in most current Hadoop systems, which can also lead to poor performance. Here, heterogeneity can be with respect to both jobs and resources.

The Hadoop scheduler is the centrepiece of a Hadoop system. Desired performance levels can be achieved by proper submission of jobs to resources. Primary Hadoop scheduling algorithms, like the FIFO algorithm and the Fair-sharing algorithm, are simple algorithms which use small amounts of system information to make quick scheduling decisions. The main concern in these algorithms is to quickly multiplex the incoming jobs on the available resources. Therefore, they use less system information. However, a scheduling decision based on a small amount of system information may cause some disadvantages such as less locality, and neglecting the heterogeneity of the system. As a result, the primary Hadoop scheduling algorithms may not be good choices for heterogeneous systems. In a heterogeneous Hadoop system, increasing parallelism without considering the difference between various resources and jobs in the system can result in poor performance. As a result of such considerations, it is useful to explore the possible performance gains by considering more sophisticated algorithms.

Gathering more system information can have significant impact on making better scheduling decisions. It is possible to gather some Hadoop system information [7], which can be used in scheduling decisions. Research at UC-Berkeley [2] provides a means to estimate the mean job execution time based on the structure of the job, and the number of map and reduce tasks in each job. Moreover, in most Hadoop systems, multiple types of jobs are repeating according to various patterns. For example, the Spam detector applications run on the Facebook Hadoop cluster every night. Therefore, it is also may be possible to estimate the arrival rates of job types in some Hadoop systems.

In this paper, we introduce a Hadoop scheduling algorithm which uses this system information to make appropriate scheduling decisions. The proposed algorithm takes into account the heterogeneity of both resources and jobs in assigning jobs to resources. Using the system information, it classifies the jobs into classes and finds a matching of the job classes to the resources based on the requirements of the job classes and features of the resources. At the time of a scheduling decision, the algorithm uses the matchings of the resources and the classes, and considers the priority, required minimum share, and fair share of users to make a scheduling decision. Our proposed algorithm is dynamic, and it updates its decisions based on changes in the parameters of the system.

We extend a Hadoop simulator, MRSIM [6], to evaluate our proposed algorithm. We implement the four most common performance metrics for Hadoop systems: locality, fairness, satisfying the minimum share of the users, and mean completion time of jobs. We compare the performance of our algorithm with two commonly used Hadoop scheduling algorithms, the FIFO algorithm and the Fair-sharing algorithm [9]. The results show that our proposed algorithm has significantly better performance in reducing the mean completion time, and satisfying the required minimum shares. Moreover, its performance in the Locality and the Fairness performance metrics is very competitive with the other two algorithms. To the best of our knowledge, there is no Hadoop scheduling algorithm which simultaneously considers job and resource heterogeneity. The two main advantages of our proposed algorithm are increasing the utilization of the Hadoop cluster, and reducing the mean completion times by considering the heterogeneity in the Hadoop system.

The remainder of this paper is organized as follows. In Section 2 we give a brief overview of a Hadoop system. Current Hadoop scheduling algorithms are given in Section 3. Our Hadoop system model is described and formally introduced in Section 4. Then, in Section 5, we formally present the performance metrics of in-

terest. Our proposed Hadoop scheduling algorithm is introduced in Section 6. In Section 7, details of the environment in which we study our algorithm are provided, and we study the performance of our algorithm in various Hadoop systems. Finally, we provide some concluding remarks and discuss possible future work in the last section.

2 Hadoop Systems

Computing todays' large scale data processing applications requires thousands of resources. Cloud computing is a paradigm to provide such levels of computing resources. However, to harness the available resources for improving the performance of large applications, it is required to break down the applications into smaller pieces of computation, and execute them in parallel. MapReduce [5] is a programming model which provides an efficient framework for automatic parallelization and distribution, I/O scheduling, and monitoring the status of large scale computations and data analysis.

Hadoop is an open-source implementation of MapReduce for reliable, scalable, and distributed computing. A distributed file system that underlies the Hadoop system provides efficient and reliable distributed data storage for applications involving large data sets. Users in Hadoop submit their jobs to the system, where each job consists of map functions and reduce functions. The Hadoop system breaks the submitted jobs into multiple map and reduce tasks. First, Hadoop runs the map tasks on each block of the input, and computes the key/value pairs from each part of the input. Then, it groups intermediate values by their key. Finally, it runs the reduce tasks on each group, which provides the jobs' final result.

The scheduler is a fundamental component of the Hadoop system. Scheduling in the Hadoop system is pool based, which means that when a resource is free, it sends a heartbeat to the scheduler. Upon receiving a heartbeat, the scheduler searches through all the queued jobs in the system, chooses a job based on some performance metric(s), and sends one task of the selected job to each free slot on the resource. The heartbeat message contains some information such as the number of currently free slots on the resource. Various Hadoop scheduling algorithms consider different performance metrics in making scheduling decision.

3 Related Work

MapReduce was initially designed for small teams in which a simple scheduling algorithm like FIFO can achieve an acceptable performance level. However, experience from deploying Hadoop in large systems shows that basic scheduling algorithms like FIFO can cause severe performance degradation; particularly in systems that share data among multiple users. As a result, the next generation scheduler in Hadoop, Hadoop on Demand (HOD) [3], addresses this issue by setting up private Hadoop clusters on demand. HOD allows the users to share a common file system while owning private Hadoop clusters on their allocated nodes. This approach failed in practice because it violated the data locality design of the original MapReduce scheduler, and it resulted in poor system utilization. To address some of these shortcomings, Hadoop recently added a scheduling plug-in framework with two additional schedulers that extend rather than replace the original the FIFO scheduler.

The additional schedulers are introduced in [9], where they are collectively known as Fair-sharing. In this work, a pool is defined for each user, each pool consisting of a number of map slots and reduce slots on a resource. Each user can use its pool to execute her jobs. If a pool of a user becomes idle, the slots of the pool are divided among other users to speed up the other jobs in the system. The Fair-sharing algorithm does not achieve good performance regarding locality. Therefore, in order to improve the data locality, a complementary algorithm for Fair-sharing is introduced in [10], called delay scheduling. Using the delay scheduling algorithm, when Fair-sharing chooses a job for the current free resource, and the resource does not contain the required data of the job, scheduling of the chosen job is postponed, and the algorithm finds another job. However, to limit the waiting time of the jobs, a threshold is defined; therefore, if scheduling of a job is post-

poned until the threshold is met, the job will be submitted to the next free resource. The proposed algorithms can perform much better than Hadoop's default scheduling algorithm (FIFO); however, these algorithms do not consider heterogeneous systems in which resources have different capacities and users submit various types of jobs.

In [1], the authors introduce a scheduling algorithm for MapReduce systems to minimize the total completion time, while improving the CPU and I/O utilization of the cluster. The algorithm defines Virtual Machines (VM), and decides how to allocate the VMs to each Hadoop job, and to the physical Hadoop resources. The algorithm formulates and solves a constrained optimization problem. To formulate the optimization problem a mathematical performance model is required for the different jobs in the system. The algorithm first runs all job types in the Hadoop system to build corresponding performance models. Then, assuming these jobs will be submitted multiple times to the Hadoop system, scheduling decisions for each job are made based on the solution of the defined optimization problem. The algorithm assumes that the job characteristics will not vary between runs, and also when a job is going to be executed on a resource, all its required data is placed on that node. The problem with this algorithm is that the algorithm can not make decisions when a new job with new characteristics joins the system. Moreover, the assumption that all of the data required by a job is available on the running resource, without considering the overhead of transmitting the data is unrealistic. Furthermore, Hadoop is very I/O intensive both for file system access and Map/Reduce scheduling, so virtualization incurs a high overhead.

In [8], a Dynamic Priority (DP) parallel task scheduler is designed for Hadoop, which allows users to control their allocated capacity by dynamically adjusting their budgets. This algorithm prioritizes the users based on their spending, and allows capacity distribution across concurrent users to change dynamically based on user preferences. The core of this algorithm is a proportional share resource allocation mechanism that allows users to purchase or be granted a queue priority budget. This budget may be used to set spending rates denoting the willingness to pay a certain amount per Hadoop map or reduce task slot per time unit.

4 Hadoop System Model

The Hadoop system consists of a cluster, which is a group of linked resources. The data in the Hadoop system is organized into files. The users submit jobs to the system, where each job consists of some tasks. Each task is either a *map task* or a *reduce task*. The Hadoop components related to our research are described as follows:

1. The Hadoop system has a cluster. The cluster consists of a set of resources, where each resource has a computation unit, and a data storage unit. The computation unit consists of a set of slots (in most Hadoop systems, each CPU core is considered as one slot), and the data storage unit has a specific capacity. We assume a cluster with M resources as follows:

$$Cluster = \{R_1, \ldots, R_M\}$$

$$R_j = < Slots_j, Mem_j >$$

   - $Slots_j$ is a set of slots in resource R_j, where each slot ($slot_j^k$) has a specific execution rate ($exec_rate_j^k$). Generally, slots belonging to one resource have the same execution rate.

$$Slots_j = \{slot_j^1, \ldots, slot_j^m\}$$

   - Mem_j is the storage unit of resource R_j, which has a specific capacity ($capacity_j$) and data retrieval rate ($retrieval_rate_j$). The data retrieval rate of resource R_j depends on the bandwidth within the storage unit of this resource.

2. Data in the Hadoop system is organized into files, which are usually large. Each file is split into small pieces, which are called slices (usually, all slices in a system have the same size). We assume that there are L files in the system, which are defined as follows:

$$Files = \{F_1, \ldots, F_L\}$$

$$F_i = \{slice_i^1, \ldots, slice_i^k\}$$

3. We assume that there are N users in the Hadoop system, where each user (U_i) submits a set of jobs to the system ($Jobs_i$).

$$Users = \{U_1, \ldots, U_N\}$$

$$U_i =< Jobs_i >$$

$$Jobs_i = \{J_i^1, \ldots, J_i^n\}$$

The Hadoop system assigns a priority and a minimum share to each user based on a particular policy (e.g. the pricing policy of [8]). The number of slots assigned to user U_i depends on her priority ($priority_i$). The minimum share of a user U_i (min_share_i) is the minimum number of slots that the system must provide for user U_i at each point in time.

In a Hadoop system, the set of jobs of a user is dynamic, meaning that the set of jobs for user U_i at time t_1 may be completely different from the set of jobs of this user at time t_2. Each job in the system consists of a number of *map tasks*, and *reduce tasks*. The sets of *map tasks*, and *reduce tasks* for the job J_i is represented with $Maps_i$, and $Reds_i$, respectively.

$$J_i = Maps_i \cup Reds_i$$

Each *map task* k of job i (MT_i^k) performs some processes on the slice ($slice_j^l \in F_j$) where the required data for this task is located.

$$Maps_i = \{MT_i^1, \ldots, MT_i^k\}$$

Each *reduce task* k of job i (RT_i^k) receives and processes the results of some of the *map tasks* of job i.

$$Reds_i = \{RT_i^1, \ldots, RT_i^k\}$$

The $mean_execTime(J_i, R_j)$ defines the mean execution time of job J_i on resource R_j, and the corresponding execution rate is defined as follows:

$$mean_execRate(J_i, R_j) = 1/mean_execTime(J_i, R_j)$$

5 Performance Metrics

In this section, first we define the following functions which return the status of the Hadoop system and will be used to define the performance metrics. Then, we introduce the performance metrics related to our scheduling problem.

- $Tasks(U, t)$ and $Jobs(U, t)$ return the sets of tasks and jobs, respectively, of the user U at time t.
- $ArriveTime(J), StartTime(J)$, and $EndTime(J)$ return the arrival time, start of execution time, and completion time of job J, respectively.
- $TotalJobs(t)$ returns the set of all the jobs which have arrived to the system up to time t.
- $RunningTask(slot, t)$ returns the running task (if there is one) on the slot *slot* at time t. If there is no running task on the *slot*, the function returns $NULL$.
- $Location(slice, t)$ returns the set of resources (R), which store the slice *slice* at time t.
- $AssignedSlots(U, t)$ returns the set of slots which are executing the tasks of user U at time t.
- $CompletedJobs(t)$ returns the set of all jobs that have been completed up to time t. The function $CompletedTasks(t)$ is defined analogously for completed tasks.
- $Demand(U, t)$ returns the set of tasks of the user U at time t which have not yet been assigned to a slot.

Using the above functions, we define four performance metrics that are useful for a Hadoop system:

1. $MinShareDissatisfaction(t)$ measures how successful the scheduling algorithm is in satisfying the minimum share requirements of the users in the system. If there is a user in the system, whose current demand is not zero, and her current share

is lower than her minimum share, then we compute her Dissatisfaction as follows:

$$IF \quad (|Demand(U,t)| > 0 \wedge U.min_share > 0 \wedge |AssignedSlots(U,t)| < U.min_share)$$

$$THEN \quad Dissatisfaction(U,t) = \frac{U.min_share - |AssignedSlots(U,t)|}{U.min_share} \times U.priority$$

$$ELSE \quad Dissatisfaction(U,t) = 0$$

$U.priority$ and $U.min_share$ denote the priority and the minimum share of the user U. $MinShareDissatisfaction(t)$ takes into account the distance of all the users from their min_share. When comparing two algorithms, the algorithm which has smaller $MinShareDissatisfaction(t)$ has better performance.

$$MinShareDissatisfaction(t) = \sum_{\forall U \in Users} Dissatisfaction(U,t).$$

2. $Fairness(t)$ measures how fair a scheduling algorithm is in dividing the resources among the users in the system. A fair algorithm gives the same share of resources to users with equal priority. However, when the priorities are not equal, then the user's share should be proportional to their priority. In order to compute the fairness of an algorithm, we should take into account the slots which are assigned to each user beyond her minimum share, which is represented with $\Delta(U,t)$.

   $$\Delta(U,t) = AssignedSlots(U,t) - U.min_share$$

   Then, the average additional share of all the users with the same priority ($Users_p$) is defined as:

   $$avg(p,t) = \frac{\sum_{U \in Users_p} \Delta(U,t)}{|Users_p|}, \quad Users_p = \{U | U \in Users \wedge U.priority = p\},$$

   and $Fairness(t)$ is computed by the sum of distances of all the users in one priority level from the average amount of that priority level.

   $$Fairness(t) = \sum_{p \in priorities} \sum_{U \in Users_p} |\Delta(U,t) - avg(p,t)|.$$

   Comparing two algorithms, the algorithm which has lower $Fairness(t)$ achieves better performance.

3. $Locality(t)$ is defined as the number of tasks which are running on the same resource as where their stored data are located. Since in the Hadoop system the input data size is large, and the *map tasks* of one job are required to send their results to the *reduce tasks* of that job, the communication cost can be quite significant. A *map task* is defined to be local on a resource R, if it is running on resource R, and its required slice is also stored on resource R. Comparing two scheduling algorithms, the algorithm which has larger $Locality(t)$ has better performance.

4. $MeanCompletionTime(t)$ is the average completion time of all the completed jobs in the system.

6 Proposed Scheduler Model

In this section we first discuss the characteristics of our proposed algorithm, comparing them with current Hadoop scheduling algorithms. Then, we present our proposed scheduling algorithm for the Hadoop system.

6.1 Motivating Our Algorithm

In this part we discuss the important characteristics of our proposed algorithm, based on the challenges of the Hadoop system.

1. **Scheduling based on fairness, minimum share requirements, and the heterogeneity of jobs and resources.**

In a Hadoop system satisfying the minimum shares of the users is the first critical issue. The next important issue is fairness. We design a scheduling algorithm which has two stages. In the first stage, the algorithm considers the satisfaction of the minimum share requirements of all the users. Then, in the second stage, the algorithm considers fairness among all the users in the system. Most current Hadoop scheduling algorithms consider fairness and minimum share objectives without considering the heterogeneity of the jobs and the resources. One of the advantages of our proposed algorithm is that while our proposed algorithm satisfies the fairness and the minimum share requirements, it further matches jobs with resources based on job features (e.g. estimated execution time) and resource features (e.g. execution rate). Consequently, the algorithm reduces the completion time of jobs in the system.

2. **Reducing communication costs.** In a Hadoop cluster, the network links among the resources have varying bandwidth capabilities. Moreover, in a large cluster, the resources are often located far from each other. The Hadoop system distributes tasks among the resources to reduce a job's completion time. However, Hadoop does not consider the communication costs among the resources. In a large cluster with heterogenous resources, maximizing a task's distribution may result in significant communication costs. Therefore, the corresponding job's completion time will be increased. In our proposed algorithm, we consider the heterogeneity and distribution of resources in the task assignment.

3. **Reducing the search overhead for matching jobs and resources.** To find the best matching of jobs and resources, an exhaustive search is required. In our algorithm, we use clustering techniques to restrict the search space. Jobs are classified based on their requirements. Every time a resource is available, it searches through the classes of jobs (rather than the individual jobs) to find the best matching (using optimization techniques). The solution of the optimization problem results in the set of suggested classes for each resource. The suggested set for each resource is used for making the routing decision. We limit the number of times that this optimization is performed, in order to avoid adding significant overhead.

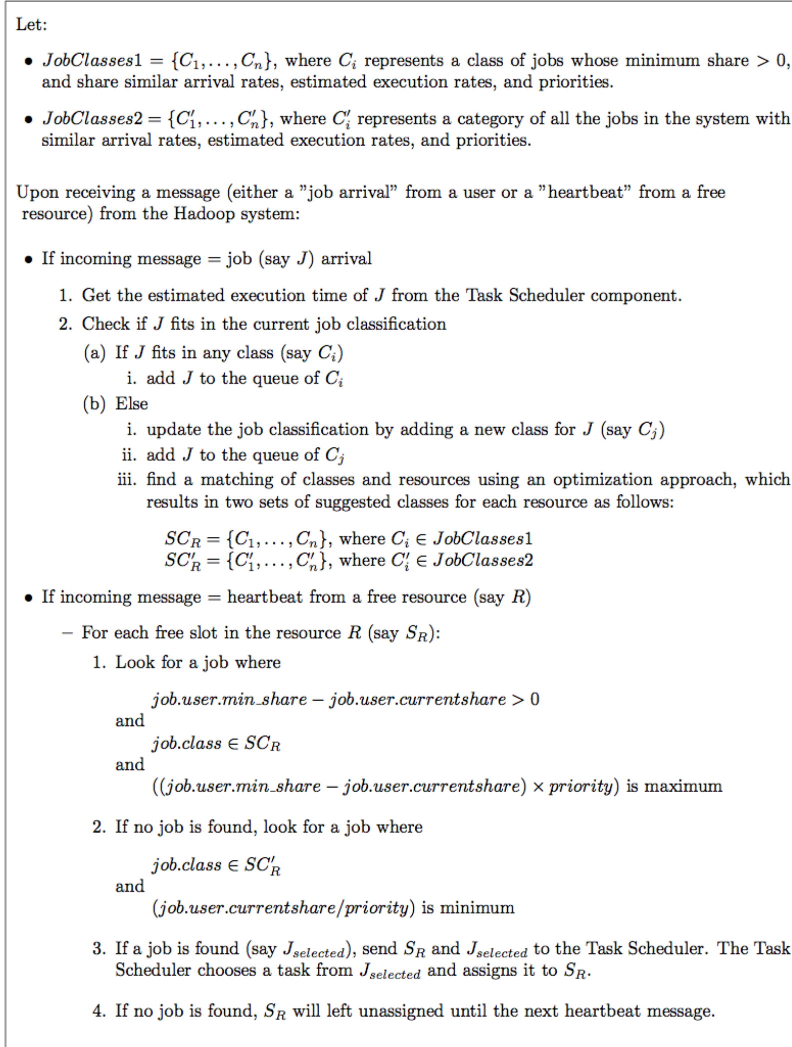
4. **Increasing locality.** In order to increase locality in a Hadoop system, we should increase the probability that tasks are assigned to resources which also store their input data. Our algorithm makes a scheduling decision based on the suggested set of job classes for each resource. Therefore, we can replicate the required data of the suggested classes of a resource, on that resource. Consequently, locality will be increased in the system.

6.2 Proposed Algorithm

In this section, we first present a high level view of our proposed algorithm (in Figure 1), and then we discuss different parts of our algorithm in more detail.

A typical Hadoop scheduler receives two main messages: a new job arrival message from a user, and a heartbeat message from a free resource. Therefore, our proposed scheduler consists of two main processes, where each process is triggered by receiving one of these messages. Upon receiving a new job from a user, the scheduler performs a queueing process to store the incoming job in an appropriate queue.

When receiving a heartbeat message from a resource, the scheduler triggers the routing process to assign a job to the free resource. Our algorithm uses the job's classification, so when a new job arrives to the system, the queueing process specifies the class of this job, and stores the job in the queue of its class. The queueing process sends the updated information of all of the classes to the routing process, where the routing process uses this information to choose a job for the current free resource. In what follows, we provide greater detail for our proposed algorithm.



Let:

- $JobClasses1 = \{C_1, \ldots, C_n\}$, where C_i represents a class of jobs whose minimum share > 0, and share similar arrival rates, estimated execution rates, and priorities.
- $JobClasses2 = \{C'_1, \ldots, C'_n\}$, where C'_i represents a category of all the jobs in the system with similar arrival rates, estimated execution rates, and priorities.

Upon receiving a message (either a "job arrival" from a user or a "heartbeat" from a free resource) from the Hadoop system:

- If incoming message = job (say J) arrival
  1. Get the estimated execution time of J from the Task Scheduler component.
  2. Check if J fits in the current job classification
     (a) If J fits in any class (say C_i)
        i. add J to the queue of C_i
     (b) Else
        i. update the job classification by adding a new class for J (say C_j)
        ii. add J to the queue of C_j
        iii. find a matching of classes and resources using an optimization approach, which results in two sets of suggested classes for each resource as follows:

$$SC_R = \{C_1, \ldots, C_n\}, \text{ where } C_i \in JobClasses1$$
$$SC'_R = \{C'_1, \ldots, C'_n\}, \text{ where } C'_i \in JobClasses2$$

- If incoming message = heartbeat from a free resource (say R)
  – For each free slot in the resource R (say S_R):
  1. Look for a job where

     $job.user.min_share - job.user.currentshare > 0$

     and

     $job.class \in SC_R$

     and

     $((job.user.min_share - job.user.currentshare) \times priority)$ is maximum
  2. If no job is found, look for a job where

     $job.class \in SC'_R$

     and

     $(job.user.currentshare/priority)$ is minimum
  3. If a job is found (say $J_{selected}$), send S_R and $J_{selected}$ to the Task Scheduler. The Task Scheduler chooses a task from $J_{selected}$ and assigns it to S_R.
  4. If no job is found, S_R will left unassigned until the next heartbeat message.

Figure 1: High level view of our proposed algorithm

1. **Job Execution Time Estimation:** When a new job arrives to the system, it is required to estimate its mean execution time on the resources. The Task Scheduler component uses the Program Analyzer to estimate the mean execution time of the new incoming job on all resources $(mean_execTime(J_i, R_j))$. The Task Scheduler component has been introduced in the AMP lab in UC Berkeley [2].

2. **Two Classifications:** The Hadoop system requires that upon a user's request at any time, it will provide her minimum share immediately. Therefore, it is critical for the system to first consider satisfying the minimum shares of all users. After satisfying the minimum shares, the system should consider dividing the resources among the users in a fair way (to prevent starvation of any user). Based on these two facts, our algorithm has two classifications: minimum share classification, and fairness classification. In the minimum share classification, only the jobs whose users have $min_share > 0$ are considered, while in the fairness classification all the jobs in the system are considered.

When a user asks for more than her minimum share, the scheduler assigns her minimum share immediately but the extra share will be assigned fairly after considering all users. As a result, users with $min_share > 0$ first should be considered in the minimum share classification, and once they receive their minimum shares, they should be considered in the fairness classification. However, the current share of a user, and consequently her minimum share satisfaction can be highly varying over time, and it is not feasible to generate a new classification each time the minimum share satisfaction changes. Therefore, we consider a job whose user has $min_share > 0$ in both classifications, and make the scheduling decision for the job based on its user's status at the time of scheduling.

In our algorithm, both the minimum share, and the fairness classifications classify the jobs such that the jobs in the same class have the same features (i.e, priority, execution rate on the resources ($mean_execRate(J_i, R_j)$), and arrival rate). We define the set of classes generated in the minimum share classification as $JobClasses1$, where each class is denoted by C_i. Each class C_i has a specific priority, which is equal to the priority of the jobs in this class. The estimated arrival rate of the jobs in class C_i is denoted by α_i, and the estimated execution rate of jobs in class C_i on resource R_j is denoted by $\mu_{i,j}$. Hence, the heterogeneity of resources is completely addressed with $\mu_{i,j}$. We assume that the total number of classes generated with this classification is F.

$$JobClasses1 = \{C_1, \dots, C_F\}$$

The fairness classification is the same as the minimum share classification; however, the difference is that this classification is done on all the jobs regardless of their users' min_share amount. We define the set of classes generated by this classification as $JobClasses2$. Each class, denoted by C'_i, has a specific priority, which is equal to the priority of the jobs in this class. The arrival rate of the jobs in class C'_i is equal to α'_i, and the execution rate of the jobs in class C'_i on resource R_j is represented with $\mu'_{i,j}$. We assume that the total number of classes generated with this classification is F'.

$$JobClasses2 = \{C'_1, \dots, C'_{F'}\}$$

For example, Yahoo uses the Hadoop system in production for a variety of products (job types) [4]: Data Analytics, Content Optimization, Yahoo! Mail Anti-Spam, Ad Products, and several other applications. Typically, the Hadoop system defines a user for each job type, and the system assigns a minimum share and a priority to each user. For example, assume a Hadoop system (called Exp1) with the parameters in Table 1. The jobs in the Exp1 system, at a given time t, are presented in Table 2, where the submitted jobs of a user are based on the user's job type (e.g., J_4 which is submitted by user1 is an advertisement product, while the job J_5 is a search ranking job). The minimum share classification of the jobs in the Exp1 system, at time t, is presented in Figure 2. Note that here we assume that there is just one resource in the system. In a system which has more than one resource, the mean execution time for each class is represented with an array, to show the execution time of the class on each resource. The fairness classification of system Exp1,

User	Job Type	min_share	priority
User1	Advertisement Products	50	3
User2	Data Analytics	20	2
User3	Advertisement Targeting	40	3
User4	Search Ranking	30	2
User5	Yahoo! Mail Anti-Spam	0	1
User6	User Interest Prediction	0	2

Table 1: The Hadoop System Example (Exp1)

User	Job Queue
User1	$\{J_4, J_{10}, J_{13}, J_{17}\}$
User2	$\{J_1, J_5, J_9, J_{12}, J_{18}\}$
User3	$\{J_2, J_8, J_{20}\}$
User4	$\{J_6, J_{14}, J_{16}, J_{21}\}$
User5	$\{J_7, J_{15}\}$
User6	$\{J_3, J_{11}, J_{19}\}$

Table 2: The job queues in Exp1 at time t

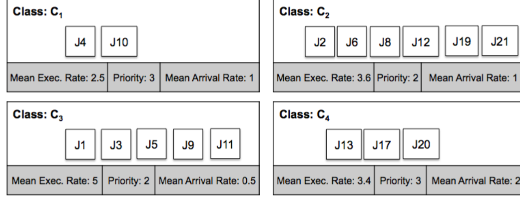


Figure 2: The minimum share classification of the jobs in Exp1 system at time t

at time t, is presented in Figure 3. Similar to the minimum share classification, we assume that there is just one resource in the system.

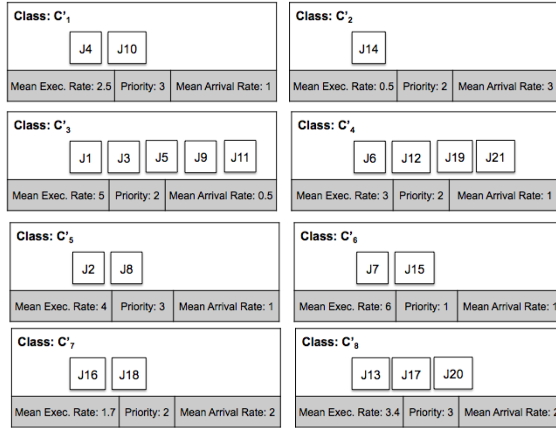


Figure 3: The fairness classification of the jobs in Exp1 system at time t

3. **Optimization approach:** In order to find an appropriate matching of jobs and resources, we define an optimization problem based on the properties of the job classes and the features of the resources. The scheduler solves the following linear program (LP) for the classes in the set $JobClasses1$. Here $\delta_{i,j}$ is defined as the proportion of resource R_j which is allocated to class C_i, and λ is the amount that we simultaneously increase arrival rates of all classes. We aim to maximize λ, while keeping the system stable.

$$\max \lambda$$

$$s.t. \sum_{j=1}^{M} C_i.\mu_{i,j} \times \delta_{i,j} \geq \lambda \times C_i.\alpha_i, \; for \; all \; i = 1, \ldots, F, \quad (1)$$

$$\sum_{i=1}^{F} \delta_{i,j} \leq 1, \; for \; all \; j = 1, \ldots, M, \quad (2)$$

$$\delta_{i,j} \geq 0, \; for \; all \; i = 1, \ldots, F, \; and \; j = 1, \ldots, M. \quad (3)$$

In the above LP, M is the total number of resources in the system, and F is the total number of minimum share classes ($|JobClasses1|$). This optimization problem tries to minimize the maximum load over all resources. After solving this LP, we have the allocation matrix $\delta_{i,j}$ for each class C_i and each resource R_j. Based on the results of this LP, we define the set SC_j for each resource R_j as follows:

$$SC_j = \{C_i : \delta_{i,j} \neq 0\}$$

Note that this is the only place we use $\delta_{i,j}$, where its value is just used to find the non zero amounts for defining the set SC_j. For example consider a system with two classes of jobs, and two resources ($M = 2$, $F = 2$), in which the arrival and execution rates are as follows:

$$\alpha = \begin{bmatrix} 2.45 & 2.45 \end{bmatrix} and \quad \mu = \begin{bmatrix} 9 & 5 \\ 2 & 1 \end{bmatrix}$$

Solving the above LP gives $\lambda = 1.0204$ and

$$\delta = \begin{bmatrix} 0 & 0.5 \\ 1 & 0.5 \end{bmatrix}.$$

Therefore, the sets for resources R_1 and R_2 will be $SC_1 = \{C_2\}$ and $SC_2 = \{C_1, C_2\}$, respectively. These two sets define the suggested classes for each resource, i.e. they suggest that upon receiving a heartbeat from resource R_1, select a job from class C_2. However, upon receiving a heartbeat from the resource R_2, either choose a job from class C_1 or C_2. Even though resource R_1 has the fastest rate for class C_1, the algorithm does not assign any jobs of class C_1 to it. This occurs because, the system is highly loaded, and since $\frac{\mu_{1,1}}{\mu_{2,1}} > \frac{\mu_{1,2}}{\mu_{2,2}}$ and $\alpha_1 = \alpha_2$, the mean completion time of the jobs is decreased if resource R_1 only executes class C_2 jobs.

A similar optimization problem is used for the classes defined in the fairness classification. The scheduler solves the following LP for classes in the set $JobClasses2$. Here $\delta'_{i,j}$ is defined as the proportion of resource R_j which is allocated to class C'_i, and λ' is the amount that we simultaneously increase arrival rates of all classes. We aim to maximize λ', while keeping the system stable.

$$\max \lambda'$$

$$s.t. \sum_{j=1}^{M} C'_i.\mu'_{i,j} \times \delta'_{i,j} \geq \lambda' \times C'_i.\alpha'_i, \; for\ all\ i = 1, \ldots, F', \quad (4)$$

$$\sum_{i=1}^{F'} \delta'_{i,j} \leq 1, \; for\ all\ j = 1, \ldots, M, \quad (5)$$

$$\delta'_{i,j} \geq 0, \; for\ all\ i = 1, \ldots, F', \; and\ j = 1, \ldots, M. \quad (6)$$

As with the LP for the minimum share classification, this linear programming problem aims to find the best classes for resource allocation based on the requirements of the jobs, the arrival rates and features of the resources. We define the set SC'_j for each resource R_j as the set of classes which are allocated to this resource based on the result of this LP. Note that this is the only place we use $\delta'_{i,j}$, and we are not using the actual values.

$$SC'_j = \{C'_i : \delta'_{i,j} \neq 0\}$$

4. **Job selection:** When the scheduler receives a heartbeat from a resource, say R_j, it triggers the routing process. The first stage in the routing process is the Job Selector component. This component selects a job for each free slot in the resource R_j, and sends the selected job for each slot to the Task Scheduler component. The Task Scheduler, introduced in [2], chooses a task of the selected job to assign to the free slot.

7 Evaluation

In this section we first describe our implemented evaluation environment, and later we provide our experimental results.

7.1 Experimental Environment

To evaluate our proposed algorithm, we use a Hadoop simulator, MRSIM [6]. MRSIM is a MapReduce simulator based on discrete event simulation, which accurately models the Hadoop environment. The simulator on the one hand allows us to measure scalability of the MapReduce based applications easily and quickly, while capturing the effects of different configurations of Hadoop setup on performance.

We extend this simulator to measure the four Hadoop performance metrics introduced in Section 5. We also add a job submission component to the simulator. Using this component we can define various users with different minimum shares, and priorities. Each user can submit various types of jobs to the system with different arrival rates. Moreover, we add a scheduler component to the simulator, which receives the incoming jobs and stores them in corresponding queues chosen by the system scheduling algorithm. Also, upon receiving a heartbeat message, it sends a task to the free slot of the resource.

Our experimental environment consists of a cluster of 6 heterogeneous resources. The resources' features are presented in Table 3. The bandwidth between the resources is 100Mbps. We define our workload using a Loadgen example job in Hadoop that is used in Hadoop's included Gridmix benchmark. Loadgen is a configurable job, in which choosing various percentages for keepMap and keepReduce, we can make the job equivalent to various workloads used in Gridmix, such as sort and filter.

Resources	Slot		Mem	
	slot#	*execRate*	*Capacity*	*RetriveRate*
R_1	1	$500MHz$	$4GB$	$40Mbps$
R_2	1	$500MHz$	$4TB$	$100Gbps$
R_3	1	$500MHz$	$4TB$	$100Gbps$
R_4	8	$500MHz$	$4GB$	$40Mbps$
R_5	8	$500MHz$	$4GB$	$40Mbps$
R_6	8	$4.2GHz$	$4TB$	$100Gbps$

Table 3: Experiment resources

We generate four types of jobs in the system: small jobs, with small I/O and CPU requirements (they have 1 Map and 1 Reduce task), I/O-heavy jobs, with large I/O and small CPU requirements (they have 10 Map

and 1 Reduce tasks), CPU-heavy jobs, with small I/O and large CPU requirements (they have 1 Map and 10 Reduce tasks), and large jobs, which have large I/O and large CPU requirements (they have 10 Map and 10 Reduce tasks). Using these jobs, we define three workloads: an I/O-Intensive workload, in which all jobs are I/O-bound; a CPU-Intensive workload; and a mixed workload, which includes all job types. The workloads are given in Table 4. Considering various arrival rates for the jobs in

Workloads	Workload Type	Jobs Included
W_1	$I/O\text{-}Intensive_i$	$small, I/O\text{-}heavy$
W_2	$CPU\text{-}Intensive_i$	$small, CPU\text{-}heavy$
W_3	$Mixed_i$	*All jobs*

Table 4: Experimental workloads

each workload, we define three benchmarks for each workload in Table 5. Here $BM_{i,j}$ shows the benchmark j of workload i; for instance, $BM_{1,1}$ is a benchmark which includes I/O-Intensive jobs, where the arrival rate of smaller jobs is higher than the arrival rate of larger ones. In total, we define nine benchmarks to run in our simulated Hadoop environment. We

Benchmarks	Arrival rate Ordering
$BM_{i,1}$	Smaller jobs have higher arrival rates
$BM_{i,2}$	Arrival rates are equal for all jobs
$BM_{i,3}$	Larger jobs have higher arrival rates

Table 5: Experiment benchmarks

submit 100 jobs to the system, which is sufficient to contain a variety of the behaviours in a Hadoop system, and is the same number of jobs used in evaluating most Hadoop scheduling systems [9]. The Hadoop block size is set to 64MB, which is the default size in Hadoop 0.21. We generate job input data size similar to the workload used in [9] (which is driven from a real Hadoop workload), where the input data of a job is defined by the number of map tasks of the job and creating a data set with correct sizes (there is one map task per 64MB input block).

7.2 Results

This section provides the results of our experiments. In each experiment we compare our

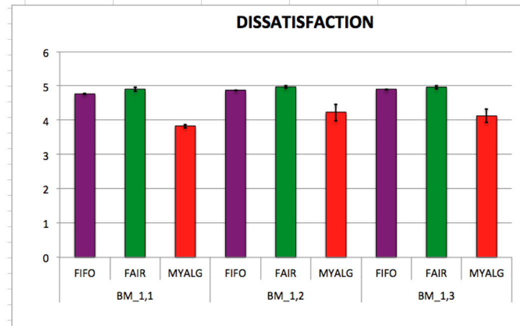


Figure 4: Dissatisfaction performance metric of the algorithms in I/O-Intensive workload

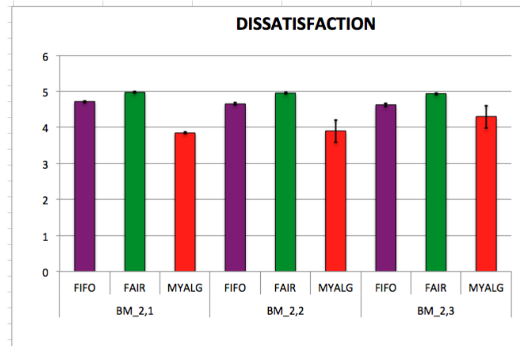


Figure 5: Dissatisfaction performance metric of the algorithms in CPU-Intensive workload

proposed algorithm with the FIFO algorithm and the version of the Fair-sharing algorithm which is presented in [9]. The comparison is based on the four performance metrics introduced in Section 5. For each experiment, we run 30 replications in order to construct 95 percent confidence intervals.

Figures 4, 5, and 6 present the Dissatisfaction metric of the algorithms running the benchmarks of the I/O-Intensive, CPU-Intensive, and Mixed workloads, respectively. The lower and upper bounds of the confidence intervals are represented with lines on each bar.

Based on these results, our proposed algorithm can lead to considerable improvement in the Dissatisfaction performance metric. There are a couple of reasons for this improvement. First, our proposed algorithm considers the minimum share satisfaction of the users as its initial goal. When receiving a heartbeat from a resource, it first satisfies the minimum shares

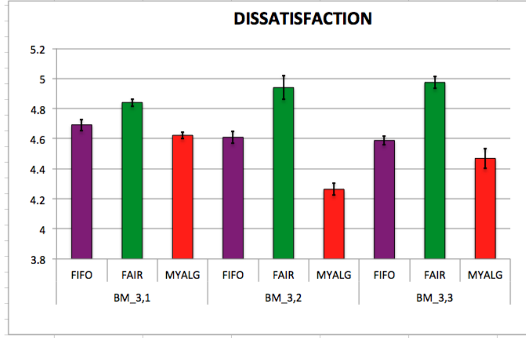


Figure 6: Dissatisfaction performance metric of the algorithms in Mixed workload

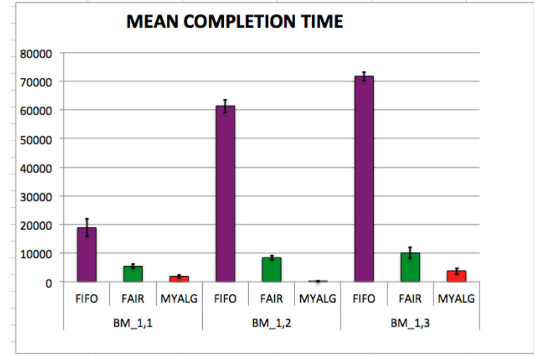


Figure 7: Mean Comp. Time performance metric of the algorithms in I/O-Intensive workload

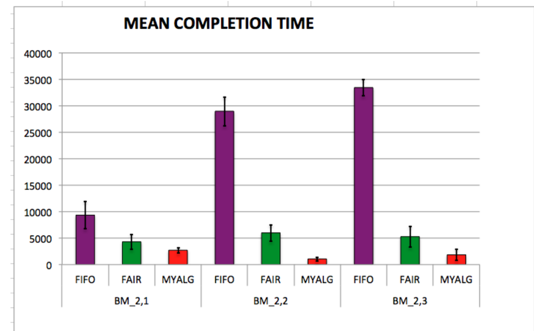


Figure 8: Mean Comp. Time performance metric of the algorithms in CPU-Intensive workload

of the users. Second, our algorithm considers the priority of the users in satisfying their minimum shares. Therefore, the highest priority user who has not yet received her minimum share will be considered first. However, since the algorithm considers the product of the remaining minimum share and the priority of the user, it does not let a high priority user with high minimum share starve lower priority users with smaller minimum shares. This is an important issue, which is not considered in the Fair-sharing algorithm. As for our algorithm, the Fair-sharing algorithm has the initial goal of satisfying the minimum shares of the users. However, since the Fair-sharing algorithm does not change the ordering of the users who have not received their minimum share, it causes higher Dissatisfaction in the system. The Fair-sharing algorithm defines pools of jobs, where each pool is dedicated to a user. Since the order of the pools (which present users) is fixed, the algorithm always checks the users' minimum share satisfaction in that order. Therefore, if there is a user at the head of this ordering who has large minimum share requirement and low priority, she may create a long delay for the other users with higher priority. Moreover, the Fair-sharing algorithm does not consider the users' priorities in the order of satisfying their minimum shares.

Figures 7, 8, and 9 present the Mean Completion Time metric of the algorithms running the benchmarks of the I/O-Intensive, CPU-Intensive, and Mixed workloads, respectively. The results show that compared to the other algorithms, our proposed algorithm achieves the best mean completion time in all the benchmarks. Compared to the FIFO algorithm, our algorithm leads to significant improvement in reducing the mean completion time of the jobs. This significant improvement can be explained by the fact that unlike the other two algorithms, our proposed algorithm considers the heterogeneity in making a proper scheduling decision based on the job requirements and the resource features.

Table 6 presents the Fairness metric of the algorithms in the various defined benchmarks. In each benchmark, the table shows the 95%-confidence interval for Fairness when the corresponding scheduling algorithm is used. Comparing the algorithms, the Fair-sharing algorithm has the best Fairness. This is as expected, because the main goal of this algorithm is improving the Fairness metric. Our proposed

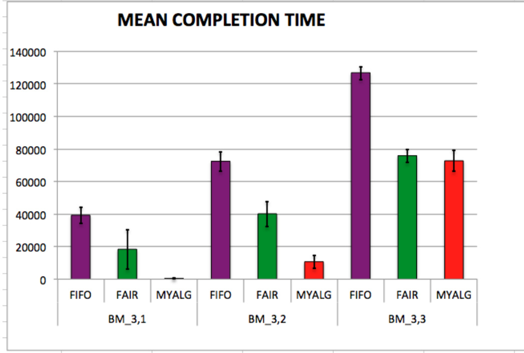


Figure 9: Mean Comp. Time performance metric of the algorithms in Mixed workload

Benchmarks	FIFO	FAIR	MyALG
$BM_{1,1}$	(14.88, 15.05)	(11.59, 11.65)	(14.68, 16.08)
$BM_{1,2}$	(14.93, 15.00)	(11.57, 11.72)	(12.68, 14.60)
$BM_{1,3}$	(14.63, 15.26)	(11.59, 11.76)	(17.23, 17.65)
$BM_{2,1}$	(14.77, 15.22)	(11.63, 11.98)	(11.99, 12.34)
$BM_{2,2}$	(14.83, 15.09)	(11.81, 12.12)	(13.99, 14.36)
$BM_{2,3}$	(14.42, 15.73)	(11.81, 11.94)	(17.37, 17.72)
$BM_{3,1}$	(14.94, 15.37)	(11.47, 12.71)	(14.11, 15.05)
$BM_{3,2}$	(14.73, 15.62)	(11.72, 12.46)	(14.41, 14.98)
$BM_{3,3}$	(15.00, 15.44)	(11.89, 12.07)	(12.11, 13.31)

Table 6: Fairness performance metric of the algorithms for all workloads

algorithm considers the heterogeneity and assigns the jobs based on the features of the resources. Therefore, it does not blindly assign each job to each free resource. Moreover, our algorithm first satisfies the minimum share of the users. Then, after receiving the minimum share, the corresponding user will be considered along with all other users (second level of classification of our algorithm) to achieve fairness in dividing the shares of the resources among the users in the system. In some benchmarks, our algorithm leads to an increase in the Fairness metric. However, because of the importance of the users with non zero minimum shares, this side effect may be considered acceptable. Generally, the minimum share of the users are assigned based on business rules, which has higher priority for most companies. As a result, a small increase in Fairness may be considered acceptable for most Hadoop systems, if it results in better satisfaction of the users' minimum shares, and significant reduction in the mean completion time of the jobs.

Benchmarks	FIFO	FAIR	MyALG
$BM_{1,1}$	(96.60, 98.03)	(98.12, 99.08)	(98.62, 99.98)
$BM_{1,2}$	(47.39, 57.81)	(89.84, 91.76)	(93.82, 95.38)
$BM_{1,3}$	(62.93, 65.07)	(71.43, 74.57)	(66.44, 71.55)
$BM_{2,1}$	(90.38, 94.42)	(97.12, 98.08)	(98.56, 99.87)
$BM_{2,2}$	(68.65, 82.15)	(93.93, 96.87)	(91.78, 95.42)
$BM_{2,3}$	(78.73, 84.07)	(94.14, 97.86)	(93.78, 97.42)
$BM_{3,1}$	(73.48, 86.92)	(78.77, 83.63)	(99.12, 100.00)
$BM_{3,2}$	(92.36, 95.24)	(81.27, 87.13)	(95.11, 99.69)
$BM_{3,3}$	(79.23, 88.37)	(78.02, 86.37)	(66.86, 76.73)

Table 7: Locality performance metric of the algorithms for all workloads

Table 7 presents the Locality metric of the algorithms in the various defined benchmarks. For each benchmark, the table shows the 95%-confidence interval for Locality when the corresponding scheduling algorithm is used. The locality of our proposed algorithm is close to, and in some cases is better than the Fair-sharing algorithm. This can be explained by the fact that our algorithm chooses the replication places based on the suggested classes for each resource.

Another significant feature of our proposed algorithm is that although it uses sophisticated approaches to solve the scheduling problem, it does not add considerable overhead. The reason is that first, we limit the number of times required to do classification, by considering the aggregate of job features (i.e. mean execution time and arrival rate). This results in considering the group of job types in each class, rather than just one job type. Also, since some jobs in the Hadoop system are submitted multiple times by users, these jobs do not require changing the classification each time that they are submitted to the system.

8 Conclusion and Future Work

The primary Hadoop scheduling algorithms do not consider the heterogeneity of the Hadoop system in making scheduling decisions. In order to keep the algorithm simple they used minimal system information in making scheduling decisions, which in some cases could result in poor performance. Growing interest in applying the MapReduce programming model in var-

ious applications gives rise to grather heterogeneity, and thus must be considered in its impact on performance. It has been shown that it is possible to estimate system parameters in a Hadoop system. Using the system information, we designed a scheduling algorithm which classifies the jobs based on their requirements and finds an appropriate matching of resources and jobs in the system. Our algorithm is completely adaptable to any variation in the system parameters. The classification part detects changes and adapts the classes based on the new system parameters. Also, the mean job execution times are estimated when a new job is submitted to the system, which makes the scheduler adaptable to changes in job execution times. We have received permission to use the workload from a high profile company, and we are currently working on defining benchmarks based on this workload, and use them to evaluate our algorithm. Finally, we aim to implement and evaluate our algorithm in a real Hadoop cluster.

ACKNOWLEDGEMENTS

This work was supported by the Natural Sciences and Engineering Research Council of Canada. A major part of this work was done while both authors were visiting UC-Berkeley. In particular, the first author would like to thank Ion Stoica and Sameer Agarwal for sharing the Task Scheduler part, and also their comments on our proposed algorithm.

References

[1] A. Aboulnaga, Z. Wang, and Z. Y. Zhang. Packing the most onto your Cloud. In *Proceedings of the first international workshop on Cloud data management*, 2009.

[2] S. Agarwal and G. Ananthanarayanan. Think global, act local: analyzing the trade-off between queue delays and locality in MapReduce jobs. Technical report, EECS Department, University of California, Berkeley, 2010.

[3] Apache. Hadoop on demand documentation, 2007. [Online; accessed 30-November-2010].

[4] R. Bodkin. Yahoo! updates from Hadoop Summit 2010. http://www.infoq.com/news/2010/07/yahoo-hadoop-summit, July 2010.

[5] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51:107–113, January 2008.

[6] S. Hammoud, M. Li, Y. Liu, N. K. Alham, and Z. Liu. MRSim: a discrete event based MapReduce simulator. In *Proceedings of the 7th international conference on Fuzzy Systems and Knowledge Discovery (FSKD 2010)*, pages 2993–2997. IEEE, 2010.

[7] Kristi Morton, Magdalena Balazinska, and Dan Grossman. ParaTimer: a progress indicator for MapReduce DAGs. In *Proceeding of the international conference on management of data*, pages 507–518, 2010.

[8] T. Sandholm and K. Lai. Dynamic proportional share scheduling in Hadoop. In *Proceedings of the 15th workshop on job scheduling strategies for parallel processing*, pages 110–131. Springer, Heidelberg, 2010.

[9] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Job scheduling for multi-user MapReduce clusters. Technical Report UCB/EECS-2009-55, EECS Department, University of California, Berkeley, April 2009.

[10] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleey, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceeding of the European conference on computer systems (EuroSys)*, Paris, France, 2010.