Mitigating the Negative Impact of Preemption on Heterogeneous MapReduce Workloads

Lu Cheng*, Qi Zhang*, Raouf Boutaba*†
*David R. Cheriton School of Computer Science, University of Waterloo, Ontario Canada
†Division of IT Convergence Engineering, POSTECH, Pohang, KB 790-784, Korea
{l32cheng, q8zhang, rboutaba}@uwaterloo.ca

***Abstract*—Modern production clusters are often shared by multiple types of jobs with different priorities in order to improve resource utilization. Preemption is a common technique employed by MapReduce schedulers to avoid delaying production jobs while allowing the cluster to be shared by other non-production jobs. In addition, it also prevents a large job from occupying too many resources and starving others. Recent literature shows that jobs in production MapReduce clusters have a mixture of lengths and sizes spanning many orders of magnitude. In this type of environments, the current preemption policy used by MapReduce schedulers can significantly delay the completion time of long running tasks, resulting in waste of resources. This paper firstly discusses the heterogeneous nature of MapReduce jobs and their arrival rates in several production clusters. Secondly, we characterize the situations where the current preemption policy causes significant preemption penalty. We then propose a simple mechanism that works in conjunction with existing job schedulers to address this problem. Finally, we evaluate our solution under various types of workloads in Amazon EC2. Experiments show our method can improve system normalized performance by 15% during busy periods by effectively avoiding unnecessary preemption while preserving fairness.**

***Index Terms*—Cloud Computing; Mapreduce; Hadoop**

I. Introduction

Cloud computing frameworks like MapReduce [1] and Dryad [2] have become the dominant programming model for data-intensive computing in recent years. In these frameworks, a job may spawn many small tasks that can be executed concurrently on multiple machines, resulting in significant reduction in job completion time. In addition, these frameworks provide fault tolerance features. Since software and hardware exceptions are common in large-scale clusters, the scheduler can automatically restart a task after the task fails due to runtime exceptions. As a result, these models are very attractive not only for running data-intensive jobs, but also for executing computation-intensive applications.

Several studies have analyzed the workload characteristics in production MapReduce clusters [3][4][5][6][7]. One important finding of these studies is that workloads are *heterogeneous*. First, the difference in job size can span several orders of magnitude. A large job typically spawns many small tasks in parallel to speed up the execution, although doing so requires a large amount of computation slots during a short period. Second, the length of the jobs can differ significantly. Although most jobs have short running time, some jobs can take a very long time to complete. Finally, like many other service systems, the arrival of job requests is not only non-uniform but also very spiky. In order to ensure acceptable response time of jobs during peak workload time, the capacity of a cluster is often much higher than what an average workload needs. Therefore, most of the resources will typically be idle if the cluster is only reserved for a single type of jobs. In order to improve resource utilization, modern clusters are often shared by a mixture of jobs with different priorities, which further increases the heterogeneity in workloads. For example, in Google's compute clusters, on the average around 30% jobs are production jobs, and the others are for research and experimental purposes (Based on internship experience at Google in 2010). Typically production jobs (i.e. jobs that generate revenue) are given higher priorities than non-production jobs (i.e. experiments and research jobs). Although production jobs account for a small percentage of the total job population, they are allowed to consume a significant portion of the cluster resources.

For heterogeneous workloads, MapReduce job schedulers typically rely on preemption to coordinate resource sharing, to achieve fairness, and to improve system performance. Specifically, preempting a task means terminating the task immediately and using the resources to schedule a different task. As a single task has very light weight compared to the job, the scheduler typically does not need to save the task progress. It simply restarts the task later. This simple preemption policy is used by Apache Hadoop, an open source, widely deployed implementation of MapReduce, and also in Google's data centers [5]. In the environment where workload is heterogeneous, preemption can help to ensure fast response of high priority jobs by allowing them to preempt low priority ones, when there is no free capacity in the cluster. Generally speaking, jobs of the same priority usually have the same fair share, and job schedulers will always assign newly freed slots to jobs that occupy fewer slots than their share. Because slots are often freed up quickly in large clusters, this policy can achieve Max-Min fairness [8]. Lastly, schedulers can also use preemption to improve system performance, such as to kill a non-local task and move it to a node with local data, or to terminate outlier tasks and restart them again [3][9].

However, we have found in a MapReduce cluster that executes heterogeneous workloads, the preemption policy mentioned above can greatly delay the completion of long running jobs, resulting in a significant waste of resources. During our research experiments and in particular our experience with Google's production clusters, from time to time we have observed some tasks that have long running time are killed repeatedly when large production jobs arrive. Specifically, the arrival of a large production job can significantly cut down the share of each non-production job, resulting in large number of tasks being preempted. Since each production job usually has short response time, after its completion the low-priority jobs are allowed to launch more tasks again. But before a long task can finish, it is often killed again when the next large production job arrives. Consequently, jobs consisting of these long tasks are heavily delayed and the resources allocated for their execution are wasted. This may partly explain why preemption is disabled in the default deployment of Apache Hadoop fair scheduler [10]. However, simply disabling preemption eliminates all the aforementioned benefits. Hence, we believe it is necessary to carefully examine the negative impact of preemption and mitigate it. Specifically, the contributions of this paper are as follows:

1) We identify and analyze the situation where preemption can significantly prolong the job completion time under heterogeneous workloads. We also propose a simple solution that can be combined with existing job schedulers to address this problem.
2) We implement our solution on top of Hadoop fair scheduler [10] and evaluate it in Amazon Elastic Compute Cloud (EC2) [11] under data intensive and mixed workloads that are consistent with the statistics shown in recent publications. Experiments show by using our method, system normalized performance improves by 15% on average during busy periods. Because system capacity is designed based on service demands during busy periods, even a few percent of improvement in the efficiency of a cluster consisting of tens of thousands of nodes can save millions of dollars a year [9].

The rest of this paper is as follows. Section II provides a brief introduction to MapReduce and Hadoop Job Scheduler. Section III discusses heterogeneous nature of MapReduce jobs and their arrival rates in several production clusters. Section IV analyzes the situations where preemption can significantly increase job completion time of long-running jobs. We then propose our solution in Section V. In Section VI we outline our experimental design and results of evaluation. After reviewing related work in Section VII, we conclude in section VIII.

II. MAPREDUCE AND HADOOP JOB SCHEDULING

This section presents a brief introduction to MapReduce and Hadoop job scheduling mechanism. We will also describe the terminologies that will be used throughout the paper. Essentially, MapReduce and Dryad exploit a well-known design pattern: divide and conquer. A MapReduce or Dryad job consists of two types of tasks: *map* and *reduce* tasks. Original input is divided into multiple small blocks and are processed by map tasks, resulting in a set of intermediate key/value pairs. The reduce tasks then combine all intermediate values associated with the corresponding keys to produce the final output. Tasks are executed in parallel on multiple machines, and every single task can be killed and started independently.

Apache Hadoop [12] is the most popular open-source implementation of MapReduce. It consists of one Job Tracker that receive job submissions from users, and multiple Task Trackers that actually execute tasks. At run-time, task trackers report the status of tasks and computing resources to the Job Tracker in heartbeats every few seconds, and the Job Tracker uses a job scheduler to assign tasks to the Task Trackers.

Table I
CDF OF MAPREDUCE / DRYAD JOB RESPONSE TIME AT FACEBOOK, THE INTERNET COMPANY AND MICROSOFT

(a) Execution Trace of a Hadoop Cluster at Facebook

% Jobs	40%	50%	60%	70%	80%
Running Time (s)	55	90	120	150	350
% Jobs	90%	95%	98%	99%	99.5%
Running Time (s)	650	1200	3000	5000	>10000

(b) Execution Trace of a Hadoop Cluster at the Internet Company

% Jobs	40%	50%	60%	70%	80%	90%
Running Time (s)	45	80	30	190	450	650

(c) Execution Trace of Microsoft Research Cluster

% Jobs	18.9%	28.0%	34.7%	51.3%	72.0%	95.7%
Running Time (min)	5	10	15	30	60	300

III. HETEROGENEITY OF WORKLOADS IN PRODUCTION MAPREDUCE CLUSTERS

Recent publications [3-7] have reported workload characteristics in production clusters at Microsoft, Google and Facebook. Combing these traces, this section analyzes the heterogeneity in MapReduce jobs in terms of job length, size, and arrival rate.

A. Bimodal Behavior of Job Lengths

Table I shows the distribution of MapReduce and Dryad job response time. The data are collected from a 600 node dedicated Hadoop cluster at Facebook [7], a 40 node Hadoop at an anonymous Internet Company (hereon referred to as IC) [4], and a production cluster used inside Microsoft's search division [3]. Specifically, Table I(a) indicates that at Facebook the median job length is 84s and the average length of jobs is between 300-450s. Table I(b) shows that the median as well as average job response times at IC are similar to those at Facebook. In the Microsoft's research cluster, the median job is approximately 30 minutes, as shown in Table I(c). The average job lengths at both Facebook and IC are similar to Google's statistic of 395s [1], while significantly shorter than the average length of jobs in Microsoft's research cluster. From Table I, it is evident that most jobs in current cloud computing

Table II
CDF OF NUMBER OF MAP TASKS IN A HADOOP CLUSTER AT FACEBOOK

% Jobs	39%	55%	69%	78%	84%	90%
Num. of Maps	1	2	20	60	150	350
% Jobs	94%	97%	98%	99%	99.5%	Largest
Num. of Maps	500	1500	3065	3846	6232	>25000

clusters are short. However, in all three clusters there are long jobs with running time at least 2 to 3 orders of magnitude longer than running time of the short ones. Moreover, the distributions of job response time show bimodal behavior, with the transition region between 200 and 400 seconds at Facebook and IC, and between 60 and 100 minutes at Microsoft.

Similar observations have been reported more recently by Mishra et. al [5], who studied the workload characteristics in Google's cloud backend. Even though their dataset contains non-MapReduce jobs, the workload characteristics reported in [5] are still similar to what we can see in Table I. Specifically, the duration of task executions follows a bimodal distribution as tasks either have short or long running time. Even though most tasks are short, long tasks are multiple orders of magnitude longer than short ones.

B. Bimodal Behavior of Jobs Sizes

Now we turn our attention to the distribution of job size. Table II describes the job size in terms of number of map and reduces tasks at Facebook [7]. The number of map tasks reflects the input file size, because in Hadoop clusters, the number of map tasks is equal to the input file size divided by the size of a block, which is normally 64MB or 128MB. From Table II, it can be observed that most jobs have a small number of map tasks. On the other hand, a small fraction of large jobs have very large number of map tasks. In the experimental workload used in [7], which is designed based on the metrics of real workloads at Facebook, among 100 jobs the 4 largest jobs accounts for 73% of total 26410 map tasks. Figure 1 shows the number of map and reduce tasks per job at IC. Around 40% jobs contain less than 10 map or reduce tasks, but another 40% jobs have more than 100 map or reduce tasks. Also, at both Facebook and IC, the smallest job only contains one task, but a single large job can have up to tens of thousands of tasks. Similar to job length, the distribution of job sizes has a bimodal behavior [4].

Based on the above discussion, we know the difference in job length and size can range across multiple orders of magnitude. Statistically, if we add the running time of all the jobs together, a few long jobs constitute a large faction of total running time; and if we add up the number of all the tasks, a few large jobs take a large proportion of the total number of tasks. These characteristics also match what have been observed at Google [5]: most resources are consumed by a few jobs with long duration that have large demands for CPU and memory.

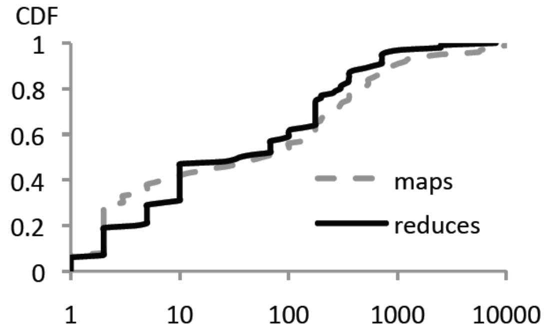

Figure 1. CDF of number of map and reduce tasks in a Hadoop cluster at Internet Company [4]

The aforementioned characteristics of MapReduce workloads share many similarities with what is already known in traditional distributed computing systems. Harchol-Balter [13] observed the workload in many real environments has a mixture of job lengths and sizes spanning many orders of magnitude. Typically there are many small jobs and a few large ones. As another example, measurements have shown that running time of Unix processes, sizes of files transferred through the Web and stored in Unix file systems are with heavy-tailed distributions [13][14][15].

C. Fluctuating Job Arrival Rates

Similar to the job size and length, the arrival rate of MapReduce jobs is also highly variable from time to time. In October 2009, the distribution of job inter-arrival times at Facebook was first reported in [7]. Chen et al. [4] also studied job inter-arrival times at Facebook and IC. For both companies, inter-arrival time exhibits an on-off pattern according to the time of the day. During day time the job arrival can be quite intense, as around 40% of the time inter-job arrival time is less than 10s. Consequently, the system is often very busy. However, at nighttime, job arrival intervals can be very long. In this case, most of the resources in the clusters become idle.

Combined with job characteristics discussed above, we can further see that slot requests in clusters are even spikier than job arrival rates. This is because the distribution of job size has bimodal behavior, and therefore a large amount of requests of computation slots will happen at the same time with the arrival of large-sized jobs.

Fluctuating job arrival rate is also commonly seen in other service systems like telecommunication networks and public transport systems. For example, in VoIP networks, the traffic during the busiest hour accounts for approximately 15 to 20 percent of the traffic for that day [16]. Similar measurements have been reported for Telecommunications networks [17][18]. In these systems, the traffic intensity during the busiest periods is 3 to 4 times higher than the average workload intensity.

IV. PROBLEM DESCRIPTION

Heterogeneity in MapReduce workloads raises new challenges for designing effective scheduling policies. As described previously, although production MapReduce jobs only account for a small percentage of total job population, they are allowed to occupy a significant portion of resources in the cluster. As resource requests of production jobs are very spiky and they typically finish much faster than non-production jobs, the amount of remaining resources not used by production jobs (which will be evenly distributed to non-production jobs) can change greatly according to their arrival and departure rates. In this case, when a production job with large resource requirements arrives, the scheduler will evenly preempt a fraction of tasks from each non-production job. However, since a production job usually has a short response time, many resources will be freed upon its completion, allowing low priority jobs to launch more tasks again. If this process repeats, long running tasks of non-production jobs can potentially be killed repeatedly. This problem is aggravated during busy hours when large production jobs arrive regularly. Therefore, the existing preemption policy used by MapReduce schedulers can significantly delay the completion of long running jobs. Consequently, the resources used by these jobs are wasted.

Let us use Facebook's dataset (described in Section III) as an example to illustrate the severity of this problem. In a 600 nodes cluster with 3100 map slots, the amount of map tasks of 3% of the jobs is larger than 50% capacity of the cluster, and the amount of map tasks of 2% of the jobs is even larger than the capacity of the whole cluster. According to Section III.C, if the job inter-arrival time during busy periods is 5s (two times the reported average speed), on average a large job will come every 170s. If we assume $1/3$ are production jobs, roughly once every $500s$ a large production job will arrive. From Table I(a) we can see that around 2% map tasks are longer than $500s$ in Facebook's dataset, and these map tasks will most likely be killed repeatedly during busy periods. Although the percentage of these tasks is small, as we have discussed, most resources are consumed by a few jobs with long duration [5]. If we can cut down wastage, the system performance will be improved.

V. Mitigating the Negative Impact of Preemption

To deal with the issue mentioned in Section IV, we propose a technique called Global Preemption (GP) that is responsible for selecting tasks to be preempted by job schedulers in order to reduce the cost of preemption. The architecture of GP is illustrated in Figure 2. The key idea behind GP is to trade short-term fairness for better efficiency. In the existing implementations of MapReduce schedulers, when a large production job arrives, the scheduler will first calculate the share of each job, and then kill overflowing tasks to release slots when there is a shortage of slots to accommodate this job. Here overflowing tasks refer to tasks of a job beyond the share of this job. Usually a certain number of most recently launched tasks of each job will be killed. With the GP policy, instead of killing newly launched tasks of each individual job, GP globally selects the most recently launched from all the running tasks rather than from an individual job to minimize the cost of preemption. When slots are freed up later, the job scheduler will ensure jobs with the biggest deficiency to get slots first. Here deficiency equals share of a job minus the number of running tasks of this job. Even though GP prefers

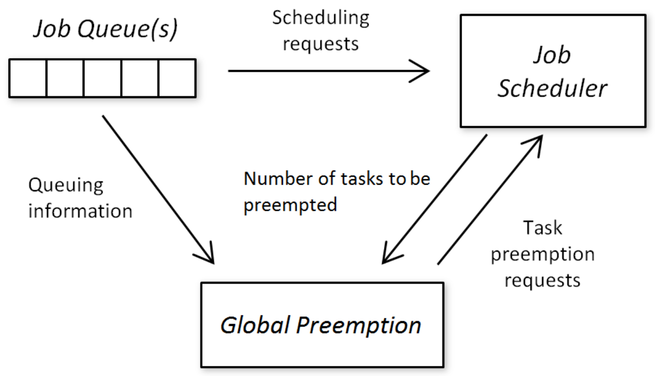

Figure 2. The architecture of the Global Preemption based Scheduler

to keep long tasks running and normally selects short tasks to be killed first, in our experiments we have observed that the negative impact on fairness is very small, and in fact most of the time short jobs will not be starved but instead finish faster. By cutting down wastage due to running jobs that will be repeatedly preempted, the resources available for non-production jobs are increased and as a result jobs are speeded up on average.

However, although GP does not hurt fairness for typical MapReduce jobs, there are cases where long jobs can starve short ones. There are two possible causes: (1) Production jobs monopolize a large portion of slots and do not release them for a long time; or (2) Long jobs are also large and take up many slots. In both cases, we preempt long jobs when they start to starve short jobs by monitoring the length of waiting queue and busy status of clusters.

We define the following conditions for preempting overflowing tasks of a long running job: (1) the job must occupy more slots than its fair share, and (2) the cluster is busy, for example, $> t\%$ time of a period (e.g. 3 minutes) more than $p\%$ slots are occupied; and (3) there are a lot of tasks to be scheduled during this period, namely, the total number of slots requested by the jobs in the queue is larger than n_{task}.

Regarding parameter t and p, we selected some intuitive values such as 80% and 90% in our experiments, as it generally is not difficult to judge the load degree of a cluster. More sophisticated methods can be used to compute more precise values in the future. The n_{task} changes with the actual size of available resources. We present the following method to help decide its value. Let T be a job's response time which is the sum of waiting time and actual service time, i.e:

$$T = T_{queueing} + T_{service}$$

Let $W = \mathbf{E}(T)$, and in an ideal condition where waiting time is zero, we have

$$W_{ideal} = \mathbf{E}(T_{service})$$

The ANP (Application Normalized Performance) [3] of a job j is the ratio of j's execution time under ideal conditions to j's execution time in the situation of interest. ANP is an important metric because other metrics are dependent on

Algorithm 1 GP: Global Preemption

```
Input  $k$: Number of tasks to be preempted
Output  taskToPreempt: The set of tasks to be preempted
 1: if now - lastPreemptCheckTime > preemptInterval then
 2:     lastPreemptCheckTime = now
 3:     if long jobs starve other jobs then
 4:         add tasks of long jobs to be killed into taskToPreempt
 5:         if |taskToPreempt| >= $k$ then
 6:             return  taskToPreempt
 7:     $k$ = $k$-|taskToPreempt|
 8:     reverse order running tasks by start time
 9:     add most recent $k$ tasks into taskToPreempt
10:     return  taskToPreempt
```

ANP (Please refer to Table IV). If we want to maintain the quality of service above a certain level, say, ensuring the average $ANP > \pi (0 \leq \pi \leq 1)$ during busy hours, i.e.:

$$\mathbf{E}(ANP) = \frac{W_{ideal}}{W} > \pi$$

According to Little's formula [19], for any $G/G/c$ queues, we have $L = \lambda W$, and

$$L_{queuing} = \lambda W_{queuing} = \lambda (W - W_{ideal}) \tag{1}$$

Here L is the mean number of jobs in the system, $L_{queuing}$ is the mean number of jobs in the queue and λ is the job arrival rate. By (1), we have:

$$\mathbf{E}(ANP) = \frac{L - L_{queuing}}{L} \tag{2}$$

From (2) we can see that in order to get an average ANP larger than π, the mean length of the queue must be less than $L(1-\pi)$. For example, if we require during idle times $\pi = 0.5$, the mean length of the queue must be less than the size of available resources, in this case, the number of slots available for non-production jobs.

When starvation occurs, a simple solution is to kill all the overflowing tasks. However, this naïve approach can cause excessive preemptions. A better method is to perform preemption according to the busy degree of the cluster. In our experiments, we use a simple heuristic timeout parameter $T_{timeout}$ that corresponds to the load degree of the cluster. If during this period we continually observe tasks being starved, we can then kill all the overflowing tasks. Before that, only a proportion (e.g. 50%) of them is killed. Finally, one may ask how to determine which tasks are long. Previous work has investigated this problem and proposed a good solution. See [9] for example.

Finally, Algorithm 1 summarizes the GP policy. The preempt interval is usually short, e.g. 10s, to ensure production jobs acquire slots quickly. GP is designed to work with any job scheduler that supports preemption.

VI. Evaluation

We have implemented GP on top of Hadoop Fair Scheduler (HFS) [10], specifically on the version with delay scheduling [7]. We evaluated GP plus HFS (with preemption disabled), and compared it with HFS (with preemption enabled) in Amazon EC2. This section describes our experiment setup as well as the evaluation results.

A. Experiment Cluster

We use 101 standard large Linux instances, in which one is configured as Hadoop master, and the other 100 are configured as Hadoop slaves. Each instance has 7.5GB of RAM and 4 EC2 Compute Units, running 64-bit Fedora. We configure each Hadoop slave to have 4 map slots and 2 reduce slots, which means the test cluster has 400 map slots and 200 reduce slots in total. We first create an Amazon EBS-backed AMI (Amazon Machine Image) that contains all the information necessary to boot instances of our software and configuration. We then launch 101 instances of this AMI. Each instance has a 15GB EBS (Elastic Block Store) volume as its boot device, and 2 disks with total capacity of 850 GB as local instance storage. The input data files are striped across both local storages. The instances are able to send 1 Gbps data to each other. To support deployment of Hadoop, we then ran a Python script to enable the master node to have all the necessary information about the slave nodes, and any node can SSH to any other node without a passphrase.

B. Experiment Applications and Workloads

We adapted the experiment workloads and settings from that of [7], which is designed based on the metrics of real workloads at Facebook. However, we also modified their setup in several ways to better emulate the heterogeneity of workloads, and to make it compatible with Hadoop 0.21. Specifically, the applications used in our experiments were:

1) `Grep`. This program scans through a huge number of records searching for a relatively rare pattern. It redistributes its input data between nodes and is inherently I/O intensive.
2) `Aggregate`. This program calculates the total advertisement revenue generated for each IP address in the user visits records. Since there are many datasets containing a particular IP address, the reduce tasks are communication intensive.
3) `Join`. This program consists of two sub-applications that perform a complex calculation on two data sets. It is used to identify the user who generated the most advertisement revenue and the average PageRank of their pages. This program is rather mixed.

To create CPU-heavy jobs, like [7] we modified the `Grep` job to run a compute-intensive function on input data and still output only a few records. Moreover, because we can only use a limited number of applications in the experiments, in order to make the task lengths span across the entire distribution, we add compute-intensive or I/O intensive user defined function (UDF) into these applications to simulate tasks of different lengths and thus make the generated workloads closer to the statistics discussed in Section III.

Regarding the workload, we extend the 100 jobs used in [7] to 130 so that we can differentiate jobs in the long-tailed area of distribution. The experimental workload is shown in Table III. Each row represents jobs that have the same size of input in terms of number of map tasks. For example, in the first row there are 49 jobs and each contains one map task whose service time ranges from 100s to 600s.

We use each one of the three experimental applications with a variety of inputs. In our experiments, an application instance refers to a particular application with a particular data set as its input. We run two types of workloads and each has a list of application instances. The I/O intensive workload contains 130 application instances, and the mix workload contains 160 application instances because a `Join` application instance will spawn three consecutive jobs. Among each of the workloads, we select a medium size job that has 250 maps as a heavy-tailed long job. The submission schedule is generated randomly, and experiments that compare schedulers running jobs in the same order. Specifically, the two types of workloads are:

1) *I/O intensive*: This type of workloads are presented by `Sort` and `Grep` jobs. These jobs are typically data-intensive, and a single map of such a job needs less than 6s for processing a 128MB block in our experiment. Similar to [7], we use different Grep instances to represent our I/O intensive workload.
2) *Mixed*: The mixed workload consists of: 25 jobs containing 15554 maps which are I/O bound `Grep` instances; 56 jobs containing 1589 maps which are CPU bound Grep instances; 34 jobs containing 10608 maps which are `Aggregate` instances; and 15 jobs which are `Join` instances. The heavy-tailed long job is an application instance of the CPU bound Grep program.

C. Evaluation Metrics

We use the metrics described in Table IV to evaluate the performance of our proposed GP policy. These metrics are originally described in [3]. However, different from [3], we use the average response time instead of makespan (i.e. the total time taken by an experiment until the last job completes) as the evaluation metric, because the average response time is more appropriate for capturing the system throughput in our experiments.

D. Experiment Settings

Since the computing the metrics in Table IV requires determining the ideal running time of each job, we start by estimating it by giving each job exclusive access to the cluster.

Once the ideal running time of each job is determined, the next step is to evaluate the performance of current preemption policy (HFS) and our proposed GP policy (HFS+GP, or GP for short) during busy and idle hours under I/O intensive and mixed workloads. We conduct six groups of experiments during busy periods and four during idle times. In these experiments, we select 30% jobs as production jobs whose share is 80% of total capacity and the others as non-production jobs whose minimum share is 20% capacity of the test cluster. We assume jobs of the same type have the same priority. Jobs of the same priority have the same resource share.

The job inter-arrival time is exponential with a mean of $\lambda = 30s$ and $\lambda = 80s$ during busy and idle periods respectively. Generally, the job arrival rate will not drop to zero instantly after busy periods, hence after we have submitted 130 jobs, we continuously submit 25 jobs but the

Table III
DESIGN OF EXPERIMENTAL WORKLOAD

No. of Jobs	Job input size in terms of no. of Maps	Job length in terms of average service time of tasks
49	1	100 seconds (16 jobs), 160 seconds (16 jobs), 300 seconds (11 jobs), 600 seconds (6 jobs)
21	2	100 seconds (7 jobs), 160 seconds (7 jobs), 300 seconds (4 jobs), 600 seconds (3 jobs)
18	10	100 seconds (7 jobs), 160 seconds (6 jobs), 300 seconds (3 jobs), 600 seconds (2 jobs)
11	50	100 seconds (4 jobs), 160 seconds (4 jobs), 300 seconds (2 jobs), 600 seconds (1 jobs)
8	100	100 seconds (4 jobs), 160 seconds (3 jobs), 300 seconds (1 jobs)
8	250	30 seconds (3 jobs), 60 seconds (2 jobs), 100 seconds (2 jobs), 2500 seconds (1 jobs)
5	400	30 seconds (2 jobs), 60 seconds (2 jobs), 100 seconds (1 jobs)
5	800	12 seconds (1 jobs), 30 seconds (2 jobs), 60 seconds (2 jobs)
2	1500	8 seconds (1 jobs), 12 seconds (1 jobs),
1	3000	20 seconds
1	5000	6 seconds
1	8000	5 seconds

Table IV
EVALUATION METRICS

Metric	Definition
System Normalized Performance (SNP)	SNP is the geometric mean of all the ANP values. Larger values of ANP and hence SNP are better, where a value of 1 is ideal.
Slowdown norms	The two norms used as evaluation metrics are: $l_1 = \frac{1}{N}\sum_j \sigma_j$ and $l_2 = \sqrt{\frac{1}{N}\sum_j \sigma_j^2}$ where $\sigma_j = 1/ANP_j$ and N is the number of jobs in the experiment. Smaller values of slowdown norms are better, where a value of 1 is ideal.
Unfairness	Unfairness is coefficient of variation of the ANPs. Smaller value is better, where a value of 0 is ideal.
Average Response Time	The total response time of all the jobs divided by the number of jobs

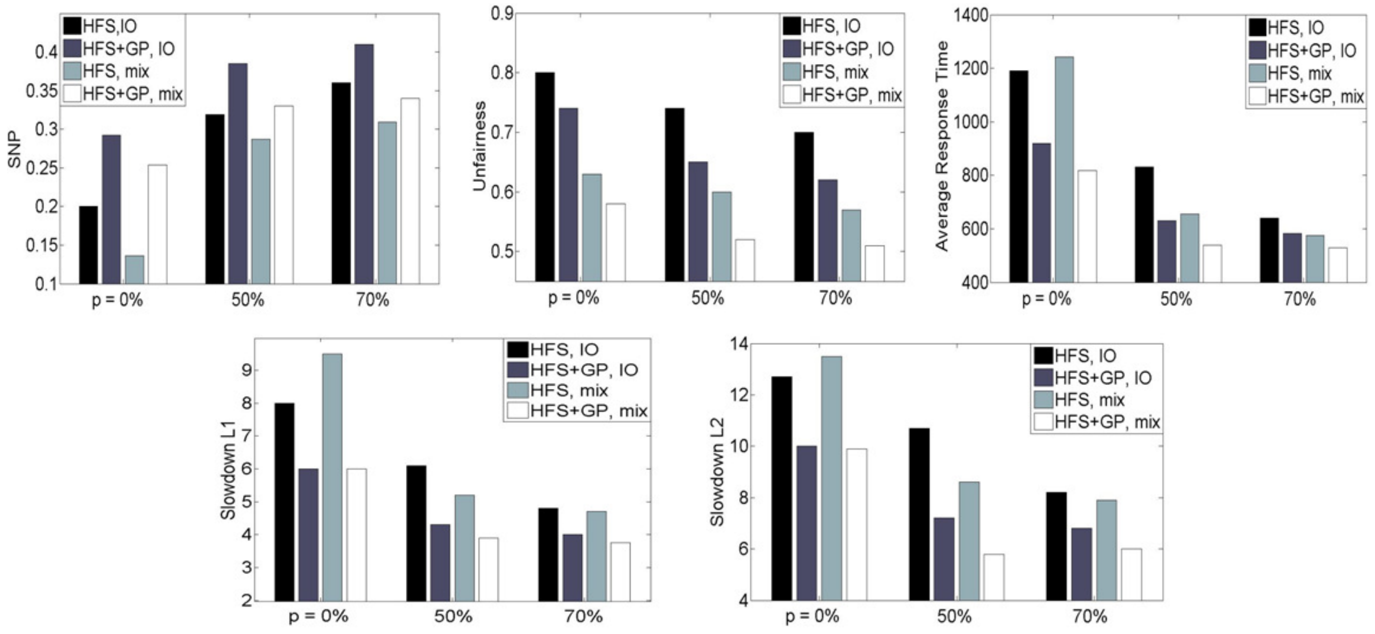

Figure 3. Performance metrics during busy hours under I/O and mixed workload. Here $p = x\%$ means the long job arrives after $x\%$ of other jobs. The system normalized performance is improved by 15% by using GP.

average job arrival rate drops to $\lambda = 80s$. The evaluate results do not include the performance of the last 25 jobs. According to Hadoop's manual and our custom settings, we set the preemption interval to be $20s$ to check for tasks to preempt. Because our test cluster is small compared to size of experimental jobs, therefore sometimes the long waiting queue is only caused by arrival of a single large-sized job. For this reason, if more than 90% of the waiting queue is attributed by a single job, we don't kill tasks of the long job to release slots. Finally, we control the long job arrive in the 1st, 66th, and 92nd place and other jobs arrive randomly. The later the long job arrives, the less impact it has on other jobs.

E. Experiment Results

During busy periods: Because only 30% are production jobs and the guaranteed share is high, we observe that production jobs come and go quickly in our experiments. Furthermore, non-production jobs occupy almost 90% of the cluster capacity most of the time. When a long job arrives, because the number of jobs in the system fluctuates, it gradually occupies 180-250 slots in less than 150s after its arrival under I/O intensive workload, even during busy periods. But we also observe that it takes longer to launch this number of tasks under mixed workload when the speed of releasing slots is lower.

When a large-size production job (above 250 tasks in this case) arrives, the total share of non-production jobs shrinks to its minimum value of 20%. Because the remaining slots are equally shared by all the running jobs, a number of tasks of the long job are killed when using the current preemption policy (HFS). Since production jobs complete quickly, several minutes later many long tasks are launched again when slots become available. But with the arrival of the next large production job, a large fraction of these long tasks are killed again. This really wastes resources especially when the workload is high and competition for resources is rather intense. But when using GP, short tasks are killed first and this makes the number of killed long tasks reduces to a minimum level.

In all the experiments, we observe only once the long job starving other jobs and the system throughput is greatly reduced because of this. In this scenario, two large non-production jobs, with 800 and 250 map tasks respectively, arrive shortly after a large production job has been launched. As a result, there is a huge number of requests for time slots. After 2 periods (6 minutes in our test) the overflowing tasks of the long job are preempted to make room for other jobs by GP. However, the tasks of the long job are not preempted any more before they finish.

The detailed results are shown in Figure 3. It can be observed that the System Normalized Performance (SNP) improves by around 15% on average. The SNP is generally better under I/O intensive workloads because the mixed workload contains more jobs. From this we can see that the improvement of SNP by using GP is greater when the system is under higher workload. With respect to fairness, we found that GP slightly outperforms HFS. We believe the reason is that the system load varies even during a short period, and GP speeds up jobs at the busiest times. As a result, the variations in job completion time become smaller. The average response time and slowdown norms reflect the average system performance and are consistent with SNP as we expected.

During idle periods: Among the four groups of experiments for idle periods, we observe that both policies deliver similar performance. This is because the job arrival rate is low in these experiments, as most of the time there are less than 5 jobs in the system. Most jobs finish quickly and therefore there are many idle slots in the cluster except when large jobs arrive. Thus, when large-sized production jobs arrive the number of tasks to be killed is small. The long job completes faster by using GP, but generally no matter what policy we use, the average values of performance metrics show little difference. We omit the corresponding figures due to space constraints.

VII. Related Work

Heterogeneity is a common characteristic in production MapReduce clusters, and ignoring it can lead to severe degradation of system performance. As a result, there have been many recent studies that attempt to deal with this issue. In this section we survey some of the representative works in this area.

In the original MapReduce paper, Dean et. al. noted that speculative execution can improve job response time by 44% [1]. Hadoop job schedulers speculatively re-execute tasks that appear to be stragglers, tasks lagging behind other tasks of a particular job. However, Hadoop job scheduler implicitly assumes cluster machines are homogeneous and tasks make progress linearly, and decides when to speculatively re-execute tasks that appear to be stragglers based on these assumptions. Zaharia et. al. [20] demonstrated that Hadoop's scheduler can cause severe performance degradation in heterogeneous clusters. They designed a new scheduling algorithm, Longest Approximate Time to End (LATE), that is robust to heterogeneity in cluster machines. The LATE scheduler uses a simple heuristic to estimate the progress of each task and launch speculative copy of tasks that have longest time to end compared to other tasks of the job on fast machines. LATE has been incorporated into Hadoop and indeed we have observed speculative tasks when there are free slots in the cluster during our experiments.

Heterogeneous machines and data skew can cause stragglers that significantly prolong job completion in an operational Microsoft MapReduce cluster. Ananthanarayanan et. al. [9] presented Mantri, a system that monitors tasks and selectively kills outliers using cause- and resource-aware techniques. One of the root causes for outliers is contention for resources, including machines and network bandwidth. Mantri preempts and restarts a task elsewhere if its remaining time is so large that there is a more than even chance that a restart would finish sooner. The "kill and restart" scheme drastically improves the job completion time without requiring extra slots. Compared to LATE, Mantri considered the impact of network congestion on tasks progress, which makes it achieve better performance since most jobs are data intensive in MapReduce clusters. We also observed in our experiments that the network bandwidth between nodes could be as low as $40-50$Kbps in Amazon EC2, although the network bandwidth can be up to 10Mbps when the cluster is not busy. Considering this, we plan in the future to integrate Mantri's method into GP and test it in Hadoop.

In traditional distributed and parallel systems, specially designed methods have also been proposed to deal with heterogeneity in workloads. Harchol-Balter et. al. [13] discovered that all workloads had a tail of long-lived jobs, i.e., the distributions have high variance, but an exponential distribution, which was used before, with the same mean would have lower variance. In [21], the authors proposed an algorithm that tries to assign jobs with similar length (for example, between 10 and 30 minutes) to selected hosts, achieving load balancing and significant reduction in job completion time. Through experiments, they showed that their algorithm is at least two orders of magnitude faster than other algorithms. However, the length of each job is assumed to be known in advance in their work. Later, in [15] Harchol-Balter presented a new method for situations where lengths of jobs are not known in advance. This method first assigned new jobs to hosts that were reserved for shortest jobs. If a job did not finish within a certain amount of time, it was killed and moved to a host for longer jobs. If the running time of this job continues to exceed time limit of that host, this job was killed again and assigned to another host until it finally reached the host that was supposed to accommodate jobs of its size. Provided the system load is not high (< 0.5), the algorithm in [15] is several orders of magnitude better than all the other techniques with respect to both mean response time and mean slowdown. However, these traditional systems are different from MapReduce clusters as jobs running on these traditional systems do not spawn multiple tasks to speed up their execution. Consequently, these solutions do not apply to MapReduce scheduling.

VIII. Conclusion

Modern MapReduce clusters are often shared by multiple types of jobs in order to improve resource utilization. Motivated by the problem caused by heterogeneity in workloads, we have analyzed the limitation of the preemption policy in MapReduce schedulers. This limitation can incur significant performance penalty when the load of the cluster is not high. To address this problem, we proposed a simple solution that can be combined with existing job schedulers. Experiments carried out in Amazon's Elastic Compute Cloud show our method can improve system normalized performance by 15% during busy periods by effectively avoiding costly preemption while preserving fairness.

Acknowledgment

This research was partially supported by the Natural Science and Engineering Council of Canada (NSERC) under its discovery program and partially by WCU (World Class University) program through the Korea Science and Engineering Foundation funded by the Ministry of Education, Science and Technology (Project No. R31-2008-000-10100-0).

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007.* ACM, 2007, pp. 59–72.

[3] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, “Quincy: fair scheduling for distributed computing clusters,” in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 261–276.

[4] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz, “Towards Understanding Cloud Performance Tradeoffs Using Statistical Workload Analysis and Replay,” *University of California at Berkeley, Technical Report No. UCB/EECS-2010-81*, 2010.

[5] A. Mishra, J. Hellerstein, W. Cirne, and C. Das, “Towards characterizing cloud backend workloads: insights from Google compute clusters,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 37, no. 4, pp. 34–41, 2010.

[6] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz, “Analysis and Lessons from a Publicly Available Google Cluster Trace,” 2010.

[7] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, “Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling,” in *Proceedings of the 5th European conference on Computer systems*. ACM, 2010, pp. 265–278.

[8] “Max-min fairness (wikipedia),” http://en.wikipedia.org/wiki/Max-min fairness.

[9] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, “Reining in the outliers in map-reduce clusters using Mantri,” in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. USENIX Association, 2010, pp. 1–16.

[10] “Hadoop fair scheduler,” http://hadoop.apache.org/common/docs/r0.21.0/.

[11] “Amazon ec2,” http://aws.amazon.com/ec2/.

[12] “Apache hadoop,” http://hadoop.apache.org/.

[13] M. Harchol-Balter and A. Downey, “Exploiting process lifetime distributions for dynamic load balancing,” *ACM Transactions on Computer Systems (TOCS)*, vol. 15, no. 3, pp. 253–285, 1997.

[14] S. Foss and D. Korshunov, “Heavy tails in multi-server queue,” *Queueing Systems*, vol. 52, no. 1, pp. 31–48, 2006.

[15] M. Harchol-Balter, “Task assignment with unknown duration,” *Journal of the ACM*, vol. 49, no. 2, pp. 260–288, 2002.

[16] C. T. Report, “Traffic analysis for Voice over IP,” 2001.

[17] J. Lempiainen and M. Manninen, *Radio interface system planning for GSM/GPRS/UMTS*. Springer Netherlands, 2001.

[18] D. Jackson and F. Kunzinger, “Calculation of system availability using traffic statistics,” *Bell Labs technical journal*, vol. 7, no. 3, pp. 139–150, 2002.

[19] J. Little, “A proof of the queuing formula $L = \lambda W$,” *Operations Research*, vol. 9, no. 3, pp. 383–387, 1961.

[20] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica, “Improving mapreduce performance in heterogeneous environments,” in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*. USENIX Association, 2008, pp. 29–42.

[21] M. Harchol-Balter, M. Crovella, and C. Murta, “On choosing a task assignment policy for a distributed server system,” *Journal of Parallel and Distributed Computing*, vol. 59, no. 2, pp. 204–228, 1999.