

# FlexSlot: Moving Hadoop into the Cloud with Flexible Slot Management

Yanfei Guo\*, Jia Rao\*, Changjun Jiang<sup>†</sup> and Xiaobo Zhou\*

\*Department of Computer Science, University of Colorado, Colorado Springs, USA  
Email: {yguo,jrao,xzhou}@uccs.edu

<sup>†</sup>Department of Computer Science & Technology, Tongji University, Shanghai, China  
Email: cjjiang@tongji.edu.cn

**Abstract**—Load imbalance is a major source of overhead in Hadoop where the uneven distribution of input data among tasks can significantly delays the job completion. Running Hadoop in a private cloud opens up opportunities for mitigating data skew with elastic resource allocation, where stragglers are expedited with more resources, yet introduces problems that often cancel out the performance gain: (1) performance interference from co-running jobs may create new stragglers; (2) there exist a semantic gap between Hadoop task management and resource pool-based virtual cluster management preventing efficient resource usage.

We present *FlexSlot*, a user-transparent task slot management scheme that automatically identifies map stragglers and resizes their slots accordingly to accelerate task execution. *FlexSlot* adaptively changes the number of slots on each virtual node to promote efficient usage of resource pool. Experimental results with representative benchmarks show that *FlexSlot* effectively reduces job completion time by 46% and achieves better resource utilization.

## I. INTRODUCTION

Hadoop, the open source implementation of the MapReduce programming model [7], is increasingly popular in Big Data analytics due to its simplicity of use and the scalability to large clusters. However, studies have shown that current use of Hadoop in enterprises and research stay in an ad hoc manner, leaving advanced features underused [19], potential performance unexploited [16], [26], [27], and resources in Hadoop clusters inefficiently utilized [18]. In particular, load imbalance (a.k.a. skew) among Hadoop tasks poses significant challenges to achieving good performance and high resource utilization [12], [15], [17]. Skew that could come from uneven data distribution or non-uniform data processing cost creates *stragglers*, tasks that runs significantly slower than others. Such sluggish tasks can take more than five times longer to complete than the fastest task [16], slowing the overall job completion.

To address hardware failure and misconfiguration, Hadoop *speculatively* runs a backup copy of a slow task on a different machine. Besides fault-tolerance, speculative execution is able to expedite stragglers due to skew to a certain extent as the backup copy may run on a better performing machine. However, the differences in machines are not significant enough to mitigate skew and job performance is still bottlenecked by stragglers. Although skew can be eliminated by using customized and job-specific data partitioners for balancing workload among tasks [15], this approach requires domain

knowledge on the structure of input data and imposes burdens on users. To this end, researchers proposed to mitigate skew by dynamically re-partitioning data during job execution [16]. Nevertheless, re-distributing data at runtime introduces overhead of moving data between machines.

We propose a new perspective on tackling the skew problem in Hadoop applications. Rather than mitigating skew among tasks, we try to balance the processing time of tasks even with the presence of data imbalance. Specifically, tasks with more data or more expensive data records are accelerated by having more resources. We purposely create heterogeneous clusters and match different machine capabilities with the actual processing demands of unbalanced tasks. Cloud computing, unlocked by virtualization, allows the creation of dynamic virtual clusters with elastic resource allocation. Hadoop clusters can be easily scaled out by adding virtual nodes with a latency of several minutes. More importantly, individual nodes can also be scaled up with more resources. A recent study found that the ability to scale-up leads to better Hadoop performance in a virtual cluster compared to a native cluster with the same settings [4].

However, moving Hadoop into the cloud introduces additional problems that can outweigh the benefit of flexible resource management. First, clouds are usually shared by multiple users in order to increase hardware utilization. Interferences from co-located users may create new stragglers in Hadoop applications [6], [14]. Second, to reduce the resources required to run a virtual cluster, private clouds often multiplex a pool of shared resources among virtual nodes [20]. Virtual cluster management such as VMware DRS (Distributed Resource Manager) [10] and OpenStack Utilization-based Scheduling [2] dynamically allocates resources to individual nodes according to their estimated demands. It eliminates the need of proactively allocating resource for slave nodes. Surprisingly, we find that there exists a semantic gap between demand-based resource allocation and the actual needs of Hadoop tasks. The virtual nodes that receive disproportionately more resources due to high demands actually run fast tasks, leaving nodes with stragglers deprived of resources. Our study exposes that running unmodified Hadoop in such a cloud does not mitigate skew but aggravates the imbalance.

In this work, we explore the possibility of using elastic resource allocation in the cloud to mitigate skew in Hadoop applications. We design *FlexSlot*, an effective yet simple extension to the slot-based Hadoop task scheduling framework. *FlexSlot* automatically identifies stragglers and resizes their

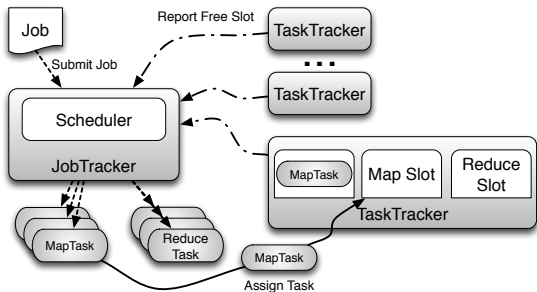


Fig. 1. The slot-based task scheduling in Hadoop.

slots. Slot resizing not only allows stragglers to receive more resources but also alleviates the interference between co-running jobs. To expose the actual demands of tasks, FlexSlot co-locates stragglers with fast tasks on a node by dynamically changing its number of slots. As fast tasks finish quickly and the node is allocated a large amount of resources, stragglers in fact receive more resources and their executions are thus accelerated.

We implemented FlexSlot on a 32-node Hadoop virtual cluster and evaluated its benefits using the Purdue MapReduce Benchmark Suite (PUMA) [3] with datasets collected from real applications. We compared the performance of FlexSlot running different workloads with that of the stock Hadoop and a recently proposed skew mitigation approach SkewTune [16]. Experimental results show that FlexSlot reduces job completion time by as much as 46% and 22% compared to stock Hadoop and SkewTune, respectively. FlexSlot also achieves better cloud resource utilization with both the flexible slot size and flexible number of slots.

The rest of this paper is organized as follows. Section II introduces the background of Hadoop, discusses existing issues, and presents a motivating example. Section III elaborates FlexSlot’s architecture and key designs. Section IV presents the implementation details of FlexSlot. Section V gives the testbed setup and experimental results. Related work is presented in Section VI. We conclude this paper in Section VII.

## II. BACKGROUND AND MOTIVATION

We first describe the basics of the Hadoop MapReduce framework and discuss the causes of skew in Hadoop tasks. Then, we show that elastic resource allocation in a cloud and an ideal task scheduling in Hadoop together help balance task execution time and improve the overall job performance significantly. Finally, we demonstrate that a naive migration of Hadoop to the cloud does not mitigate skew but aggravates task imbalance.

### A. Skew in Hadoop MapReduce

The data processing in MapReduce is expressed as two functions: *map* and *reduce*. The map function takes an input pair and produces a list of intermediate key/value pairs. The reduce function processes the intermediate key with the list of its values and generates the final results. In the implementation of Hadoop, the *map* and *reduce* functions are implemented in `MapTask` and `ReduceTask`. Each MapReduce job is divided into multiple map tasks and reduce tasks.

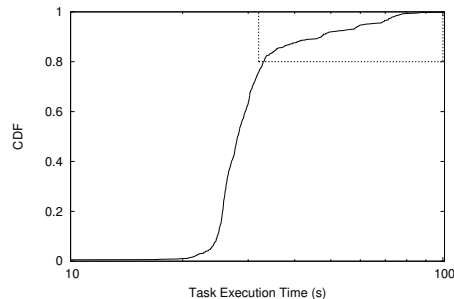


Fig. 2. The skewed task execution time of *wordcount*.

TABLE I. TASK SKEWNESS OF *wordcount* ON THREE CLUSTERS.

	Homo	Heter	Heter + Skew-Sched
Skewness	2.65	2.25	0.31

Figure 1 shows MapReduce execution environment in Hadoop with one master node and multiple slave nodes. The master node runs `JobTracker` and manages the task scheduling on the cluster. Each slave node runs `TaskTracker` and controls the execution of individual tasks. Hadoop uses a slot-based task scheduling algorithm. Each `TaskTracker` has a preconfigured number of map slots and reduce slots. Task trackers report their number of free slots to the job tracker through heartbeat [1]. The job tracker assigns a task to each free slot.

Ideally, each task should take approximately the same time to complete if it is assigned the same amount of data. However, there is no guarantee that data is evenly distributed among tasks or the same amount of input data will take the same time to process. Hadoop tries to divide the input data set into a collection of input splits, ideally each matching the block size of Hadoop Distributed File System (HDFS). Map tasks usually process block of input at a time using the default `FileInputFormat`. If the input data contains file with various sizes (e.g., files smaller or larger than the HDFS block size), taking one block of data as input inevitably leads to input skew in map tasks. Although it is possible to combine small blocks into a map input (e.g., using a custom input reader), there is only a small portion of hadoop users customizes input handling while data skew in map tasks is quite common in production hadoop clusters [19]. Similarly, reduce tasks may also have imbalanced input size as the partitions generated by map tasks can contain different number of key/value pairs [8], [15], [16]. Moreover, the same input size can lead to distinct task execution time as some records are inherently more expensive to process. As a result, unbalanced task runtime is prevalent in Hadoop applications [19].

Figure 2 shows an example of skewed task execution in the *wordcount* benchmark. The job was run on a homogeneous cluster with 32 nodes. We can see that nearly 20% of tasks run at least 20% longer than the others. Such an imbalance leads to prolonged job completion time and wasted cluster resources on nodes that were idle waiting for the stragglers. Next, we show that how a re-organization of cluster resources, i.e., using a heterogeneous cluster, improves Hadoop performance.

## B. Mitigating Skew with Heterogeneous Resource Allocation

In this subsection, we study how does heterogeneous resource allocation to map tasks help the overall job performance. The data skew in reduce tasks can be mitigated by repartitioning and rebalancing intermediate data [11], [15], [16]. But a similar approach cannot be easily applied to map tasks. We show that purposely allocating more resources to nodes that run stragglers effectively mitigates map input skew. If not otherwise stated, tasks refer to map tasks throughout this paper.

We created three 32-node Hadoop clusters, each with a total capacity of 153.6 GHz CPU resource and 128 GB memory in our university cloud. The first cluster (denoted as *Homo*) emulates a physical cluster with homogeneous configurations on each node. The resources were evenly distributed to nodes, resulting in an uniform node capacity of 4.8 GHz CPU and 4 GB memory. The second cluster (denoted as *Heter*) contained nodes with heterogeneous resource allocations. We profiled the resource demands of individual Hadoop tasks and created powerful nodes for stragglers. We first ran a job in a homogeneous cluster and determined the stragglers. We then scaled up the nodes running stragglers according to their runtime statistics. Specifically, we increased the CPU resource and memory size of these nodes proportionally based on their CPU time and I/O wait time in the last run. We then re-ran the job on the adjusted cluster. Although stragglers had their input data on the powerful nodes, Hadoop’s task scheduler may still launch them remotely on less powerful nodes if there are no available slots on the powerful ones. In the third cluster (denoted as *Heter + Skew-sched*), we forced that stragglers only run on powerful nodes.

We calculated the statistical skewness in task runtimes in a job  $J$  as  $\frac{1}{n} \sum_{i \in J} \frac{(x_i - \mu)^3}{\sigma^3}$ , where  $x_i$  is the runtime of individual tasks,  $\mu$  and  $\sigma$  represent the average and deviation, respectively. The lower the skewness, the more balanced task execution. Table I lists the task skewness of *wordcount* on the three clusters. As expected, cluster *Homo* shows significant skew in task completion with a skewness of 2.65. We can also see that creating a heterogeneous cluster alone only mitigated the skew to a certain extent (e.g., reducing skewness to 2.25) because Hadoop task scheduler may not run stragglers on powerful nodes. In contrast, strictly requiring stragglers be run on powerful nodes significantly improved task balance with a skewness of 0.30. This confirms previous findings [27] that allocating more data and work on powerful nodes optimizes Hadoop performance in a heterogeneous cluster environment.

Unfortunately, it is difficult to predict which tasks will be the stragglers and determine their resource demands before actually running these tasks. There is a need for dynamic resource allocation in Hadoop clusters in response to stragglers.

## C. Automatically Tuning Hadoop with Demand-based Resource Allocation

Modern cloud management middleware, such as VMware DRS, XenServer and OpenStack, supports demand-based resource allocation, where a virtual cluster shares a resource pool and individual nodes receive resources according to their estimated demands. This technique promotes efficient usage

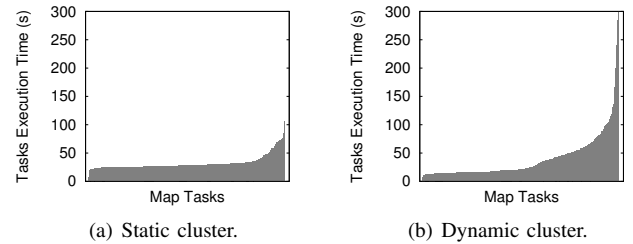


Fig. 3. Task runtime of *wordcount* on two clusters.

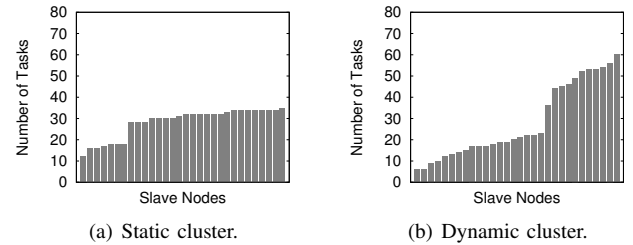


Fig. 4. Task distribution of *wordcount* on two clusters.

of cluster resource and avoids the potential resource wastage of manual configured resource reservation. Ideally, if nodes running stragglers show high demands, demand-based resource allocation is able to automatically tune a Hadoop cluster into a heterogeneous cluster that accelerates stragglers. However, we found that there exists a semantic gap between demand-based resource allocation and the actual needs of Hadoop tasks, making incorrect decisions in resource allocations.

We created a resource pool of 153.6 GHz CPU resources and 128 GB memory, which is shared by a 32-node Hadoop cluster. VMware DRS was used to automatically allocate resources on these virtual nodes. Figures 3(a) and 3(b) show the execution time of individual map tasks on a static homogeneous cluster and the dynamic cluster with demand-based resource allocation, respectively. From the figure, we can see that demand-based resource allocation unexpectedly aggravated skewness in task execution time. Although the tasks on the left of Figure 3(b) ran faster than those in the static cluster, stragglers in the dynamic cluster (tasks on the right tail) appeared to be significantly slower.

Figure 4 shows the number of finished tasks on each virtual node in the two clusters. We find that in the dynamic cluster, some nodes ran 10X more tasks than the nodes with fewest tasks. An examination of the Hadoop execution log and the VMware resource allocation log revealed that these “hot spot” nodes ran mostly fast tasks and received disproportionately more resources than the “idle” nodes that turned out to host straggler tasks. We attribute the counterintuitive resource allocations to the semantic gap between the VM resource demands estimated by the cloud management and the actual Hadoop task demands. VMware DRS computes a node’s CPU demand based on its recent actual CPU consumption and estimates the memory demand according to the ratio of touched pages in a set of randomly-selected pages.

However, the demand-based allocation does not meet stragglers’ needs. Study [19] has shown that production Hadoop clusters are still dominated by I/O intensive jobs and stragglers

with more data are likely to spend a significant amount of time waiting for disk I/O. Thus, nodes running stragglers appear to be less busy than nodes with fast tasks and are allocated less resource by the demand-based cloud management. For memory, a straggler is unable to use the additional memory allocated because its memory usage is upper bounded by the JVM heap size of its slot. The heap size of a slot is statically set in the cluster configuration file. The selection of the heap size is usually conservative to prevent a node from memory thrashing.

**[Summary]** We have shown that skew in Hadoop map tasks can be effectively mitigated by accelerating stragglers with more resources. However, demand-based dynamic resource allocation in a cloud does not meet stragglers’ needs and the fixed slot configuration in Hadoop prevents a flexible use of dynamic resources. These findings motivated us to make Hadoop run more efficiently in the cloud in the presence of skew and the in place cloud management. We found that a simple extension to Hadoop’s map slot management effectively directs resource to map stragglers in a cloud environment. Next, we present FlexSlot, a user-transparent flexible slot management scheme for Hadoop.

### III. FLEXSLOT DESIGN

We aim to mitigate skew in Hadoop applications by leveraging dynamic resource allocation in the cloud. We focus on a private cloud platform in which dynamic resource allocation to virtual clusters is automated by estimating the demands of individual virtual nodes and adjusting their resources accordingly. In this section, we present *FlexSlot*, an effective yet simple extension to the Hadoop’s slot management scheme that automatically identifies map stragglers at run-time and adjusts both the map slot size and number on the nodes hosting the stragglers. The dynamic adjustment to slot configurations effectively exposes the actual demands of stragglers to the low-level virtual resource management and guides the demand-based allocation to efficiently allocate resources to stragglers.

#### A. Overview

FlexSlot provides two functionalities: on-the-fly straggler identification and automated slot reconfiguration.

**Identifying Stragglers.** FlexSlot continuously monitors two task-specific metrics during task execution: *progress rate* and *input processing speed*. Based on a synthesis of these metrics, FlexSlot is able to identify tasks that are abnormally slower than their peers due to either uneven data distribution, expensive record or cloud interference. Once having determined stragglers, FlexSlot infers their resource bottlenecks based on resource usages obtained on the straggler nodes.

**Proactively changing the size of slots.** If FlexSlot determines that a straggler’s performance is bottlenecked by I/O operations, it proactively terminates the straggler and restarts it with a larger slot size. Since it is hard to predict the memory requirement of a task, FlexSlot uses a trial-and-error approach to change the slot size at incremental steps. The increment continues until the overhead of task restarting outweighs the improvement on task processing speed.

**Adaptively adjusting the number of slots.** FlexSlot bridges the semantic gap between Hadoop tasks and the

demand-based resource allocation by adaptively changing the number of slots on Hadoop nodes. Contrary to intuition, FlexSlot adds more slots to straggler nodes in order to expose the actual straggler demand to the cluster-level resource management.

#### B. Identifying Stragglers

There are three causes of straggler tasks in a cloud environment: (1) uneven data distribution among tasks; (2) non-uniform data processing time; and (3) performance interference from co-running jobs. We measure two performance metrics during task execution. Hadoop provides *progress score* to represent the fraction of work completed. It is computed as the ratio of finished input size and the original input size. *Progress rate* [27], which is defined as the change in progress score in unit time, is a good online measure of task performance. If all tasks process input data at the same speed, the tasks with smaller progress rates are likely to have large input sizes. Another important metric is the *input processing speed*  $D$ . It counts how many bytes are processed in unit time. Ideally, all tasks should have the same input processing speed given that the data distribution is uniform across tasks and it takes the same amount time to process the same amount of data. If some tasks exhibit apparently slower processing speed, it is likely that their record are more expensive to process or they are experiencing interferences.

Neither of the two metrics alone can reliably determine stragglers. Thus, we create a composite metric based on these two. We normalize both metrics against their maximum values among all tasks, both in the range of 0 to 1, and simply use their sum as a heuristic to identify stragglers. Since progress rate can be expressed by  $\frac{D}{S}$ , where  $S$  is the input data size and  $D$  is the input processing speed. Therefore, we use  $P = (1 + \frac{1}{S}) \cdot D$  to measure task performance.

We consider straggler tasks as outliers compared to regular tasks. For every measurement interval, we calculate the value of  $P$  for all tasks and perform a *k-means* clustering algorithm on the  $P$  values. The clustering algorithm divides all tasks into two clusters. The tasks in cluster with smaller  $P$  values are considered significantly different from the rest of tasks.

#### C. Determining Performance Bottlenecks

Having stragglers identified by the proposed heuristic, FlexSlot then determines how to accelerate the execution of stragglers. Depending on the cause of straggler, these tasks can be bottlenecked either by disk I/O performance or insufficient CPU resources.

**Disk I/O Bottleneck due to Data Skew.** Previous studies show that the number of input records can largely affect the task completion time even with the same amount of input data [24], [25], [28]. The *expensive record* is a major type of data skew in map tasks. A map task processes a collection of key value pairs, each pair as one record. Due to data skew, some records may require more resources to process than others, leading to more processing time. These expensive records can simply contain large data that fills the output buffer quickly. When the output buffer is full, intermediate results need to be written to disk. Thus, expensive records cause more frequent disk I/O operations.

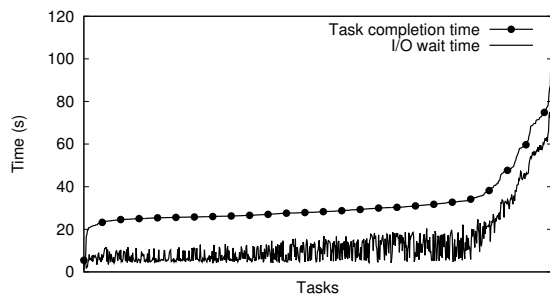


Fig. 5. Task completion time and `iowait` time.

The *inverted-index* benchmark contains non-uniform records and incurs skewed task processing time. Figure 5 shows the breakdown of task execution time for all tasks. Since expensive records incur excessive I/O operations, we are interested in how much does I/O time (e.g., I/O wait) contribute to the total task completion time. Tasks are sorted in the ascending order of their completion time, with tasks on the right tail considered as stragglers. We find that I/O wait contributed most to the prolonged completion time in stragglers. We observed as much as 8X more I/O wait time than that of regular tasks. Thus, a large portion of time spending in waiting for I/O is a good indicator of stragglers with I/O bottleneck. A large slot memory size, e.g., a large output buffer, can effectively reduce I/O operations.

#### CPU Starvation due to Inaccurate Demand Estimation.

As shown in Figure 4, demand-based resource allocation incurs severe imbalance on task distribution, with some nodes heavily loaded and some nodes hosting only a few stragglers. The culprit is that nodes running faster tasks received more resources from cloud resource management. It enters a loop that nodes with more resources ran tasks even faster. Stragglers usually incur more I/O operations and appear to be less busy in terms of CPU usage. Thus, such tasks may become victims of CPU starvation in demand-based resource allocation. Figure 6 shows the breakdown of task runtime of benchmark *wordcount*. It plots each task's actual `cputime`, I/O wait and `steal` time, which is the time that a task's node is ready to run but fails to get scheduled by the hypervisor. In the figure, we can see that `steal` time contributed most to the straggler runtime. If given sufficient CPU, these stragglers can be effectively accelerated. Thus, a large portion of time being ready but not running is a good indicator of insufficient CPU resources. Next, we show that purposely coupling stragglers with faster tasks exposes the demands of stragglers to the cluster-level resource manager.

#### D. Resizing Slot Memory

Increasing the memory size of a slot can reduce the number of I/O operations and effectively expedites stragglers caused by data skew. However, the memory that can be used by a slot is limited by the heap size of its JVM and the output buffer size of the task running on it. Although the JVM heap size can be dynamically changed through JVM memory ballooning, the output buffer size of a task cannot be altered without restarting the task. Thus, it is difficult to increase a slot's memory size during task execution. We rely on the straggler identification

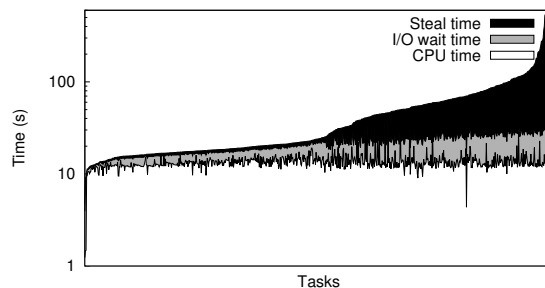


Fig. 6. Task completion time and the `cputime`.

---

#### Algorithm 1 Flexible slot size.

---

```

1: Variables: Heartbeat interval  $T$ ; Data split size  $S$ ;
2:
3: The killcount of task  $k$  is initialized to 0.
4: /* Only apply to stragglers */
5: function RESIZESLOTMEM( $k$ )
6:   get current input processing speed  $D$ 
7:   if  $k.killcount == 0$  or  $\frac{S}{D} - \frac{S}{k.D_{prev}} > T$  then
8:     kill  $k$ , free slot  $s$ 
9:      $s.size = s.size * (1 + \alpha)$ 
10:    launch  $k$  on slot  $s$ 
11:     $k.killcount = k.killcount + 1$ 
12:   end if
13:    $k.D_{prev} = D$ 
14: end function

```

---

to determine stragglers in a timely manner. As such, we can afford to kill stragglers and restart them on larger slots.

Another question is how to efficiently determine the slot size for straggler tasks. We propose an algorithm that is automatically invoked on all straggler tasks at every heartbeat. It determines the slot memory size for a straggler task. The pseudo-code for this algorithm is shown in Algorithm 1. The goal of this algorithm is to increase the slot memory size for straggler tasks so that the performance of these tasks becomes close to that of regular tasks. For some straggler tasks, it may require multiple iterations to reach the desired slot memory size.

Once a straggler is identified, the function `ResizeSlotMem` is called to determine the appropriate memory size of the task. If the task has not gone through a memory resizing (i.e., first time killing and  $killcount = 0$ ), we kill it immediately and increase the memory size of the slot at a step of  $\alpha$  (empirically set to 0.2). Otherwise, we test if an additional round of killing and memory resizing is needed. Suppose a task  $k$ 's input data size is  $S$ . For each round (at heartbeat intervals), we keep increasing the memory size of the slot until the resulting decrease in the input processing time (i.e.,  $\frac{S}{D} - \frac{S}{k.D_{prev}}$ ) no longer outweighs the wasted processing time due to task killing (i.e., the last heartbeat interval  $T$ ). Then, the slot resizing process is stopped.

#### E. Adjusting Slot Number

The key rationale behind adjusting slot number on Hadoop nodes is to couple stragglers with fast tasks and expose

---

**Algorithm 2** Flexible slot number.

---

```
1: Variables: list of node  $L$ ; list of steal time  $ST$  on these nodes
2:
3: while true do
4:   calculate the average steal time  $\mu$ 
5:   calculate the standard deviation of steal times  $\sigma$ 
6:   if  $\sigma \leq \lambda$  then
7:     break
8:   end if
9:   for  $st_i$  in  $ST$  do
10:     $\Delta st_i = st_i - \mu$ 
11:   end for
12:   sort list  $\Delta ST$  in ascending order
13:   find node  $l_{max}$  that has the maximum  $\Delta st$ 
14:   find node  $l_{min}$  that has the minimum  $\Delta st$ 
15:    $l_{min}.removeMaxMapSlots(1)$ 
16:    $l_{max}.addMaxMapSlots(1)$ 
17: end while
```

---

stragglers' demand to cloud management. Once stragglers are identified and their bottlenecks are determined as insufficient CPU resource, FlexSlot tries to move slots from nodes running fast tasks to nodes running stragglers. We design a greedy algorithm for automating the slot adjustment. The pseudo-code is shown in Algorithm 2. The objective is to minimize the steal time (defined in Section III-C) on the Hadoop cluster. To reduce the steal time on straggler nodes, which are either caused by CPU starvation or interference, we increase the slot number on these nodes in hopes that the additional slots will bring faster tasks and finally increase the nodes' CPU allocation. We follow a simple heuristic of moving slots from the node with the smallest steal time to the nodes with the largest. We ensure that the number of total task slots in the cluster remain the same. The algorithm keeps calculating the average node steal time  $\mu$  and its standard deviation  $\sigma$ . When  $\sigma$  is greater than the threshold  $\lambda$ , the algorithm decides that the imbalance of CPU resource needs to be adjusted. Changing  $\lambda$  allows us to trade-off between the resource imbalance and the converging speed of the algorithm.

Changing the slot number on nodes not only affects cloud resource allocation but also influences Hadoop task scheduling, leading to a better coordination between the two. Without slot adjustment, each Hadoop node is configured with the same number of slots but is allocated imbalanced resources. Dynamically changing slot number on each node directs both task and resources to the same node. For example, adding slots to a node brings more resources and tasks. The performance interference from co-running jobs is also mitigated as the objective of the algorithm is to minimize steal time.

One concern of this algorithm is that newly added slots can be assigned with a straggler task, aggravating the skew and worsening the starvation on straggler nodes. However, as majority of the tasks are regular tasks, the new slot is likely to be assigned to a regular task. Currently, all nodes are monitored for the automated slot memory resizing. If we find that the newly scheduled task is a straggler task, we kill it and reassign it to another node that has no stragglers while trying to preserve data locality.

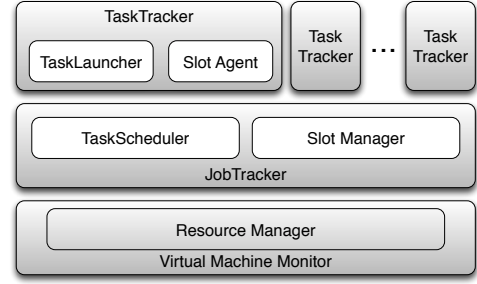


Fig. 7. The architecture of FlexSlot.

#### IV. IMPLEMENTATION

Figure 7 shows the architecture of FlexSlot. It consists of two main components: SlotAgent and SlotManager. The SlotAgent is implemented in the TaskTracker. It provides the slot management interface that allows online change of both map slot memory size and the number of slots. It also collects performance metrics in a slave node. The SlotManager is implemented in the JobTracker. It identifies straggler map tasks based on the performance metrics reported by the SlotAgent, determines the slot memory size for each straggler task and the number of slots on each slave node. These two components are organized in a client/server model and communicate via RPC (remote procedure call). To support the functionalities of these two components, we also modified the implementations of TaskTracker, TaskLauncher, and TaskScheduler.

In Hadoop, the `mapred-site.xml` file contains all the configurations of the MapReduce sub-system. All cluster parameters are set according to this file during the Hadoop initialization and cannot be changed online. Therefore, we modified TaskTracker and TaskLauncher to support online parameter update. Table II lists the functions that are related to the slot management interface.

FlexSlot maintains a hash table for tasks and their corresponding task slot configurations. These configurations can be changed with the `setTaskConfiguration()` function. We have implemented `setSlotMemory` function for updating the slot memory allocation of a specific task. We modified the `startNewTask` function to apply the updated configuration before launching a new task. By updating the configuration table, we enable changing the configuration of a Hadoop cluster and its tasks online.

We added two functions to modify the map slot number: `addMaxMapSlots()` and `removeMaxMapSlots()`. They will change the variable `maxMapSlots` to increase or decrease the number of map slots. The task launcher also keeps a local copy of `maxMapSlots`. We have also modified the task launcher so that their max slot variables can be correctly updated via `addMaxSlots()` and `removeMaxSlots()` functions.

Through these interfaces, we can change the slot memory size and the number of slots without restarting the task tracker. These changes are compatible with the slot-based task scheduling in Hadoop. Any scheduler can use these interfaces to perform the slot management during the task scheduling.

TABLE II. THE SLOT MANAGEMENT INTERFACE.

Function	Class	Functionality
<code>setTaskConfiguration(TaskID tid, JobConf newConf)</code>	TaskTracker	Update the configuration table for a specific task
<code>setSlotMemory(TaskID tid, int memSize)</code>	TaskTracker	Set memory size of the slot of a specific task
<code>addMaxSlots(int numSlots)</code>	TaskLauncher	Add slots to the task launcher.
<code>removeMaxSlots(int numSlots)</code>	TaskLauncher	Remove slots from the task launcher
<code>addMaxMapSlots(int numSlots)</code>	TaskTracker	Add map slots to the task tracker
<code>removeMaxMapSlots(int numSlots)</code>	TaskTracker	Remove map slots from the task tracker

TABLE III. PUMA BENCHMARK DETAILS.

Benchmark	Input Size (GB)	Input Data
tera-sort	150	TeraGen
inverted-index	150	Wikipedia
term-vector	150	Wikipedia
wordcount	150	Wikipedia
grep	150	Wikipedia
k-means	30	Netflix data, $k = 6$
histogram-movies	100	Netflix data
histogram-ratings	100	Netflix data

## V. EVALUATION

### A. Testbed Setup

We performed evaluations of FlexSlot on our university cloud. It consists of 8 HP BL460c G6 blade servers. Each server is equipped with 2-way Intel quad-core Xeon E5530 CPUs and 64GB memory. The servers are connected with 10 Gbps Ethernet. VMware vSphere 5.1 is used to provide the server virtualization.

We used a 32-node virtual Hadoop cluster to evaluate FlexSlot. Each node was initially configured with 4 VCPU and 4 GB memory. Depending on different experiments, the resource allocation to individual Hadoop nodes can be fixed or managed by the demand-based resource allocation in VMware DRS. We deployed Hadoop stable release version 1.1.1 and each VM ran Ubuntu Linux with kernel 2.6.24. Two nodes were configured as the JobTracker and NameNode, respectively. The rest 30 nodes ran as slave nodes for HDFS storage and MapReduce task execution. We set the HDFS block size to its default value 64 MB. Each slave node was initially configured with 4 map slots and 2 reduce slots and parameters `io.sort.mb` and `mapred.child.jvm.opts` were set to 100 MB and 200 MB, respectively. These Hadoop task settings were dynamically adjusted by FlexSlot during task execution.

For comparison, we also implemented a recently proposed skew mitigation approach SkewTune [16]. SkewTune parallelizes a straggler task by repartitioning and redistributing its input data. It mitigates the data skew and improves the job completion time. It assumes that all slave nodes have the same processing capacity. SkewTune evenly distributes the unprocessed data across all available nodes to mitigate the data skew. However, in a virtualized environment, the resource allocation of different slave nodes can be different, especially in demand-based resource allocation. Evenly distributing the unprocessed data may not be the best option. The existence of hotspot nodes also incurs unnecessary data movements.

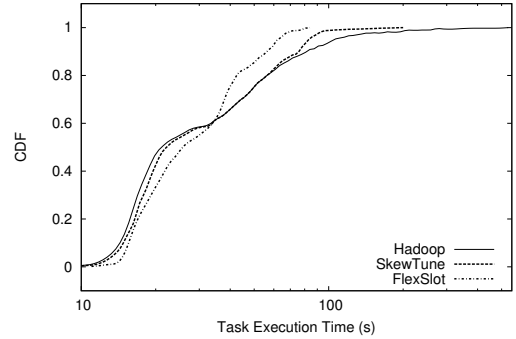


Fig. 8. The CDF of task completion time.

### B. Workloads

We used the PUMA benchmark suite [3] for evaluation. It contains various MapReduce benchmarks and real-world test inputs. Table III shows the benchmarks and their configurations used in our experiments.

These benchmarks are divided into three categories based on the content of their input data. The *tera-sort* benchmark uses a data set that is generated by *TeraGen*. The data has a relatively uniform distribution due to the randomized generation process. The *inverted-index*, *term-vector*, *wordcount* and *grep* benchmarks use the data extracted from Wikipedia. The data contains records with different sizes. Some of them are significantly larger than the average. This provides a good example of data skew. The *k-means*, *histogram-movies* and *histogram-ratings* benchmarks use the data from Netflix. They are good examples of expensive records. The content of the records can significantly affect input processing time even with the same record size.

### C. Mitigating Data Skew

In this subsection, we study the effectiveness of FlexSlot in mitigating data skew. If tasks take about the same time to finish even in the presence of skew, we consider that the skew has been mitigated. We measured the distribution of the task completion time of different benchmarks. We used the stock Hadoop with demand-based resource allocation in the cloud as the baseline. The total number of task slots in FlexSlot is set to be the same as that in stock Hadoop and SkewTune for a fair comparison.

Figure 8 shows the distribution of the task completion time of the *inverted-index* benchmark due to three different approaches. The skewness of the task execution time due to the stock Hadoop, SkewTune, and FlexSlot are 4.98, 1.68 and 0.96, respectively. The result shows that FlexSlot is the most

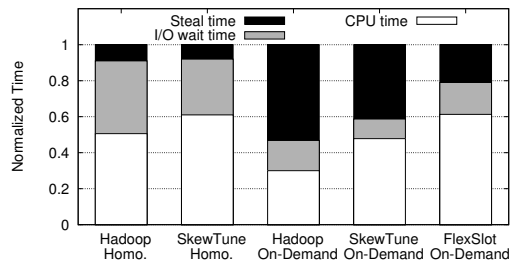


Fig. 9. Breakdown of task runtime of *inverted-index*.

effective in mitigating data skew in all three approaches. Compared to SkewTune, FlexSlot significantly reduced the task completion time of straggler tasks. FlexSlot and SkewTune have similar performance on those regular tasks, but FlexSlot brings more improvement to the performance of straggler tasks than SkewTune does.

FlexSlot outperformed SkewTune for two reasons. First, FlexSlot continuously detects straggler tasks and reduces the execution time of straggler tasks with automated slot memory resizing. On the other hand, SkewTune only mitigates a straggler task when there are free slots to parallel the straggler task. Second, SkewTune does not have the coordination between the Hadoop task scheduler and the cloud infrastructure. It does not eliminate the hotspot nodes. SkewTune tends to use the hotspot nodes to parallelize the execution of a straggler task because these nodes hold more resources. This introduces additional data movement for the straggler tasks that can be finished quickly with simply more resource. FlexSlot does not have the problems mainly because it moves slots, instead of tasks, across slave nodes. It preserves the data locality and minimizes the data movement for mitigating data skew.

The results in Figure 8 also show that FlexSlot has longer completion time for fast tasks than stock Hadoop has. It is due to the fact that running stock Hadoop with demand-based resource allocation created “hot spot” nodes that boost these fast tasks. FlexSlot avoided “hot spot” nodes from receiving disproportional more resources. Note that making fast tasks run even faster does not help improve overall job completion time, which is bottlenecked primarily by stragglers.

To further study how skew is mitigated, we show detailed breakdown of task completion time. For comparison, we also ran benchmarks in Hadoop and SkewTune on static homogeneous virtual clusters (denoted as HOMO). Figure 9 shows the breakdown of average task execution time of *inverted-index* on different clusters. FlexSlot reduced the portion of steal time by 60.5% and 49.1% when compared to stock Hadoop and SkewTune with demand-based resource allocation. The huge steal time of stock Hadoop and SkewTune with demand-based resource allocation is the sign of imbalanced CPU resource allocation between fast tasks and straggler tasks. The reduction of steal time suggests that FlexSlot was effective in preventing CPU starvation on stragglers. Moreover, FlexSlot reduced the I/O wait time compared to stock Hadoop and SkewTune in homogeneous clusters. FlexSlot achieved 39.7% and 26.7% lower I/O wait time than stock Hadoop and SkewTune, respectively. It suggests that FlexSlot reduce the I/O operations for straggler tasks with larger slot memory size.

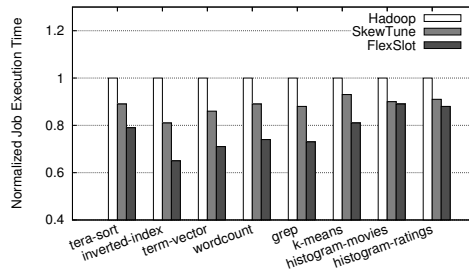


Fig. 10. The normalized job completion time.

#### D. Reducing Job Completion Time

We have shown that FlexSlot is effective in mitigating skew. In this subsection, we study how does the mitigation help improve overall job completion time. Similarly, we use the job completion time in the stock Hadoop as the baseline and compare the normalized job completion time of FlexSlot and SkewTune. Figure 10 shows the normalized job completion time of all benchmarks due to these three approaches. The results show that for benchmarks with expensive records, e.g., *inverted-index*, *term-vector*, *wordcount* and *grep*, FlexSlot outperformed stock Hadoop by 35.1%, 29.3%, 26.7% and 27.4%, respectively. FlexSlot also outperformed SkewTune by 21.6%, 17.1%, 16.9% and 17.1% in these benchmarks.

Benchmarks such as *k-means*, *histogram-movies* and *histogram-ratings* use data from Netflix. The input data is relatively uniform in record size. In the experiments with these benchmarks, FlexSlot outperformed the stock Hadoop by 25%, 13% and 12%, respectively. However, FlexSlot has less performance improvement on *histogram-movies* and *histogram-ratings* than on *k-means*, because those benchmarks have small memory demand due to the small volume of their intermediate results. The default configuration of slot memory size already provides sufficient output buffer.

The *k-means* benchmark has a large volume of intermediate data. It requires large output buffer and more memory to reduce I/O operations. It contains computation intensive tasks that require a lot of CPU resource. FlexSlot achieved much shorter job completion time than stock Hadoop did because FlexSlot was able to improve both the slot memory size and CPU resource allocation for *k-means*.

FlexSlot achieved 17.5%, 1.2% and 3.3% shorter job completion time than SkewTune in *k-means*, *histogram-movies* and *histogram-ratings* benchmarks, respectively. FlexSlot only had marginal performance improvement over SkewTune in *histogram-movies* and *histogram-ratings* benchmarks because their data skew is not severe and the CPU requirement is also low. But for the job with significant data skew and high CPU consumption, FlexSlot clearly showed a significant advantage compared to SkewTune.

#### E. Mitigating Performance Interference

Note that adjusting slot size and number can possibly resolve the interference between co-running jobs. To create interferences, we consolidated two clusters with the same configuration (see Section V-A for details) to the shared physical infrastructure. We submitted the same jobs to these two clusters



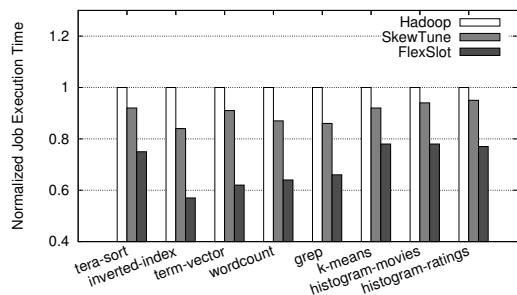


Fig. 11. The normalized job completion time with interferences of co-running jobs.

and calculated their average job completion times. We used the job completion time in stock Hadoop as the baseline, and normalized FlexSlot and SkewTune’s performance against it.

Figure 11 shows the normalized job completion time of all benchmarks due to three approaches with performance interference from co-running jobs. The results show that FlexSlot outperformed stock Hadoop and SkewTune by as much as 43.8% and 33.2%, respectively. FlexSlot achieved 34.2% ~ 43.8% shorter job completion time than stock Hadoop in the *inverted-index*, *term-vector*, *wordcount* and *grep* benchmarks. FlexSlot also outperformed SkewTune by 24.3% ~ 33.2% in these benchmarks. Due to the performance among co-running jobs, the difference of job completion time between FlexSlot, SkewTune, and stock Hadoop is larger than the results in Figure 10.

For benchmarks with less data skew, *histogram-movies* and *histogram-ratings*, FlexSlot outperformed stock Hadoop by 22% and 23% respectively, and outperformed SkewTune by 17.1% and 18% respectively. The difference of job completion time between SkewTune and stock Hadoop is around 5%, because there is no much data skew to mitigate. The results clearly show that FlexSlot can further improve the job completion time by mitigating the performance interference from co-running jobs in a cloud environment.

### F. Improving Resource Utilization

By default, FlexSlot keeps the total number of task slots unchanged in a Hadoop cluster. Thus, any addition of task slots on one node should be coupled with the removal of slots on another node. As discussed in previous experiments, FlexSlot’s slot movement effectively mitigates skew and improves job performance. In this subsection, we extend FlexSlot to handle unlimited number of slots. That is, FlexSlot adds or removes slot to/from a node only based on the performance of tasks running on the node, without the constraint of maintaining the total number of slots in the cluster. Although it is a common practice to match the map slot number with the number of CPUs on a node, it is not considered the optimal configuration. With this experiment, we study if an even more flexible slot management could further improve cluster resource utilization and job performance.

We compare two FlexSlot variations (i.e., with limited slots and unlimited slots) with stock Hadoop. Figure 12 shows the normalized job completion time of all benchmarks due to the three approaches. FlexSlot with an unlimited

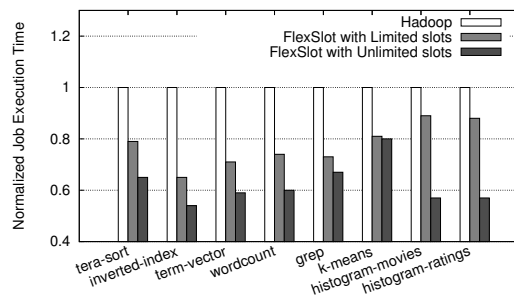


Fig. 12. Job completion time with an unlimited number of slots.

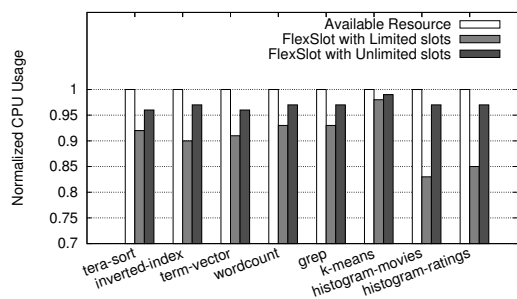
number of slots achieved up to 46% shorter job completion time than the stock Hadoop. It also outperformed the FlexSlot using a limited number of slots by 35%. The improvement varies depending on the job characteristics. For most jobs, using an unlimited number of slots can bring approximately 17% shorter job completion time than using a limited number of slots. But for CPU intensive jobs like *k-means*, removing the limitation on slots makes no major difference because the job performance is bottlenecked by the number of physical CPUs. Adding more slots does not allow more tasks to run concurrently. In contrast, job such as *histogram-movies* and *histogram-ratings* have a significant portion of I/O time. With a limited number of slots, the resources of the virtual cluster, especially the CPU resource, are not fully utilized. Figure 12 shows that FlexSlot with unlimited slots achieved on average 35% shorter job completion time compared to FlexSlot with limited slots.

Next, we compare the resource utilizations in different FlexSlot variations. Figures 13(a) and 13(b) show the CPU and memory usage of FlexSlot with and without a limit on the number of total slots. The results show that FlexSlot without a slot limit resulted in 10% and 11% higher in CPU and memory utilization, respectively. As discussed above, these resources were used to accelerate job execution. Note that due to the different job characteristics of *k-means*, *histogram-movies*, and *histogram-ratings* benchmarks, the changes in resource utilizations of these jobs were different. For the *k-means* benchmark, the resource utilization was high even with a limited number of total slots. For *histogram-movies* and *histogram-ratings* benchmarks, FlexSlot with a slot limit left roughly 15% and 26% CPU and memory resources unused, respectively. Removing the limit improved the resource utilization significantly.

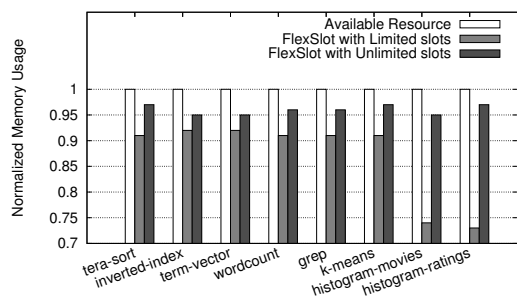
These results suggest that FlexSlot with unlimited slots can be a viable approach for Hadoop in the cloud. It improves the performance by maximizing the resource utilization. It does not affect the performance of jobs that have high resource demand because the slots will only be added when there is available resource.

### G. Slot Resizing Overhead

FlexSlot uses task-killing-based approach in the slot memory resizing and allows tasks to be killed multiple times. The task killing inevitably incurs overhead as the killed stragglers lose already performed work. We measure the overhead using the number of kills per straggler. The fewer the kills the smaller



(a) CPU



(b) Memory

Fig. 13. Resource consumption of FlexSlot.

the overhead. Figure 14 shows the average number of kills per straggler in different benchmarks. The results show that most straggler only required 1 kill to run normally as fast as other tasks.

The slot memory resizing algorithm increases a slot’s memory size by step  $\alpha$  for each task kill. The average number of task kills is affected by the value of  $\alpha$ . Figure 15 shows the average task kills due to different  $\alpha$  values. For *inverted-index* and *k-means* benchmarks, increasing  $\alpha$  resulted in a significant reduction in the average number of task kills. These benchmarks are very sensitive to the memory size because they have large intermediate data. Using a larger  $\alpha$  increases the amount of memory resource that will be allocated to them in each task kill. The *histogram-movies* benchmark, which does not have large intermediate data, is less sensitive to the memory size. Allocating more memory to it during task killing does not have a great impact on its task execution time. A larger  $\alpha$  value leads to more memory resource waste. According to the observation shown in Figure 15, we empirically set  $\alpha$  to 0.2 in FlexSlot because benchmarks with different types benefit from it while not incurring significant memory waste.

## VI. RELATED WORK

MapReduce is a popular programming model for large-scale data processing [7]. Hadoop, the open-source implementation of MapReduce, provides a software framework to support the distributed processing of large datasets [1]. YARN [23] is the second generation of Hadoop. It uses containers to replace task slots and provides a finer granularity of resource management. But it only allows different jobs to have different containers. The tasks in one job still have the containers of one fixed size. Therefore, YARN cannot mitigate data skew with flexible size of containers.

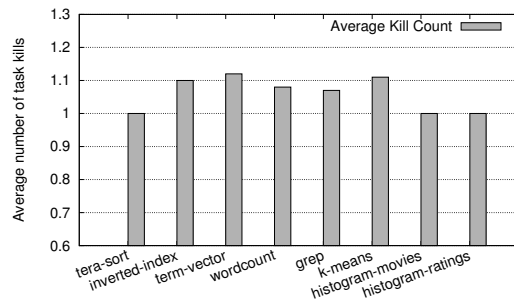
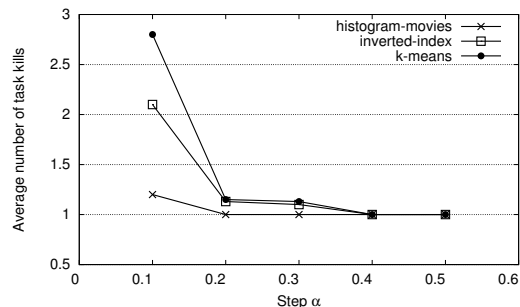


Fig. 14. Average number of task kills for different benchmarks.

Fig. 15. Average number of task kills per straggler due to different step  $\alpha$ .

There are a few studies on data skew mitigations. SkewReduce [15] alleviates the computational skew by balancing data distribution across nodes using an user-defined cost function. SkewTune [16] repartitions the data of stragglers to take the advantage of idle slots freed by short tasks. However, moving repartitioned data to idle nodes requires extra I/O operations, which could aggravate the performance interference. More importantly, these approaches do not have the coordination between Hadoop and the cloud infrastructure.

A Hadoop cluster in the cloud can be seen as a heterogeneous Hadoop cluster because of the dynamic resource allocation. A number of studies proposed different task scheduling algorithms to improve Hadoop performance for heterogeneous environments [5], [9], [27]. DynMR [22] enables interleaved MapReduce execution that overlaps the reduces tasks with map tasks. By opportunistically schedule all tasks, DynMR significantly increases both performance and efficiency of Hadoop. For instance, PIKACHU focuses on achieving optimal workload balance for Hadoop [9]. It presents guidelines for the trade-offs between the accuracy of workload balancing and the delay of workload adjustment. But, these studies focus on hardware heterogeneity in physical machine based clusters. They are not designed for VM-based clusters where the heterogeneity can be changed by dynamic resource allocation. FlexSlot is able to adapt to the change of heterogeneity with flexible slot size and number.

There are recent studies that focus on improving performance of applications in the cloud by dynamic resource allocation [13], [20], [21]. For example, Bazaar is a cloud framework that predicts the resource demand of applications based on the high-level performance goals [13]. It translates the performance goal of an application into multiple combinations of resources and selects the combination that is most suitable

for a cloud provider. One recent work focuses on demand-based resource allocation [20]. It efficiently distributes the resource by dynamically allocating the overall capacity among VMs based on their demands. But these approaches focus on VM management only and are less effective than FlexSlot for Hadoop in the cloud because of the semantic gap between demand-based resource allocation and the actual needs of Hadoop tasks.

## VII. CONCLUSIONS

Hadoop provides an open-source implementation of the MapReduce framework. But its design poses challenges to attain the best performance in the cloud environment due to the data skew. Moving Hadoop into the cloud offers the possibility of mitigating data skew with dynamic resource allocation. But Hadoop lacks of the coordination between its task scheduler and the cloud management, which brings new challenges due to the performance interference and demand-based resource allocation. In this paper, we propose and design *FlexSlot*, an effective yet simple extension to the Hadoop's slot management that provides the flexibility to change the slot memory size and the number of slots in a slave node online. We have implemented FlexSlot in Hadoop and evaluated its effectiveness on a 32-node virtual Hadoop cluster with various workloads. Experimental results show that FlexSlot is able to reduce job completion time by as much as 46% compared to stock Hadoop and by as much as 22% compared to SkewTune. An extension to FlexSlot with unlimited slots further improves the resource utilization of the virtual cluster.

Our future work will be on extending the flexible slot management approach to the resource management in Apache Hadoop YARN.

### Acknowledgement

This research was supported in part by U.S. NSF CAREER award CNS-0844983, research grants CNS-1422119, CNS-1320122 and CNS-1217979, and NSF of China research grant 61328203.

## REFERENCES

- [1] Apache Hadoop Project. <http://hadoop.apache.org>.
- [2] OpenStack Wiki: Utilization based scheduler spec. <https://wiki.openstack.org/wiki/UtilizationBasedSchedulingSpec>.
- [3] PUMA: Purdue mapreduce benchmark suite. <http://web.ics.purdue.edu/~fahmad/benchmarks.htm>.
- [4] *Virtualizing Apache Hadoop*. <http://www.vmware.com/files/pdf/Benefits-of-Virtualizing-Hadoop.pdf>.
- [5] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar. Tarazu: optimizing mapreduce on heterogeneous clusters. In *Proc. of the Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.
- [6] R. C. Chiang and H. H. Huang. TRACON: interference-aware scheduling for data-intensive applications in virtualized environments. In *Proc. of Int'l Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.
- [7] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proc. of the USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2004.
- [8] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Communication of ACM*, 35(6):85–98, 1992.
- [9] R. Gandhi, D. Xie, and Y. C. Hu. PIKACHU: How to rebalance load in optimizing mapreduce on heterogeneous clusters. In *Proc. of the USENIX conference on Annual Technical Conference (ATC)*, 2013.
- [10] A. Gulati, A. Holler, M. Ji, G. Shanmuganathan, C. Waldspurger, and X. Zhu. Vmware distributed resource management: Design, implementation and lessons learned. In *VMware Technical Journal*, 2012.
- [11] Y. Guo, J. Rao, and X. Zhou. iShuffle: Improving hadoop performance with shuffle-on-write. In *Proc. of the USENIX Int'l Conference on Autonomic Computing (ICAC)*, 2013.
- [12] S. Ibrahim, H. Jin, L. Lu, S. Wu, B. He, and L. Qi. LEEN: Locality/fairness-aware key partitioning for mapreduce in the cloud. In *Proc. of the IEEE Int'l Conference on Cloud Computing Technology and Science (CLOUDCOM)*, 2010.
- [13] V. Jalaparti, H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Bridging the tenant-provider gap in cloud services. In *Proc. of the ACM Symposium on Cloud Computing (SOCC)*, 2012.
- [14] H. Kang and J. L. Wong. To hardware prefetch or not to prefetch?: a virtualized environment study and core binding approach. In *Proc. of the Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.
- [15] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *Proc. of the ACM Symposium on Cloud Computing (SOCC)*, 2010.
- [16] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. SkewTune: Mitigating skew in mapreduce applications. In *Proc. of the ACM SIGMOD*, 2012.
- [17] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy. A platform for scalable one-pass analytics using mapreduce. In *Proc. of the ACM SIGMOD*, 2011.
- [18] S. Rao, R. Ramakrishnan, A. Silberstein, M. Ovsiannikov, and D. Reeves. Sailfish: A framework for large scale data processing. In *Proc. of the ACM Symposium on Cloud Computing (SOCC)*, 2012.
- [19] K. Ren, Y. Kwon, M. Balazinska, and B. Howe. Hadoop's adolescence: An analysis of hadoop usage in scientific workloads. In *Proc. of VLDB*, 2013.
- [20] G. Shanmuganathan, A. Gulati, and P. Varman. Defragmenting the cloud using demand-based resource allocation. In *Proc. ACM SIGMETRICS*, 2013.
- [21] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Proc. of the ACM Symposium on Cloud Computing (SOCC)*, 2011.
- [22] J. Tan, A. Chin, Z. Z. Hu, Y. Hu, S. Meng, X. Meng, and L. Zhang. DynMR: Dynamic mapreduce with reduced task interleaving and map task backfilling. In *Proc. of the ACM European Conference on Computer Systems (EuroSys)*, 2014.
- [23] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O. Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet another resource negotiator. In *Proc. of the ACM Symposium on Cloud Computing (SOCC)*, 2013.
- [24] A. Verma, L. Cherkasova, and R. H. Campbell. ARIA: automatic resource inference and allocation for mapreduce environments. In *Proc. of the ACM Int'l Conference on Autonomic Computing (ICAC)*, 2011.
- [25] A. Verma, L. Cherkasova, and R. H. Campbell. Resource provisioning framework for mapreduce jobs with performance goals. In *Proc. of the ACM/IFIP/USENIX Int'l Conference on Middleware*, 2011.
- [26] J. Wolf, D. Rajan, K. Hildrum, R. Khandekar, V. Kumar, S. Parekh, K.-L. Wu, and A. balmin. FLEX: a slot allocation scheduling optimizer for mapreduce workloads. In *Proc. of the ACM/IFIP/USENIX Int'l Conference on Middleware*, 2010.
- [27] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.
- [28] Z. Zhang, L. Cherkasova, and B. T. Loo. AutoTune: Optimizing execution concurrency and resource usage in mapreduce workflows. In *Proc. of the USENIX/ACM Int'l Conference on Autonomic Computing (ICAC)*, 2013.