iShuffle: Improving Hadoop Performance with Shuffle-on-Write

Yanfei Guo, *Member, IEEE*, Jia Rao, *Member, IEEE*, Dazhao Cheng, *Student Member, IEEE*, and Xiaobo Zhou, *Senior Member, IEEE*

Abstract—Hadoop is a popular implementation of the MapReduce framework for running data-intensive jobs on clusters of commodity servers. *Shuffle*, the all-to-all input data fetching phase between the map and reduce phase can significantly affect job performance. However, the shuffle phase and reduce phase are coupled together in Hadoop and the shuffle can only be performed by running the reduce tasks. This leaves the potential parallelism between multiple waves of map and reduce unexploited and resource wastage in multi-tenant Hadoop clusters, which significantly delays the completion of jobs in a multi-tenant Hadoop cluster. More importantly, Hadoop lacks the ability to schedule task efficiently and mitigate the data distribution skew among reduce tasks, which leads to further degradation of job performance. In this work, we propose to decouple shuffle from reduce tasks and convert it into a platform service provided by Hadoop. We present *iShuffle*, a user-transparent shuffle service that pro-actively pushes map output data to nodes via a novel *shuffle-on-write* operation and flexibly schedules reduce tasks considering workload balance. Experimental results with representative workloads and Facebook workload trace show that iShuffle reduces job completion time by as much as 29.6 and 34 percent in single-user and multi-user clusters, respectively.

Index Terms—MapReduce, shuffle, dataskew, task scheduling

1 Introduction

Hadoop is a popular open-source implementation of the MapReduce programming model for processing large volumes of data in parallel [13]. Each job in Hadoop consists of two dependent phases, each of which contains multiple user-defined *map* or *reduce* tasks. These tasks are distributed independently onto multiple nodes for parallel execution. The decentralized execution model is essential to Hadoop's scalability to a large number of nodes as map computations can be placed near their input data stored on individual nodes and there is no communication between map tasks.

There are many existing studies focusing on improving the performance of map tasks. For example, work has been done to preserve locality via map scheduling [36] or input replication [9], as the data locality is critical to map performance. Others also designed interference [11] and topology [22] aware scheduling algorithms for map tasks. While there is extensive work exploiting the parallelism and improving the efficiency in map tasks, only a few studies have been devoted to expedite reduce tasks.

The *shuffle* phase performs an all-to-all copying of intermediate data from the map phase to the reduce phase. It involves intensive communications between nodes and can significantly delay job completion. Hadoop employs *slow-start* that strives to hide the latency incurred by the shuffle phase by starting reduce tasks as soon as map output files are available. Existing work tries to overlap shuffle with map by proactively sending map output [12] or fetching map output in a globally sorted order [33].

Unfortunately, the coupling of *shuffle* and *reduce* phases in a reduce task presents challenges to attaining high performance in multi-tenant environments. First, the fairness between multiple jobs is enforced by limiting the number of concurrent tasks of each job, which means the reduce tasks may not be able to run at the same time. In fact, the reduce tasks in a multi-tenant cluster tend to run in multiple wave. The shuffle phase will not start utill the corresponding reduce task is scheduled to run, and only the first wave of reduce can be overlapped with map, leaving the potential parallelism in shuffle unexploited. Second, tasks scheduling in Hadoop is oblivious of the data distribution skew among reduce tasks [14], [15], [20], [21]. Machines running shuffle-heavy reduce tasks become bottlenecks. Finally, one user's long-running shuffle may occupy the reduce slots that would otherwise be used more efficiently by other users, lowering the utilization and throughput of the cluster.

In this paper, we propose to decouple the *shuffle* phase from reduce tasks and convert it into a platform service provided by Hadoop. Therefore, the shuffle phase can start without schedule any reduce task. We present *iShuffle*, a user-transparent shuffle service that overlaps the data shuffling of any reduce task with the map phase, addresses the input data skew in reduce tasks, and enables efficient reduce scheduling. iShuffle features a number of key designs: (1) proactive and deterministic pushing shuffled data from map to Hadoop nodes when map output files are materialized to local file systems, a.k.a, *shuffle-on-write*. (2) automatic predicting reduce execution time based on the

• *The authors are with the Department of Computer Science, University of Colorado, Colorado Springs, CO 80918.*
E-mail: {yguo, jrao, dcheng, xzhou}@uccs.edu.

Manuscript received 4 Dec. 2015; revised 25 Apr. 2016; accepted 26 Apr. 2016. Date of publication 7 July 2016; date of current version 17 May 2017. Recommended for acceptance by B. He.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2016.2587645

input partition size and placing the shuffled data to mitigate the *partition skew* and to avoid hotspots. (3) binding reduce tasks with data partitions only when reduce is scheduled to realize the load balancing enabled by the partition placement. (4) preemptive reduce scheduling to ensure fairness between the reduce tasks from different MapReduce jobs.

We implemented iShuffle on a 32-node Hadoop cluster and evaluated its benefits using benchmark jobs from the Purdue MapReduce Benchmark Suite (PUMA) [3] and the HiBench [2] with datasets collected from real applications. We compared the performance of iShuffle running both shuffle-heavy and shuffle-light workloads with that of stock Hadoop and three recently proposed approach (i.e., Hadoop-A in [33], DynMR in [29] and Sailfish in [25]). Experimental results show that iShuffle reduces job completion time by 29.6, 28.8, and 22.4 percent compared with stock Hadoop, Hadoop-A and DynMR, respectively. We also used the workload trace from Facebook's Hadoop cluster and evaluated iShuffle in a multi-user environment. The results show that iShuffle significantly improves the completion time for regular jobs without affecting the performance of map-only jobs. iShuffle also outperform Sailfish in multi-tenant environments with 16 percent less performance impact on small jobs.

A preliminary version of this paper appeared in the Proc. of USENIX ICAC'2013 [19]. It was awarded the best paper. In this significantly extended manuscript, we have redesigned and developed iShuffle with multi-user support and preemptive reduce scheduling. We have updated the existing experiments with more comprehensive benchmarks and compared the performance of iShuffle with a recently published approach. We have also performed new experiments on a multi-user Hadoop cluster using workload trace from Facebook.

The rest of this paper is organized as follows. Section 2 introduces the background, discusses existing issues, and presents a motivating example. Section 3 elaborates iShuffle's key designs. Section 4 gives the testbed setup, experimental results and analysis. Related work is presented in Section 5. We conclude this paper in Section 6.

2 Background and Motivation

2.1 Hadoop MapReduce Framework

The data processing in MapReduce [13] model 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 intermediate values associated with the same key are grouped together and then passed to the same reduce function via *shuffle*, an all-map-to-all-reduce communication. The reduce function processes the intermediate key with the list of its values and generate the final results.

Hadoop's implementation of the MapReduce programming model pipelines the data processing and provides fault tolerance. Fig. 1 shows an overview of job execution in Hadoop. The Hadoop runtime partitions the input data and distributes map tasks onto individual cluster nodes for parallel execution. Each map task processes a logical split of the input data that resides on the Hadoop Distributed File System (HDFS) and applies the user-defined map function on each input record. The map outputs are partitioned according to

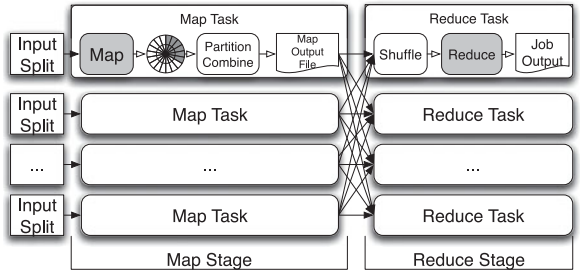


Fig. 1. An overview of data processing in Hadoop MapReduce framework.

the number of reduce tasks and combined into keys with associated lists of values. A map task temporarily stores its output in a circular buffer and writes the output files to local disk every time the buffer becomes full (i.e., buffer *spill*).

A reduce task consists of two phases: *shuffle* and *reduce*. The shuffle phase fetches the map outputs associated with a reduce task from multiple nodes and merges them into one reduce input. An external merge sort algorithm is used when the intermediate data is too large to fit in memory. Finally, a reduce task applies the user-defined reduce function on the reduce input and writes the final result to HDFS. The reduce phase cannot start until all the map phases have finished as the reduce function depends on the output generated by all the map tasks. To overlap the execution of map and reduce, Hadoop allows an early start of the shuffle phase (by scheduling the corresponding reduce task) as soon as 5 percent of the map tasks have finished.

2.2 Input Data Skew Among Reduce Tasks

The output of map tasks is a collection of intermediate keys and their associated value lists. Hadoop organizes each output file into partitions, one per reduce task and each containing a different subset of the intermediate key space. Hadoop determines which partition a key-value pair will go to by computing a hash value. Since the intermediate output of the same key are always assigned to the same partition, skew in the input data set will result in disparity in the partition sizes. Such a *partitioning skew* is observed in many applications running in Hadoop [14], [20], [21]. Although Hadoop can aggregate multiple keys in the same partition for one reduce task to process, the default hash-based partitioner lacks the ability to balance partitions for uniform workload distribution. Some user-defined partitioner may mitigate the skew but does not guarantee an even data distribution among reduce tasks. As a result, some reduce tasks take significant longer time to finish, slowing down the job.

2.3 Inflexible Scheduling of Reduce Tasks

Reduce tasks are created and assigned a task ID by Hadoop during the initialization of a job. The task ID is then used to identify the associated partition in each map output file. For example, shuffle fetches the partition that matches the reduce ID from all map tasks. When there are reduce slots available, reduce tasks are scheduled in the ascending order of their task IDs. Although such a design simplifies task management, it may lead to long job completion time. Due to the strict scheduling order, it is difficult to prioritize reduce tasks that are predicted to run longer than others.

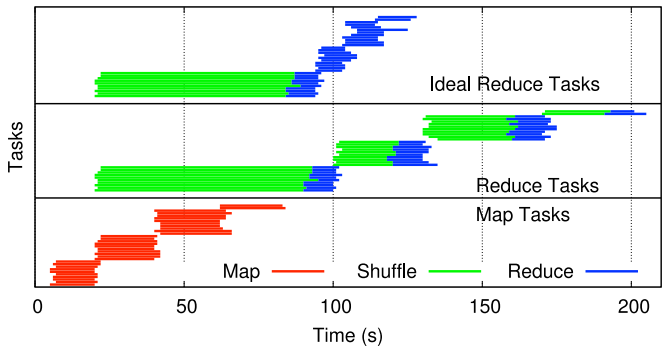


Fig. 2. *Tera-sort* job execution.

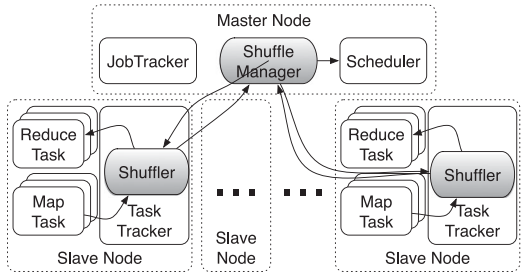


Fig. 3. The architecture of iShuffle.

Further, partitions required by a reduce task may not be generated at the time it is scheduled, occupying the reduce slot and wasting cluster cycles which would otherwise be used by another reduce with all partitions ready.

2.4 Tight Coupling of Shuffle and Reduce

As part of a reduce task, shuffle cannot start until the corresponding reduce is scheduled. Besides the inefficiency of job execution, the coupling of shuffle and reduce also leaves the potential parallelism between within and between jobs unexploited. In a production environment, a MapReduce cluster is shared by many users and multiple jobs [36]. Each job only gets a portion of the execution slots and often requires multiple execution waves, each of which consists of one round of map or reduce tasks. Because of the coupling, data shuffling in later reduce waves cannot be overlapped with map waves.

Fig. 2 shows the execution of one *tera-sort* job with 4 GB dataset in a 10-node Hadoop cluster. Each node was configured with one map slot and one reduce slot. The job was divided into 32 map tasks and 32 reduce tasks [13], [31], resulting in four map and reduce waves. We use the duration of the shuffle phase between last execution wave and next reduce phase, termed as *shuffle delay*, to quantify how data shuffling affects the completion of reduce tasks. Due to the overlapped execution, the first reduce wave experienced a shuffle delay of 11 seconds. Unfortunately, remaining reduce waves had on average a delay of 23 seconds before the reduce phase could start. Given that the average length of the reduce phase was 25 seconds, the reduce waves would have been completed in less than half the time if the shuffle delay can be completely overlapped with map.

Fig. 2 also suggests that although the overlapping of reduce and map reduced the shuffle delay from 23 to 11 seconds, the first reduce wave occupied the slots three times longer than the following waves. Most time was spent in the shuffle phase waiting for the completion of map tasks. In production systems, allowing other jobs to use these slots may outweigh the benefits brought by the overlapped execution.

The map slots that are freed after the completion of all map tasks may seem to be usable for the second and the third wave of reduce tasks. But, map slots and reduce slots are two different resources in Hadoop. The default design does not allow running reduce tasks on map slots or vice versa. Most importantly, in multi-tenant Hadoop clusters, the map slots that released by one job may be occupied by another job. In this case, it it also hard to reuse the map slots for reduce tasks.

These observations revealed the negative impacts of coupling shuffle and reduce on job execution and motivated us to explore a new shuffling design for Hadoop. We found that decoupling shuffle from reduce provides a number of benefits. We are can shuffle the data for all reduce tasks regardless the number of concurrent reduce tasks, which makes the task scheduling approximates the ideal case in Fig. 2. It also enables skew-aware placement of shuffled data, flexible scheduling of reduce tasks, and complete overlapping the shuffle phase with map tasks. In Section 3, we present *iShuffle*, a decoupled shuffle service for Hadoop.

3 iShuffle Design

We propose *iShuffle*, a job-independent shuffle service that pushes the map output to its designated reduce node. It decouples shuffle and reduce, and allows shuffle to be performed independently from reduce. It predicts the map output partition sizes and automatically balances the placement of map output partitions across nodes. iShuffle binds reduce IDs with partition IDs lazily at the time reduce tasks are scheduled, allowing flexible scheduling of reduce tasks.

3.1 Overview

Fig. 3 shows the architecture of iShuffle. iShuffle consists of three components: *shuffler*, *shuffle manager*, and *task scheduler*. The shuffler is a background thread that collects intermediate data generated by map tasks and predicts the size of individual partitions to guide the partition placement. The shuffle manager analyses the partition sizes reported by all shufflers and decides the destination of each partition. The shuffle manager and shufflers are organized in a layered structure which is similar to Hadoop's `JobTracker` and `TaskTrackers`. The task scheduler extends existing Hadoop schedulers to support flexible scheduling of reduce tasks. We briefly describe some major features of iShuffle.

User-Transparent Shuffle Service. A major requirement of iShuffle design is the compatibility to existing Hadoop jobs. To this end, we design shufflers and the shuffle manager as job-independent components, which are responsible for collecting and distributing map output data. This design allows the cluster administrator to enable or disable iShuffle through the options in the configuration files. Any user job can use iShuffle service without modifications.

Shuffle-on-Write. The shuffler implements a shuffle-on-write operation that proactively pushes the map output

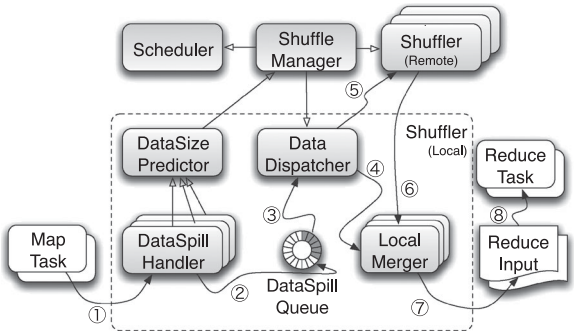


Fig. 4. Workflow of Shuffle-on-Write.

data to different nodes for future reduce tasks every time such data is written to local disks. The shuffling of all map output data can be performed before the execution of reduce tasks.

Automated Map Output Placement. The shuffle manager monitors the growth rate of each partition and predicts the final size of the partitions on all slave nodes. With this information, an automated partition placement algorithm is used to determine the destination for each map output partition. The objective is to balance the global data distribution and mitigate the non-uniformity reduce execution time.

Flexible Dispatching of Reduce Tasks. The task scheduler in iShuffle assigns a partition of a reduce task only when the task is dispatched to a node with available slots. To minimize reduce execution time, iShuffle always associates partitions that are already resident on the reduce node to the scheduled reduce.

Preemptive Reduce Scheduling. The fair scheduler in iShuffle supports preemption of reduce tasks. To ensure the max-min fairness, the scheduler pauses some running reduce tasks to make room for incoming jobs. To reduce the performance impact of the preemption, iShuffle employs a checkpoint-on-preemption mechanism and a preemptive selection algorithm that exploits the unfilled execution slots of a job.

3.2 Shuffle-on-Write

iShuffle decouples *shuffle* from a *reduce* task and implements data shuffling as a platform service. This allows the shuffle phase to be performed independently from map and reduce tasks. The introduction of iShuffle to the Hadoop environment presents two challenges: user transparency and fault tolerance.

Besides user-defined *map* and *reduce* functions, Hadoop allows customized *partitioner* and *combiner*. To ensure that iShuffle is user-transparent and does not require any change to the existing MapReduce jobs, we design the *Shuffler* as an independent component in the `TaskTracker`. It takes input from the combiner, the last user-defined component in map tasks, performs data shuffling and provides input data for reduce tasks. The shuffler performs data shuffling every time the output data is written to local disks by map tasks, thus we name the operation *shuffle-on-write*.

Fig. 4 shows the workflow of the Shuffler. It has three stages: (1) map output collection (step ①②); (2) data shuffling (step ③④⑤⑥); (3) map output merging (step ⑦⑧).

Map Output Collection. The shuffler contains multiple `DataSpillHandler`, one per map task, to collect map output that has been written to local disks. Map tasks write the stored partitions to the local file system when a spill of the in-memory buffer occurs. We intercept the writer class `IFile.Writer` used in the spill thread and add a `DataSpillHandler` class to it. While the default writer writing a spill to local disk, the `DataSpillHandler` copies the spill to a circular buffer, `DataSpillQueue`, from where data is shuffled/dispatched to different nodes in Hadoop. A map output has two paths for writing output. One path is writing output directly, the other path is writing through combiner, which combines key-value pairs before writing them to disk. Intercepting the output in the spill thread enables iShuffle to handle the output for both paths. During output collection, the `DataSizePredictor` monitors input data sizes and resulted partition sizes, and reports these statistics to the shuffle manager.

Data Shuffling. The shuffler proactively pushes data partitions to nodes where reduce tasks will be launched. Specifically, a `DataDispatcher` reads a partition from the `DataSpillQueue` and queries the shuffle manager for its destination. Based on the placement decision, a partition can be dispatched to the shuffler on a different node or to the `local merger` in the same shuffler.

Map Output Merging. The map output data shuffled at different times needs to be merged to a single reduce input file and sorted by key before a reduce task can use it. The `local merger` receives remotely and locally shuffled data and merges the partitions belonging to the same reduce task into one reduce input. To ensure correctness, the `merger` only merges partitions from successfully finished map tasks.

3.3 Balanced Partition Placement

The shuffle-on-write workflow relies on key information about the partition placement for each running job. The objective of partition placement is to balance the distribution of map output data across different nodes, so that the reduce workloads on different nodes are even. The optimal partition placement can be determined when the sizes of all partitions are known. However, this requires that all map tasks are finished when making the placement decisions, which effectively enforce a serialization between map tasks and the shuffle phase. iShuffle estimates the final partition sizes based on the amount of processed input data and current partition size, and uses the estimation to guide partition placement.

3.3.1 Prediction of Partition Sizes

The size of a map output partition depends on the size of its input dataset, the map function, and the partitioner. The final size of each partition can be predicted with decent accuracy during the early stage of job execution [28]. Verma et al. [32], found that the ratio of map output size and input size, also known as *map selectivity*, is invariant given the same job configuration. As such, the partition size can be determined using the metric of map selectivity and input data size. The shuffle manager monitors the execution of individual map tasks and estimates the map selectivity of a job by building a mathematical model between input and output sizes.

For a given job, the input dataset is divided into a number of logical splits, one per map task. Since individual map tasks run the same map function, each map task shares the same map selectivity with the overall job execution. By observing the execution of map tasks, where a number of input/output size pairs are collected, shuffle manager builds a model estimating the map selectivity metric. Shuffle manager makes k observations of the size of each map output partition. As suggested in [32], it derives a linear model between partition size and input data size: $p_{i,j} = a_j + b_j \cdot D_i$, where $p_{i,j}$ is the jth partition size in the ith observation and D_i is the corresponding input size. We use linear regression to obtain the parameters for m partitions, one per reduce task. In practice, we measure the partition size and the input size from the beginning of the job and keep updating them every one second. Since MapReduce jobs contain many more map tasks than reduce tasks (as shown in Table 2, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2016.2587645), we are able to collect sufficient samples for building the model. Once a model is obtained, the final size of a map output partition is calculated by replacing D_i with the actual input size of the map task.

3.3.2 Partition Placement

With predicted partition sizes, the shuffle manager determines the optimal partition placement that balances reduce workload on different nodes. Because the execution time of a reduce task is linear to its input size, evenly placing the partitions leads to balanced workload. Formally, the partition placement problem can be formulated as: given m map output partitions with sizes of $p_1, p_2, \ldots, p_m$, find the placement on n nodes, $S_1, S_2, \ldots, S_n$, that minimizes the placement difference

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\mu - \sum_{j \in S_i} p_j\right)}, \tag{1}$$

where μ is the average data size on one node.

Algorithm 1. Partition Placement

```
1: Variables: List of partitions p, list of nodes, has the size of
   all allocated partitions S.
2:
3: sort list p in descending order of partition sizes
4: for i ← 1 to m do
5:    min_node ← S[1]
6:    for j ← 1 to n do
7:       if S[j].size < min_node.size then
8:          min_node ← S[j]
9:       end if
10:   end for
11:   min_node.place(p[i])
12: end for
```

Partition placement problem can be viewed as the load balancing problem in multiprocessor systems [17] and is thus NP-hard. While the optimal solution can be prohibitively expensive to attain, we propose a heuristic-based approach to approximate an optimal placement. Detail of this approach is presented in Algorithm 1. This algorithm is based on two heuristics, the *largest partition first* for picking partitions and the *less workload first* for picking destination nodes. It sorts the partitions in the descending order of size and assigns the largest partition to the nodes with the least aggregate data size. It repeats until all the partitions are assigned.

3.4 Flexible Reduce Dispatching

In Hadoop, reduce tasks are assigned map output partitions statically during job initialization. When there are reduce slots available on idle nodes, reduce tasks are dispatched according to the ascending order of their task IDs. This restriction on reduce task dispatching leads to inefficient execution where reduces that are waiting for map tasks to finish occupy the slots for a long time. Because iShuffle proactively pushes output partitions to nodes, it requires that reduce tasks are launched on nodes that hold the corresponding shuffled partitions. To this end, iShuffle breaks the binding of reduce tasks and map output partitions and provides flexible reduce dispatching.

An intuitive approach for flexible reduce dispatching is to traverse the task queue and find a reduce that has shuffled data on the requesting node. However, this approach does not guarantee that there is always a "local" reduce available for dispatching. iShuffle employs a different approach that assigns partitions to reduce tasks at the time of dispatching. For single-user clusters, we modified Hadoop's FIFO scheduler to support the runtime task-partition binding. When a node with available reduce slots requests for new reduce tasks, the `task scheduler` first check with the shuffle manager to obtain the list of partitions that reside on this node. The scheduler picks the first partition in the list and associates its ID with the first reduce task in the waiting queue. The selected reduce task is then launched on the node. As such, all reduce tasks are guaranteed to have local access to their input data.

For multi-user clusters with heterogeneous workloads, we add the support for runtime task-partition association to the Hadoop Fair Scheduler (HFS). The minimum fair share allocated to individual users can negatively affect the efficiency of iShuffle as reduce tasks may be launched on remote nodes to enforce fairness. We disable such fairness enforcement for reduce tasks to support more flexible task dispatching. This allows some users to temporarily run more reduce tasks than others. We rely on the following designs to reduce unfairness among users and avoid starvation. First, the fair share of map tasks is still in effect, guaranteeing fair chances for users to generate map output partitions. Second, while records are sorted by key within each partition after shuffling, partitions belonging to different users are placed in the list with random order, giving each user an equal opportunity to launch reduce tasks. Finally and most importantly, reduce tasks are started only when all their input data is available. This may temporarily violates fairness, but prevents wasted cluster cycles spent in waiting for unfinished maps and results in more efficient job execution. We further employed the preemptive reduce scheduling to minimized the fairness impact of disabling fair share for reduce tasks.

3.5 Preemptive Reduce Scheduling

The random ordering of partitions in the reduce phase improves the fairness between launched MapReduce jobs. But it does not effectively prevent small jobs from being starved by long-running large jobs. Since the execution time of small jobs are short, letting reduce tasks of small jobs preempt long-running jobs does not being significant overhead. This would ensure the timely scheduling of small jobs. Preemption requires temporary suspension of a running task and resumes it at a later time. Modern operating systems support preemptive task scheduling by providing a way to save the state of a running task so that it can be resumed from where it is preempted. However, it is difficult to provide task preemption in MapReduce. There is no checkpoint or snapshot feature in popular frameworks such as Hadoop. Further, the coupling of the *shuffle*and *reduce* phases makes it difficult to implement preemption.

Algorithm 2. Preemptee Selection

1: **Variables:** Jobs that have more reduce running than their fair shares L_{over}; Jobs that running reduces at their fair shares L_{fair}; r_k processing rate of task k; w_j number of task waves of job j; n_j remaining tasks of job j; s_j number of task slots of job j;
2:
3: update L_{over} and L_{fair}.
4: **if** L_{over} is not empty **then**
5: GetPreemptee(L_{over})
6: **else**
7: GetPreemptee(L_{fair})
8: **end if**
9:
10: **function** GetPreemptee (L)
11: **for** each job j in L **do**
12: **for** each task k in j **do**
13: $t_k = \frac{D_{remain,k}}{r_k}$
14: **end for**
15: $candidate = \text{argmax}_k(t_k)$
16: **if** $\left\lceil \frac{n_j}{s_j} \right\rceil = \left\lceil \frac{n_j+1}{s_j-1} \right\rceil$ **then**
17: return $candidate$
18: **end if**
19: **end for**
20: **end function**

The decoupling of shuffle and reduce in iShuffle provides an easy way to preempt reduce tasks. Decoupled from the shuffle phase, reduce tasks simply read pushed intermediate data from local disks and output to local HDFS. Thus, the preemption of reduce tasks does not need to save the state of the complex shuffle phase. iShuffle first implements a checkpoint-on-preemption mechanism to save the state of reduce tasks and modifies the Hadoop Fair Scheduler to select a proper candidate of the preemption.

Checkpoint-on-Preemption. Since we only need to save reduce state upon the preemption, tracking task execution and making checkpoints throughout the life time of a reduce task is unnecessary. To avoid tracking on-the-fly reduce state, we only save the state that has been committed to persistent storage from preempted reduce tasks. Each time a output buffer is full, the reduce task flushes the current key and value list to HDFS. When iShuffle receives a preemption request, it first flushes the output buffer to HDFS. Then it saves the file offset of the record that is being processed in the checkpoint. Next, iShuffle suspends the reduce task and finishes the preemption. The preempted reduce is later restarted from the checkpoint. The process is resumed by seeking to the offset of last unprocessed record. It does not need to scan or reprocess the records that have already been committed to the HDFS.

Preemptee Selection. To enforce fairness among multiple jobs, MapReduce scheduler needs to select preemptees from the jobs that are running more tasks than their fair shares. The selection of preemptees has significant impact on overall job performance. Some existing systems select the task with the longest remaining time or largest remaining data as the preemptee to reduce the impact on the job completion time [34]. But the tasks of a job often finishes in multiple waves in a multi-user Hadoop cluster, which complicates the potential performance impact of preemption. When the tasks of the last wave filled all execution slots, the preemption of a task may create an additional wave, which can significantly prolong job completion. On the other hand, if the last wave has unfilled execution slots, the preempted task can be combined with the last wave of tasks and avoid the overhead of the additional wave.

iShuffle exploit the unfilled execution slots in the last wave and reduce the performance impact of preemptive reduce scheduling. It carefully selects preemptees (tasks) that would not cause extra waves. Algorithm 2 shows the details of how iShuffle selects the preemptee. All running jobs are divided into two categories. The *over* category contains jobs that run more tasks than their fair shares. The rest of jobs are put in the *fair* category. Whenever the scheduler needs to select a preemptee, it first calculates the fair share and updates the list of *over* and *fair* jobs. If the list of *over* jobs is not empty, the algorithm will invoke the `GetPreemptee` function to obtain a task from these jobs and use it as the preemptee. When there are no *over* jobs, the algorithm will choose the preemptee from *fair* jobs using the same function. The `GetPreemptee` function works in two steps. First, it estimates the remaining time of all tasks for each job using the task progress rate and the size of remaining input data. The task with the longest remaining time is picked as the preemptee candidate. Then, it calculates the number of tasks waves after the preemption. If the number of task waves of a job does not increase after preemption, the algorithm will choose this job and use its preemptee candidate as the final preemptee.

3.6 Design Trade-Offs of iShuffle

Fault Tolerance. Since iShuffle alters the workflow of Hadoop, it may affect fault tolerance mechanisms in Hadoop. Hadoop employs a detect-restart fault tolerance model that re-launches failed tasks on different nodes. The failed task is rerun from the beginning, and the data generated by the failed run is discarded. For map task, this is easy as the rescheduled task only needs to read input from HDFS. For reduce tasks, the shuffle phase in the reduce task will re-fetch the map output from all nodes and merge them as reduce input.

In iShuffle, the recovery workflow for map tasks remains the same while reduce recovery needs some changes. The

reduce task can read the input file from local disk, if it is scheduled on the same node as the failed task. When the reduce task is scheduled on a different node, the shuffler need to copy the shuffled data from a remote node (if the remote copy is still available), or redo the shuffle (if the remote copy is gone due to a node failure). This complexity of reduce recovery workflow helps reduce the data movement when the task is restarted on the same node because it does not need reshuffle. In the worst case where reshuffling is required, iShuffle does not incur overhead compared to conventional Hadoop.

Speculative Execution. Speculative execution is introduced in the original design of MapReduce to address stragglers, which are usually due to faulty hardware, misconfigurations, and resource contentions. Running a backup task on a different node is likely to complete the task faster, thereby mitigating the overall job slowdown due to these stragglers.

Since there may be multiple copies of the same tasks running simultaneously, Hadoop keeps the output of each task separately. As iShuffle shuffles and merges map output as soon as it is generated, we need sophisticated output deduplication to support speculative execution. Currently, iShuffle does not support speculative execution.

Nevertheless, in a multi-tenant MapReduce cluster, the scheduler is often optimized for multiple factors, such as fairness between users and cluster utilization. In this case, a job may not have any available slot for backup tasks. Moreover, speculative execution can be triggered by temporal performance fluctuation, especially in virtualized clusters. Thus, the common practice is to disable speculative execution [4]. Not supporting speculative execution does not poses limitation to iShuffle's usability in multi-tenant clusters.

Map Overhead. Decoupling shuffle and reduce, and using *Shuffle-on-Write* technique essentially moves the shuffle operation to the map phase. It is expected to have some performance impact on map tasks. However, the shuffler is mainly performing I/O operations, and should not interfere with map tasks substantially. We present overhead analysis on map tasks in Section 4.6.

4 Evaluation

The performance evaluation of iShuffle is done in two parts. First, we use a set of representative MapReduce jobs to study the reduction in shuffle delay, improvement of performance. Then, we use the workload trace from Facebook to evaluate the performance of iShuffle on real-world workloads. We also evaluate the performance impact of the automated partition placement with different balancing approaches. We further explore the effectiveness of the flexible reduce dispatching in shared Hadoop environment.

4.1 Testbed Setup

Our testbed was a 32-node Hadoop cluster. Each node had one 2.4 GHz 4-core Intel Xeon E5530 processor and 4 GB memory. All nodes were interconnected by a Gigabit Ethernet. We deployed Hadoop v1.1.1 on Ubuntu Linux with kernel 2.6.24. Two nodes were configured as the `JobTracker` and `NameNode`, respectively. The rest 30 nodes were configured as slave nodes. We set the HDFS block size to its default value 64 MB. Each slave node was configured with four map slots and four reduce slots, resulting in a total capacity of running 120 map and 120 reduce tasks simultaneously in the cluster.

We compare iShuffle with three different approaches. The first approach is Hadoop-A [33]. It enables reduce tasks to access map output files on remote disks through the network. By using a priority queue-based merge sort algorithm, Hadoop-A eliminates repetitive merge and disk accesses, and removes the serialization between the shuffle and reduce phases. However, Hadoop-A requires the remote direct memory access (RDMA) feature on Infiniband interconnections for fast remote disk access. We implemented Hadoop-A using remote procedure calls on our testbed with Gigabit Ethernet and compared its performance with iShuffle on commodity hardware.

The second approach is DynMR [29]. It replaces the map and reduce task execution slots with a unified task execution slots. During the job execution, map tasks will occupy all available execution slots, and reduce tasks only preempt map tasks when there is enough data to shuffle. The interleaved execution of map and reduce tasks allows the map output data to be shuffled as soon as possible and reduces the shuffle delay. However, DynMR does not decouple shuffle and reduce, the shuffle cannot be fully overlapped with map tasks. With the unified task execution slots, a reduce task can also preempt the map task of a map only job. This can result in significant performance interference in multi-user Hadoop clusters.

The third approach is Sailfish [25]. It proposed I-File, a data aggregation system for intermediate data. It is implemented on top of Kosmos File System. By saving map output in I-File, Sailfish is also able to overlap the shuffle phase and map phase. In order to mitigate the data skew, Sailfish repartitions the intermediate data and dynamically determined the number of reduce tasks. However, Sailfish lacks the support of preemptive reduce scheduling. In a multi-tenant environment, the long running reduce tasks for large jobs may starve the small jobs, and Sailfish has not way to mitigate it.

4.2 Workloads

We mainly used the Purdue MapReduce Benchmark Suite [3] to compose workloads for evaluation. In order to cover the area PUMA benchmark suite does not cover some new but popular applications such graph processing and machine learning. Thus we also include the *pagerank* and *bayes* benchmark jobs from the HiBench benchmark suite [2] to provide a more comprehensive collection of benchmark jobs.

The PUMA and HiBench benchmark jobs can be divided into two categories: shuffle-heavy and shuffle-light. Shuffle-heavy benchmarks are *self-join*, *tera-sort*, *k-means*, *inverted-index*, *term-vector*, *wordcount*, *pagerank*, and *bayes*. They have have high map selectivity and generate a large volume of data to be exchanged between map and reduce. Their intermediate data size ranges from 32 GB to 300 GB. Thus, such benchmarks are sensitive to optimizations on the shuffle phase. For shuffle-light benchmarks, such as *histogram-movies*, *histogram-ratings*, and *grep*, there is little data to be shuffled. We used both benchmark types to evaluate the effectiveness of iShuffle and its overhead on workloads with little communications.

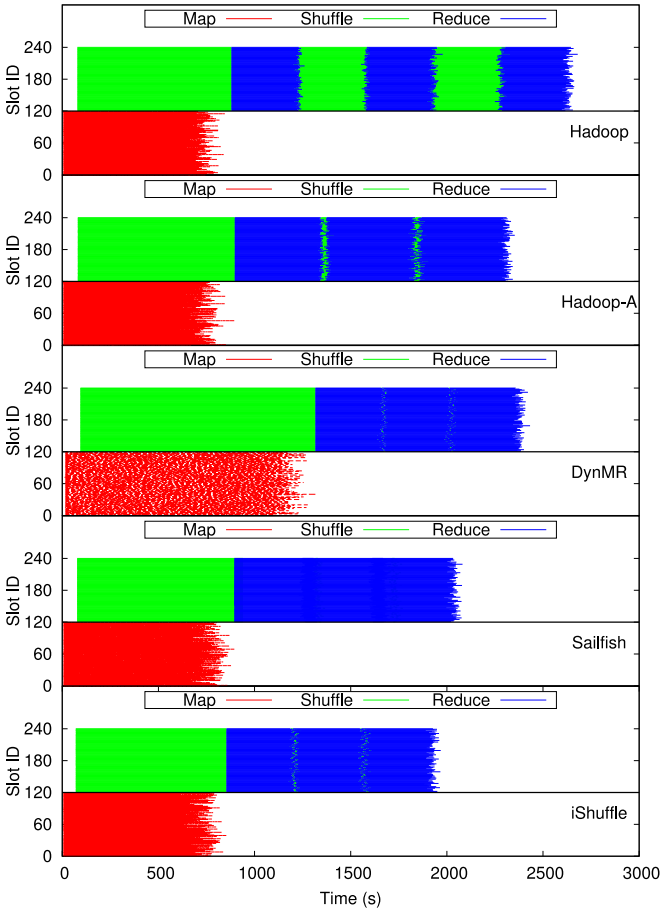


Fig. 5. Execution trace of *tera-sort* using stock Hadoop, Hadoop-A, DynMR, Sailfish and iShuffle approaches.

For experiments with multi-user environment, we used the Statistical Workload Injector for MapReduce (SWIM) [5] to replay the trace from Facebook [10]. We have include the details of the benchmark jobs in Appendix A, available in the online supplemental material.

4.3 Reducing Shuffle Delay

Recall that we defined shuffle delay as the duration between the last wave of execution and the next reduce wave. Shuffle delay measures the shuffle period that cannot be overlapped with the previous wave. The smaller the shuffle delay, the more efficient the shuffling scheme. We ran *tera-sort* on stock Hadoop, Hadoop-A, DynMR, Sailfish and iShuffle, and recorded the start and completion times of each map, shuffle and reduce phase.

Fig. 5 shows the trace of the *tera-sort* job execution under different approaches. The X-axis is the time span of job execution and Y-axis represents the map and reduce slots. The results show that iShuffle had the best performance with 29.6, 20.8, and 22.4 percent shorter job execution time than stock Hadoop, Hadoop-A and DynMR, respectively. As shown in Fig. 5, there is a significant delay of the reduce phase for every reduce task in stock Hadoop. Due to proactive placement of map output partitions, iShuffle had almost no shuffle delays. Note that Hadoop-A also significantly reduced shuffle delay because it operates on globally sorted partitions and can greatly overlap the shuffle and reduce phase.

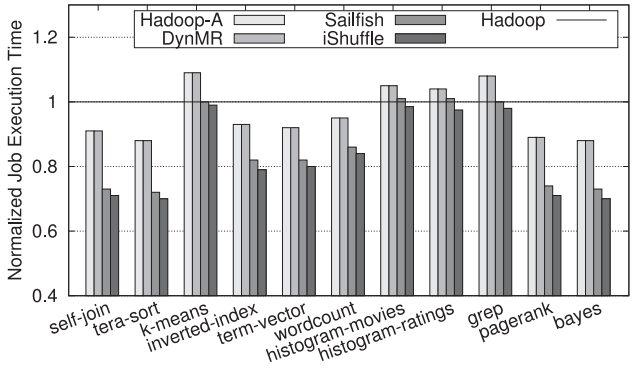


Fig. 6. Job performance using four different approaches.

iShuffle outperformed Hadoop-A on our testbed for two reasons. First, the building of the priority queue poses extra delay, e.g., the shuffle delay before the second and third reduce waves in Hadoop-A, to each reduce task. Second, the remote disk access in an Ethernet environment is significant slower than that in an Infiniband network, which leads to much longer reduce phases in Hadoop-A.

iShuffle also significantly outperformed DynMR. DynMR does not decouple shuffle and reduce, which still requires reduce tasks to be scheduled to perform shuffle. In order to reduce the shuffle delay, DynMR used interleaved execution between map and reduce tasks. The reduce tasks can preempt the map tasks, so that the intermediate data can be shuffled as soon as they are generated. But it also prolongs the completion time of all map tasks. In iShuffle, the independent shuffler can work with the need of scheduling reduce tasks or interrupt map tasks. It is able to fully overlap shuffle with map tasks.

The performance of iShuffle and Sailfish are close in this experiment. Both approaches overlapped the shuffle phase with the map phase. However, due to the completion time of the map phase in Sailfish is slightly longer than it in iShuffle, which results increases the job completion time in Sailfish. We discuss the overhead to map tasks later in Section 4.6.

4.4 Reducing Job Completion Time

We study the effectiveness of iShuffle in reducing overall job completion time with more comprehensive benchmarks. We use the job completion time in stock Hadoop implementation as the baseline and compare the normalized performance of iShuffle, Sailfish, DynMR and Hadoop-A. Fig. 6 shows the normalized job completion time of all shuffle-heavy benchmarks, which are *self-join*, *tera-sort*, *pagerank*, and *bayes*. iShuffle outperformed the stock Hadoop by 29.1-29.6 percent in these four benchmarks iShuffle outperformed DynMR by 15.5-15.7 percent. iShuffle outperformed Hadoop-A by 21.9-22.7 percent in these benchmarks. The performance of iShuffle in these four benchmarks is similar to the case in other shuffle-heavy benchmarks. As we expected, iShuffle and Sailfish performed similarly. The iShuffle achieved only 3-4 percent shorter job completion time than Sailfish.

Benchmarks like *inverted-index*, *term-vector*, and *word-count* also fit in the shuffle-heavy category, but the shuffle volumes are smaller than other shuffle-heavy benchmarks. These benchmarks had less shuffle delay than other shuffle-heavy benchmarks simply because there was less data to be

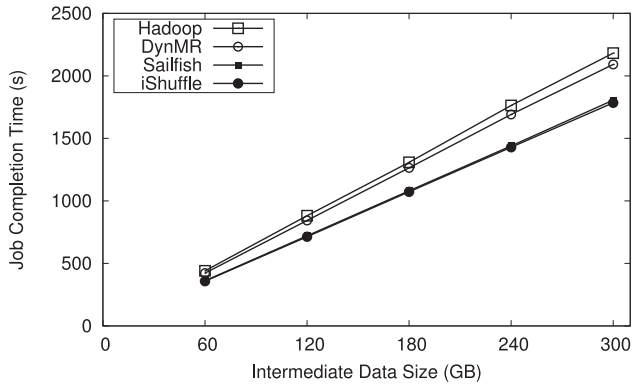


Fig. 7. Job completion time with different intermediate data size.

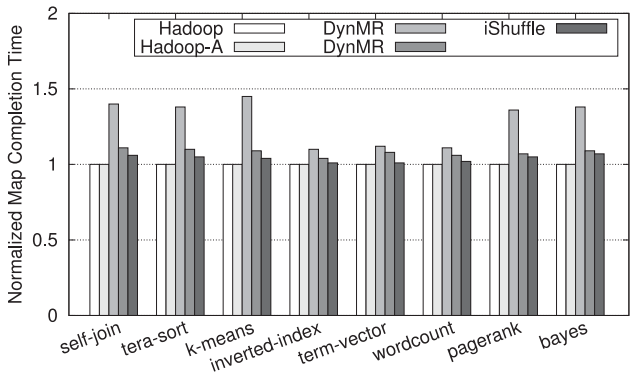


Fig. 9. Map completion time due to four different approaches.

copied during the shuffle phase. Therefore, the performance improvement due to iShuffle was less. iShuffle achieved 15.6-20.3 percent better performance than stock Hadoop with these benchmarks, respectively. For these benchmarks, the delay of map tasks in DynMR is significantly reduced due to the small shuffle volume. Thus DynMR achieved similar performance as iShuffle did. However, shuffle introduced additional delay to the map phase. The DynMR has 4.8-6.3 percent longer job completion time than iShuffle in these three benchmarks, respectively. For these benchmarks, Hadoop-A still gained some performance improvement over stock Hadoop as the reduction on shuffle delay outweighed the prolonged reduce phase. However, the performance gain was marginal with 5.5-8.6 percent improvement, respectively.

For the shuffle-light benchmarks, the shuffle delay is negligible. iShuffle, DynMR, and Hadoop-A achieved almost no improvement over the stock Hadoop.

One special case of the experiments is *k-means* benchmark, which only has six reduce tasks (one reduce wave). With only one wave of reduce tasks, stock Hadoop is already good enough at overlapping the shuffle phase with map tasks, thus it had similar performance as iShuffle and Sailfish. However, due to the additional delay of remote disk access, Hadoop-A had longer reduces, thus longer overall completion time. DynMR also has longer completion time due the interleaved execution of map and reduce tasks, which prolongs the completion time of map phase.

To understand the performance of these approaches under different intermediate data size, we also measured the job completion time of *wordcount* with intermediate data size ranging from 60 to 300 GB. Fig. 7 shows that the job completion time of all four approaches scales linearly with the intermediate data size. While the job completion time of both iShuffle and Sailfish are consistently less than Hadoop-A and DynMR.

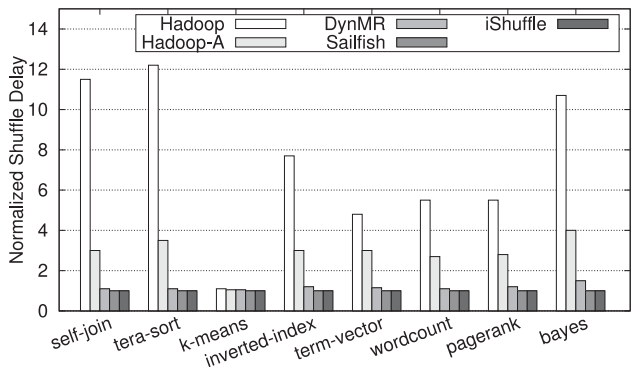


Fig. 8. Shuffle delay due to four different approaches.

4.5 Reducing Shuffle Delay

The reduction on job completion time is the result of interplay of different execution stages in a MapReduce job. To understand how shuffle delay affects the job performance, we compare the shuffle delay between these approaches. Fig. 8 shows the comparison of normalized shuffle delay. We used the shuffle delay of iShuffle as the base line. The results agree with the observation we made in previous experiments. iShuffle was able to reduce the shuffle delay significantly if the job had large volumes of shuffled data and multiple reduce waves. DynMR is able to achieve a shuffle delay that is similar to iShuffle. For benchmarks that have the largest shuffle-volume, the reductions in shuffle delay were more than 10× compared with stock Hadoop. For benchmarks with medium shuffle volume, the improvement on shuffle delay was from 4.5× to 5.5×. Fig. 8 also suggests that iShuffle was on average 2× more effective in reducing shuffle delay than Hadoop-A. iShuffle and Sailfish are equally effective on reducing shuffle delay in these benchmarks.

4.6 Overhead in Map Phase

Although iShuffle, Sailfish and DynMR achieved significant reduction in shuffle delay, they had quite different job completion time. We conclude that the difference is due to the changes in the map completion time. As Fig. 9 shows iShuffle has around 3 percent overhead in map phase due to the independent shuffler. Although the overhead seems to be high, it only diminishes the performance gain due to reduced shuffle delay. Sailfish has 4-6 percent longer map completion time than iShuffle mainly due to the overhead introduced in writing map output to a I-File. Sailfish directly appends each record to its corresponding data chunk in a distributed file system results in small writes to multiple data chunks on different servers. It also need to update the metadata server with the updates. These operations increase the cost of writing map output. While in iShuffle, the map output is shuffled to the nodes for future reduce tasks. Each reduce input contains all the keys associated to one reduce task, thus iShuffle does not have the

overhead of tracking the key distribution across on multiple chunk servers. Eventually, this brings less overhead to map phase in iShuffle. DynMR has 31.4-39.4 percent longer map completion time than iShuffle in all shuffle-heavy benchmarks. The reason is that DynMR still needs to schedule reduce tasks for shuffle, and the interleaved execution of map and reduce tasks cannot fully overlap the shuffle I/O with map computation. On the contrary, the map computation needs to be paused for shuffle I/O. This results in significant increase in map completion time and eventually prolongs the job completion time.

4.7 Multi-User Performance with Facebook Trace

We proposed the preemptive reduce scheduling to improve scheduling fairness and the job performance for multi-user Hadoop clusters. First, we show the performance improvement due to the preemptive reduce scheduling. We used the Statistical Workload Injector for MapReduce [5] to replay the trace from Facebook [10] to study the performance of iShuffle under real-world workloads.

Fig. 12 shows the normalized job completion time of all ten types of jobs in the Facebook trace. The results show that the job with high popularity in the trace, such as Job-1 and Job-6, benefit most from using the preemptive reduce scheduling. For this type of jobs, iShuffle using the preemptive reduce scheduling achieved up to 16 percent shorter job completion time than using the flexible reduce dispatching alone. Note that Job-1 has a significant increase in job completion time when iShuffle is using the flexible reduce scheduling. This is due to the fact that Job-1 is small in size and a shuffle-light job usually takes less than one minute to finish. Thus it barely benefits from the reduction of shuffle delay, but can be significantly affected by scheduling unfairness because large jobs monopolize the task slots. On the contrary, the preemptive reduce scheduling allows small jobs like Job-1 to preempt large jobs so that they can achieve expected performance.

Using the preemptive reduce scheduling also improves the performance of shuffle-heavy jobs. For Job-6 to Job-9, iShuffle with preemptive reduce scheduling achieved up to 14.2 percent shorter job completion time than using the flexible reduce dispatching. Note that Job-10 is a low popularity (1 in 24,442) long running job. Temporary unfairness in reduce scheduling does not quite affect its completion time. Also due to its small shuffle volume, it does not have a long shuffle delay, which does not benefit from iShuffle.

Second, we compare the performance of iShuffle, Sailfish, DynMR, Hadoop-A, and stock Hadoop on the Facebook workload. Fig. 13 shows the normalized job completion time of all four approaches. For Job-6 to Job-9, iShuffle outperformed, Sailfish, DynMR, and Hadoop-A by 5, 9.5, and 18.4 percent, respectively. The results with these shuffle-heavy jobs are consistent as previous experiments. For Job-1 to Job-5, iShuffle only has less than 3 percent overhead in Job-1 to Job-5, while Sailfish has up to 15 percent overhead in these jobs. Job-1 to Job-5 are high popularity small jobs. Sailfish's performance on these jobs are as the results of starvation of these small jobs. Although Sailfish can determine the ideal number of reduce tasks for a single job in an automated fashion, is does not consider the fairness in resource allocation in multi-tenant clusters. The number of reduce tasks spawned by Sailfish for one large job can easily exceeds its fair-share of reduce slots. The results is small jobs are starved because there is no available reduce slot. Moreover, Sailfish also lacks the abilities, such as preemptive reduce scheduling, that can timely yield the resource from large jobs to small jobs. It further aggravates performance impact of starvation.

iShuffle also outperformed DynMR in these jobs. The overhead comes from the prolonged completion time of the map phase. Recall that DynMR requires map tasks to be paused during shuffle. In multi-user clusters, this not only affects the performance of regular jobs, but also hurts the performance of map-only jobs. Map-only jobs do not have any shuffle delay that can be reduced to offset the increment in the map completion time due to DynMR. They will have performance degradation as long as they share the same cluster with regular jobs. iShuffle overlaps shuffle I/O with map computation. No map task needs to be paused in order to do shuffling. Therefore, it improves the performance of regular jobs without interfering map-only jobs.

We further compare the performance impact due to different preemptee selection algorithms. We compare iShuffle with the remaining time/data based preemptee selection algorithm proposed in [34]. It picks the task with the largest remaining data and the longest remaining execution time. It also monitors the predicted completion time of the jobs to avoid repetitively preempting tasks from the same job. Fig. 14 shows the performance of different preemptee selection algorithms. The results show that iShuffle achieved 3-22 percent shorter job completion time than the remaining time/data based preemptee selection algorithm did. Since the remaining time/data based algorithm does not consider the unfilled slots in the last wave of tasks, it missed the opportunity to preempt a reduce task with virtually no overhead to overall job performance.

4.8 Balanced Partition Placement

We have shown that iShuffle effectively hides shuffle latency by overlapping map tasks and data shuffling. In this section, we study how the balanced partition placement affects job performance. To isolate the effect of partition placement, we first ran benchmarks under stock Hadoop and recorded dispatching history of reduce tasks. Then, we configured iShuffle to place partitions on nodes in a way that leads to the same reduce execution sequence. As such, job execution enjoys overlapped shuffle provided by iShuffle, but bears the same partitioning skew in stock Hadoop. We compare the performance with balanced partition placement and stock Hadoop.

Fig. 10 shows the performance improvement due to balanced partition placement. The results show that iShuffle achieved 8-12 percent performance improvement over stock Hadoop. We attribute the performance gain to the prediction-based partition placement that mitigates the partitioning skew. It prevents straggler tasks from prolonging job execution time. The partition placement in iShuffle relies on accurate predictions of the individual partition sizes. Fig. 11 shows the differences between measured partition sizes and the predicted ones. The results suggest that for all the shuffle-heavy

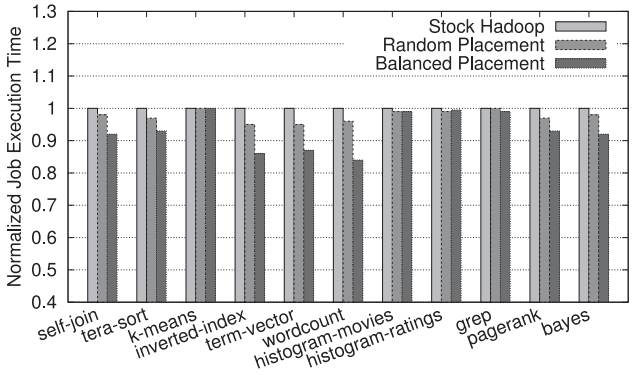


Fig. 10. Performance of automated map output placement.

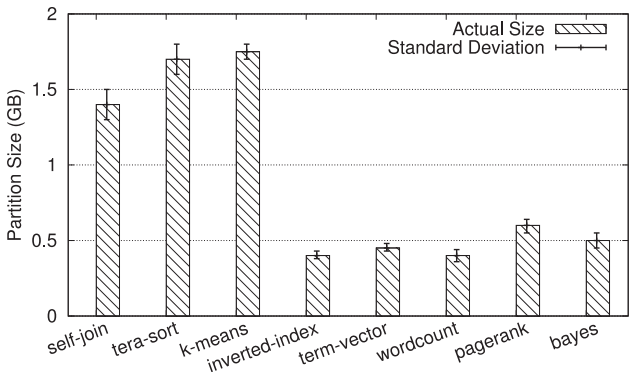


Fig. 11. Accuracy of iShuffle partition size prediction.

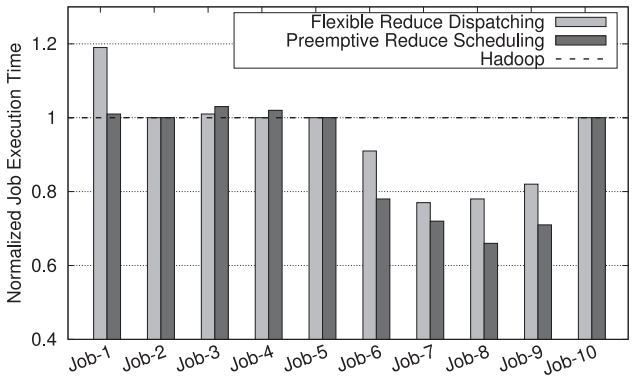


Fig. 12. Multi-user performance using flexible reduce dispatching and preemptive reduce scheduling.

benchmarks, iShuffle was able to estimate the final partition size with no more than 2 percent prediction errors.

4.9 Flexible Reduce Dispatching

We further evaluate the flexible reduce dispatching of iShuffle in a multi-user Hadoop environment. We created multiple workload mixes, each contained two different MapReduce jobs. We ran one workload at a time with two jobs sharing the Hadoop cluster. We modified the Hadoop Fair Scheduler (i.e., *iShuffle w/HFS_mod*) to support runtime task-partition binding. For comparison, we also study the performance of iShuffle with the original HFS that enforces a minimum fair share on reduce tasks (i.e., *iShuffle w/HFS*) and iShuffle running a single job on a dedicated cluster (i.e., *Separate iShuffle*).

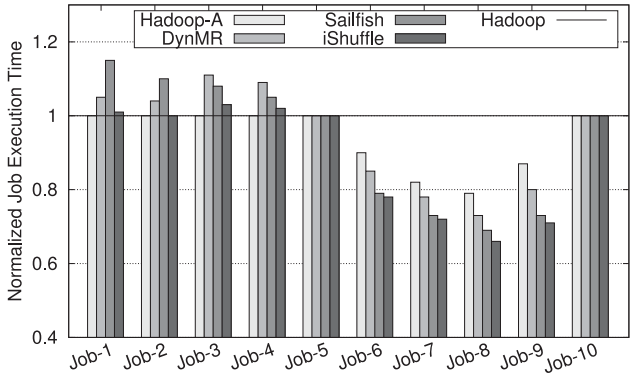


Fig. 13. Job performance due to different approaches on a multi-user Hadoop cluster.

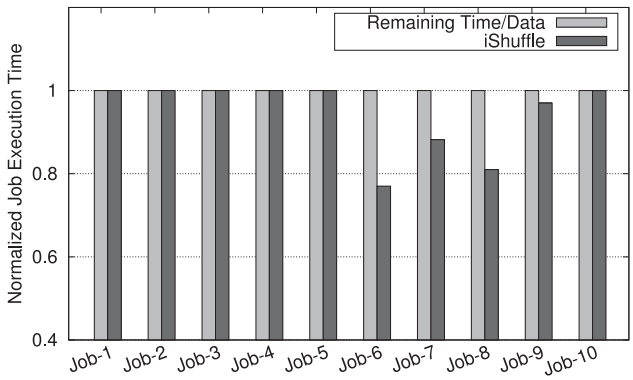


Fig. 14. Job completion time due to different preemptee selection algorithms.

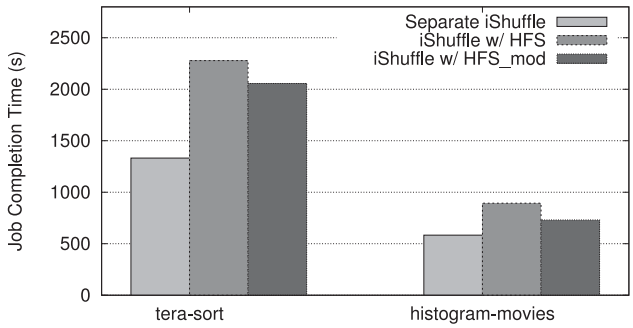


Fig. 15. Performance of job mix of *tera-sort* and *histogram-movies*.

The first experiment used the combination of a shuffle-heavy job and a shuffle-light job. Fig. 15 shows the result of workload mix of *tera-sort* and *histogram-movies*. The results suggest that the modified HFS outperformed the original HFS by 16 and 25 percent for *tera-sort* and *histogram-movies*, respectively. Unlike the original HFS, which guarantees *max-min* fairness to jobs, iShuffle allows the reduce of one job to use more reduce slots. iShuffle prioritizes shuffle-light jobs because the execution time of their reduce tasks is short. Allowing shuffle-light jobs to run with more slots boosted their performance significantly. Although shuffle-heavy jobs suffered unfairness to a certain degree, their overall performance under the modified HFS was still better than that under the original HFS.

Next, we perform the experiment with two shuffle-heavy jobs. Fig. 16 shows the performance of *tera-sort* and *inverted-index*. It shows that iShuffle improved job execution times by 8 and 23 percent over the original HFS in these two

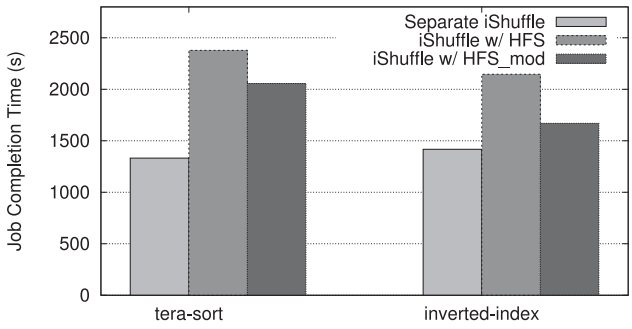


Fig. 16. Perf. of job mix of *tera-sort* and *inverted-index*.

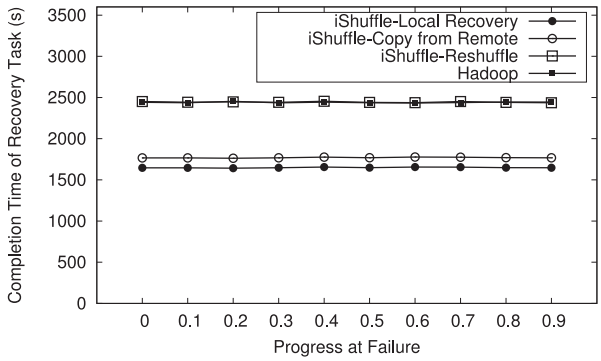


Fig. 17. Recovery task completion time for reduce tasks.

benchmarks. Although the size of input datasets of these two benchmarks are similar, *inverted-index* has a smaller shuffle volume. Therefore, its reduce tasks can be started earlier as their partitions required less time to shuffle. *tera-sort* had less improvement in this scenario because some of its reduce tasks are delayed by *inverted-index*. Table 1 shows more results of iShuffle with heterogeneous workloads compared with stock Hadoop. For most workload mixes with two jobs, iShuffle w/modified HFS was able to reduce the job completion time for both jobs. The performance gain depends on the amount of shuffled data in these co-running jobs.

However, iShuffle had poor performance with workload mix *tera-sort* + *k-means*. We ran *tera-sort* with a 300 GB dataset and *k-means* with a 15 GB dataset. The result of *k-means* does not agree with previous observations for shuffle-light workloads. The co-running of *tera-sort* and *k-means* significantly degraded the performance of *tera-sort*. An examination of the execution trace revealed that although *k-means* has little data to exchange between map and reduce, it is compute intensive. iShuffle started *k-means* earlier than *tera-sort* and *k-means* occupied the reduce slots for a long time delaying the execution of *tera-sort*. The culprit was that for *k-means*, the partition size is not a good indicator of the execution time of its reduce tasks. Thus, iShuffle failed to balance the reduce workload on multiple nodes. A possible solution is to detect such outliers earlier and restart them on different nodes. Since such outliers often have small shuffle volume, the migration is not expensive.

4.10 Performance of Recovery Reduce Tasks

We manually injected a failure at different progress of a reduce task and measured the recovery time of this reduce. Fig. 17 shows the recovery time of Hadoop and iShuffle. Note that the input of a reduce task can be on the local disk, on a remote node, or unavailable. When the reduce input is available on either the local disk or a remote node, the recovery time of a reduce task in iShuffle is significantly shorter than that in Hadoop. In the case of the reduce input is on a remote node, the recovery time also includes the time to copy data from a remote node to the local disk. When the reduce input is not available, iShuffle needs to perform reshuffling to create the missing reduce input, which falls back to the same recovery model as Hadoop.

TABLE 1
Job Completion Time of Co-Running Jobs

Workload Mix			Stock Hadoop		iShuffle	
A	+	B	A	B	A	B
tera-sort	+	grep	2,210 s	1,247 s	2,144 s	1,038 s
tera-sort	+	histogram-ratings	2,308 s	653 s	1,976 s	530 s
tera-sort	+	term-vector	2,576 s	2,183 s	2,349 s	1,845 s
tera-sort	+	wordcount	2,341 s	1,433 s	2,126 s	1,197 s
tera-sort	+	k-means	1,723 s	3,764 s	3,685 s	3,748 s

5 Related Work

YARN [30] is the second generation of Hadoop. It introduces *container* for better resource management in large Hadoop clusters. However, it does not have proactive shuffle like Shuffle-on-Write nor the ability to dynamically assign data partitions to different reduce tasks.

A number of studies proposed different task scheduling algorithms to improve Hadoop performance. The Longest Approximate Time to End (LATE) scheduling algorithm [37] improved the job performance in heterogeneous environments. FLEX [35] is a scheduling algorithm that enforces fairness between multiple jobs in a Hadoop cluster. It optimized the performance of each job under different metrics. Zaharia et al., proposed delay scheduling [36] as an enhancement to Hadoop Fair Scheduler. It exploited data locality of map task and significantly improved performance. Ghodsi et al. proposed dominant resource fairness [18] for fair resource allocation that considers multiple resource types. Wang et al. proposed preemptive task scheduling for Hadoop [34]. It is able to preempt long running reduce tasks to ensure fairness. It picks preemptee by the remaining work of each task. It does not consider the impact on number of reduce task waves, which can significantly increase the job completion time.

There are a few studies on skew mitigations and balancing workloads [7], [16], [20], [21], [24], [27], Tarazu balances the workload using data repartitioning [7]. It repartitions the intermediate data and distributes the workload of reduce phase to meet the performance difference of heterogeneous clusters. PIKACHU focuses on achieving optimal workload balance for Hadoop [16]. It presents guidelines for the trade-offs between the accuracy of workload balancing and the delay of workload adjustment. SkewReduce [20] alleviated the computational skew problem by applying a user-defined cost function on the input records. Partitioning across nodes relies on this cost function to optimize the data distribution. SkewTune [21] proposed a framework for

skew mitigation. It repartitioned the long tasks to take advantage of idle slots freed by short tasks. But, moving repartitioned data to idle nodes requires extra I/O.

Some recent work focused on the improvement of shuffle and reduce. MapReduce Online [12] proposed a push-based shuffle mechanism to support the online aggregation and continuous queries. ShuffleWatcher [6] proposed a scheduler that help shaping the shuffle traffic based on network load. Themis [26] employed a novel job workflow for MapReduce that aims to minimize the disk I/O operations. It implement the shuffle in the map phase and saves unsorted intermediate data in disk. But, it only hides the cost of data movement. The sorting of intermediate data still poses delay in reduce phase. The shuffle service in iShuffle handles both the data movement and the sorting in shuffle, and completely overlap the shuffle phase with the map phase. MaRCO [8] overlaps the reduce and shuffle. But the early start of reduce generates partial reduces which could be the source of overhead for some applications.

6 Conclusion

Hadoop design poses challenges to attain the best performance in job execution due to tightly coupled shuffle and reduce, partitioning skew, and inflexible scheduling. In this paper, we propose *iShuffle*, a novel user-transparent shuffle service that provides optimized data shuffling to improve job performance. It decouples shuffle from reduce tasks and proactively pushes data to be shuffled to Hadoop node via a novel *shuffle-on-write* operation in map tasks. iShuffle further optimizes the scheduling of reduce tasks by automatic balancing workload on multiple nodes with flexible reduce dispatching and preemptive reduce scheduling. We implemented iShuffle as a configurable plug-in in Hadoop and evaluated its effectiveness on a 32-node cluster with various workloads. Experimental results show that iShuffle is able to reduce job completion time by as much as 29.6 percent in representative benchmarks. iShuffle also significantly improves job performance in a multi-user Hadoop cluster by as much as 34 percent in Facebook workload trace.

Acknowledgments

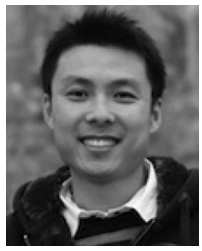
This research was supported in part by U.S. National Science Foundation CAREER award CNS-0844983, research grants CNS-1422119, CNS-1320122, and CNS-1217979, and NSF of China research grant 61328203. The authors are grateful to the anonymous reviewers for their valuable suggestions. Xiaobo Zhou is the corresponding author.

References

[1] Apache Hadoop Project. (2013). [Online]. Available: http://hadoop.apache.org
[2] HiBench. (2015). [Online]. Available: https://github.com/intel-hadoop/HiBench
[3] PUMA: Purdue mapreduce benchmark suite. (2012). [Online]. Available: http://web.ics.purdue.edu/~fahmad/benchmarks.htm
[4] Manual of Elasticsearch for Apache Hadoop. (2015). [Online]. Available: https://www.elastic.co/guide/en/elasticsearch/hadoop/current/configuration-runtime.html
[5] SWIM: Statistical Workload Injection for MapReduce. (2012). [Online]. Available: https://github.com/SWIMProjectUCB/SWIM/wiki
[6] F. Ahmad, S. T. Chakradhar, R. Anand, and T. N. Vijaykumar, "ShuffleWatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters," in *Proc. USENIX Conf. USENIX Annu. Techn. Conf.*, 2014, pp. 1–12.
[7] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar, "Tarazu: Optimizing MapReduce on heterogeneous clusters," in *Proc. Int Conf. Archit. Support Program. Languages Operating Syst.*, 2012, pp. 61–74.
[8] F. Ahmad, S. Lee, M. Thottethodi, and T. N. Vijaykumar, "MapReduce with communication overlap (marco)," *J. Parallel Distrib. Comput.*, vol. 73, no. 5, pp. 608–620, May2013.
[9] G. Ananthanarayanan, et al., "Scarlett: Coping with skewed content popularity in MapReduce clusters," in *Proc. ACM Eur. Conf. Comput. Syst.*, 2011, pp. 287–300.
[10] Y. Chen, S. Akspaugh, and R. Katz, "Interactive analytical processing of big data systems: A cross-industry study of mapredyce workloads," in *Proc. VLDB Endowment*, 2012, pp. 1802–1813.
[11] R. C. Chiang and H. H. Huang, "TRACON: Interference-aware scheduling for data-intensive applications in virtualized environments," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal. (SC)*, 2011, pp. 1–12.
[12] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "MapReduce online," in *Proc. 7th USENIX Conf. Netw. Syst. Des. Implementation*, 2010, pp. 21–21.
[13] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. 6th Conf. Symp. Opearting Syst. Des. Implementation*, 2004, pp. 10–10.
[14] D. DeWitt and J. Gray, "Parallel database systems: The future of high performance database systems," *Commun. ACM*, vol. 35, no. 6, pp. 85–98, 1992.
[15] K. Elmeleegy, C. Olston, and B. Reed, "Spongefiles: Mitigating data skew in MapReduce using distributed memory," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 551–562.
[16] R. Gandhi, D. Xie, and Y. C. Hu, "PIKACHU: How to rebalance load in optimizing MapReduce on heterogeneous clusters," in *Proc. USENIX Conf. Annu. Tech. Conf.*, 2013, pp. 61–66.
[17] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, CA: Freeman, 1990.
[18] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2011, pp. 323–336.
[19] Y. Guo, J. Rao, and X. Zhou, "iShuffle: Improving hadoop performance with shuffle-on-write," in *Proc. USENIX Int. Conf. Autonomic Comput.*, 2013, pp. 107–117.
[20] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skew-resistant parallel processing of feature-extracting scientific user-defined functions," in *Proc. ACM Symp. Cloud Comput.*, 2010, pp. 75–86.
[21] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "SkewTune: Mitigating skew in MapReduce applications," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2012, pp. 25–36.
[22] M. Li, D. Subhraveti, A. R. Butt, A. Khasymski, and P. Sarkar, "CAM: A topology aware minimum cost flow based resource manager for MapReduce applications in the cloud," in *Proc. 21st Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2012, pp. 211–222.
[23] M. Mitzenmacher, "The power of two choices in randomized load balancing," PhD dissertation, Graduate Division, Univ. California, Berkeley, CA, 1996.
[24] S. R. Ramakrishnan, G. Swart, and A. Urmanov, "Balancing reducer skew in MapReduce workloads using progressive sampling," in *Proc. ACM Symp. Cloud Comput.*, 2012, Art. no. 16.
[25] S. Rao, R. Ramakrishnan, A. Silberstein, M. Ovsiannikov, and D. Reeves, "Sailfish: A framework for large scale data processing," in *Proc. ACM Symp. Cloud Comput.*, 2012, Art. no. 4.
[26] A. Rasmussen, V. T. Lam, M. Conley, G. Porter, R. Kapoor, and A. Vahdat, "Themis: An I/O-efficient mapreduce," in *Proc. ACM Symp. Cloud Comput.*, 2012, Art. no. 13.
[27] M. A. Shah, J. M. Hellerstein, and E. Brewer, "Highly available, fault-tolerant, parallel dataflows," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2004, pp. 827–838.
[28] B. Sharma, T. Wood, and C. R. Das, "HybridMR: A hierarchical MapReduce scheduler for hybrid data centers," in *Proc. IEEE Int. Conf. Distrib. Comput. Syst.*, 2013, pp. 102–111.
[29] J. Tan, et al., "DynMR: Dynamic MapReduce with reducetask interleaving and maptask backfilling," in *Proc. ACM Eur. Conf. Comput. Syst.*, 2014, Art. no. 2.

[30] V. K. Vavilapalli, et al., "Apache Hadoop YARN: Yet another resource negotiator," in *Proc. ACM Symp. Cloud Comput.*, 2013, Art. no. 5.
[31] A. Verma, L. Cherkasova, and R. H. Campbell, "ARIA: Automatic resource inference and allocation for MapReduce environments," in *Proc. 8th ACM Int. Conf. Autonomic Comput.*, 2011, pp. 235–244.
[32] A. Verma, L. Cherkasova, and R. H. Campbell, "Resource provisioning framework for MapReduce jobs with performance goals," in *Proc. ACM/IFIP/USENIX Int. Conf. Middleware*, 2011, pp. 235–244.
[33] Y. Wang, X. Que, W. Yu, D. Goldenberg, and D. Sehgal, "Hadoop acceleration through network levitated merge," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2011, Art. no. 57.
[34] Y. Wang, J. Tan, W. Yu, X. Meng, and L. Zhang, "Preemptive reducetask scheduling for fair and fast job completion," in *Proc. USENIX Int. Conf. Autonomic Comput.*, 2013, pp. 279–289.
[35] J. Wolf, et al., "FLEX: A slot allocation scheduling optimizer for MapReduce workloads," in *Proc. ACM/IFIP/USENIX Int. Conf. Middleware*, 2010, pp. 1–20.
[36] 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 *Proc. ACM Eur. Conf. Comput. Syst.*, 2010, pp. 265–278.
[37] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving MapReduce performance in heterogeneous environments," in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2008, pp. 29–42.

Yanfei Guo received the BS degree in computer science and technology from the Huazhong University of Science and Technology, China, in 2010, and the PhD degree in computer science from the University of Colorado, Colorado Springs in 2015. He is currently a Postdoc fellow in the Argonne National Lab. His research interests include cloud computing, big data processing, and HPC. He is a member of the IEEE.



Jia Rao received the BS and MS degrees in computer science from Wuhan University in 2004 and 2006, respectively, and the PhD degree from Wayne State University in 2011. He is currently an assistant professor in the Department of Computer Science, University of Colorado, Colorado Springs. His research interests include the areas of resource auto-configuration, machine learning and CPU scheduling on multi-core systems. He is a member of the IEEE.

Dazhao Cheng received the BS degree in electronic engineering from the Hefei University of Technology in 2006. He received the MS degree in electronic engineering from the University of Science and Technology of China in 2009. Currently, he is working toward the PhD degree in computer science at the University of Colorado, Colorado Springs. His research interests include sustainable cloud computing and autonomic resource management. He is a student member of the IEEE.

Xiaobo Zhou received the BS, MS, and PhD degrees in computer science from Nanjing University, in 1994, 1997, and 2000, respectively. He is currently a professor of the Department of Computer Science, University of Colorado, Colorado Springs. His research include Cloud computing and datacenters, BigData distributed processing, autonomic and sustainable computing. He received US National Science Foundation CAREER Award in 2009. He is a senior member of the IEEE.