Virtual Shuffling for Efficient Data Movement in MapReduce

Weikuan Yu, *Senior Member*, *IEEE*, Yandong Wang, Xinyu Que, and Cong Xu

Abstract—MapReduce is a popular parallel processing framework for large-scale data analytics. To keep up with the increasing volume of datasets, it requires efficient I/O capability from the underlying computer systems to process and analyze data in two phases (mapping and reducing). Between these phases, MapReduce requires a shuffling phase to globally exchange the intermediate data generated by the mapping phase. We reveal that data shuffling, by physically moving segments of intermediate data across disks, causes significant I/O contention and compounds the I/O problem. In this paper, we propose a novel *virtual shuffling* strategy to enable efficient data movement and reduce I/O for MapReduce shuffling, thereby reducing power consumption and conserving energy. Virtual shuffling is realized through a combination of three techniques including a three-level segment table, near-demand merging, and dynamic and balanced merging subtrees. Our experimental results show that virtual shuffling significantly speeds up data movement in MapReduce and achieves faster job execution. Particularly, its reduction in disk I/O accesses results in as much as 12% savings in power consumption for MapReduce programs.

Index Terms—Hadoop, MapReduce, virtual shuffling, near-demand merging

1 Introduction

MapReduce [1] has emerged as a popular and easy-to-use programming model for large-scale data analytics in data centers. It is an important application for numerous organizations to process explosive amounts of data, perform massive computation, and extract critical knowledge out of big data for business intelligence. The efficiency of MapReduce performance and scalability can directly affect our society's ability to mine knowledge out of raw data. In addition, energy consumption accounts for a large portion of the operating cost of data centers in analyzing such big data. While business and scientific applications are increasingly relying on the MapReduce model, the energy efficiency of MapReduce is also critical for data centers' energy conservation.

Hadoop [2] is an open-source implementation of MapReduce, currently maintained by the Apache Foundation, and supported by leading IT companies such as Facebook and Yahoo!. It implements the MapReduce model by distributing user inputs as data splits across a large number of compute nodes. Hadoop uses a master program (called the JobTracker) to command many TaskTrackers (a.k.a slaves) and schedule map tasks (MapTasks) and reduce tasks (ReduceTasks) to the TaskTrackers, A Hadoop program processes data through two main functions (map and reduce). Accordingly, the analytic functions are performed in two phases: mapping and reducing. In the mapping phase, the input dataset of a program is divided into many *data split*s. Each split is organized as many records of key and value ($< k, v >$) pairs. One MapTask is launched per data split to convert the original records into intermediate data in the form of many ordered $< k', v' >$ pairs. These ordered records are stored as a MOF (Map Output File) split. A MOF is organized into many data partitions, one per ReduceTask. In the reducing phase, each ReduceTask applies the reduce function to its own share of data partitions (a.k.a segments).

Between the mapping and reducing phases, a ReduceTask needs to fetch a segment of the intermediate output from all finished MapTasks. Globally, this leads to a *shuffling* of intermediate data (in segments) from MapTasks to ReduceTasks. For data-intensive MapReduce programs such as TeraSort, data shuffling can add a significant number of disk accesses, contending for the limited I/O bandwidth. In order to elaborate this problem, we conduct a data-intensive MapReduce test, where we run 200 GB TeraSort on 10 slave nodes. We have examined the wait (queuing) time and the service time of I/O requests during the execution. As shown in Fig. 1, the wait time can be more than 1,100 milliseconds. Worse yet, most I/O requests are spending close to 100% of this time waiting in the queue, suggesting that the disk is not able to keep up with the requests. The shuffling of intermediate data competes for disk bandwidth with MapTasks. This can significantly overload the disk subsystem. It results in a serious bottleneck along with the severe disk I/O contention in data-intensive MapReduce programs, which entails further research on efficient data shuffling techniques.

Although a number of recent efforts have investigated data locality of MapReduce by either preventing stragglers (slow MapReduce tasks) [3]–[6] or applying high-performance interconnects to transfer data in large-scale Hadoop cluster [7], [8], few studies have addressed the need of efficient I/O during data shuffling in the Hadoop MapReduce framework. Condie et al. [9] have proposed a MapReduce online architecture to

• *The authors are with the Department of Computer Science and Software Engineering, Auburn University, AL 36849. E-mail: wkyu@auburn.edu.*

Manuscript received 13 Apr. 2012; revised 05 Nov. 2013; accepted 05 Nov. 2013. Date of publication 19 Nov. 2013; date of current version 16 Jan. 2015. Recommended for acceptance by S. Ranka.
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/TC.2013.216

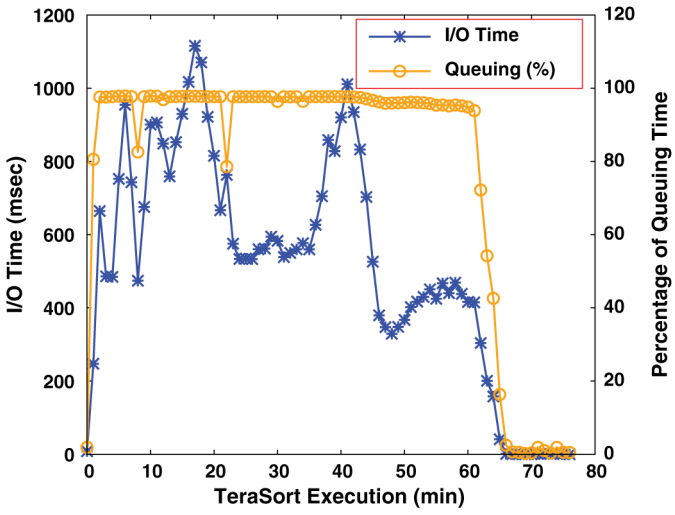


Fig. 1. Disk I/O contention in a MapReduce program.

open up direct network channels between MapTasks and ReduceTasks and speed up the delivery of data from MapTasks to ReduceTasks. While their work reduces job completion time and improves system utilization, it cannot accommodate a gigantic dataset that does not fit in memory, and also complicates the fault tolerance handling of Hadoop tasks. It often falls back to spilling data to the disks for large data sets. Our prior work [8] has offered a network-levitated merging scheme to keep the data shuffling phase above disks. But the aggressive use of memory buffers in network-levitated merging makes it unable to cope with MapReduce jobs with tens of thousands or even millions of data splits.

In this work, we undertake another effort to investigate the issue of disk I/O contention in data shuffling. As shown in Fig. 2, we take a new perspective at data shuffling of MapReduce programs. In the default Hadoop implementation, intermediate data segments are pulled by ReduceTasks in their entirety to local disks, and then merged before being reduced for final results. This is shown by Fig. 2(a). Because the physical movement of entire segments across disks, we refer to this strategy as physical shuffling.

Inspired by the classic concepts of virtual memory and demand paging, we propose a virtual shuffling strategy to enable efficient shuffling for MapReduce programs. Fig. 2(b) shows the general idea. Instead of moving entire segments to local disks before starting the reduce function, virtual shuffling allows a ReduceTask to fetch only a minimal set of segment attributes and create a virtual segment table that records the actual locations of remote segments. Once the global segment table is fully constructed, the ReduceTask can start the final merge and generate data for the reduce function. When the reduce function starts, the majority of segments still reside on remote disks. In another word, segments will not be brought in locally through the global segment table until they are needed by the reduce function. Virtual shuffling delays the actual movement of data until the ReduceTask requests data. Compared to physical shuffling, when the reduce function demands more data input, virtual shuffling employs near-demand merging to fetch data in small blocks into memory, merge and send them directly to the reduce function. In doing so, virtual shuffling greatly reduces the number of disk accesses of physical shuffling, and enables efficient data movement. Such optimizations in I/O and program execution also benefit MapReduce programs in terms of power consumption and energy savings. The design of virtual shuffling is described in detail in Section 3. Accordingly, we have implemented virtual shuffling, tuned a number of performance-critical parameters, and evaluated its benefits to job execution an I/O reduction. Furthermore, we have measured power consumption during the execution of MapReduce programs to quantify the energy savings.

In summary, this paper makes the following research contributions:

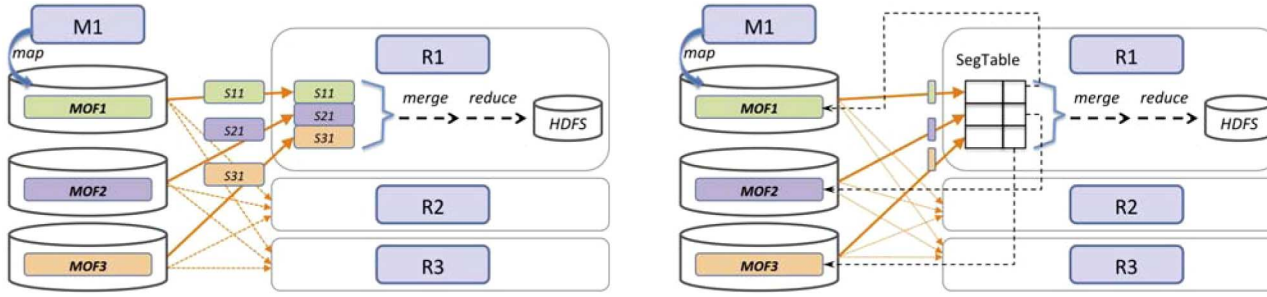
- A novel virtual shuffling strategy that enables efficient data movement and relieves the disk I/O contention problem in the original Hadoop MapReduce;
- The design and implementation of virtual shuffling using a three-level segment table, near-demand merging, and dynamic and balanced merging subtrees;
- Systematic evaluation and documentation of the benefits provided by virtual shuffling in terms of job execution time and disk access. Particularly, we show that virtual shuffling can lead to 12% savings in power consumption for MapReduce programs.

The rest of the paper is organized as follows. Section 2 gives a review of related work. Section 3 presents virtual shuffling in detail, followed by Section 4 that describes the implementation of virtual shuffling. Section 5 introduces experimental setup and benchmarks. Sections 6, 7 and 8 provide our tuning and evaluation of virtual shuffling. Section 9 discusses a few relevant issues of virtual shuffling. Finally, we conclude the paper in Section 10.

2 Related Work

MapReduce is popularized by Google as a very simple but powerful program model that offers parallelized computation, fault-tolerance and distributing data processing [1]. Its open-source implementation, Hadoop, provides a software framework for distributed processing of large datasets [2]. We review related work in a number of directions.

MapReduce Performance Tuning: Several studies were published on tuning the performance of MapReduce. These include [4], [10]–[12] that tuned different parameters of Hadoop MapReduce for performance. Dai et al. [13] developed HiTune for Hadoop performance analysis and tuning. Herodotou et al. [14] designed a cost-based optimizer with performance knobs to help choose better Hadoop configurations. Zaharia et al. [6] proposed a new scheduling algorithm, called Longest Approximate Time to End (LATE), for environments with heterogeneous server configurations. Ananthanarayanan et al. proposed Mantri [3] to monitor tasks and cull outliers for better job completion time, and later proposed Scarlett [15] to replicate data blocks to alleviate hotspots. Jahani et al. [16] applied compiler techniques for Hadoop optimizations. Tan et al. [12] documented and extensively analyzed the performance problem of delay tails in Hadoop MapReduce programs caused by long ReduceTasks. However, none of these works investigated the I/O problem caused by MapReduce intermediate data shuffling. Our work takes on a different perspective to investigate new strategies for efficient data movement in MapReduce, relieving its I/O contention.



(a) Physical Shuffling (b) Virtual Shuffling

Fig. 2. Comparisons of different shuffling strategies.

MapReduce Data Communication: Kim et al. [17] improved the performance of MapReduce by reducing redundant I/O in the software architecture. But it did not study the I/O issue caused by data shuffling between MapTasks and ReduceTasks. The closest work to our project is *MapReduce Online* as proposed by Condie et al. [9]. This work focused on enabling instant shuffling (so called online) of intermediate data from MapTasks to ReduceTasks. Essentially, MapReduce Online introduces direct data shuffling channels between MapTasks and ReduceTasks to avoid the creation of intermediate Map Output Files. In doing so, it requires the direct coupling of each MapTask with all ReduceTasks, and completely changes the fault handling mechanism of Hadoop. A failure of a MapTask or a ReduceTask is no longer a local event that can easily be recovered by re-launching the failed task. In addition, MapReduce Online requires a large number of sustained TCP connections, which severely limits its scalability. In contrast, our work does not require close coupling of data flow between MapTasks and ReduceTasks, thus allowing separated recovery from failures of either MapTasks or ReduceTasks.

Spark [18] is an emerging MapReduce-based system for big data analytics. It recognizes the disk I/O bottleneck issue during the data shuffling and relaxes the sorting/merging requirement at the reduce sides. i.e., *it is not necessary to sort intermediate data before a ReduceTask starts processing them*. By relaxing such constraint, data shuffling and computation can be pipelined and accomplished in memory. Spark requires very high memory consumption for shuffling and merging in memory. In addition, by retaining intermediate results in memory, Spark can efficiently accelerate many data-intensive programs, such as weather prediction applications that require iterative algorithms [19].

Power and Energy of MapReduce Programs: Leverich et al. [20] modified Hadoop to allow scale-down of operational clusters which could save between 9% and 50% of energy consumption. They also outlined further research into the energy-efficiency of Hadoop. Lang et al. [21] closely examined two techniques, namely Covering Set (CS) and All-In Strategy (AIS), which could be used for the management of MapReduce clusters. They showed that AIS was the right strategy for energy conservations. Chen et al. [22] presented a statistics-driven workload generation framework which distilled summary statistics from production MapReduce traces and realistically reproduced representative workloads. This methodology could be useful for understanding design trade-offs in MapReduce. The same team also exploited and analyzed how compression could improve performance and energy efficiency for MapReduce workloads [23]. They proposed an algorithm that examines per-job data characteristics and I/O patterns, and decides when and where to use compression. Our work does not directly study energy conservation techniques, but evaluates the benefits of virtual shuffling in energy savings. This is complementary to previous research efforts. Our work documents a case study in conserving energy by reducing other related system activities such as disk access.

3 Virtual Shuffling

In this section, we describe in detail the design of virtual shuffling that can enable efficient data movement for MapReduce. In order to overcome the disk I/O problem of physical shuffling, virtual shuffling needs to address three important issues: (1) how to scalably represent intermediate data segments in a virtual manner, (2) how to minimize the impact of actual shuffling of data; and (3) how to dynamically coordinate and balance data shuffling and merging without degrading the performance.

3.1 A Three-Level Segment Table

We draw our inspiration from the classic concept of virtual memory in designing virtual shuffling. To manage many intermediate data segments produced by MapTasks, we design a three-level segment table to organize them in a scalable manner. The hierarchical table includes three kinds of directories: the Segment Table Directory, the Segment Middle Directory, and the Segment Global Directory. These directories and their associated data structures are listed in Table 1.

Fig. 3 shows the organization and relationship among these three levels. At the completion of a MapTask, its data segment is not physically fetched all at once by a ReduceTask. Instead, a Segment Table Entry (STE) is created at the lowest level-Segment Table Directory (STD)-to represent the segment in a virtual manner. The STE includes several attributes of the segment such as its total length, its source MapTask, as well as its physical location on the remote disk. The number of STEs in an STD is a tunable parameter based on the computation, memory, and I/O resources. Many STDs are organized into a Segment Middle Directory (SMD), in which each entry (SME) represents an STD. Many SMDs in turn are organized as a Segment Global Directory (SGD) with each entry (SGE) referring to an SMD. Within a ReduceTask, we allocate

TABLE 1
Segment Table Terms and Their Definitions

Level	Directory	Entry	Merging Buffer
First	Segment Table Directory (STD)	STD Entry (STE)	STD Merging Buffer (STB)
Second	Segment Middle Directory (SMD)	SMD Entry (SME)	SMD Merging Buffer (SMB)
Third	Segment Global Directory (SGD)	SGD Entry (SGE)	SGD Merging Buffer (SGB)

Segment Entry Buffers (SEBs) for each STE. SEBs are used to store partial blocks of data for incoming segments. In addition, three kinds of memory buffers are used as temporary merging buffers. For example, an SGD will merge the data from its constituent SMDs to the SGD Buffer (SGB); an SMD will merge data from its STDs to its SMD Buffer (SMB); and an STD will merge data from its constituent SEBs to its STD Buffer (STB). With this three-level hierarchical table, if there were memory pressure, we can keep only a few active STDs and their ancestral SMDs and SGDs in memory, while other STDs are temporarily stored on disk.

3.2 Near-Demand Merging

Physical shuffling causes frequent I/O accesses because ReduceTasks fetch all segments and merge them using local memory/storage before they are needed. Because of the pressure of hosting more data than available memory, ReduceTasks often have to spill the intermediate data to the disks. To adress this issue, virtual shuffling mimics the concept of demand paging and realizes near-demand merging to reduce the pressure of data spilling and minimize the impact of actual data movement. With near-demand merging, we wait until it is clear which segments are needed by the reduce function of ReduceTasks. Near-demand merging does not really wait until the last moment to fetch data. Instead, it works as part of virtual shuffling to form a pipeline of fetching, merging and reducing data segments, with an emphasis on hiding the cost of data transfer over the network. With near-demand merging, virtual shuffling is not designed to eliminate data movement, but to hide its cost within the reduce phase.

Fig. 4 shows the operation of near-demand merging. When a ReduceTask needs to reduce some data, it initiates a data request to the segment table, which in turn triggers the fetching of data blocks (which contain more intermediate $< \text{key}, \text{val} >$ pairs from MapTasks) from remote segments. These blocks will then be buffered at SEBs. Based on the virtual segment table, these $< \text{key}, \text{val} >$ pairs in SEBs will then be merged through STBs, SMBs, and finally into SGBs. The ordered $< \text{key}, \text{val} >$ pairs in SGBs are ready to be reduced by the reduce function.

To avoid synchronously waiting on the completion of these steps, two sets of buffers are provided at each interface. This enables double buffering and overlaps the near-demand merging of incoming data with the reducing of previous data. Data from each segment is brought in sequentially as small blocks. One block will be fetched into an SEB only when it is the next block to be merged. Near-demand merging is built on top of our previous work *network-levitated merging* [8]. While network-levitated merging strives to lift the merging process of segments above disks, near-demand merging emphasizes the importance of minimizing and hiding the cost of shuffling to the reduce phase. Near-demand merging does not preclude the need of flushing data to disks, as will be described in Section 3.3.

3.3 Dynamic and Balanced Subtrees

With a hierarchical segment table, all virtual segments are essentially organized into a merging tree in which the leaves are the SEBs. If near-demand merging with double buffering were to activate all leaves, there would be a need of $2N$ SEBs, where N is the number of total virtual segments. For an application with a dataset (S) and a data split size (B), it will then have $N = \frac{S}{B}$ segments. Assume a split size of 64 MB, SEB of 32KB, and the use of double buffering for blocks from each segment, the amount of memory needed for all SEBs would be

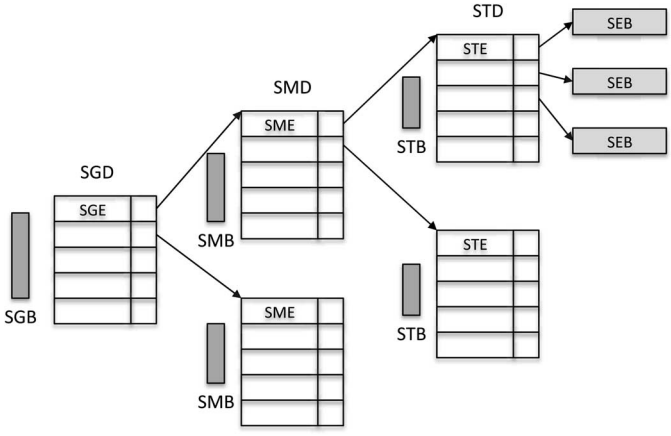


Fig. 3. Design of a three-level segment table for virtual shuffling.

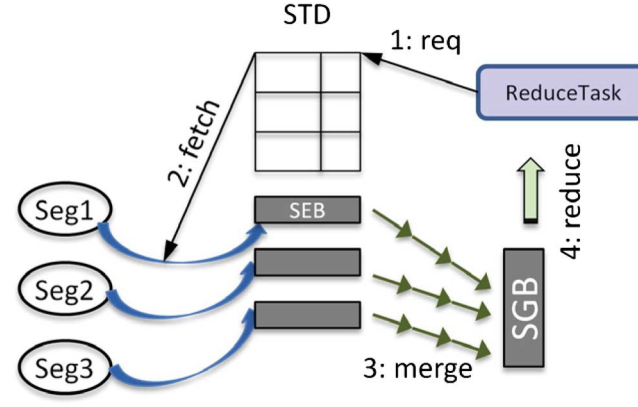


Fig. 4. Near-demand merging.

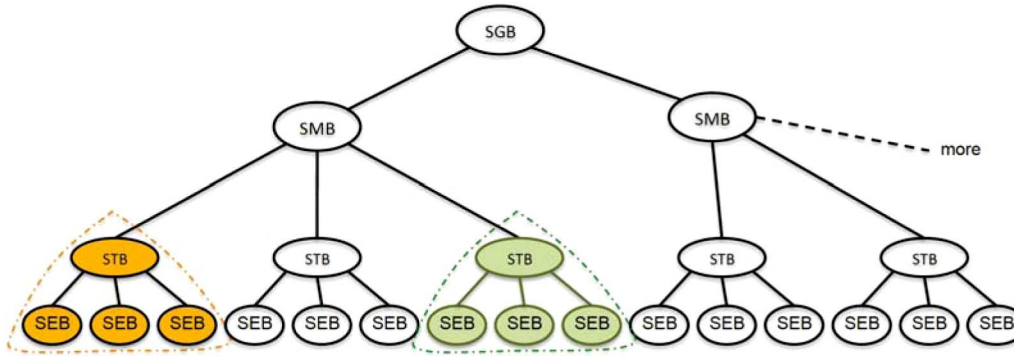


Fig. 5. Dynamic and balanced subtrees for concurrent merging.

$\frac{S}{64\,MB} \times 2 \times 32KB$ bytes per ReduceTask. This means that even if we reserve 1 GB memory per ReduceTask, we can only support an application with 1 Terabyte of data. Clearly, there is a need of better scalability for applications with petascale data and beyond. Because of this issue, our network-levitated merging technique faces a dilemma between the growing need of scalability and the need to keep data levitated, i.e., in memory. We employ a dynamic orchestration mechanism to manage the merging of virtual segments. Instead of activating all leaves, we organize the whole tree as many subtrees, each composed of an STD and its SEBs. At any time, only a limited number of subtrees are actively merging data. As shown in Fig. 5, two subtrees are currently active in merging its SEBs into STBs. The merged data will be further merged to SMBs and/or SGBs. To balance the merging progress at different subtrees, previously active subtrees will be deactivated to allow other subtrees to make progress.

There is an intriguing issue here. At the time when a subtree is to be deactivated, their SEBs usually contain data not yet merged to the STB. Worse yet, the use of double buffering means that, for any segment, one SEB has data left to be merged while the other SEB is waiting on data to be fetched remotely. A decision needs to be made on either dumping the data (including the remaining data in one SEB and the data in flight for the other SEB) or flushing the data to the disk, to make memory available for other subtrees. To improve the utilization of data in memory, we prevent a subtree from fetching more data into SEBs when its STB is already 70% full, and also provide a configurable period of 0.5 second (a configurable parameter) in the deactivation of a subtree, allowing more data in SEBs to be consumed (merged) into the STB. For a subtree that still has data left in its SEB, we by default dump the data. A user option is also introduced to allow the flushing of data to disk. The reason for dumping data by default is to avoid frequently writing small data blocks to, and reading them back from, disks.

4 Implementation

In this section, we describe the detailed implementation of virtual shuffling. The design of virtual shuffling is built on top of our previous work, Hadoop-A [8], which provides an acceleration framework for performance enhancement and protocol optimizations. We start with some background on Hadoop-A and then explain in detail the implementation of virtual shuffling.

4.1 MOFSupplier and NetMerger

The Hadoop-A framework optimizes Hadoop with two main plugin components, *MOFSupplier* and *NetMerger*. As shown in the Fig. 6, the MOFSupplier and NetMerger are standalone native C processes. They are launched by the local TaskTracker and connected back to TaskTracker through asynchronous loop-back sockets.

MOFSupplier: A MOFSupplier communicates with all local MapTasks and manages all the MOFs generated by these MapTasks. It employs a network server to serve the requests from remote ReduceTasks. Each ReduceTask will fetch one segment of each MOF. Since all MOFs are stored on the disk and the incoming requests incur frequent disk accesses. Two cache structures, IndexCache and DataCache, are provided in the MOFSupplier to facilitate I/O. IndexCache is provided to speed up the retrieval of segment locations of MOFs. The DataCache is used to prefetch and buffer segment data thereby reducing the actual disk I/O upon requests. Each MOFSupplier also provides multiple I/O threads to serve requests in parallel.

NetMerger: A NetMerger works with local ReduceTasks on the same node. It contains a client thread, an upload thread, as well as one merging thread for each ReduceTask. The client thread issues fetch requests on behalf of local ReduceTasks. When segments arrive, it forwards them to the merging thread to be merged. The upload thread will in turn push the merged segments to Hadoop Java-side ReduceTask for data processing by the reduce function.

4.2 Leveraging the Hierarchical Segment Table

We implement virtual shuffling by extending the user-configurable plugin components, *MOFSupplier* and *NetMerger* in Hadoop-A. The bulk of virtual shuffling implementation centers around the three-level segment table. As mentioned earlier in Section 1, when a ReduceTask is notified about the availability of MOFs, it starts fetching the segments from all the MOFs and merges them. Since not all segments can be stored in the limited memory of a ReduceTask, it is crucial to decide when and how to shuffle, order and merge the intermediate segments of $< \mathrm{key}, \mathrm{val} >$ pairs. The original Hadoop employs a heap-like structure called *priority queue*, to manage the ordering and merging of segments. A segment descriptor is used to represent each segment in the priority queue. Compared to the cost of data shuffling and merging, the storage and computation overheads for constructing the priority queue of segment descriptors are negligible. Virtual

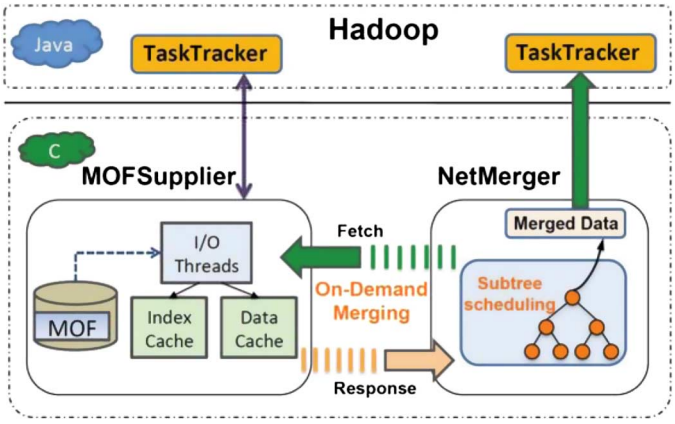


Fig. 6. Software implementation of virtual shuffling.

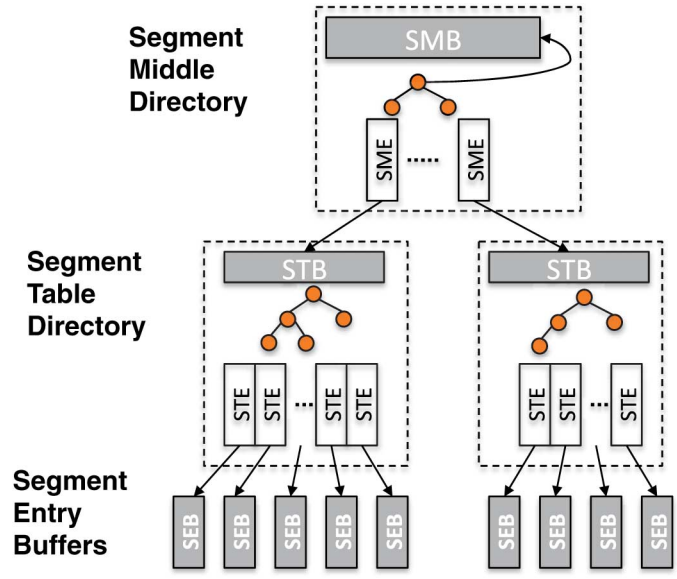


Fig. 7. Implementation of near-demand merging.

shuffling takes advantage of the fact that it is really not necessary to fetch all segments to the local node before merging them. Instead we can allow partial fetching and merging of segments, thereby reducing the demand on memory and disk I/O. Through this accumulative procedure, virtual shuffling completes the fetching and merging of all segments.

In our implementation of virtual shuffling, we use a three-level hierarchical table to manage near-demand merging of segments. The initialization of multiple level segment table is shown in Fig. 7. Here we show only two levels of directories in the table for simplification. At the very bottom, a set of SEBs are prepared for fetching remote segments on demand. When more and more segment headers (the first 64KB $< \text{key}, \text{val} >$) are available, an STD is created to manage and merge available segments. Each STE entry in the STD directory represents such a partial segment. Once an STD is fully established, the merging of SEBs in this STD will be triggered on demand by the reduce function, as described in Section 3.2. The merged $< \text{key}, \text{val} >$ pairs are stored in the STB.

More arriving segment headers will lead to more STDs, in turn leading to more STBs. To manage and merge $< \text{key}, \text{val} >$ pairs contained in these STBs, an SMD will be created. Each SMD entry points to an STD and the data location points to the STB. The SMB is the memory buffer that will host the merged $< \text{key}, \text{val} >$ pairs from STBs. For a three-level segment table, similar to the organization of STBs into an SMD, many SMDs will form an SGD in which the data for each entry is stored in an SMB. The SGB is then the memory buffer that will host the final merged $< \text{key}, \text{val} >$ pairs from SMBs.

To relieve the memory pressure and achieve scalability, the merging process of different merging trees is orchestrated dynamically based on resource availability. The design of virtual shuffling only allows a limited number of sub-merging trees to be activated as shown in Fig. 5. In addition, we recycle all buffers in the hierarchical table. All subtrees are monitored and ordered based on the merging progress of their data segments. Once the progress of SEBs in one subtree falls below a threshold, it is then scheduled to fetch and merge more data. All these merging tasks are undertaken by the same pool of merging threads. The reduce function can get delayed if there is no data in SGBs. To minimize such delays, we grant a higher priority to the task of merging STBs into SGBs, allocate and activate a thread when there is a need to refill SGBs with more $< \text{key}, \text{val} >$ pairs.

4.3 Scalability of Our Implementation

Hadoop intermediate data is fetched from remote MOFs and stored in SEBs before being merged through the three-level segment table. As previously analyzed in Section 3.3, there must be at least one SEB (M_b) for each segment (we use two for double buffering). For a Hadoop application, the total number of segments (N) is $\frac{S}{B}$, where S is the size of application's dataset and B the size of a data split. If we use only a segment table with only one-level directory, all SEB must be loaded with active memory while their $< \text{key}, \text{val} >$ pairs are merged through the priority queue. The amount of needed memory (M_t) is then given by $M_t = 2 \times M_b \times \frac{S}{B}$. Conversely, the application data size is given by $S = \frac{M_t}{M_b} \times \frac{B}{2}$. There are usually multiple ReduceTasks per computer server. As mentioned in Section 3.3, on commodity servers with GBs of memory, we can only support Hadoop programs with terabytes of data. Furthermore, a key issue here is that the memory requirement grows linearly (i.e., on the order of $O(N)$) with respect to the number of segments.

In virtual shuffling, we can organize all segments into a three-level segment table, which is essentially a hierarchy of many priority queues. We can activate the data shuffling for as few as one subtree (which has an embedded priority queue), and leave the other subtrees temporarily inactive, i.e., not holding any data in memory. For simplicity, we can assume that the SGD, the SMDs and STDs have the same number of entries. If virtual shuffling is designed with only a two-level segment table, the number of segments in one STD is then on the order of $O(\sqrt{N})$. When we keep only one STD active, the memory requirement becomes $M_t = 2 \times M_b \times \sqrt{\frac{S}{B}}$. With such a hierarchical organization, the maximum supported data size for a single Hadoop job can be calculated as 16 petabytes, using the same numbers as before. If virtual shuffling is designed with a three-level segment table, the number of segments in one STD is then in the order of $O(\sqrt[3]{N})$.

When we keep only one STD and one SMD active, the memory requirement becomes $M_t = 2 \times M_b \times \sqrt[3]{\frac{S}{B}}$. With such a hierarchical organization, the maximum supported data size for a single Hadoop job would be 256 exabytes. However, a deeper hierarchy implies that each segment has to go through multiple levels of directories. While it can improve memory scalability, a deeper hierarchy implies more memory copies. Based on the above analysis, provided enough computer nodes, the two-level segment table can theoretically support a single MapReduce application with tens of petabytes under a reasonable consumption of 1 GB memory per ReduceTask. The size of data splits and that of SEBs (see Section 6) can be adjusted further to support applications with 100s of petabytes. We consider such theoretical scalability as sufficient, therefore adopt the two-level segment table in our implementation of virtual shuffling.

5 Experimental Setup and Test Benchmarks

We conduct all our experiments on a cluster of 21 compute nodes. Each node is equipped with dual-socket quad-core 2.13 GHz Intel Xeon processors and 8 GB of DDR2 800 MHz memory, along with 8x PCI-Express Gen 2.0 bus. Four cores on a socket share 4 MB L2 cache. These nodes run Linux 2.6.18-164.el5 kernels. All nodes are equipped with Mellanox ConnectX-2 QDR Host Channel Adapters and are connected to both a 108-port InfiniBand QDR switch and a 48-port 10GigE Vantage switch. We use the InfiniBand software stack, OFED [24] version 1.5.3.2, as released by Mellanox. Each node has one 500 GB, 7200 RPM, Western Digital SATA hard drive and Hadoop version 0.20.0 is used. On each node, we have configured 4 map slots and 2 reduce slots. Each MapTask is assigned 512 MB maximal heap size, while 2 GB for each ReduceTask.

We evaluate virtual shuffling with a number of popular public benchmarks. These include the *TeraSort*, *Grep*, and *WordCount* test programs that are distributed as part of Hadoop. Grep searches in a text file for a predefined expression and creates a file with matches. WordCount counts the number of occurrences of different words in a data file. TeraSort is a popular benchmark that measures the capability of a program in sorting a large-scale dataset. In addition, we examine a benchmark of Hive [25]—a high-level query language that is designed to facilitate user queries for data processing and analysis over Hadoop.

6 Parameter Tuning of Virtual Shuffling

Virtual shuffling enables a seamless flow of data, starting from MOF segments, going through a series of steps including fetching, buffering, and merging, and finally reaching the reduce function at ReduceTasks. A number of important parameters, such as SEB and STB buffer sizes and the number of virtual segments in a subtree, need to be tuned for this pipeline to work efficiently. In our tuning tests, we have examined a variety of different data sizes on a number of different nodes. For brevity, we present several representative case studies, running the TeraSort program with the data size being 128 GB and the data split size 128 MB. This results in a total of 1024 splits in the job, equal to the number of data segments per ReduceTask.

6.1 SEB and STB Buffer Sizes

In the three-level segment table, SEB and STB are the primary interfaces for merging data segments into the SGB. Their sizes can affect the effectiveness of the fetching and merging processes. It is important to understand how they impact the performance of MapReduce programs. To do this, we first fix the number of virtual segments in a subtree, which we choose as the square root of the total number of segments.

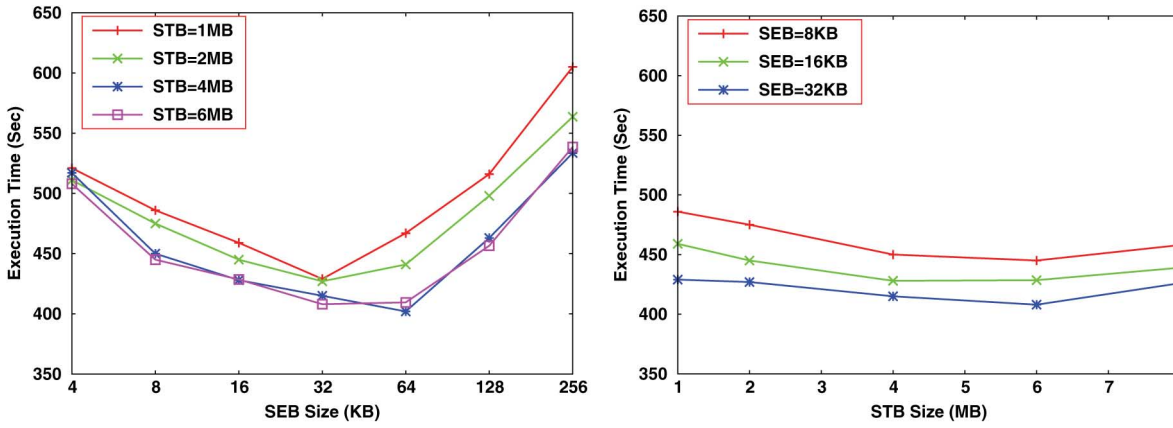
Fig. 8(a) shows the tuning of SEB buffer size. For a fixed STB buffer size, with an increasing SEB buffer size, the execution time decreases. This is because, when a large SEB buffer is used, the data fetching speed can catch up with the merging speed of the active subtree, which benefits the pipeline of fetching and merging. However, when the SEB buffer size goes further up, the execution time becomes worse. This is because more data from SEBs are dumped when a subtree is deactivated. The same data often have to be re-fetched from remote segments, resulting in wasteful and repetitive disk accesses. In addition, this delays the effectiveness of data fetching and stalls the entire pipeline. As shown in the figure, with different STB buffer sizes, these performance curves reach their minimum at different points. The best SEB buffer sizes are 32KB and 64KB for different STB buffers. Since the STB buffer sizes are different, the fetching speed and merging speed on the subtrees are different as well. The lowest point in a curve suggests a balance where the data fetching speed matches up with the merging speed.

To reveal more performance impact of the STB buffer size, we conduct another experiment to tune the STB buffer size with a fixed SEB buffer size. Fig. 8(b) shows the results of this tuning experiment. With an increasing STB buffer size, the job execution time gets shorter and reaches the lowest point when the STB size is between 4 MB and 6 MB. After that, the job execution stays roughly flat. The underlying cause of this behavior is again the interplay between the speed of data fetching and that of merging. In another word, there is a competition between the speed of data fetching and data merging. The optimal performance occurs when the data fetching and merging speeds are well balanced. Thus, when the size of the subtree is fixed, the job execution time can benefit from a large STB buffer until it can hold all SEBs buffers. For the rest experiments of the paper, we use 4 MB STB buffers and 64KB SEB buffers unless otherwise specified.

6.2 Virtual Segments in a Subtree

The number of virtual segments in a subtree is another factor that can directly affect the concurrency of data merging, thus impact the performance. It is important to understand its performance implications. Fig. 9 shows the results of our tuning experiment, in which we fix the STB size to be 4 MB. As subtrees grow in size, the number of total subtrees is decreased, which consequently reduces the load of merging threads. That is to say, the merging threads are able to merge active subtrees efficiently without having to deactivate many of them. If subtrees grow further in size, one active subtree will trigger more SEBs to fetch data from remote segments, which in turn causes more disk contention on the accessing of MapOutput files. The program execution is then affected, resulting in longer execution time.

Given this intriguing behavior, we speculate that there is a theoretical relationship between the subtree size and the job



(a) SEB Buffer Size (b) STB Buffer Size

Fig. 8. Tuning of memory buffer sizes.

execution time. On one hand, smaller subtrees will lead to more of them, causing a linear increasing complexity in managing subtrees (activating and deactivating). On the other hand, virtual segments in a subtree need to be fetched remotely, thus more of them will lead to a linear increasing cost of data movement. In addition, Hadoop uses a heap-based priority queue for merging $< \text{key}, \text{val} >$ pairs, which is inherited by virtual shuffling as well. This implies that there is a logarithmic complexity in merging SEBs into STBs and the same in merging STBs to the SGB.

To gain more insight on the relationship, we perform an analysis on the scalability trend. Let us denote the total number of segments as M, and the subtree size as x. The total number of subtrees is then $\frac{M}{x}$. The time for managing these subtrees and merging their STBs can be denoted as $f_1 = O(\frac{M}{x} + log\frac{M}{x})$. The total time for fetching SEBs of a subtree and merging them into the STB can be denoted as $f_2 = C{*}O(x) + O(logx)$, where $C\,(C > 1)$ is a constant factor that represents the longer time for fetching data remotely, compared to that of managing subtrees locally. Ideally, the merging processes conducted on STB and SEB would be completely asynchronous and be able to achieve a pipeline without any stall, when the two time durations are the same. The lower bound for x can be solved by letting $f_1 = f_2$. Approximately, x is equal to $\sqrt[(C+1)]{M}$. With the total number of segments being 1024, the theoretical lower bound would be a number close to 16, depending on the exact value of C. As shown in Fig. 9, this conjecture matches well with our empirical results.

7 Benefits to Job Execution

7.1 Overall Performance

We run Hadoop TeraSort benchmark with different data sizes and different numbers of nodes. Each slave runs 8 MapTasks and 4 ReduceTasks concurrently. Fig. 10 shows the performance of TeraSort on 20 nodes using different shuffling strategies, where the total amount of physical memory is 160 GB. Three different cases are included in the comparison: virtual shuffling as implemented in Hadoop-A, physical shuffling in Hadoop-A, and physical shuffling in the original Hadoop. Hadoop-A tests were run with InfiniBand RDMA (Remote Direct Memory Access) and RoCE (RDMA over Converged Ethernet) transport protocol. The original Hadoop was run with InfiniBand IPoIB and 10GigE. Because of its scalability limitation, we did not include our previous work network-levitated merge in this comparison.

As shown in the figure, physical shuffling in Hadoop-A performs slightly better than the original Hadoop for all cases. This is because the use of the high-speed RDMA protocol compared to the IPoIB and 10GigE protocol. While RDMA is beneficial to data movement on the network, the performance bottleneck lies with the disk I/O performance when physical shuffling is used. Therefore, marginal benefits are observed when the transport protocol is replaced. Among the three cases, virtual shuffling consistently performs the best and improve the overall performance by up to 27%, which demonstrates that virtual shuffling is able to speed up the data movement and boost the performance. Virtual shuffling also shows good scalability on different networks, which demonstrates consistent performance improvement compared to the physical shuffling. Because of the comparable performance, we use the original Hadoop as the representative implementation of physical shuffling for the rest of the paper and we conduct all the rest tests on InfiniBand network.

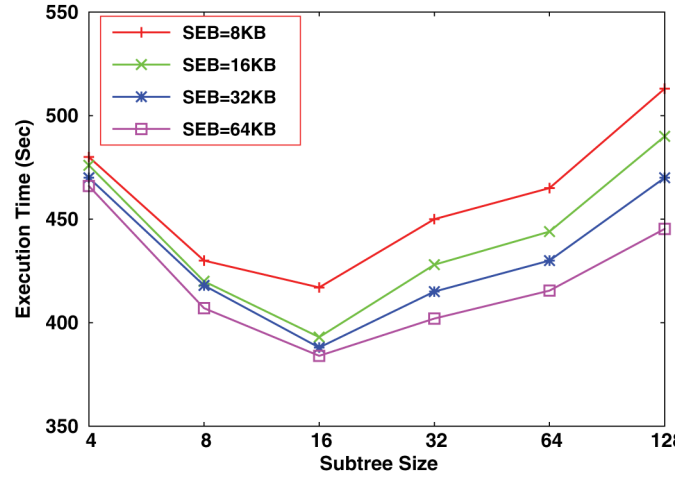


Fig. 9. Tuning of virtual segments in a subtree.

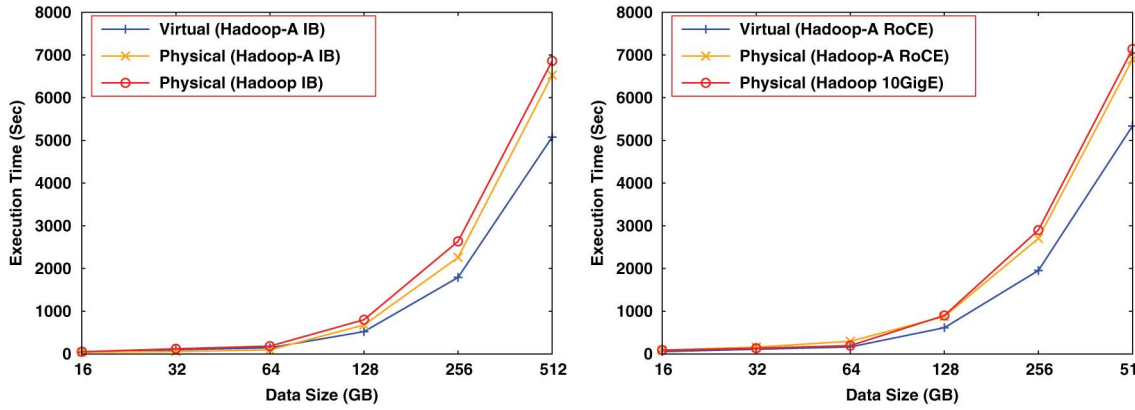


Fig. 10. Performance comparison of different shuffling strategies.

Fig. 11 shows the performance of different shuffling strategies on four representative benchmarks, namely WordCount (256 GB), Grep (256 GB), Hive Benchmarks (Order By, 60 GB), and TeraSort (512 GB). As shown by the figure, virtual shuffling speeds up the total execution time significantly for the Hive Benchmarks and TeraSort, by 54.6% and 26%, respectively, compared to physical shuffling. For two other common web crawling benchmarks, Grep and WordCount, virtual shuffling does not lead to much performance benefit. This is because these programs generate very little intermediate data, requiring little data movement during data shuffling. Thus, their data shuffling phase is mostly CPU bound. Overall, these performance results indicate that virtual shuffling can significantly improve the performance of data-intensive MapReduce programs, and the benefit dwindles for the programs that do not have much intermediate data.

7.2 Progress of TeraSort Execution

We compare the progress of TeraSort program execution using different shuffling strategies. The results are shown in Fig. 12. The Y-axis shows the percentage of completion for Map and Reduce Tasks. The X-axis shows the progress of time during execution. Fig. 12(a) shows that MapTasks of TeraSort complete much faster with virtual shuffling, especially when the percentage of completion goes over 50%. This is because MapTasks are launched as multiple waves of tasks. Right after the first wave of MapTasks finished, under the physical shuffling strategy, ReduceTasks are launched and compete for the disk bandwidth with later waves of MapTasks. This leads to severe disk contention and a cascading impact to the job execution time. In contrast, virtual shuffling eliminates such disk contention to MapTasks, which consequently benefits the progress of MapTasks. Similarly, Fig. 12(b) shows that ReduceTasks are completed faster with virtual shuffling. Note that in the case of physical shuffling, the progress of ReduceTasks is reported while the data are being merged. However, with virtual shuffling, we do not report progress until the completion of all MapTasks, and then near the completion of merging all segments. This is reflected in the figure as seemingly slow initial progress for virtual shuffling. Virtual shuffling actually still makes progress on ReduceTasks. Once it begins reporting, the progress in terms of percentage jumps up quickly, first at the completion of MapTasks and then at the end of merging, as indicated by the two jumps in the figure.

7.3 Scalability

Being able to leverage more nodes to process large amounts of data is an essential feature of Hadoop. We want to ensure that virtual shuffling deliver scalability in a similar manner. So we measure the total execution time of TeraSort in two scaling patterns: one with fixed amount of total data (200 GB) and an increasing number of nodes, and the other with fixed data (10 GB) per node and an increasing number of nodes. The aggregated throughput is calculated by dividing the total size with the program execution time.

Fig. 13 shows the scalability comparison between virtual shuffling and physical shuffling with a fixed data size (10 GB) per node. Both of them can achieve linear scalability. Virtual shuffling can speed up the execution time by approximately 30% and improve throughput by 43%. Fig. 14 shows the scalability comparison between virtual shuffling and physical shuffling with a fixed size of total data (200 GB). Again both of them can achieve good scalability. Virtual shuffling can cut the execution time by up to 33%, compared to physical shuffling. Conversely, this results in a throughput improvement of 49.2%. To summarize, compared to physical

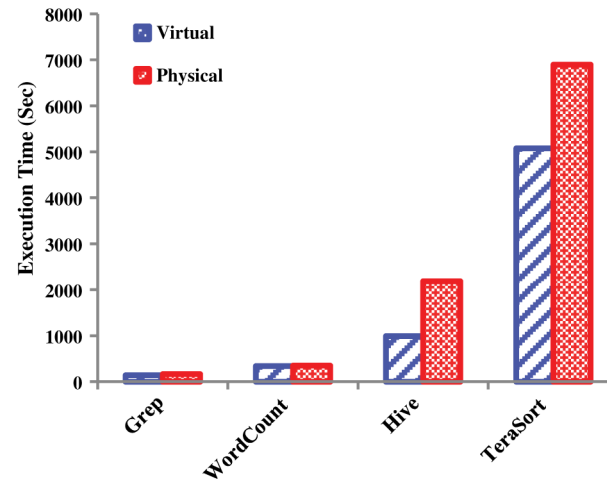


Fig. 11. Performance of different benchmarks.

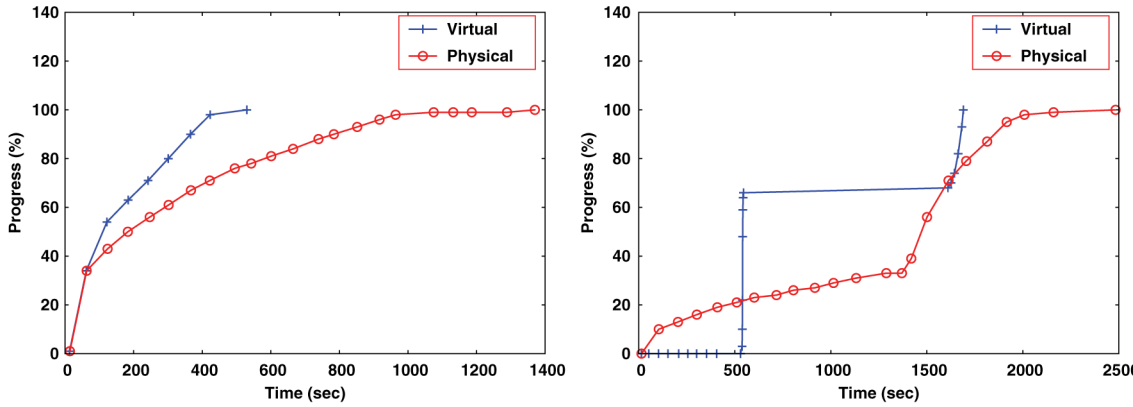


Fig. 12. Progress diagrams of terasort.

shuffling, these results adequately demonstrate better scalability of virtual shuffling for large-scale data processing.

8 Benefits to I/O and Power Consumption

Virtual shuffling is designed to alleviate the severe disk contention problem in MapReduce infrastructure. In this section, we analyze the detailed I/O behavior of virtual shuffling and compare it to physical shuffling. The benchmark used is TeraSort and we conduct the experiments on 12 slaves nodes with a fixed size of input data (250 GB).

8.1 Profile of I/O Accesses

We trace the *vmstat* output every second on all the slave nodes at run time. Table 2 shows the average total number of read and write blocks on a slave node. Physical shuffling aggressively fetches the entire intermediate to local disk for shuffling and merging, which has to use external sort. However, virtual shuffling pulls the data only when needed and merges data through memory for the reduction function, avoiding the extra disk accesses. Thus, we have significantly reduced the disk accesses for both read and write operations, by 30.9% and 36.2% respectively. The total reduction by using virtual shuffling technique is up to 34.1% as shown in the table.

Fig. 15 provides the run-time profile of read and write blocks, which indirectly reflects the number of requests issued during the program execution. The more read and write blocks are issued, the more traffic is generated which essentially increases the disk contention and hurts the performance. Note that total disk read and write blocks are different because intermediate files are read and written in different ways and these I/O activities are concurrent with those to the Hadoop Distributed File System.

When disk bandwidth is a scarce resource, high disk I/O traffic can lead to long queuing time of I/O requests which essentially degrades the performance of the original Hadoop. However, virtual shuffling is able to reduce the disk I/O traffic and support efficient data movement. Fig. 16 illustrates the number of I/O requests across the test. At the beginning of the figure, it shows the map phase, virtual shuffling shares similar number of I/O requests as current Hadoop due to all the MapTasks still materialize the map output into the disks. However, once ReduceTasks are launched, we observe significant drop in the number of I/O requests in the virtual shuffling due to its in-memory merging algorithm. In order to further understand the benefits of disk accesses reduction, we analyze the service time and the wait time of I/O requests. The service time is the time taken to complete one I/O request and the wait time includes the queuing time and the service time. Fig. 17(a) shows the details for both virtual shuffling and physical shuffling. Several I/O behaviors can be observed from this figure. First, virtual shuffling has the similar I/O

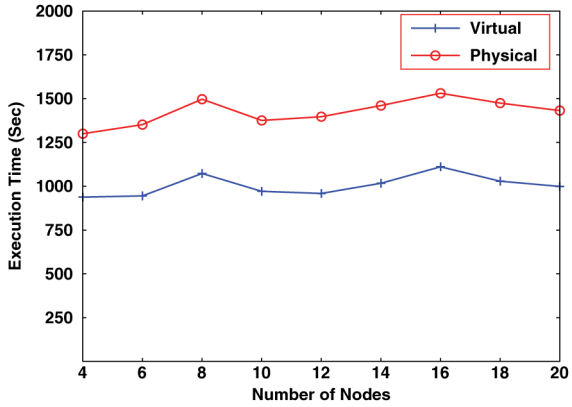


Fig. 13. Scalability with a fixed data size per node.

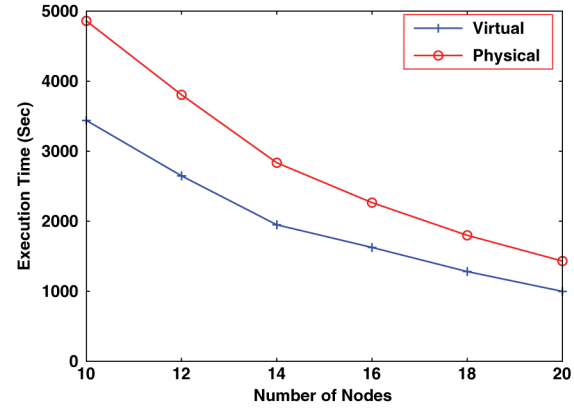


Fig. 14. Scalability with a fixed data size for the program.

TABLE 2
I/O Blocks

	READ	WRITE	Total (Kblocks)
Virtual	31,088	40,833	71,921
Physical	45,031	64,039	109,070

service time as physical shuffling. Second, virtual shuffling leads to similar or lower I/O wait time during the first 20 minutes, which correspond to the mapping phase of the execution. As the execution progresses into the reducing phase, the I/O wait time is significantly reduced. This is because virtual shuffling significantly reduces disk accesses. Third, for physical shuffling, especially during the reducing phase, most of the I/O requests spend more than 95% of their turnaround time waiting in the queue, which means the disk is not being able to keep up with the requests. On the contrary, I/O requests only spend around 40% of the total time waiting in the queue with virtual shuffling.

Taken together, the experiment demonstrates that virtual shuffling is able to efficiently alleviate disk contention and leave it in an efficient working status. It achieves so via completely eliminating the disk I/O and conducting all of the data shuffling and merging in memory, thereby significantly reducing the number of I/O requests.

8.2 Power Consumption

To examine the energy implication of virtual shuffling, we attach WattsUp PRO/ES power meters to several compute nodes and measure their power consumptions at a per-second interval. The WattsUp meter has a simple serial-USB interface that allows us to record the power profile of MapReduce programs in a fine-grained manner into a tracefile. We then plot the power profile based on the trace files. The power is recorded every second. For clarity, we plot the power consumption profile on a per-minute basis.

Fig. 18 shows the run-time profile of the power consumption per minute. The average draw for virtual shuffling was 160 watts with a standard deviation of about 8 watts, while 180 watts on average with a standard deviation 23 watts for physical shuffling. Compared to physical shuffling, the average power consumption of virtual shuffling is reduced by 12%. It suggests that, by reducing disk accesses, virtual shuffling can lead to significant savings on run-time power consumption for MapReduce programs.

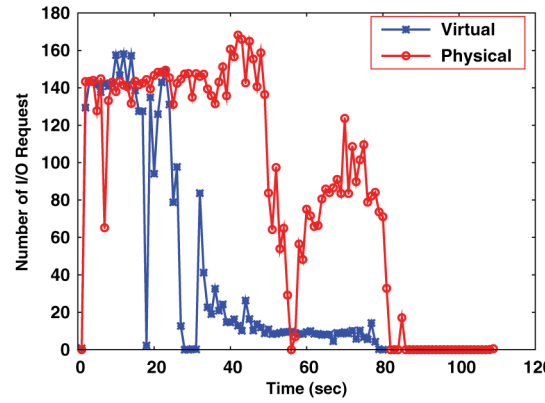


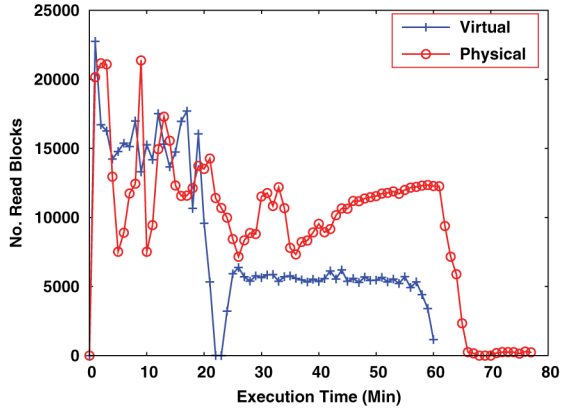
Fig. 16. Number of I/O requests.

9 Discussion

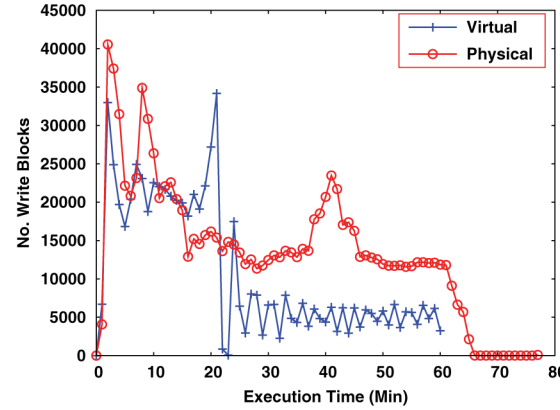
9.1 Active Subtrees

The main objective of virtual shuffling is to keep the merging process scalable with limited memory consumption and without disk spilling. With an increasing application data size, it is crucial to efficiently use and manage the memory buffers. The virtual segment table is to mimic the classic paging mechanism, which builds a global index on the description of all the segments.

When the number of subtrees grows significantly, it is unrealistic to keep all subtrees active due to the memory pressure. Thus we only allow a limited number of active subtrees. Our tree-based management of virtual segments is able to achieve high scalability. To shed light on the impact of memory management, it is important to study parameters such as the buffer size for SEBs and STBs and the size of subtrees. Thus we conduct the performance tuning experiments and analyze the impact of different data sizes as provided in Section 6. These tuning experiments help us find the optimal choices for these parameters. We adopt these parameters to achieve a smooth pipeline in shuffling, merging and reducing virtual segments while keeping the memory consumption scalable.



(a) Read Blocks



(b) Write Blocks

Fig. 15. Run-time profile of I/O accesses.

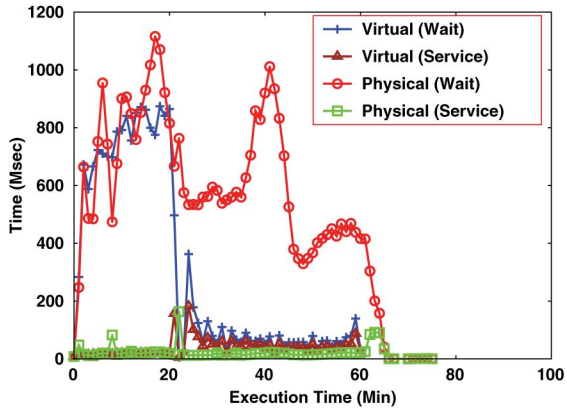


Fig. 17. Dissection of request wait and service times.

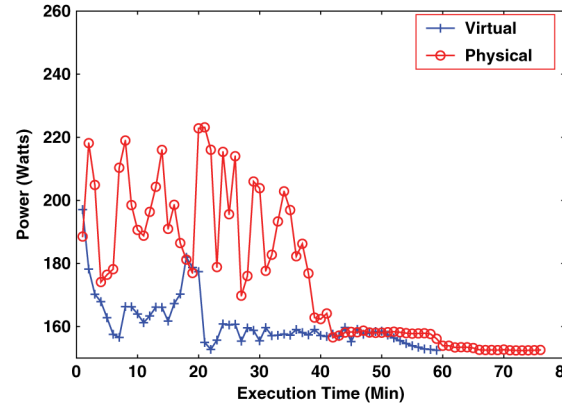


Fig. 18. Comparison of power consumption.

9.2 Fault Tolerance

In addition, the handling of execution faults is worth mentioning here. Hadoop adopts a simple fault-tolerance model through task re-execution to handle node failures. During the job execution, the JobTracker periodically communicates with TaskTrackers through heartbeat messages. If a TaskTracker fails to communicate with the JobTracker for a period of time (by default, 1 minute in Hadoop), JobTracker will assume that this TaskTracker has crashed. The JobTracker chooses another TaskTracker to re-execute all MapTasks, if it is in map phase (or ReduceTasks if it is in reduce phase), that were previously executed at the failed TaskTracker. The MapReduce online work significantly complicates the fault tolerance of Hadoop. It directly couples map tasks with reduce tasks, hence called online. A faulty reduce task means that all coupled map tasks and the associated data are potentially at fault. In contrast, virtual shuffling does not change the fault tolerance features of Hadoop, which are continued to be enabled by speculative tasks and task re-execution. If a job fails in the map phase, the JobTracker will choose another TaskTracker to re-execute the MapTasks. Meanwhile, all ReduceTasks will re-fetch the segment headers belonging to the failed MapTasks, which will be updated to the global segment table. Similarly, if a jobs fails in the reduce phase, another TaskTracker will re-execute the failed ReduceTasks and the global segment table will be created and all the intermediate data will be shuffled again. Virtual shuffling only renovates the shuffling algorithm, with which ReduceTasks determine the time and manner for fetching and merging of data segments. This algorithm is external to the TaskTrackers and does not complicate the speculative execution and re-execution of tasks by TaskTrackers.

10 Conclusions

We have proposed virtual shuffling as a new strategy to enable efficient data movement for MapReduce applications. Accordingly, we have designed and implemented virtual shuffling as a combination of three techniques including a three-level segment table, near-demand merging, and dynamic and balanced merging subtrees. Our experimental results show that virtual shuffling significantly relieves the disk I/O contention problem and speeds up data movement in MapReduce programs. In addition, it reduces power consumption of MapReduce programs by as much as 12%. For future work, we are interested in the applicability of virtual shuffling over different network protocols. we also plan to investigate the benefits of virtual shuffling for more commercial and scientific workloads on large-scale commercial cloud computing systems.

Acknowledgments

This work is funded in part by a Mellanox Grant to Auburn University, and by National Science Foundation Awards CNS-0917137, CNS-1059376, and CNS-1320016.

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. 6th Symp. Oper. Syst. Design Implementation (OSDI)*, Dec. 2004, pp. 137–150.

[2] Apache Software Foundation, *Apache Hadoop Project*. [Online]. Available: http://hadoop.apache.org/.

[3] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in Map-Reduce clusters using Mantri," in *Proc. 9th USENIX Symp. Oper. Syst. Design Implementation (OSDI)*, R. H. Arpaci-Dusseau and B. Chen, eds., Oct. 4–6, 2010, pp. 265–278. [Online]. Available: http://www.usenix.org/event/osdi10/tech/full_papers/osdi10_proceedings.pdf.

[4] P. Balaji, S. Aameek, L. Ling, and J. Bhushan, "Purlieus: Locality-aware resource allocation for MapReduce in a cloud," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal. (SC'11)*, Nov. 2011, pp. 58:1–58:11.

[5] 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. 5th Eur. Conf. Comput. Syst. (EuroSys'10)*, 2010, pp. 265–278. [Online]. Available: http://doi.acm.org/10.1145/1755913.1755940.

[6] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving MapReduce performance in heterogeneous environments," in *Proc. 8th USENIX Symp. Oper. Syst. Design Implementation (OSDI)*, Dec. 8–10, 2008, pp. 29–42. [Online]. Available: http://www.usenix.org/events/osdi08/tech/full_papers/zaharia/zaharia.pdf.

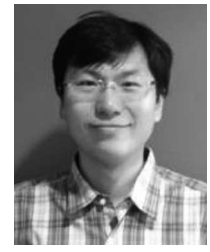
[7] G. Mackey, S. Sehrish, J. Lopez, J. Bent, S. Habib, and J. Wang, "Introducing Map-Reduce to high end computing," in *Proc. 3rd Petascale Data Storage Workshop Held Together with Supercomput.*, Nov. 2008, pp. 1–6.

[8] 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. (SC'11)*, Nov. 2011, pp. 57:1–58:10.

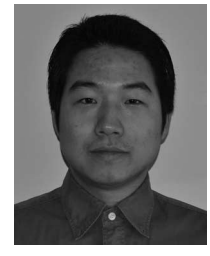
[9] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "MapReduce online," in *Proc. 7th USENIX Symp. Netw. Syst. Design Implementation (NSDI)*, Apr. 2010, pp. 312–328.

[10] D. Jiang, B. C. Ooi, L. Shi, and S. Wu, "The performance of MapReduce: An in-depth study," in *Proc. 36th Int. Conf. Very Large Data Bases (VLDB)*, 2010, vol. 3, no. 1, pp. 472–483.
[11] S. Babu, "Towards automatic optimization of MapReduce programs," in *Proc. 1st ACM Symp. Cloud Comput. (SoCC'10)*, 2010, pp. 137–142.
[12] J. Tan, X. Meng, and L. Zhang, "Delay tails in MapReduce scheduling," in *Proc. 12th ACM SIGMETRICS/PERFORMANCE Joint Int. Conf. Meas. Modeling Comput. Syst.*, 2012, pp. 5–16. [Online]. Available: http://doi.acm.org/10.1145/2254756.2254761.
[13] J. Dai, J. Huang, S. Huang, B. Huang, and Y. Liu, "HiTune: Dataflow-based performance analysis for big data cloud," in *Proc. USENIX Ann. Tech. Conf. (ATC'11)*, 2011, p. 87.
[14] H. Herodotou and S. Babu, "Profiling, what-if analysis, and cost-based optimization of MapReduce programs," *Proc. 37th Int. Conf. Very Large Data Bases*, vol. 4, no. 11, pp. 1111–1122, 2011.
[15] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. G. Greenberg, I. Stoica, D. Harlan, and E. Harris, "Scarlett: Coping with skewed content popularity in MapReduce clusters," in *Proc. 6th Eur. Conf. Comput. Syst. (EuroSys)*, Apr. 10–13, 2011, pp. 287–300.
[16] E. Jahani, M. J. Cafarella, and C. Re, "Automatic optimization for MapReduce programs," in *Proc. Very Large Data Bases Endowment (PVLDB)*, 2011, vol. 4, pp. 385–396.
[17] S.-G. Kim, H. Han, H. Jung, H. Eom, and H. Y. Yeom, "Harnessing input redundancy in a MapReduce framework," in *Proc. ACM Symp. Appl. Comput. (SAC'10)*, 2010, pp. 362–366. [Online]. Available: http://doi.acm.org/10.1145/1774088.1774167.
[18] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. 9th USENIX Conf. Netw. Syst. Design Implementation (NSDI'12)*, 2012, p. 2. [Online]. Available: http://dl.acm.org/citation.cfm?id=2228298.2228301.
[19] X. Yang, Z. Yu, M. Li, and X. Li, "Mammoth: Autonomic data processing framework for scientific state-transition applications," in *Proc. ACM Cloud Autonomic Comput. Conf. (CAC'13)*, 2013, pp. 13:1–13:10. [Online]. Available: http://doi.acm.org/10.1145/2494621.2494633.
[20] J. Leverich and C. Kozyrakis, "On the energy (in)efficiency of Hadoop clusters," *ACM SIGOPS Oper. Syst. Rev.*, vol. 44, no. 1, pp. 61–65, 2010.
[21] W. Lang and J. M. Patel, "Energy management for MapReduce clusters," *J. Very Large Data Bases (VLDB)*, vol. 3, no. 1, pp. 129–139, 2010. [Online]. Available: http://www.comp.nus.edu.sg/~vldb2010/proceedings/files/papers/R11.pdf.
[22] Y. Chen, A. S. Ganapathi, A. Fox, R. H. Katz, and D. A. Patterson, "Statistical workloads for energy efficient MapReduce," EECS Dept., Univ. of California, Berkeley, CA, Tech. Rep. UCB/EECS-2010–6, Jan. 2010. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-6.html.
[23] Y. Chen, A. Ganapathi, and R. H. Katz, "To compress or not to compress—Compute vs. IO tradeoffs for MapReduce energy efficiency," in *Proc. Green Netw.*, 2010, pp. 23–28.
[24] Open Fabrics Alliance. *OFED Resources*, 2012 [Online]. Available: http://www.openfabrics.org.
[25] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy, "Hive—A Petabyte scale data warehouse using Hadoop," in *Proc. Int. Conf. Data Eng. (ICDE)*, 2010, pp. 996–1005.

Weikuan Yu received the master's degree in developmental biology from the Ohio State University, Columbus, the bachelor's degree in genetics from Wuhan University, China, and the PhD degree in computer science from the Ohio State University, in 2006. He is an associate professor in the Department of Computer Science and Software Engineering, Auburn University, Alabama. Prior to that, he served as a research scientist at Oak Ridge National Laboratory (ORNL) until January 2009. At Auburn University, he leads the Parallel Architecture and System Laboratory (PASL) for research and development on high-end computing, parallel and distributing networking, storage and file systems, as well as interdisciplinary topics on computational biology.



Yandong Wang received the master's degree in computer science from Rochester Institute of Technology, New York, in 2010. He is a PhD student of Parallel Architecture and System Laboratory (PASL) in the Department of Computer Science and Software Engineering, Auburn University, Alabama. His research interests include cloud computing, big data analytics, high speed networking, file and storage systems.



Xinyu Que graduated from the Auburn University, Alabama. Before that, he received the master's degree in computer science from the University of Connecticut, Storrs, in 2009. He joined IBM T.J. Watson as a postdoctoral researcher. His research interests include global address space programming model and runtime system, cloud computing and mapreduce programming model, parallel graph algorithm, and next-generation memory architecture.

Cong Xu is a PhD student of Parallel Architecture and System Laboratory (PASL) in the Department of Computer Science and Software Engineering, Auburn University, Alabama. He received the master's degree from the Auburn University, in 2012, and the bachelor's degree from the Beijing University of Posts and Telecommunications, China, in 2009. His research interests include high performance computing, cloud computing, and big data analytics.