CooMR: Cross-Task Coordination for Efficient Data Management in MapReduce Programs

Xiaobing Li Yandong Wang Yizheng Jiao Cong Xu Weikuan Yu

Department of Computer Science and Software Engineering
Auburn University, AL 36849, USA
{xbli,wangyd,yzj0018,congxu,wkyu}@auburn.edu

ABSTRACT

Hadoop is a widely adopted open source implementation of MapReduce programming model for big data processing. It represents system resources as available map and reduce slots and assigns them to various tasks. This execution model gives little regard to the need of cross-task coordination on the use of shared system resources on a compute node, which results in task interference. In addition, the existing Hadoop merge algorithm can cause excessive I/O. In this study, we undertake an effort to address both issues. Accordingly, we have designed a cross-task coordination framework called CooMR for efficient data management in MapReduce programs. CooMR consists of three component schemes including cross-task opportunistic memory sharing and log-structured I/O consolidation, which are designed to facilitate task coordination, and the key-based *in-situ* merge (KISM) algorithm which is designed to enable the sorting/merging of Hadoop intermediate data without actually moving the <key, value> pairs. Our evaluation demonstrates that CooMR is able to increase task coordination, improve system resource utilization, and significantly speed up the execution time of MapReduce programs.

1. INTRODUCTION

MapReduce [7] has emerged as a popular and easy-to-use programming model for numerous organizations to process explosive amounts of data, perform massive computation, and extract critical knowledge for business intelligence. Hadoop [2] provides an implementation of the MapReduce programming model. It includes two categories of components: a JobTracker and many TaskTrackers. The JobTracker commands TaskTrackers (a.k.a slaves) to process data through two main functions (map and reduce) and schedules map tasks (MapTasks) and reduce tasks (ReduceTasks) to TaskTrackers. For convenient parallelization and scalable data processing, Hadoop-based MapReduce divides input data into many splits and assigns such splits in parallel to map tasks.

In this execution model, Hadoop represents system resources as available map and reduce slots for MapTasks and ReduceTasks. Once assigned, these tasks are taken as not only individual processing entities but also static resource containers. However, when such resource containers are assigned, there is little regard of these tasks as concurrent processing entities and the need to coordinate their changing demands on resources such as processor cores, available memory, network throughput, and disk bandwidth on a compute node. To reconcile the dependence between MapTasks and ReduceTasks, a number of Hadoop schedulers such as delay scheduler [22], coupled scheduler [18] and fast completion schedulers [20] are introduced. In addition, the LATE scheduler [24] can monitor the progress variations across different tasks, and take measures to mitigate the impact of straggler tasks. All these techniques introduce the intelligence at the schedulers and enable them to become more responsive to the changing dynamics of available resource capacity in a MapReduce environment. However, there has been little work that examines the dynamic resource allocation and sharing during execution and supports cross-task coordination at runtime among concurrent MapReduce tasks.

In this paper, we undertake an effort to examine the lack of task coordination and its impact to the efficiency of data management in MapReduce programs. With an extensive analysis of Hadoop MapReduce framework, particularly cross-task interactions, we reveal that Hadoop programs face two main performance-critical issues to exploit the best capacity of system resources. Both issues, namely task interference and excessive I/O, can be attributed to the lack of task coordination. The former can cause prolonged execution time for MapTasks and ReduceTasks; the latter can cause dramatic degradation of disk I/O bandwidth. These problems prevent the system resources on a compute node from being effectively utilized, constraining the efficiency of MapReduce programs.

Based on these findings, we have designed a cross-task coordination framework called *CooMR* for efficient data management in MapReduce programs. CooMR is designed with three new techniques to enable close coordination among tasks while not complicating the task execution model of Hadoop. These techniques are **C**ross-task **O**pportunistic **M**emory **S**haring (COMS), **LO**g-structured I/O **C**on**S**olidation (LOCS), and **K**ey-based **I**n-**S**itu Merge (KISM). The COMS component is designed with a shared memory region to increase coordination among MapReduce tasks in their memory usage. The LOCS component provides a new organization of intermediate data with a log-based append-only format. This scheme not only helps consolidate small write operations from many concurrent MapTasks, but also provides a server-driven shuffling technique for sequential retrieval of intermediate data for the shuffle phase. A new *Key-based In-Situ Merge (KISM)* algorithm is introduced to enable the merging of Hadoop <k, v> pairs without actual movement of their data blocks (values). By doing so, it offers an alternative solution to address the issue of excessive I/O caused by the current merge algorithm in Hadoop.

We have carried out an extensive set of experiments to evaluate the performance of CooMR compared to the original Hadoop. Our evaluation demonstrates that CooMR is able to increase task coordination, mitigate task interference, improve system resource utilization, thereby speeding up the execution times for both MapTasks and ReduceTasks. Such benefits have been demonstrated for a number of data-intensive MapReduce programs. Overall, CooMR is able to speed up the execution of MapReduce programs such as TeraSort by as much as 44%. The contributions of our research can be summarized as follows:

- We have carefully examined the execution of Hadoop tasks and quantified the performance issues of task interference and excessive I/O.
- CooMR is designed to enable cross-task coordination through two component techniques: cross-task opportunistic memory sharing and log-structured I/O consolidation.
- A novel merge algorithm, key-based in-situ merge, has been designed to enable the sorting/merging of Hadoop <k, v> pairs without actual movement of their data blocks (value).
- A systematic evaluation of CooMR is conducted using data-intensive MapReduce programs. Our results demonstrate that CooMR can improve the performance of these programs by as much as 44% compared to the original Hadoop.

The rest of the paper is organized as follows. Section 2 provides the background and motivation. We then describe the two main coordination techniques in Section 3, followed by Section 4 that details the key-based in-situ merge algorithm. Section 5 describes the implementation. Section 6 provides experimental results. Section 7 reviews related work. Finally, we conclude the paper in Section 8.

2. BACKGROUND AND MOTIVATION

In this section, we start with a brief description of the Hadoop MapReduce framework. Then we discuss the existence of task contention and interference among Hadoop tasks and the occurrence of excessive I/O due to the current Hadoop merging algorithm.

2.1 The Hadoop MapReduce Framework

The Hadoop MapReduce framework supports the execution of MapReduce programs in several phases: map, shuffle, and reduce. When a user's job is submitted to the JobTracker, its input dataset is divided into many *data split*s. In a split, user data is organized as many records of key value pairs, each denoted as <k, v>. In the first phase, one MapTask is launched per data split, which converts the original records into intermediate data in the form of <k', v'> pairs. These new data records are stored as a MOF (Map Output File). A MOF is organized into many data partitions, one per ReduceTask. In the second phase, each ReduceTask fetches its partition (a.k.a segment) from these MOFs. A ReduceTask needs to fetch segments from all finished MapTasks. As more remote segments are fetched and merged locally, a ReduceTask has to spill, i.e. store some segments to local disks in order to reduce memory consumption. This phase is commonly referred as the **shuffle** (or shuffle/merge) phase. In the third, or **reduce** phase, each ReduceTask loads and processes the merged segments using the reduce function. The final result is then stored to the Hadoop Distributed File System [17].

Among the three execution phases, initially multiple MapTasks in the same MapReduce job are launched as a group. The exact number is decided by the Fair Scheduler on the Hadoop production environment [4, 10, 22]. More MapTasks will be launched when some complete. Hadoop is also designed to overlap the execution of MapTasks and ReduceTasks to pipeline the processing of map and reduce functions. While these tasks are launched concurrently (some on the same node), they share computational resources such as processors, cache, memory, storage, and network.

2.2 Task Contention and Interference

Although modern commodity machines provide rich computational resources to accommodate many MapReduce tasks on a single node, our examination reveals that, when multiple data-intensive tasks are running concurrently on the same node, task interference can severely impact their execution, causing substantial performance degradation and variations. To illustrate such interference in detail, we employ *Grep* and *TeraSort* as representative compute- and data-intensive programs, respectively. We conduct experiments on a cluster of 10 nodes, among which 5 nodes host MapTasks only, and another 5 nodes for ReduceTasks. We maintain the uniformity of the input data for each task so as to avoid performance variations from skewed input.

To investigate the interference among MapTasks, we use a fixed number (10) of ReduceTasks while varying the number of MapTasks on each node. As shown in Figure 1 (a), when the number of concurrent running MapTasks on each node increases from 1 to 8, computation-intensive Grep has a slight increase of execution time. Running 8 computation-intensive tasks on each node slows down the execution by 2.2×, compared to the time for one MapTask. On the contrary, for the data-intensive TeraSort, MapTasks interfere with each other, exhibiting dramatic performance degradation with an increasing number of tasks. When 8 MapTasks are running concurrently, the average execution time is increased by as much as 14.2×.

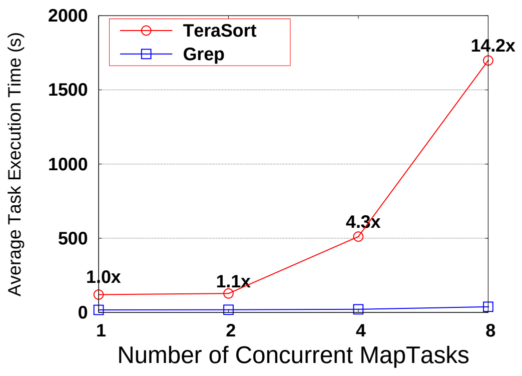
Furthermore, we also observe interference among ReduceTasks. Its impact can be even worse than that among MapTasks due to the presence of multiple phases in ReduceTasks. Because computation-intensive MapReduce programs usually have very short reduce phase and small ReduceTasks demand, we focus on data-intensive MapReduce programs for this examination. As shown in Figure 1(b), when there are only 4 ReduceTasks running concurrently on the same node, their average task execution time is slowed down by as much as 10.9×, compared to the time for one ReduceTask. Our further dissection shows that the execution times of shuffle/merge and reduce phases are increased by 11.1× and 24.8×, respectively. These results suggest that the performance degradation is caused by contention on shared disk and network resources for intermediate data shuffling, merging and reading.

A simple approach to mitigate such task interference is to allow only a small number of concurrent tasks per node for data-intensive MapReduce programs. However, the number of CPU cores and the memory capacity (24 cores with 24 GB memory in our system) are constantly increasing. Limiting MapReduce programs to a few concurrent tasks can substantially hurt the utilization of data center resources. Therefore, it is important to find an alternative solution that help task coordination in MapReduce programs to improve the utilization of shared resources.

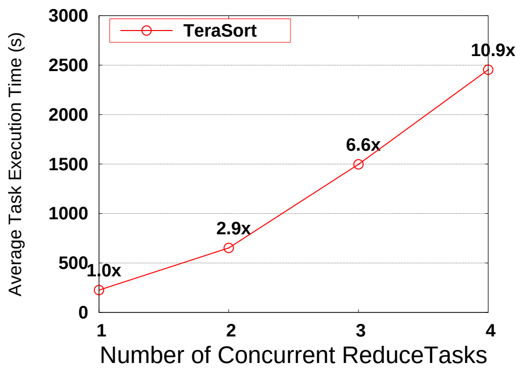
2.3 Excessive I/O During Merging

As mentioned earlier, Hadoop ReduceTasks fetch intermediate data partitions generated by remote MapTasks as local data segments, merge these segments into a stream of <k, v> pairs, and reduce them to produce final results. The merging of data segments is important to reduce the total number of files for the smooth execution of the reduce phase. However, the disk I/O operations are amplified by the repetitive merge behavior [11, 19] caused by the current Hadoop merge algorithms.

To illustrate the excessive I/O caused by the existing merge algo-



(a) MapTask Execution Time under Different Concurrency



(b) ReduceTask Execution Time under Different Concurrency

Figure 1: Task Interference in Hadoop MapReduce Runtime

Table 1: Profile of Excessive I/O During Merging

Total Number of Segments	480
Intermediate data per ReduceTask	5.69GB
Percentage of segments that are merged once	100%
Percentage of segments that are merged Twice	98.1%

rithm in Hadoop, we have conducted an experiment running TeraSort in the same environment as Section 2.2 with 120GB input data across 20 nodes. We count the number of partitions that are merged at least once and measure the data size involved in the merging process. Table 1 shows the profiling results. On average, each ReduceTask needs to fetch 480 partitions from all the MapTasks and processes up to 6GB intermediate data. Before a ReduceTask starts its reduce phase, we observe that all the partitions are merged at least once from memory to disk and up to 98.1% of partitions are merged twice. With an average intermediate data size of 5.69GB, each ReduceTask has to write such data to and read from the disks several times. Such excessive I/O aggravates the interference among tasks and delay the execution of MapReduce programs.

2.4 Proposed Solutions

In this paper, we carry out a study to address the aforementioned issues on task interference and excessive I/O of data-intensive MapReduce programs. Accordingly, we investigate the feasibility of cross-task coordination and new merging algorithms. Based on our findings, we have designed a cross-task coordination framework called CooMR for efficient data management in MapReduce programs.

As shown in Figure 2, two main components are introduced in CooMR to tackle the issue of task interference including **C**ross-task **O**pportunistic **M**emory **S**haring (COMS) and **LO**g-structured I/O **C**on**S**olidation (LOCS). The COMS component is designed with a shared memory region to increase coordination among MapReduce tasks in their memory usage. The LOCS component provides a new organization of intermediate data using a log-based append-only format. This component not only helps consolidate small write operations for many concurrent MapTasks, but also provides a server-driven shuffling technique for sequential retrieval of intermediate data during the shuffle phase. In addition, to mitigate the excessive I/O caused by the current merging algorithm, we introduce the **K**ey-based **I**n-**S**itu **M**erge (KISM) algorithm to enable the sorting/merging of Hadoop <k, v> pairs without actually moving their data blocks (values).

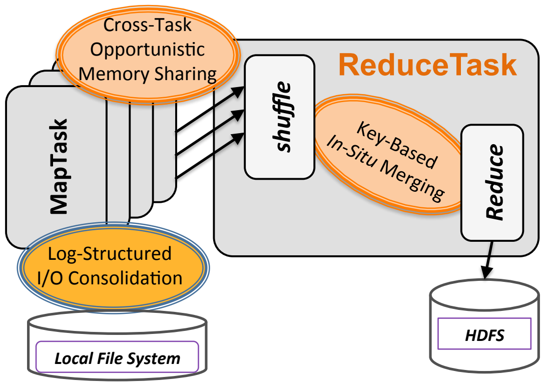

Cross-Task Opportunistic Memory Sharing
ReduceTask
shuffle
Key-Based In-Situ Merging
Reduce
MapTask
Log-Structured I/O Consolidation
Local File System
HDFS

Figure 2: Proposed Techniques in Cross-Task Coordination Framework for Efficient Data Management

3. CROSS-TASK COORDINATION

In the CooMR framework, COMS and LOCS are designed to address the aforementioned lack of task coordination in Hadoop. While these two are integral components of the framework, we describe their details separately for a clear treatment.

3.1 Cross-task Opportunistic Memory Sharing (COMS)

In the original Hadoop, each MapTask processes the input data and generates intermediate <k, v> pairs through a ring of memory buffers. Periodically, a MapTask spills intermediate <k, v> pairs from the ring buffer to local disks during its execution. As depicted in Figure 3, we have designed cross-task opportunistic memory sharing to facilitate the coordination across MapTasks. COMS is equipped with a large area of dedicated memory for cross-task sharing. When <k, v> pairs are to be spilled from MapTasks, the COMS component intercepts the data spills and buffers them in the reserved memory. At a periodic interval, COMS will sort and merge all spilled <k, v> pairs that are available in the reserved area, and then store them to the local disks through the companion component log-structured I/O consolidation (c.f. Section 3.2).

The generation speeds of intermediate data and memory requirements of MapTasks can vary substantially due to various reasons, e.g. the progress skew [13]. To avoid the situation in which some MapTasks dominates the use of shared memory and blocks the execution of other tasks, COMS is designed with a memory balancer

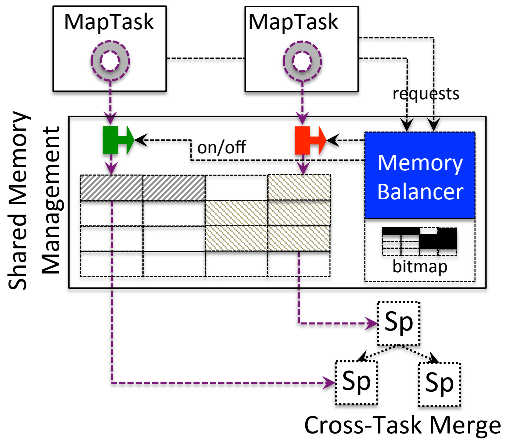


Figure 3: Cross-task Opportunistic Memory Sharing

to monitor the runtime status of each MapTask and balance the amount of memory for each task, thereby ensuring the fair use of memory and commensurate progress across MapTasks.

To achieve balanced memory usage, COMS partitions the shared memory into many equal-size units and leverage the *max-min* fair sharing [8] to allocate memory blocks to MapTasks. Each running task possesses a fair share of the available blocks. In the meantime, the memory balancer records the unit consumption of each running task. When multiple tasks are requesting the memory, the balancer always allocates an available block to the task with the minimum unit usage. When a task has a less requirement than its fair share, memory blocks can be assigned to other tasks with high demand. In doing so, the COMS component allows that a task to obtain more units than its fair share when memory is available. When all tasks are requiring more units than their fair shares, all will be allocated with a fair share of available blocks.

When a MapTask completes, its intermediate data may still be held in the shared memory for a short period of time before being flushed. This can cause later MapTasks to have less memory, detrimental to their performance. COMS addresses this issue by monitoring the memory share and the arrival/completion times of tasks. When a new task does not have its fair share and some blocks are still owned by a completed task, COMS merges more aggressively the intermediate data of completed tasks and writes to the disks.

COMS also enables memory sharing across ReduceTasks so that they can get memory from the shared memory region dynamically. In the original Hadoop, many threads per ReduceTask issue shuffling requests simultaneously to fetch the intermediate data from remote nodes to the task's local memory space. Upon memory pressure or the pressure from many small data segments [19], ReduceTask will merge some of data segments together and spill them to local disks. Without cross-task sharing, even though some ReduceTasks have sufficient memory in its heap, other ReduceTasks cannot make use of it because of the statically configured memory heaps. By enabling cross-task memory sharing, ReduceTasks are no longer limited by the size of individual memory space. The memory balancer for MapTasks is also used to control the allocation of shared memory to each ReduceTask, thereby smoothening out their execution progress.

3.2 Log-Structured I/O Consolidation

Another component designed in CooMR is the log-structured I/O consolidation (LOCS) that addresses I/O contention from concurrent tasks.

In the original Hadoop, when many tasks are running simultaneously on a node, many small random write requests are issued to disks to store intermediate data, resulting in many small dispersed write operations. Such I/O pattern can disrupt the sequential locality of disk access and adversely interfere the performance of running tasks. To overcome such interference problem raised by small random writes, LOCS is designed to consolidate many small blocks of <k, v> pairs into large blocks and append them into a single intermediate data file in a log-structured manner [16].

We design a hierarchical organization to log the intermediate data, which is shown in Figure 4(a). When there is a need to store the intermediate data, a new logical repository (Repo) is created at the tail of the log to hold the intermediate data. Each repository is partitioned into many buckets, one per ReduceTask. A bucket contains a group of sorted intermediate <k, v> records for the same ReduceTask. Since many buckets can be generated for the same ReduceTask in different repositories, these buckets can be spread into different Repos in the log. Thus, to locate the buckets that belong to the same ReduceTask, LOCS maintains a bucket index table which maps each ReduceTask to its list of buckets as shown by Figure 4(b). Each element in the list maintains the information on the location and length of the corresponding bucket in the log.

In the original Hadoop, when a shuffle request is received, the TaskTracker will find the correct partition for that ReduceTask from the specified MOF. Although our log-structured data organization consolidates I/O requests, the <k, v> pairs are no longer stored contiguously as one partition for a ReduceTask. Instead, they are dispersedly stored as different buckets in the log. In order to efficiently retrieve intermediate data for shuffling, a new design is needed to serve the shuffle requests. Otherwise, many seeks will be triggered to read small buckets for a ReduceTask.

To address this issue, we introduce a server-driven shuffling mechanism as part of the LOCS scheme. The main idea is that the TaskTracker is responsible for serving the intermediate data determines when to retrieve data and where to send the data. Thus the data serving thread in the TaskTracker always retrieves data sequentially from the latest *read start point*, which is set at the boundary of repositories toward the tail of the log. To keep track of the retrieval progress and the corresponding read start points, another table structure is maintained in LOCS. As shown in Figure 4(c). each entry in the table maps a *repository starting offset* to an array of <bucket_id, length> tuples, which share the same order as those buckets stored inside the repository. A key feature of this table is that all the elements are sorted according to the repository offsets, which follow the appending order, so that the data server in LOCS always reads from the last read start point, and sends to the corresponding ReduceTask. When the buckets for a ReduceTask are too small to efficiently utilize the network bandwidth, we aggregate buckets to improve the utilization of network.

Note that the design of server-driven shuffling mechanism within LOCS takes into account the presence of asynchronous ReduceTasks, due to task re-execution or speculative execution. In such cases, server-driven shuffling is optional, and does not conflict with the fault tolerance and speculative execution. LOCS can work by enabling server-driven shuffling for some or all ReduceTasks, except that the speculative or restarted ReduceTasks cannot receive <k, v> records via the server-driven shuffling. The on-demand mechanism is adopted instead for such ReduceTasks, i.e., they send their requests to the LOCS module and obtain records on a per-request basis. Nonetheless, LOCS works the best when it is possible for the server on the TaskTracker to opportunistically connect with all ReduceTasks and send retrieved records.

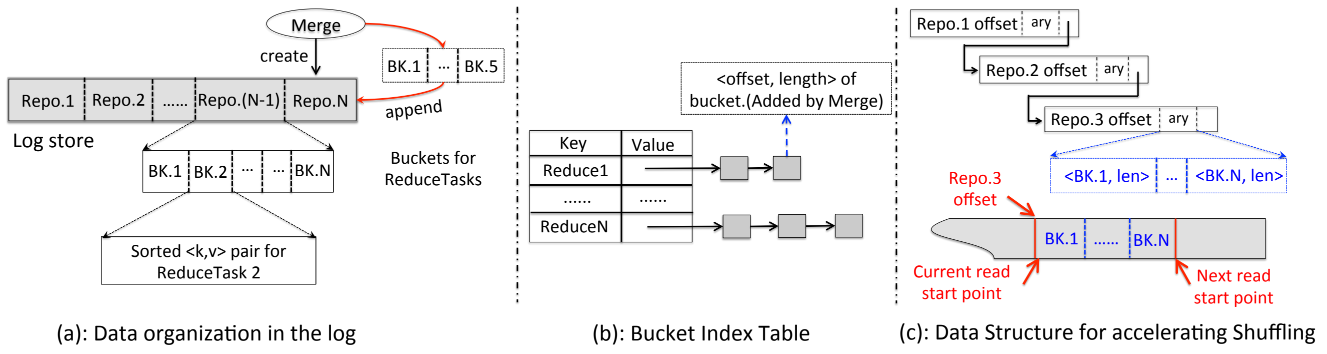


Figure 4: Log-structured I/O Consolidation

Overall, with the log-structured organization and consolidated writes and reads, LOCS provides three main benefits. First, tasks that need to write intermediate results will not be disrupted by writes from other tasks, thus alleviating the interference problem. Second, the number of write requests will be substantially reduced because of the consolidation of small writes, thereby relieving the disk bottleneck. Third, because of server-driven shuffling and its use of large sequential reads, the number of read requests will also be greatly reduced.

4. KEY-BASED IN-SITU MERGE

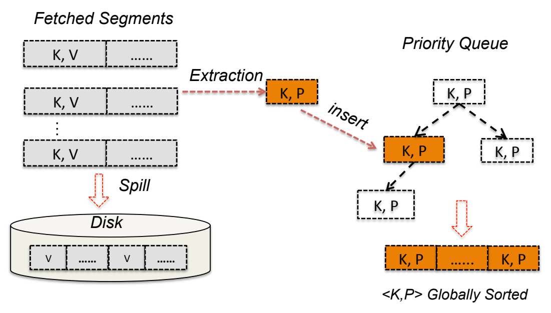


Figure 5: Diagram of In-Situ Merge

As discussed in Section 2, the original Hadoop suffers from excessive I/O due to the occurrence of repetitive merges. In order to reduce the probability of repetitive merge, it is critical to use memory wisely. Another indication of the need of better memory efficiency is that many MapReduce applications exhibit disproportionate intermediate key-value sizes. For instance, during the iteration phase of PageRank [12], it needs to find the adjacent links of each webpage. The intermediate key is the link of a webpage, while the corresponding values can be of an arbitrary size, including all the incoming and outgoing links of that webpage. In such scenarios, the existing merge algorithm in Hadoop sorts and merges each <k, v> pair as a whole. This is very inefficient for memory usage because long-length values consume a significant amount of memory but are not relevant to the operation of the merge algorithm.

In view of above issues, we introduce a memory-efficient **K**ey-based **I**n-**S**itu **M**erge (KISM) algorithm as depicted in Figure 5. Before elaborating our algorithm, we briefly describe the original merge algorithm because our algorithm inherits its concept and some of its data structures. In the current merge algorithm, several segments with sorted <k, v> pairs are combined into one large segment with all <k, v> pairs globally sorted. At the beginning of a ReduceTask, sorted segments are fetched from MapTasks. A heap (*a.k.a Priority Queue*) is constructed to merge multiple segments into a larger segment by traversing all the <k, v> pairs of the input segments. The newly created segment would be merged again with other segments in the future.

The key concept of our algorithm is to decouple the sorting of keys and the spilling of values. Since sorting only operates on the keys, we extract all keys from all segments and record the location of its corresponding value with a simple pointer *p*, which points to the offset of the value on the disk. For each <k, v> pair, a pair <k, p> is maintained in memory, as shown in Figure 5. The value of original <k, v> pair is retained in its original place, i.e., *in-situ*, and flushed to the disk with other values together when necessary. The new <k, p> pairs are inserted to the Priority Queue and finally inserted to the result. The merging result is then a special segment composed of <k, p> pairs. Since each <k, p> pair is much smaller than the original <k, v> pair, a much bigger Priority Queue can be kept in memory.

While the sorting/merging of <k, v> is handled through the new Priority Queue of <k, p> pairs, the management of data values is completely orthogonal to the merging process. The placement of values (in memory or to disks) is only determined by whether there is enough memory to hold all data. If values are spilled to the disk, it will not be read back until it is needed by the reduce function, thus completely eliminating the excessive I/O caused by repetitive merges on all the values.

The advantages offered by this approach are four-fold. First, our algorithm stores <k, p> pairs in the memory while leaving values in-situ on the disk, so more memory is spared to hold the input segments. Second, the new *p* pointer can be much smaller than the value in a <k, v> pair. It is likely that all <k, p> pairs can be completely stored in memory, causing little I/O and accelerating the merge of segments. When large data blocks for values are generated by some MapTasks, they will no longer cause excessive I/O in KISM. Our algorithm can also perform faster because its use of smaller <k, p> pairs and less data manipulation. Finally, decoupling the sorting of keys and the spilling of values allows these two processes to be optimized separately. Future techniques that accelerate disk I/O can be used for spilling without introducing side effects to sorting.

Under failure scenarios, our KISM does not affect the fault handling mechanism in current Hadoop framework either. Once detecting the failure of a single ReduceTask, local TaskTracker reclaims the files that contain data values on local disks, and JobTracker then launches another attempt of the same ReduceTask on an available node. For a machine failure, all the running tasks on such node are re-executed on other nodes, following the exact same procedure in current framework. Preemptive ReduceTask introduced in [20] provides an interesting solution to checkpoint the intermediate data, so that fault recovering can be accelerated. We have not explore this technique in the CooMR, and we intend to pursue it as a future work.

5. IMPLEMENTATION

We have implemented the three components of CooMR as described in Sections 3 and 4. The resulting CooMR framework coordinates the execution of tasks on the same node. From the perspective of global shuffling, CooMR cooperates with the JobTracker and takes into account the status of all the participating nodes in a job to determine when to use server-driven or on-demand shuffling. While the main design goal of CooMR is to reduce the task interference, it is critical that the new framework maintains the original Hadoop APIs. We achieve this by designing the CooMR as a transparent library that can be easily enabled or disabled by users without having to modify their Hadoop applications. Our previous efforts [19, 21] have demonstrated that native C is beneficial to use to reduce the overhead of Java Virtual Machine for Hadoop data management. Thus we use the Java Native Interface in CooMR to bridge the Hadoop Java code with our plug-in library.

Memory Allocation APIs: CooMR partitions the reserved shared memory as many blocks and provides a simple set of API functions for Hadoop tasks to request and release them. Two main functions: *attach* and *detach*, are provided to the upper level Hadoop tasks. With these memory access functions, MapTasks can buffer its intermediate data in the shared memory by *attach*-ing blocks. When such data is ready to be spilled, MapTasks can *detach* these blocks from the shared memory. ReduceTasks can also retrieve and release their segments by calling the same attach and detach functions.

6. EXPERIMENTAL RESULTS

In this section, we evaluate the performance of CooMR in comparison to the original Hadoop. We first demonstrate that CooMR can efficiently enhance the overall performance for a number of MapReduce programs. We then shed light on the impact of COMS memory block size. In addition, we discuss the benefits of our techniques to the execution of MapTasks and ReduceTasks. We demonstrate that CooMR can mitigate the impact of Hadoop block size on the program execution. Furthermore, we analyze the impact of CooMR on the utilization of various system resources including CPU utilization, memory consumption, network throughput, and I/O request numbers.

6.1 Experimental Setup

Cluster setup: We have conducted the experiments in a cluster of 25 nodes connected with both 10 Gigabit Ethernet and 1 Gigabit Ethernet. Each node is equipped with four 2.67 GHz hex-core Intel Xeon X5650 CPUs, 24 GB memory, and 2 Western Digital SATA hard-drivers featuring 1 TB storage space.

Hadoop setup: Many configuration parameters can substantially impact the performance of Hadoop runtime system. In general, we use the default parameter settings for our Hadoop evaluation unless otherwise specified. Table 2 lists the major configuration parameters in our experiments.

Benchmarks: We use TeraSort and WordCount from the official Hadoop release; InvertedIndex, AdjacencyList, TermVector, RankedInvertedIndex and SelfJoin from Tarazu benchmark suite [3] in our evaluation. One thing to note is that the criticality of data management is closely related to the ratio of intermediate data size and input data size. This ratio is determined not only by the map function but also by the inherent characteristics of input data. In our experiments, the ratio for the listed benchmarks ranges from 70% to 110%, except for WordCount, whose ratio is as low as 25%.

Table 2: List of Hadoop Configuration Parameters

Parameter Name	Description	Value
mapred.child.java.opts	task heap size	2GB
io.sort.mb	k-v ring buffer size	800MB
io.file.buffer.size	io buffer size	8MB
dfs.block.size	size of input split	256MB

6.2 Overall Performance Improvement

Figure 6 shows the overall performance of several benchmarks using both CooMR and Hadoop. Also provided in the figure is the factor of improvement achieved by CooMR. For the WordCount benchmark, CooMR performs comparably to the original Hadoop. This is because the low ratio of intermediate data to the initial input data. All data for WordCount can be efficiently managed within the program memory, causing little contention or interference. This test case also suggests that, for MapReduce programs that are not generating intensive intermediate data, our CooMR techniques are pretty light-weight and causing no overhead.

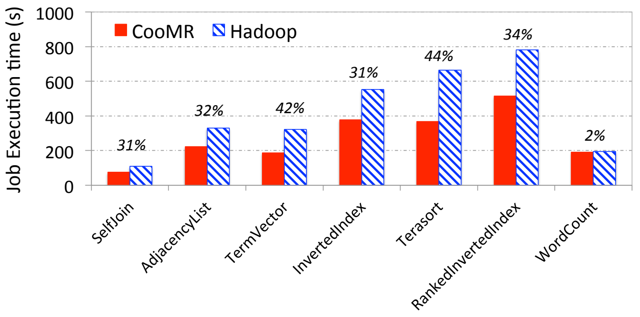


Figure 6: Overall Performance of Different Benchmarks

For the rest of benchmarks, CooMR outperforms the original Hadoop, with an improvement factor of more than 30%, up to 44% for TeraSort. Several factors together contribute to the improvement of execution time. First, these benchmarks generate a large amount of intermediate data, for which the cross-task coordination techniques in CooMR are a good fit. Second, CooMR is very effective in combining small blocks of intermediate data generated by MapTasks for log-structured I/O consolidation, thereby shortening the I/O time of MapTasks. Finally, by decoupling the sorting/merging and the movement of intermediate key-value pairs, the KISM algorithm in CooMR overcomes the overhead incurred by the original merging algorithm in the original Hadoop.

Taken together, these results demonstrate that CooMR can improve the execution of data-intensive MapReduce programs.

6.3 Tuning of Memory Block Size

As described in section 3, CooMR is designed with a key configuration parameter, called memory block size, in addition to the default tunable parameters in the Hadoop framework. This parameter, *coomr.block.size*, specifies the size of memory blocks for the cross-task shared memory. We use TeraSort as a program to tune and analyze the impact of coomr.block.size.

With smaller block sizes, the management overhead grows because of too many small blocks. Larger block sizes cause more wasted memory in some blocks, hence the utilization of shared memory becomes low. Figure 7 shows that using a block size of

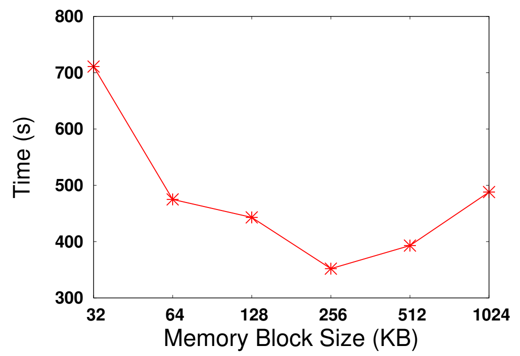


Figure 7: Tuning of Memory Block Size for CooMR

256 KB leads to the best execution time for TeraSort. In the rest of our experiments, we use 256 KB as the choice of memory block size for our CooMR tests.

6.4 Improvement on MapTask Execution

COMS and LOCS are designed to reduce task interference and improve the I/O performance of MapTasks in CooMR. To evaluate the effectiveness, we use TeraSort as a representative data-intensive application and measure its performance. For these experiments, we use a total of 10 computer nodes, 5 for MapTasks, and 20 ReduceTasks on the other 5 nodes. This configuration is chosen to avoid the performance interference incurred by ReduceTasks to the MapTasks, so we can focus on the execution time of MapTasks.

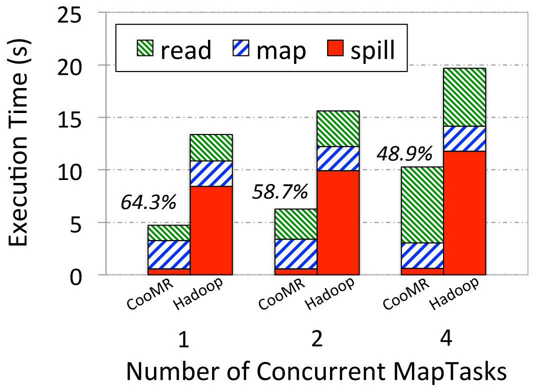


Figure 8: Improvement of MapTask Execution

We first measure the performance of concurrent MapTasks to evaluate the impact of potential interference. Figure 8 shows the average execution time of MapTasks. Compared to Hadoop, CooMR is able to reduce the execution time by 64.3%, 58.7% and 48.9%, for 1, 2 and 4 MapTasks, respectively. We further break down the execution time of MapTask into three portions (read, map and spill) based on the data processing steps during the map phase. Figure 8 shows the dissection of execution time. Compared to the default Hadoop, CooMR improves the execution time of TeraSort because of its effectiveness in mitigating task interference. By using shared memory for coordinated buffering and spilling, CooMR spends a small amount of time for spilling the data compared to its time in reading and mapping the data. In contrast, Hadoop MapTasks exhibit serious disk contention when spilling their intermediate data. We also observe that CooMR achieves less improvement with an increasing number of MapTasks. This is because there is contention among MapTasks when they are reading input data from the shared HDFS.

Hadoop MapTasks are launched repetitively and run with short durations, a dynamic behavior often described as waves of MapTasks. To achieve fast turnaround, the effective completion of MapTasks is dependent not only on the last MapTask but on a fast turnaround rate of MapTasks. We have measured the cumulative distribution function (CDF) of 240 MapTasks using CooMR. Figure 9 shows that CooMR can complete 50% of MapTasks in less than 15 seconds. The original Hadoop, however, takes 22 seconds for the same percentage.

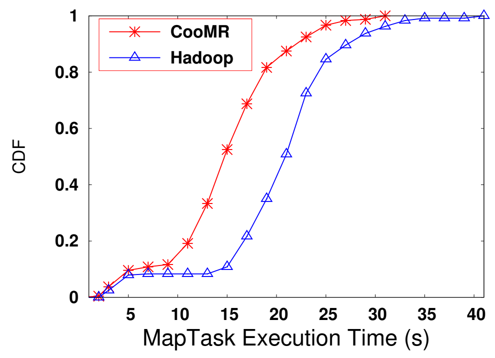


Figure 9: CDF of Progressive MapTask Completion

6.5 Improvement on ReduceTask Execution

We have also investigated the benefit of CooMR on the performance of ReduceTasks. Similar configuration as in section 6.4 is used, except that we fix 20 MapTasks on the first 5 nodes and vary the number of ReduceTasks on the other 5 nodes. In addition, given that the progress of shuffle/merge phase within the ReduceTask is strongly correlated with the progress of map phase, we launch all ReduceTasks after the completion of the map phase to achieve a fair comparison between CooMR and original Hadoop.

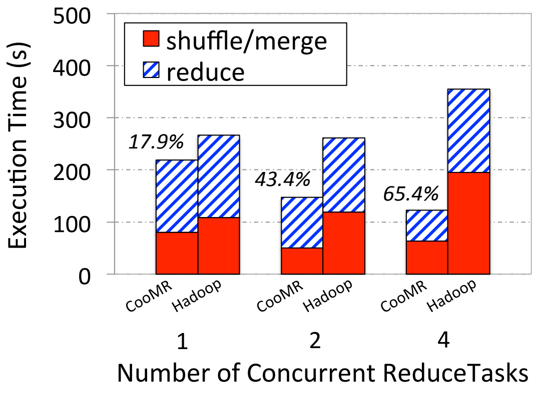


Figure 10: Improvement to ReduceTask Execution Time

As shown in Figure 10, CooMR efficiently accelerates the execution of ReduceTasks. When there are 1, 2 and 4 ReduceTasks running simultaneously on each node, CooMR outperforms the original Hadoop by 15.4%, 40.2% and 63.5%, respectively. More importantly, in contrast to the original Hadoop in which the performance of ReduceTasks becomes worse with more concurrent ReduceTasks on each node, CooMR demonstrates much better scalability. When the number of concurrent ReduceTasks on a node is increased from 1 to 4, the average task execution time is reduced by up to 42.9% in CooMR. When using Hadoop, the same execution time is increased by 32.3%. This better scalability is attributed to the new merge algorithm and the I/O consolidation in CooMR.

To further study the performance impact of CooMR on different stages of the pipeline, we have dissected the time spent on different phases during the execution of ReduceTasks. As also shown in Figure 10, compared to the original ReduceTask, CooMR drastically cuts down on the shuffle/merge time of all the ReduceTasks by up to 68.6% on average. Two factors contribute to such significant improvement. First, our in-situ merge algorithm avoids the expensive disk-based merging operations. Second, the hash-based fast retrieval of intermediate data on the MapTask side helps improve the throughput of data shuffling. In addition, Figure 10 also shows that CooMR is able to accelerate the reduce phase by as much as 57.4% on average.

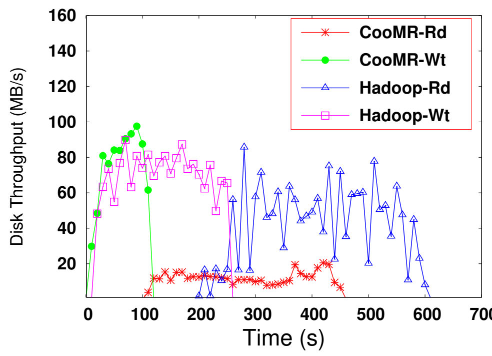


Figure 11: Profile of ReduceTask Disk Throughput

In the original Hadoop, we observe that multiple ReduceTasks can saturate the disk with I/O requests during the execution. As shown in Figure 11, under a heavy load of disk requests, the effective read and write throughputs fluctuates significantly in the case of Hadoop, especially after the start of reduce phase. In contrast, CooMR can achieve very stable throughput during the reduce phase. Because of the use of the KISM merge algorithm, ReduceTasks in the case of CooMR generate very light I/O workload to the disks.

6.6 Mitigate the Impact of Block Sizes

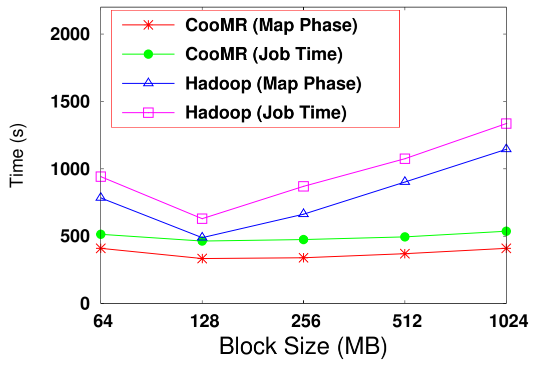


Figure 12: The Impact of Data Block Size

MapTasks in Hadoop take input data from the underlying HDFS. The input data size is determined by the HDFS block size. Tuning the block size appropriately can provide a good tradeoff between task parallelism and I/O. As shown in Figure 12, the TeraSort program achieves the best performance with a block size of 128MB under the original Hadoop. A smaller block size of 64 MB leads to poor performance because it results in a large number of MapTasks each with a small data split as input, thus increasing the scheduling overhead and reducing the chance of sequential read. The ensuing task interference and I/O contention from small writes also limits the overall performance. In addition, the performance decreases for block sizes larger than 128MB, with a loss up to 112.1% when the block size becomes 1024MB. Large blocks negatively affect the performance of data-intensive applications, due to the long intermediate data generation time of MapTasks and the resulting poor parallelism.

The techniques we have designed in CooMR can help mitigate the sensitivity of Hadoop to the granularity of input data splits. As shown in Figure 12, CooMR achieves a flat execution time for the map phase of TeraSort using different block sizes. The same trend holds for the total execution time. This is because CooMR buffers the data from MapTasks with its shared memory and writes the intermediate data as log-structured sequential I/O. Therefore, local disks can be efficiently utilized for storing intermediate data.

6.7 Profile of Resource Utilization

We have also examined the performance impact of CooMR from the perspective of resource utilization. These experiments were conducted using 10 Gigabit Ethernet. The statistics of resource utilization shown in Figure 13 is extracted from the execution of TeraSort with 200GB input on 20 slave nodes, each of which is specified with 4 map slots and 4 reduce slots. Compared to Hadoop, CooMR improves the total execution time of TeraSort by 38.7% from 643 seconds to 394 seconds. Particularly, the map execution time is improved by 42.3% from 392 seconds to 226 seconds as shown in Figure 13(b). We have conducted a detailed analysis on CPU utilization, memory consumption, I/O requests, and network throughput.

As shown in Figure 13(a), CooMR is able to achieve higher utilization of CPU in both map and reduce phases compared to Hadoop. Because of its dedicated shared memory and I/O consolidation, CooMR is able to keep the processors busy and speed up the execution.

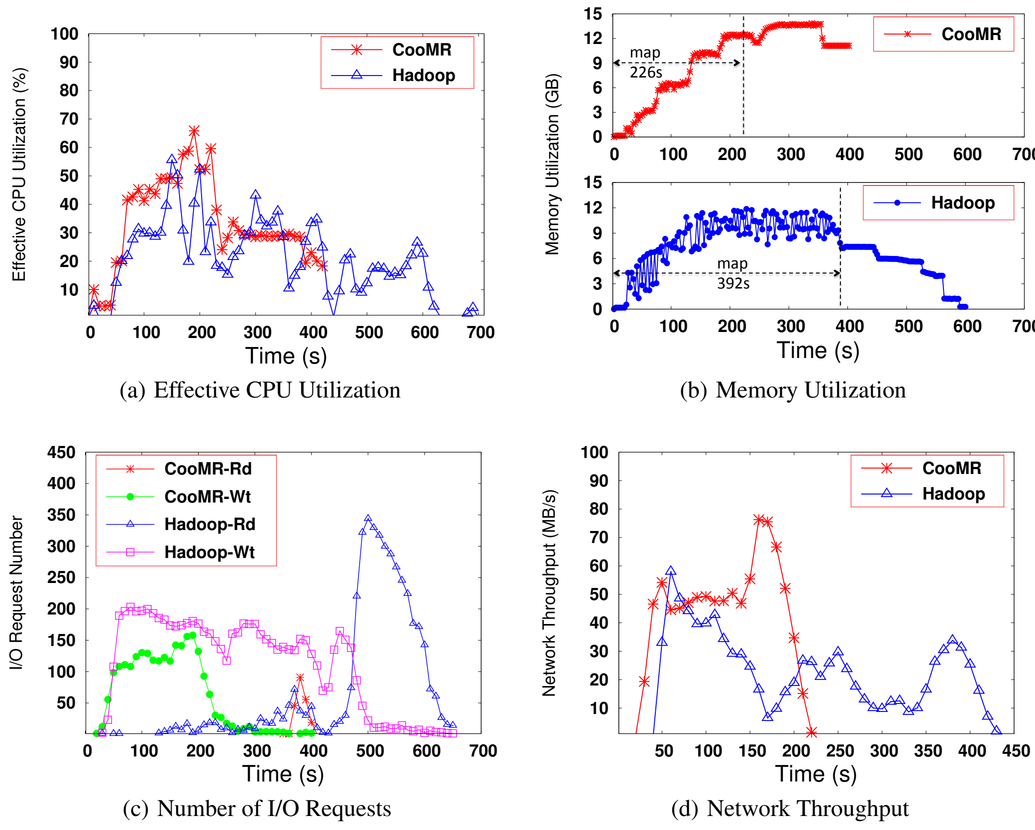
As shown in Figure 13(b), Hadoop causes a fluctuation in memory utilization during the map phase and the utilization gradually goes down in the reduce phase compared to CooMR. This is because Hadoop statically partitions the memory resource to tasks. By coordinating the use of shared memory, CooMR achieves better memory utilization, especially when compared to Hadoop during the reduce phase. In the case of Hadoop, completing ReduceTasks gradually releases their memory heap, but remaining ReduceTasks are not able to use the newly available memory.

CooMR also improves the disk utilization compared to Hadoop. We measure the number of I/O requests during the execution of TeraSort. These results are shown in Figure 13(c). Compared to Hadoop, CooMR incurs a much smaller number of read requests throughout the execution. This profile demonstrates that CooMR can dramatically reduce the number of read requests for data shuffling. CooMR reduces the number of disk reads during the reduce phase because its in-situ merging algorithm. As also shown in the figure, CooMR reduces the number of write requests. This is because of its design of log-structured I/O consolidation.

Figure 13(d) shows the profile of network throughput. CooMR delivers a network throughput 25 MB/s higher than Hadoop. During the program execution, the network throughput with CooMR is also more stable. This is due to our log-structured format for intermediate data and the server-driven shuffling technique for data retrieval. These techniques improve the coordination among shuffle requests and lead to high and sustained network throughput.

7. RELATED WORK

Condie *et al.* have introduced MapReduce Online [6] to support online aggregation and continuous queries by pushing the in-



(a) Effective CPU Utilization (b) Memory Utilization

(c) Number of I/O Requests (d) Network Throughput

Figure 13: Profile of Various Resource Utilization

termediate data to reduce side aggressively. However, it lacks a mechanism to coordinate concurrent running tasks. Hadoop-A [19, 21] introduces a new pipeline of shuffle, merge, and reduce phases via a network-levitated merge algorithm. But intermediate data generation by MapTasks can still lead to a large number of disk operations, causing severe disk contention when writing the data to local disks and serving the fetch requests. Themis [15] builds a deep pipeline to take advantage of the memory resource to accelerate data processing. In its implementation, the partition size is restricted to fit the memory space to minimize the I/O times of each KV pair. The major drawback of Themis is that the existing MapReduce programs cannot continue execution under its framework. In our work, CooMR optimizes the data process pipeline from the perspective of task coordination. It alleviates the interference among MapTasks and ReduceTasks by consolidating I/O from the upper layer and reduces I/O traffic by utilizing Key-based In-Situ Merge algorithm. Its server-driven shuffling overcomes the disk contention not addressed by Hadoop-A.

Many researchers have proposed techniques to better utilize the resources in the MapReduce cluster from the perspective of task scheduling. Delay scheduling [22] optimizes the execution of MapTasks by improving the data locality from multiple levels. When the task at the head of scheduling list cannot achieve desired data locality, Hadoop fair scheduler skips it and prioritizes the later tasks that can meet expected locality requirement. Fast completion scheduler [20] is designed to address the monopolizing behavior of long running ReduceTasks by a lightweight preemption mechanism. It avoids small job starvation issue by prioritizing ReduceTasks of jobs with the least amount of remaining workload. LATE scheduler [24] profiles the progress variations across different tasks, and mitigates the impact of straggler tasks in heterogeneous environment. These techniques focus on the improvement of scheduler and make it more responsive to the dynamics of available resource in a MapReduce environment. However, CooMR examines the the actual resource allocation, sharing and coordination among concurrent MapReduce tasks on each node to improve the system's efficiency of data processing. Bu [5] *et al.* have introduced ILA scheduling strategy for Hadoop to alleviate the task interference issue in a virtual MapReduce cluster, meanwhile solving the problem that existing Hadoop optimizations for data locality is ineffective in virtual environments. However, such solution is designed for virtual environment and still lacks the task coordination mechanism on each physical machine.

Several studies have investigated the performance of I/O issue in MapReduce. Sailfish [14] is an endeavor to address disk contention through decoupling the external sorting out of MapTasks. The kosmos distributed filesystem (KFS) is employed to stage the intermediate data. But sorting the segments on KFS triggered both disk and network contention. In addition, ReduceTasks still suffer from the repetitive I/O. Li *et al.* [11] have proposed to use hash functions instead of merge-sort to mitigate the disk bottleneck. During the shuffle phase, only append I/O operations are issued. But the recursive partitioning policy can still lead to repetitive disk I/O within the reduce phase. Different from them, our work coalesces the disk access from all local tasks and leverages a log-structured organization to improve disk utilization.

YARN [1] and Mesos [9] are two cluster management solutions designed to allow different computation and data processing frameworks, such as Hadoop, MPI, and Spark [23] to share a same pool of nodes. Spark extends the MapReduce programming model. It alleviates the burden from users to write a pipeline of MapReduce jobs by offering MapReduce abstractions as many parallel easy-to-use collections. Internally, MapReduce jobs are organized as a dataflow graph, or directed acyclic graph (DAG). However, Spark does not take into account the need of task coordination on each machine. PACMan [4] provides coordinated data access to the distributed caches according to the wave-width of MapReduce jobs so that all or none of the input splits for one job are cached. However, ReduceTasks that take input from intermediate data gain little help from this framework. Our work is complementary to the features in PACMan where we optimize all three phases of Hadoop execution to increase coordination and reduce contention.

8. CONCLUSION

Hadoop MapReduce is a proven scalable framework for processing massive-scale data on commodity off-the-shelf systems. The growing computation capability of modern commodity machines attracts more users to run many MapReduce tasks on a compute node. However, our detailed examination reveals that, due to the lack of coordination among tasks, current Hadoop suffers from severe cross-task interference when multiple data-intensive tasks are running concurrently. Moreover, excessive I/O caused by the merge algorithm during the execution of ReduceTasks further degrades the system performance. To overcome these issues, we systematically study the feasibility of cross-task coordination in this work. Accordingly, we introduce two techniques, cross-task memory sharing and log-structured I/O consolidation, to coordinate the execution of co-located tasks and consolidate their I/O operations. In addition, we introduce a key-based in-situ merge algorithm to sort/merge Hadoop <k, v> pairs without moving the actual data. Our experimental evaluation results demonstrate that our solutions can effectively improve the performance of data-intensive jobs by up to 44% and accelerate the execution of both MapTasks and ReduceTasks with better resource utilization compared to the original Hadoop.

Acknowledgments
Xiaobing Li and Yandong Wang have contributed equally to the paper. This work is funded in part by National Science Foundation awards 1059376 and 1320016.

9. REFERENCES

[1] Apache hadoop nextgen mapreduce (yarn). http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[2] Apache Hadoop Project. http://hadoop.apache.org/.

[3] Faraz Ahmad, Srimat T. Chakradhar, Anand Raghunathan, and T. N. Vijaykumar. Tarazu: optimizing mapreduce on heterogeneous clusters. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'12, pages 61–74, New York, NY, USA, 2012. ACM.

[4] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Wang, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. Pacman: Coordinated memory caching for parallel jobs. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pages 3–3, Berkeley, CA, USA, 2012. USENIX Association.

[5] Xiangping Bu, Jia Rao, and Cheng-zhong Xu. Interference and locality-aware task scheduling for mapreduce applications in virtual clusters. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, HPDC '13, pages 227–238, New York, NY, USA, 2013. ACM.

[6] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[7] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[8] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 24–24, Berkeley, CA, USA, 2011. USENIX Association.

[9] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: a platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 22–22, Berkeley, CA, USA, 2011. USENIX Association.

[10] Soila Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. An analysis of traces from a production mapreduce cluster. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 94–103, Washington, DC, USA, 2010. IEEE Computer Society.

[11] Boduo Li, Edward Mazur, Yanlei Diao, Andrew McGregor, and Prashant Shenoy. A platform for scalable one-pass analytics using mapreduce. In *Proceedings of the 2011 International Conference on Management of Data*, SIGMOD'11. ACM, 2011.

[12] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: bringing order to the web. 1999.

[13] Smriti R Ramakrishnan, Garret Swart, and Aleksey Urmanov. Balancing reducer skew in mapreduce workloads using progressive sampling. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC'12. ACM, 2012.

[14] Sriram Rao, Raghu Ramakrishnan, Adam Silberstein, Mike Ovsiannikov, and Damian Reeves. Sailfish: a framework for large scale data processing. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 4:1–4:14, New York, NY, USA, 2012. ACM.

[15] Alexander Rasmussen, Michael Conley, Rishi Kapoor, Vinh The Lam, George Porter, and Amin Vahdat. Themis: An i/o efficient mapreduce. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC'12. ACM, 2012.

[16] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, February 1992.

[17] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[18] Jian Tan, Xiaoqiao Meng, and Li Zhang. Delay tails in mapreduce scheduling. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international*

conference on Measurement and Modeling of Computer Systems, SIGMETRICS '12, pages 5–16, 2012.

[19] Yandong Wang, Xinyu Que, Weikuan Yu, Dror Goldenberg, and Dhiraj Sehgal. Hadoop acceleration through network levitated merge. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 57:1–57:10, New York, NY, USA, 2011. ACM.

[20] Yandong Wang, Jian Tan, Weikuan Yu, Xiaoqiao Meng, and Li Zhang. Preemptive reducetask scheduling for fair and fast job completion. In *Proceedings of the 10th International Conference on Autonomic Computing*, ICAC'13, June 2013.

[21] Yandong Wang, Cong Xu, Xiaobing Li, and Weikuan Yu. Jvm-bypass for efficient hadoop shuffling. In *27th IEEE International Parallel and Distributed Processing Symposium*, IPDPS'13. IEEE, 2013.

[22] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, EuroSys'10, pages 265–278, New York, NY, USA, 2010. ACM.

[23] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[24] Matei Zaharia, Andrew Konwinski, Anthony D. Joseph, Randy H. Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. Technical Report UCB/EECS-2008-99, EECS Department, University of California, Berkeley, Aug 2008.