I/O Throttling and Coordination for MapReduce

Siyuan Ma, Xian-He Sun, Ioan Raicu
Department of Computer Science
Illinois Institute of Technology
Chicago, IL 60616, USA
{sma9, sun, iraicu}@iit.edu

***Abstract*— As a leading framework for data intensive computing, MapReduce has gained enormous popularity in large-scale data analysis. With the increasing adoption of multi/many core platform, more and more MapReduce tasks are now running on the same node and sharing the same storage resources. The concurrency of tasks raises the issue of I/O stream congestion. We have observed significant throughput drops and task delays caused by I/O stream congestion in the MapReduce framework.**

In this paper, we propose two techniques to address the I/O stream congestion in MapReduce tasks. First, I/O stream throttling is presented to limit the number of concurrent I/O streams, and avoid throughput drops. Furthermore, to alleviate the I/O contention among multiple MapReduce jobs, I/O coordination orders the I/O streams in accordance to job priority. By exclusively granting I/O resources to streams with higher priorities, the coordination effectively shortens the average job completion time. Experimental results from Hadoop confirm that the proposed techniques improve the average job completion time by up to 33.74%. In addition, the proposed techniques greatly accelerate the execution of high priority jobs; thereby, showing it is capable of fostering QoS in the MapReduce framework.

Keywords- I/O stream; MapReduce; I/O scheduling; throttling; coordination

I. INTRODUCTION

MapReduce [1] is an emerging programming model for large data processing, which gained popularity due to its merits of easy programming, ad-hoc parallel processing, and fault tolerance. A MapReduce job consists of a group of map tasks (mapper) and reduce tasks (reducer). One mapper deals with one fixed-size block. Then, its output is collected and further processed by a reducer [2]. To scale up a job, one generally needs to simply add computing resources. There are two ways to increase the computing capabilities: add more computing nodes, or add more computing power per node. The latter is achieved via adopting many-core processors. The ever-growing concern in energy efficiency calls for the adoption of many-core systems in data centers and commercial clouds [3], which signifies the dominance of many-core platforms in future.

The evolution toward many core systems inevitably results in resource contention, a major factor which ultimately enlarges the execution time of applications. I/O systems, often the bottleneck of data intensive applications, are a key factor in these slowdowns. Since MapReduce tasks often start and finish in waves [17], multiple tasks tend to compete for limited I/O resources even when the total utilization of the resources is low.

This group competition raises two problems. One is a throughput drop on mechanical storage like hard disks. To enhance data locality, MapReduce adopts a large block size; this causes large sequential I/O streams. When multiple I/O streams execute on the same node concurrently, it is observed that the aggregate throughput fluctuates with a differing number of concurrent streams. These fluctuations generally cause a drop in total throughput. Another problem is I/O resource competition between jobs. As shown in Figure 1, tasks are first scheduled to the nodes holding their task-related data. For this reason, many jobs may be started on the same node. Therefore, each job will have to fight for resources amongst its peers. This competition greatly slows all the jobs involved. Our experiments show that even in a single 8 core node, I/O contention can quadruple the average completion time for map tasks, thereby greatly increasing the job response time.

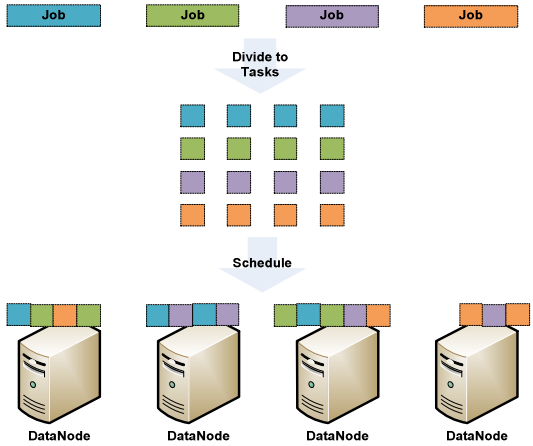

Figure 1. I/O Resource Competition among Jobs

By borrowing our experience from parallel file systems, this study applies I/O scheduling to the MapReduce Framework in order to improve average job completion time. The basic idea is that I/O Throttling reduces contention on a single node and I/O Coordination reduces average job completion time [12]. Combining the two techniques reduces the overall job execution time considerably. The solution is

tested out on Hadoop. Experimental testing on popular MapReduce benchmarks affirms the strength and potential of our solution in reducing I/O contention related job delay.

The rest of this paper is organized as follows. Section II provides the motivation of our work. In Section III, we analyze the influence of our design on job response time. Meanwhile, Section IV presents the design and implementation of our I/O throttling and coordination method. Section V evaluates the experimental results. Section VI reviews the related work. At the end, Section VII concludes this study and proposes our future work.

II. MOTIVATION AND SOLUTION

In this section, we first present the two observations about **I/O congestions** in MapReduce tasks that motivate our work. The observations suggest that I/O congestion can strikingly delay MapReduce job completion. To address this issue, we present our solution, I/O throttling and I/O coordination. The relationship of these two techniques is discussed at the end of the section.

OBSERVATION 1 *I/O Throughput Drop*

In MapReduce, almost all the non-trivial I/O streams read or write entire blocks. Benefiting from the large block size, I/O system is able to serve such streams efficiently. However, when the system serves multiple streams in parallel, an aggregate throughput fluctuation is observed. Figure 2 presents the results of a MapReduce benchmark tested on HDD (Hard Disk Drive). Each I/O stream in the benchmark will read or write one block. For read I/O streams, the increase in concurrent stream leads to a drop of total throughput.

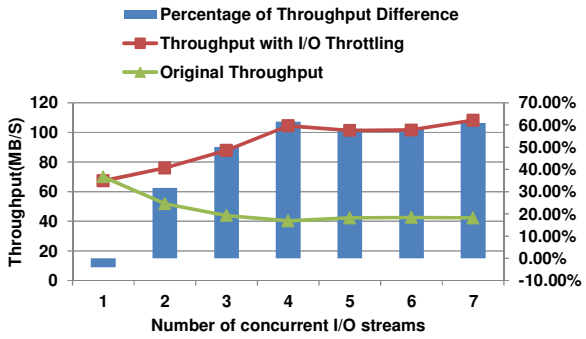

Figure 2. Throughput Improvement with I/O throttling on TestDFSIO – read Benchmark

SOLUTION *I/O Stream Throttling*

In above results, a relationship between the throughput and the number of concurrent I/O streams is observed. The throughput drop can be attributed to the mechanical structure of hard disks. The concurrent I/O streams cause non-sequential I/O accesses which reduce disk performance. Therefore, by controlling the number of concurrent I/O streams, throttling is able to avoid throughput drops caused by contention. According to the example in Figure 2, I/O throttling effectively boosts the total throughput up to 60%.

Another reason for using throttling is that a few long sequential I/O streams create almost all of the I/O in MapReduce jobs. These streams read or write an entire MapReduce block. Figure 3 gives statistics of three benchmarks handling 5 GB data. Since sequential streams maintain locality, throughput becomes more predictable.

Figure 3. Percentage of Data Processed by Long Sequential I/O Streams

OBSERVATION 2 *MapReduce Task Delay*

Contention is not rare in multi/many core systems, where memory, persistent storage, bus and network capacity are shared by all the cores. The many/multi-core systems running the MapReduce framework also suffer from the aforementioned contention. Figure 4 demonstrates such contention on an eight-core computing node running one Hadoop Terasort job. The node is equipped with 8G memory and one 7200RPM SATA hard drive.

In the graph, the red line shapes the change of average mapper completion time, which parallels the rise of task concurrency. Since each map task performs sorting upon a 128MB block, each task should take the same amount of time if there is no contention. The huge map task delay clearly indicates that contention is present in the system. In the worst case where 7 cores are in use, the task concurrency reaches above 6, quadruples the map task length, and prolongs the Map Phase by 3 times. The Map Phase is the time period from the start of the first Map task to the end of the last Map task. The Mapper delay indicates that the Mapper completion time increases from one core to multiple cores. In order to focus on the effect of contention, the Reduce tasks are intentionally neglected due to their dependency on Map tasks.

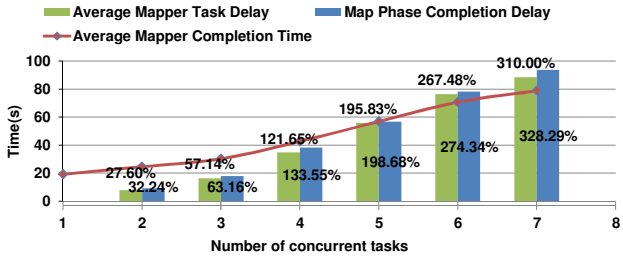

Figure 4. One Terasort Job Running upon a Single DataNode with 128MB blocks (HDD)

Though there are many resources shared among tasks, the I/O system has the largest contribution to task delay. To underpin this point, the above test was repeated using a PCI-E X4 SSD (Solid State Disk). As shown in Figure 5, the average task completion time fluctuated slightly, hereby permitting great scalability duo to the removal of the bottleneck in the I/O system. On the other hand, the task concurrency suggests low compute resource utilization. This is because the overhead of MapReduce framework becomes significant when task execution time is reduced. An increase of block size could reduce the proportion of framework overhead within the task length. So, the core utilization could be improved by setting a larger block size.

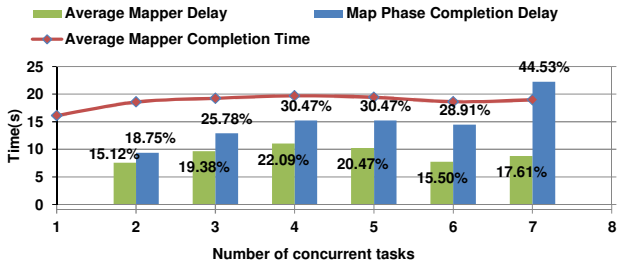

Figure 5. One Terasort Job Running upon a Single DataNode with 128MB blocks (SSD)

To further clarify this point, the TestDFSIO benchmark was used to solely test the I/O degradation brought by contention. In Figure 6, the benchmark starts a MapReduce job with little computation but significant I/O in each task. For the read test, 128MB was initially chosen as the file size read for a task. In the test, the task shows no obvious completion time difference as the number of jobs per node is increased. This is due to the task completing too fast to maintain enough concurrency. Therefore, a larger file was chosen. In this test, a similar performance was observed. As predicted, the higher the concurrency, the longer the task execution time. Depending on the amount of concurrency that can be achieved, the write test experiences task delays up to 267% due to I/O system contention.

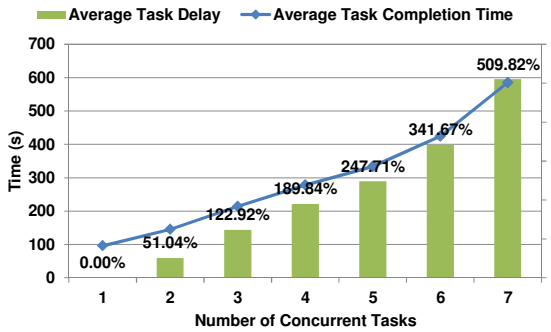

Figure 6. TestDFSIO –write benchmark

SOLUTION *I/O Stream Coordination*

The above observations suggest that by controlling contention, limiting the number of concurrent I/O streams, the ability to accelerate certain tasks at the expense of others is gained. If we can accelerate tasks belonging to one job on all the nodes, then the response time of the job will be improved.

Admittedly, I/O Coordination is not useful for systems dedicated to one job. The reason is the absence of job contentions. However, a shared environment is a different story. When multiple jobs are executing concurrently, the contention can be devastating to execution time. The I/O system treats tasks from various jobs as "one job", and slows them equally. As shown in Figure 7, where a task consists of 5 seconds of I/O and 10 seconds of computation, the number of slots caps is equal to the number of concurrent tasks. Therefore at most four tasks can be executed simultaneously on each node. The graph depicts three jobs, A, B, and C, each has a varying number of tasks, running upon the four nodes. Due to the sharing of I/O resources, the I/O time for all tasks is increased by three times and doubles the job execution time to 30 seconds. In this scenario, all tasks start at the same time, and I/O operations are performed at the end of each task.

In the example above, though each task is able to grab a fair share of I/O resources by default, a general slowdown is unnecessary. Through I/O stream coordination, we are able to reduce the task delay caused by contention. By coordination, tasks with a higher priority can exclusively access I/O resources before other low priority tasks. In this case, Job A will first process the I/O resource exclusively, before releasing it to other jobs. So, the resource competition only doubles the I/O time of tasks in Job A. Coordination saves 10 seconds for Job A without delaying the other jobs. Similarly, B1 saves 5 seconds by blocking tasks of Job C during its I/O. On average, I/O Coordination reduces the average job completion time by 1/6.

Relationship *of I/O Throttling and I/O Coordination*

On one hand, these two techniques work for different purposes. I/O Throttling tries to avoid throughput drops due to I/O stream contention; I/O coordination is used to reduce job delay caused by contention. On the other hand, these two techniques are complementary. I/O throttling suggests the number of I/O streams that optimizes system throughput. I/O coordination enforces orders to select streams by priority. By combining the two techniques, we are able to:

- Reduce the average job completion time.
- Shorten the response time for high priority jobs.

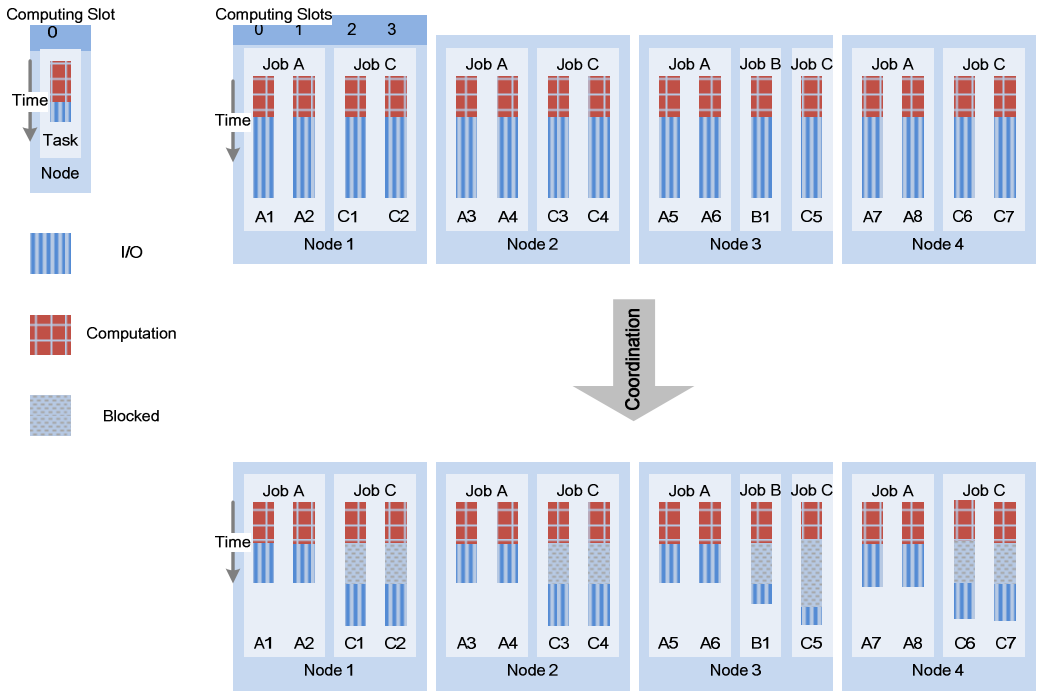

Figure 7. An example of I/O Coordination

III. Analysis on I/O Coordination and Throttling

In this section, we begin by analyzing the components that determine the job completion time and task completion time. Next we discuss how I/O coordination and throttling take effect. Finally, factors and techniques related to the proposed method are investigated. To begin the analysis, we define the following terminologies:

a) System state S involves the runtime state, static resource characteristics, and submitted jobs. It is a shared variable between all the tasks. Any change on task scheduling and resource usage policy will yield a different system state. For convenience, let S_0 be the system without resource contention.

b) An uppercase letter is used to represent a job, and a letter with a number to stand for a task. For instance, A and B are two jobs, and A1, A2, A3 are tasks of job A.

c) $C(A, S)$ is the expected number of concurrent tasks in job A running on system S.

d) $N(A)$ is the number of tasks forming job A.

e) $T(A_i, S)$ indicates the completion time of task A_i on system S; $T_{task}(A, S)$ is the average completion time of tasks belonging to job A on system S; $T(A, S)$ represents the the completion time of job A.

When all terms in one formula refer to the same system, the system variable is neglected for neatness.

A. *Anatomy of MapReduce Job Completion Time*

By definition, job A's completion time is

$$T(A,S) = T_{task}(A,S) \times \frac{\mathrm{N}(A)}{\mathrm{C}(A,S)} \qquad (1)$$

Apparently, the larger the concurrency, the faster the job will finish. Unfortunately, coordination and throttling has almost nothing to do with this factor. The task scheduler and the job structure mostly determine the expected task concurrency of a job.

The default task scheduler works in a first-come-first-serve manner, and dedicates the system to at most one job at any time. Hence if system S has enough resource, $\mathrm{C}(A, S) = \mathrm{N}(A)$. While other schedulers may raise the resource utilization by running multiple jobs simultaneously. Therefore, $\mathrm{C}(A, S)$ is generally smaller than $\mathrm{N}(A)$.

Job structure describes the dependency between map and reduce tasks. The dependencies are caused by the fact that reduce task can only be launched after certain percentage of map tasks (e.g. 90%) have completed. In Hadoop, mapred.reduce.slowstart.completed.maps can be set to adjust this percentage. When the value is small, more map tasks and reduce tasks can be launched at the same time, which implies a larger $\mathrm{C}(A, S)$. Because the execution of reduce tasks requires the output data from map tasks, if only a few map tasks have finished, some reducers cannot proceed. The reducer will still occupy certain amount of resources and possibly prolonging the average task length. For simplicity,

we assume both the task concurrency and structure of a job are unchanged in our analysis.

B. Task Completion Time and Stretch Factor

Let's shift our focus onto $T_{task}(A)$. Certainly, we have $T_{task}(A,S) = \frac{\sum_i T_{task}(A_i,S)}{\mathrm{N}(A)}$

The motivation section has showed that the contention on a single node mostly lies in its local I/O system. This fact suggests a division of completion time based on I/O. Therefore, when task j is running without contention, $T_{task}(A_j)$ can be dissected into two parts, non-I/O and the I/O parts.

$$T_{task}(A_j,S_0) = T_{non-I/O}(A_j,S_0) + T_{I/O}(A_j,S_0)$$

$T_{non-I/O}(A_j)$ involves normal computation and network communication, which is immune to the influence of local I/O contention. Let $\mathrm{ST}(A_j,S) = \frac{T_{I/O}(A_j,S)}{T_{I/O}(A_j,S_0)}$ stands for the I/O stretch factor for the task A_j running on system S. Apparently, $\mathrm{ST}(A_j,S_0) = 1$. The above formula simplifies into

$$\mathrm{T}(A_j,S) = T_{non-I/O}(A_j,S) + T_{I/O}(A_j,S_0) \times ST(A_j,S)$$

C. An Example for I/O Coordination

Assume throughput is constant in Figure 7 and job priorities satisfy$\mathrm{Priority(A)} > Priority(\mathrm{B}) > Priority(\mathrm{C})$. All tasks then have $\mathrm{ST}_{\mathrm{I/O}}^{\mathrm{task}}(\mathrm{task}) = 4$ before coordination. After coordination, the stretch factor of task $\mathrm{A1} - \mathrm{A8}$ drops to 2. Furthermore to measure the stretch factor change for other tasks, the blocked time is considered as part of the I/O time, hence $\mathrm{ST}_{\mathrm{I/O}}^{\mathrm{task}}(\mathrm{B1})$ declines to 3. The stretch factors for other tasks remain unchanged. Note that if the blocking time becomes large, $\mathrm{ST}_{\mathrm{I/O}}^{\mathrm{task}}(\mathrm{task})$ can exceed its original value. This phenomenon has three prerequisites:

1) The task scheduler keeps launching high priority tasks
2) The node always has free slots to hold the new task
3) The data used by the new task exists on the current node. The extra delay is warranted because it allows system to sacrifice the low priority tasks to accelerate other high priority tasks.

D. How does I/O Throttling Work

Given I/O throughput is constant and computing slots are **fully utilized**, then

$$\frac{\mathrm{size(data\ accessed)}}{\sum_{\mathrm{t\ on\ Node}} \mathrm{T_{I/O}(t,S_{orig})}} = \frac{\mathrm{size(data\ accessed)}}{\sum_{\mathrm{t\ on\ Node}} \mathrm{T_{I/O}(t,S_{coord})}}$$

$\mathrm{S_{orig}}$ and $\mathrm{S_{coord}}$ stand for the system before and after coordination. By replacing $\mathrm{T_{I/O}}$ with the I/O stretch factor, then

$$\sum ST(\mathrm{t},S_{orig}) = \sum ST(\mathrm{t},S_{coord}) \qquad (2)$$

This implies that the total time tasks are blocked for coordination is equal to the time saved in precedent tasks. In such cases, I/O coordination cannot decrease the average task completion time.

I/O throttling works when throughput is not a constant. The motivation section states that a different number of concurrent I/O streams may yield different throughput. Therefore, I/O stream throttling, is necessary to avoid possible throughput drops by coordination. And by throttling in extreme contention cases, the increases of throughput can further reduce the stretch factor for all tasks. With expected I/O throttling, formula (2) would evolve into

$$\sum ST(\mathrm{t},\mathrm{S_{orig}}) \geq \sum ST(\mathrm{t},\mathrm{S_{coord+throttling}})$$

E. How does I/O Coordination Work

The purpose of I/O coordination is to reduce the I/O stretch factor for tasks of high priority jobs. By cutting down the stretch factor from $ST(t,S_{orig})$ to $ST(t,S_{coord})$, the time saved for a given task is

$$\mathrm{T}(task,S_{orig}) - \mathrm{T}(task,S_{coord}) = T_{I/O}(task,S_0)\ (ST(task,S_{orig}) - ST(task,S_{coord}))$$

$$\mathrm{P_s(task,S_{orig},S_{coord})} = \mathrm{P_{I/O}(task)(ST_{I/O}^{task}(task,S_{orig}) - ST_{I/O}^{task}(task,S_{coord}))}$$

To analyze the acceleration for one job, we introduce

1) I/O portion of the task
$$P_{I/O}(task) = T_{I/O}(task,S_0)/T(task,S_0)$$
2) Average drop of I/O stretch factor for tasks in job A
$$\Delta\mathrm{ST(A,S_{orig},S_{coord})} = \sum_{\mathrm{j}\in\mathrm{A}}(\mathrm{ST(j,S_{orig})} - \mathrm{ST(j,S_{coord})})/\mathrm{N(A)}$$
3) Average task slowdown in job A
$$\mathrm{SLOWDOWN(A,S_{orig})} = \mathrm{T_{task}(A,S_0)/T_{task}(A,S_{orig})}$$

Assume all tasks in job A have the same amount of I/O $T_{I/O}(A,S_0) \equiv T_{I/O}(task,S_0)$ and the same proportion of I/O $\mathrm{P_{I/O}(A)} \equiv \mathrm{P_{I/O}(task)}$. Since task concurrency and structure of a job won't be changed by above assumptions, coordination would reduce the average task execution time by the following percentage:

$$\mathrm{P_s(A,S_{orig},S_{coord})} = \frac{\sum_{\mathrm{j}\in\mathrm{A}}(\mathrm{T(j,S_{orig})} - \mathrm{T(j,S_{coord})})}{\mathrm{N_{task}(A)} \times \mathrm{T(A,S_{orig})}} = \mathrm{P_{I/O}(A)} \times \Delta\mathrm{ST(A,S_{orig},S_{coord})} \times \mathrm{SLOWDOWN(A,S_{orig})} \qquad (3)$$

According to formula (1), $\mathrm{P_s(A,S_{orig},S_{coord})}$ is also the percentage of time saved for the entire job A.

F. Influential Factors

1) The number of concurrent tasks on each node: Since MapReduce tasks perform synchronized I/O, the number of concurrent tasks is the expected upper bound of $ST(task,S_{orig})$.

2) The priority of the job: Tasks in high priority job tend to have smaller $ST(task)$ after throttling and coordination.

3) The I/O portion of a task measured in a contention free environment, $P_{I/O}(A)$: Formula (3) suggests that $P_{I/O}(A)$ is linear to $T(A, S_{orig}) - T(A, S_{coord})$. This fact meets our intuition that the proposed method works under I/O intensive scenarios.

4) The number of active jobs: When fewer jobs are active, chances will rise that tasks running upon the same node belong to the same job. In this case, the coordination may become less effective due to less contention among jobs.

5) Storage Hardware: The speed of the storage system directly affects the I/O portion of any MapReduce task. An upgrade from a single HDD to RAID or SSD can decrease the $P_{I/O}(A)$ in proportion to the bandwidth increase. Furthermore, with the reduction of task completion time, the task concurrency will also drop accordingly. This was witnessed in the motivation section. Since the expected I/O concurrency of a task cannot exceed the task concurrency, the rise of storage speed will also cause an adverse effect upon the $ST(task, S_{orig})$. Generally speaking, the adoption of faster storage can reduce the I/O contention; hence making I/O coordination less effective for MapReduce. Because the usage of I/O throttling strongly relies on the characteristics of the storage hardware and non-mechanical storages like SSD are less sensitive to I/O stream concurrency than HDD, I/O throttling will be rendered less useful.

6) CPU Speed: In opposition to the increased speed of storage, a faster slot (core) will shorten the computation time in a task, hence increasing the $P_{I/O}$. Under intensive workload, rise of $P_{I/O}$ will lead to higher $ST(task, S_{orig})$. Therefore, fast computation renders the I/O coordination more effective by introducing more contention.

7) Block Compression: As stated in [14], block compression shifts the bottleneck from I/O to CPU. It can reduce both $P_{I/O}(A)$ and $ST(task, S_{orig})$. But it has two major drawbacks. One is that the compression only works for compressible data. Another is the computation overhead it introduces. Therefore, the adoption of block compression is somewhat limited. Meanwhile, our method works without such a limitation and it will work along with the block compression since the technique won't remove all I/O contention on platforms with increasing number of cores.

8) Network Satuation: the load imbalance and data transfer between mappers and reducers can saturate the network bandwidth. In this case, $ST(task, S_{orig})$ will be enlarged since the flows of I/O streams are blocked on the network. Throttling down the number of concurrent I/O streams can alleviate the situation. In the future work, we prepare to distinguish local I/O streams and remote I/O streams to further address this issue.

IV. Design and Implementation

A. I/O Coordination

As mentioned in the motivation, the coordination grants MapReduce tasks with higher priority exclusive access to I/O resources, and blocks accesses from the lower priority tasks. So, the first work of coordination is to order the I/O streams. The stream priority is determined by the priority of corresponding jobs set by the user. For instance, if Job A has a higher priority than Job B, then the I/O streams of A will be able to access I/O resources easier than Job B. For jobs with the same priority, the streams, whose job started earliest, will have higher priority during coordination. Therefore, the priority of a stream s can be set as a triple:

$$\text{priority}(s) = (s.preempted, s.\text{Task.Job.priority}, s.\text{Task.Job.startTime})$$

In reference to Figure 7, each active I/O stream can only stay at one of three states. These are IDLE (Computation), BLOCKED, and Being Served (I/O). The work of coordination includes:

1) Maintain two lists of active I/O streams $\{R_i\}$ and $\{W_i\}$, ordered by priority.

2) Guarantee the rule: In either stream list $\{S_i\}$, if $priority(S_i) < priority(S_j)$, then S_j is BLOCKED implies S_i is BLOCKED.

$s.preempted$ is *false* by default. To avoid job starvation, $s.preempted$ is set *true* when the stream has been BLOCKED for too long.

B. I/O Throttling

Observations of throughput fluctuations motivated the idea of I/O Throttling. The complexity of real MapReduce workloads invalidates the throughput control by directly manipulating the number of streams. As presented in Figure 8, an I/O stream could stall at waiting for data to write, or waiting for buffers to be empty. In either case, the stream will enter the IDLE state. The duration of IDLE is normally short for write streams since programmers tend to let reducers do less, and mappers do more, because it increases the scalability.

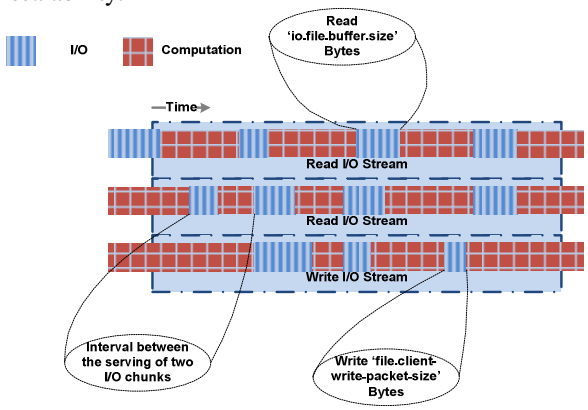

Figure 8. I/O stream dissection

To perform throttling, the following calculations are required:

1) For any active stream S_i, keep a queue of the lengths of the last ten time intervals that are depicted in Figure 8.

2) Calculate the average value of the elements in the queue $E_{int}(S_i)$. The throughput contributed by S_i then equals $E_{int}(S_i)^{-1} \times chunk.size$.

C. Algorithm

Given	the number of prioritized jobs N_{p-job}
	maximum BLOCKED time for a stream T_0
	chunk size C
	throughput estimate function f
	estimate function error e
Input	m active I/O streams $S_1, S_2, \ldots, S_m$, which are ordered by priorities
Trigger	before each chunk access in S_i; after each chunk access in $S_1, \ldots, S_{i-1}$
Steps	
1	If($S_i.preempted = true$) enter BEING_SERVED state
2	If($S_i.BLOCKED.time > T_0$) set $S_i.preempted = true$; Goto step 1
3	If($\#jobs\ in\ S_1, \ldots, S_{i-1} \leq N_{p-job}$) enter BEING_SERVED state
4	If($(1+e)f(S_1, \ldots, S_{i-1}) \geq f(S_1, \ldots, S_i)$) enter BLOCKED state
	Else enter BEING_SERVED state

D. Implementation

Our implementation in Hadoop only involves modifications to the code of DataNode. As presented in Figure 9, we mainly deal with three classes, DataXceiver, BlockSender and BlockReceiver.

DataXceiver.java

```
readBlock(...) {
    ...
    blockSender = new BlockSender(...);
    blockSender.TCEnabled = true;
    blockSender.sendBlock(...);
    ...
}

writeBlock(...) {
    ...
    blockReceiver = new BlockReceiver(...);
    blockReceiver.TCEnabled = true;
    blockReceiver.receiveBlock(...);
    ...
}
```

BlockSender.java

```
sendBlock(...) {
    ...
    while(endOffset > offset) {
        if(TCEnabled)
            coordination();
        len = sendChunks(...);
        offset += len;
    }
    ...
}
```

BlockReceiver.java

```
receiveBlock(...) {
    ...
    while(receivePacket()>0);
    ...
}

receivePacket(){
    readNextPacket();
    if(TCEnabled)
        coordination();
    out.write(pktBuf, dataOff, len);
}
```

Figure 9. Modification in Hadoop

The DataXceiver contains two methods, readBlock() and writeBlock(), to read and write a block respectively. The readBlock method then hinges on a BlockSender instance to perform the read operation. Finally, calling the sendBlock method, the BlockSender instance reads the block to an output stream chunk-by-chunk. The sendBlock method is basically a loop of sendChunks(). The maximum data performed by sendChunks() is capped by io.file.buffer.size, 64KB by default.

In our implementation, we perform coordination before every sendChunks call to determine whether the current task owns proper priority to make the call, or if it should step aside for higher priority I/O streams. A similar modification is made in writeBlock(), which will create a BlockReceiver instance to perform the write operation.

The coordination procedure itself is a critical section. Assuming n concurrent streams and m chunks for each stream, the coordination time complexity is $O(n)$. Each stream calls it $O(mn)$ times. Therefore, the total time overhead is $O(n^3 m)$. Consider that n is proportional to the number of cores per node, which is small, thereby the overhead is linear to the size of data to be accessed.

V. EXPERIMENTS

We conduct our experiments upon a 65-node SUN Fire Linux based cluster. It involves one head node and 64 computing nodes. All nodes are equipped with Gigabit Ethernet interconnection. The head node's model is Sun Fire X4240. It is equipped with dual 2.7 GHz Opteron quad-core processors, 8GB memory, and 12 500GB 7200RPM SATA II disk drives configured as RAID5 disk array. The computing nodes are Sun Fire X2200 servers. Each has dual 2.3GHz Opteron quad-core processors, 8GB memory, and a 250GB 7200RPM SATA hard drive. All 65 nodes are connected with Gigabit Ethernet, and run Ubuntu 9.04 (Linux kernel 2.6.28.10) operating system.

A. Hadoop Configuration

The implementation is based on Hadoop 0.20.204.0. To investigate the I/O resource competition among jobs, we adopt *the* FAIR scheduler as the task scheduler. The runtime system has one NameNode and one Secondary NameNode working on a dedicated node. Each computing node holds one TaskTracker and one DataNode. Assuming n is the number of cores available, each experimental workload launches 4n mappers in order to test with enough load intensity. Meanwhile to immediately start the reducers once a mapper finishes, we set the number of reducers in one job as $\min(\#\text{Computing Nodes}, \#\text{map tasks})$. Since the throttling and coordination will block some I/O operations and cause a timeout using the default Hadoop configuration, we disabled the timeout in our experiments.

TABLE I. BENCHMARK SUMMARY

Benchmark	*Description*	*I/O Intensity*
TestDFSIO	Benchmark that tests HDFS throughput	Pure I/O
Terasort	Benchmark that sorts large set of data	High
wordcount	Benchmark that counts the occurrences of any words in a given text file	Low

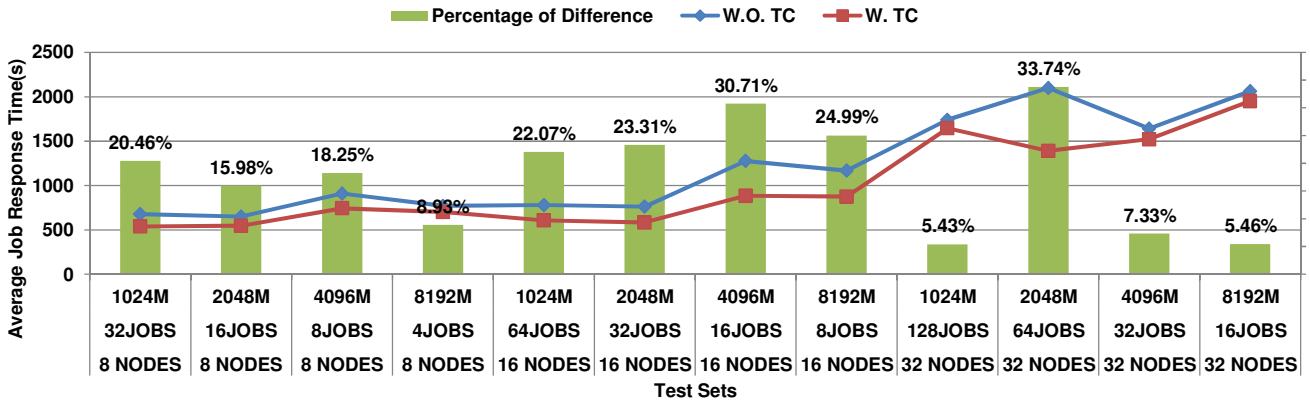

Figure 10. Terasort Benchmark

B. *Experiments Setup*

The experiments are performed at three different scales: 8, 16 and 32 nodes. Therefore, each scale setting has 64, 128, and 256 cores, respectively. Three benchmarks, TestDFSIO, Terasort and wordcount were used to measure the performance. In analyzing the experiments, we compare the results between two sets of experiments. One disables the coordination and throttling (W.O. TC); the other enables them (W. TC). Both adopt the same configuration. Performance metrics include average job completion time, average task completion time, deviation of task length, and job completion time distribution. The comparison gives a positive feedback to the proposed method. In all the experiments, N_{p-job} equals 2 and e is set to 30%.

C. *Average Job Response Time*

One major purpose of our work is to reduce the average response time of MapReduce jobs. So we tested our design with the Terasort benchmark in three different platform scales. Each of the test sets is solely identifiable by the job size and the number of jobs. Figure 10 shows that the improvement on average job response time ranges from 5.43% to 33.74%.

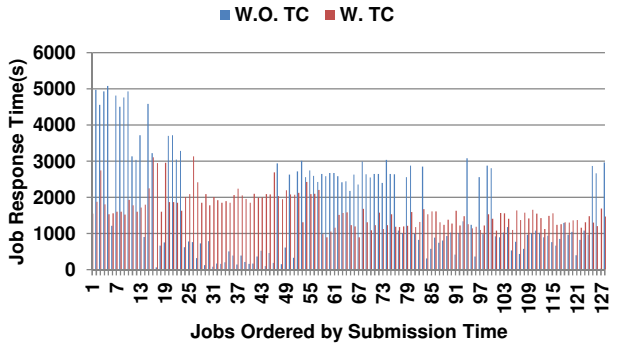

Figure 11. Terasort Test with 128 Jobs of Size 1024M

The smallest improvement comes from the scenario involving the most jobs. As presented in Figure 11, the I/O competition from concurrent jobs drastically increases the execution time of the task. Due to serious contention, the time saved for one job will immediately cause the delay of others, and throughput becomes hard to control since jobs are relatively small compared to their scale. Although the savings on average job response times are trivial, jobs submitted earlier in time do suffer much less from delay, and the distribution of job completion time is flattened. The largest improvements are witnessed in the case of the 32-nodes scale. The time savings can be attributed to the acceleration of reduce tasks, which cut almost one third of average task completion time. Since reduce tasks in Terasort involve large amounts of data output, the coordination becomes most effective when reducers of different jobs are competing on the same node. Within these tests, I/O throttling and coordination favors jobs of medium size and job sets of moderate scale. Jobs of medium size can yield long execution reducers for better I/O throttling, and a medium amount of jobs could provide enough competition for I/O coordination.

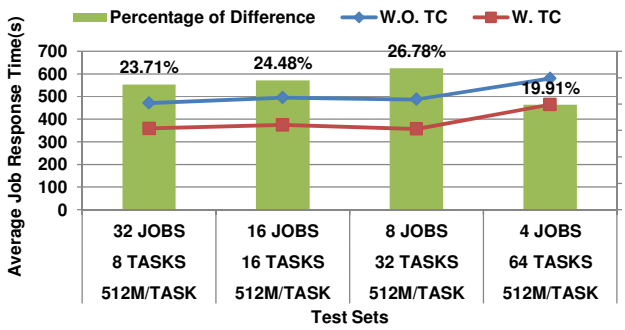

Figure 12. TestDFSIO -write running on 16 computing nodes

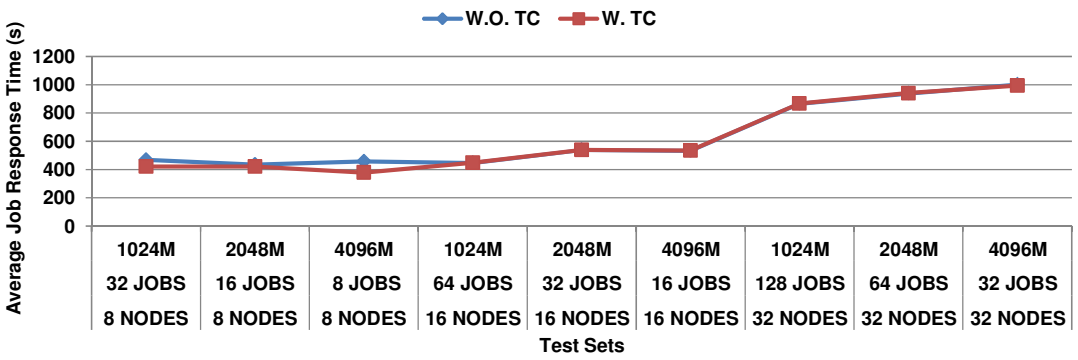

Figure 13. Wordcount benchmark with small output

Next we test the benchmark TestDFSIO where $P_{I/O}(\text{task}) \equiv 1$. In Figure 12, each test set is characterized by the number of jobs involved, the data size handled by each mapper (512M) and the number of mappers in each job. In this case, the proposed optimizations are able to reduce the average job response time by approximately 20% for the TestDFSIO benchmark. Due to the decrease in the number of tasks (512 reduce tasks to 128 map tasks), it shows a smaller improvement in performance compared to the Terasort benchmark.

The wordcount benchmark is also tested in all three scales. The input file is generated by repeatedly duplicating a 70MB dictionary. For this reason, the output of the wordcount benchmark is quite limited. Due to the low I/O intensity, especially the low intensity of write operations, I/O throttling and coordination does not improve average response time greatly.

Figure 14. Job Response Time Distribution [Test Set: Benchmark=Terasort; #Nodes = 8; Job Size = 4096M] [smaller Job ID implies higher priority]

D. Response Time and Job Priority

The scheme of I/O coordination emphasizes high priority jobs; hence, the response time of each job should reflect its priority. Figure 14 gives a typical distribution of job response time after coordination. Since original HDFS has no preference for tasks from any job, the job completion time tends to be even when $\frac{\text{N(job)}}{\#jobs}$ is large. At the same time, I/O coordination makes job response time uneven since prioritized jobs now complete much faster. Consistent with the example in Figure 14, the acceleration of high priority jobs is accompanied with delay of low priority jobs. The total job response time is 20% less after coordination.

VI. Related Work

In this section, the related works are presented in three categories: task scheduling in MapReduce, contention handling on multi/many-core systems, and I/O scheduling in parallel file systems. The discussion compares these works with our research, and highlights the differences and our contribution.

A. Task Scheduling in MapReduce

In the MapReduce paradigm, task schedulers have a huge impact on MapReduce jobs. For this reason, most work concerning performance and QoS in MapReduce demonstrate new scheduling algorithms. To fairly share the resources among jobs, Facebook and Yahoo proposes the FAIR scheduler [15] and the CAPACITY scheduler respectively. Noticing the lack of concern for general metrics, [16] describes an add-on for the FAIR scheduler, which takes into account standard scheduling metrics like response time, makespan, stretch and SLAs (Service Level Agreements). In addition, [17] introduces the LATE scheduling scheme to launch speculative execution for tasks that is critical to job completion time. Meanwhile, [18] presents a task scheduler using estimated job completion time to enforce QoS.

Although schedulers can directly affect the performance and QoS of MapReduce jobs, their influence is limited once the scheduling completes. When a task starts on a computing slot, no scheduler decision can interrupt its execution. Furthermore, task schedulers in MapReduce are unaware of any runtime contention and generally only consider computing slot and memory usage. Such decisions using

minimal information help the system scale up, but at the same time fail to avoid the slowdowns caused by contention. While speculative execution can save time on slow task by redundant computing, its effects are not positive in a busy system, where slowdowns are common and free computing slots are rare.

To our best knowledge, this is the first work that tried to improve MapReduce performance from the perspective of the I/O system. By controlling I/O accesses of MapReduce tasks, the I/O contention is reduced during the runtime; thereby, improving the performance and QoS. Since I/O throttling and coordination are only made after task scheduling, our research makes a good complement to any MapReduce task scheduler.

B. *Contention Handling on Multi/Many-Core System*

The popularity of multi/many-core system has inspired additional research into the contention issue. [4] and [5] summarize the contention impact upon multicore system performance. Its scope covers L2 cache, front-side bus and memory controllers. In [6], researchers handle the cache contention with cache partitioning and page coloring. [8] presents scheduling algorithms to reduce the memory controller contention and memory bus contention. Moreover, effort has been made on the integration of MapReduce framework into many/multi-core systems. [9] describes the Phoenix, an implementation of MapReduce for shared memory systems. Another research in [10] depicts a MapReduce framework on GPUs. In addition, asymmetric multi-core processors are considered in [11].

In multi/many core system, I/O contention fails to draw as much attention as cache and memory. One reason is that the underlying storage stack works well for general I/O requests. There are memory cache to make write consecutive, read ahead mechanism to reduce disk accesses, and optimized arm movement to cut down seeking time. Also, SSDs render the system insensitive to non-contiguous I/O streams. More importantly, the complexity of the I/O system and randomness of I/O requests make performance hard to predict.

Our research focuses on the I/O contention issue in multi/many-core systems for two reasons. One is that I/O is significant in MapReduce tasks and the numbers of cores determines the number of maximum concurrent MapReduce tasks. Therefore, the system has many concurrent I/O streams, which leads to strong I/O contention. Secondly, the I/O streams are long in MapReduce tasks and long I/O streams generally make the I/O performance more predictable. By applying I/O throttling, we are able to limit the I/O throughput drop caused by contention.

C. *I/O Scheduling*

With the rising importance of resource sharing environments like grids and clouds, the QoS in reference to I/O has drawn numerous research attentions. [19,20] address this issue with respect to virtualized platform. Solutions in [21] and [22] adopt deadline-driven strategies to schedule I/O requests in large batches and [23] applies I/O throttling to guarantee the interests of resource owners in a Condor-like system. All these works address QoS of I/O under the MapReduce file system, while our solution strengthens the I/O guarantee within the MapReduce framework.

To achieve high throughput in PFS (Parallel File System), many I/O scheduling techniques have been proposed to improve server-side I/O efficiency. Techniques like disk-directed I/O [24], server-directed I/O [26], and stream-based I/O [25], [27] have optimized either the disk access or network traffic. Most of them schedule I/O requests in groups to further exploit spatial locality of data on the disk. While [12] and [13] address the resource competition among I/O requests, they schedule I/O requests with inter-server coordination either to serve requests or in order to improve the spatial locality.

We noticed similarities between PFS and MapReduce File System. For this reason, the I/O coordination technique is applied into MapReduce. The method, complemented with I/O throttling, improves the average job response time, and accelerates the execution of high priority jobs.

VII. CONCLUSION AND FUTURE WORK

Due to data locality and the increasing adoption of multi-core processors, I/O resource contention has become a common phenomenon in MapReduce applications. Borrowing some recent results from parallel file systems, we proposed a combined method that throttles and coordinates I/O streams to reduce job completion time of MapReduce applications. Experimental results show that I/O throttling and coordination can reduce average job response time by up to 33.74% for I/O intensive applications. Even for less data intensive applications, the method is able to find its usefulness in improving the response time of prioritized jobs. In addition, a detailed analysis is presented to illustrate and guide the design of the coordination and throttling method. Our implementation in Hadoop yields a code patch that reinforces the system when working under heavy I/O load.

One short term future work is to make the design more effective for cloud environments where multiple MapReduce clusters run on the same physical machine. [19] presents work that modifies the VMM scheduler to avoid performance degradation when hosting multiple MapReduce clusters. Inspired by their work, migrating our work to the cloud will need the support of the VMM scheduler. The reason is that I/O throttling and coordination can only work knowing all concurrent I/O streams. Another future task is to extend our method to handle network contention. Although MapReduce emphasizes the locality by pushing computation closer to data, it also tries to access remote blocks when free computing slots are available. This behavior improves the performance in most cases. However, reduce phases always consume large amount of network bandwidth. So sharing

network bandwidth among multiple jobs will lead to the similar contention problem s as those addressed in this paper.

REFERENCES

[1] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, Berkeley, CA, USA, 2004, pp. 10–10.

[2] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in Proceedings of the nineteenth ACM symposium on Operating systems principles, New York, NY, USA, 2003, pp. 29–43.

[3] A. Beloglazov, R. Buyya, Y. C. Lee, and A. Zomaya, "A Taxonomy and Survey of Energy-Efficient Data Centers and Cloud Computing Systems," arXiv:1007.0066, Jul. 2010.

[4] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems, New York, NY, USA, 2010, pp. 129–142.

[5] R. Hood et al., "Performance impact of resource contention in multicore systems," in 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS), 2010, pp. 1-12.

[6] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm, "RapidMRC: approximating L2 miss rate curves on commodity systems for online optimizations," in Proceeding of the 14th international conference on Architectural support for programming languages and operating systems, New York, NY, USA, 2009, pp. 121–132.

[7] X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multicore cache management," in Proceedings of the 4th ACM European conference on Computer systems, New York, NY, USA, 2009, pp. 89–102.

[8] S. Blagodurov, S. Zhuravlev, and A. Fedorova, "Contention-Aware Scheduling on Multicore Systems," ACM Transactions on Computer Systems, vol. 28, pp. 1-45, Dec. 2010.

[9] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for multi-core and multiprocessor systems," In HPCA '07: Proceedings of the 13th International Symposium on High-Performance Computer Architecture, p. 13--24, 2007.

[10] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a MapReduce framework on graphics processors," in Proceedings of the 17th international conference on Parallel architectures and compilation techniques, New York, NY, USA, 2008, pp. 260–269.

[11] M. M. Rafique, B. Rose, A. R. Butt, and D. S. Nikolopoulos, "Supporting MapReduce on large-scale asymmetric multi-core clusters," ACM SIGOPS Operating Systems Review, vol. 43, p. 25, Apr. 2009.

[12] Huaiming Song, Yanlong Yin, Xian-He Sun, Rajeev Thakur, Sam Lang. "Server-Side I/O Coordination for Parallel File Systems." In the Proc. of the ACM/IEEE SuperComputing Conference (SC'11), Nov. 2011.

[13] Xuechen Zhang, K. Davis, and Song Jiang, "IOrchestrator: Improving the Performance of Multi-node I/O Systems via Inter-Server Coordination," in High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for, 2010, pp. 1-11.

[14] Y. Chen, A. Ganapathi, and R. H. Katz, "To compress or not to compress - compute vs. IO tradeoffs for mapreduce energy efficiency," 2010, p. 23.

[15] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Job Scheduling for Multi-User MapReduce Clusters," EECS Department, University of California, Berkeley, UCB/EECS-2009-55, Apr. 2009.

[16] J. Wolf et al., "FLEX: A Slot Allocation Scheduling Optimizer for MapReduce Workloads," in Middleware 2010, vol. 6452, I. Gupta and C. Mascolo, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1-20.

[17] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving MapReduce performance in heterogeneous environments," in Proceedings of the 8th USENIX conference on Operating systems design and implementation, Berkeley, CA, USA, 2008, pp. 29–42.

[18] J. Polo et al., "Performance-driven task co-scheduling for MapReduce environments," in 2010 IEEE Network Operations and Management Symposium (NOMS), 2010, pp. 373-380.

[19] H. Kang, Y. Chen, J. L. Wong, R. Sion, and J. Wu, "Enhancement of Xen's scheduler for MapReduce workloads," in Proceedings of the 20th international symposium on High performance distributed computing, New York, NY, USA, 2011, pp. 251–262.

[20] A. Gulati, I. Ahmad, and C. A. Waldspurger, "PARDA: proportional allocation of resources for distributed storage access," in Proccedings of the 7th conference on File and storage technologies, Berkeley, CA, USA, 2009, pp. 85–98.

[21] A. Povzner, D. Sawyer, and S. Brandt, "Horizon: efficient deadline-driven disk I/O management for distributed storage systems," in Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, New York, NY, USA, 2010, pp. 1–12.

[22] M. Karlsson, C. Karamanolis, and X. Zhu, "Triage: Performance differentiation for storage systems using adaptive control," Trans. Storage, vol. 1, no. 4, pp. 457–480, Nov. 2005.

[23] K. D. Ryu, J. K. Hollingsworth, and P. J. Keleher, "Efficient network and I/O throttling for fine-grain cycle stealing," in *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, New York, NY, USA, 2001, pp. 3–3.

[24] D. Kotz, "Disk-directed I/O for MIMD multiprocessors," ACM Trans. Comput. Syst., vol. 15, no. 1, pp. 41–74, Feb. 1997.

[25] . I. Ligon, W. B. and R. B. Ross, "Implementation and performance of a parallel file system for high performance distributed applications," in Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing, Washington, DC, USA, 1996, p. 471–.

[26] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett, "Server-Directed Collective I/O in Panda," in Supercomputing, 1995. Proceedings of the IEEE/ACM SC95 Conference, 1995, pp. 57- 57.

[27] R. B. Ross and W. B. L. Iii, "Server-Side Scheduling in Cluster Parallel I/O Systems," Calculateurs Parallèles Journal Special Issue on Parallel I/O for Cluster Computing, 2001.