Modeling I/O Interference for Data Intensive Distributed Applications

Sven Groot Kazuo Goda Daisaku Yokoyama

Miyuki Nakano Masaru Kitsuregawa

Institute of Industrial Science, The University of Tokyo
4-6-1 Komaba, Meguro-ku, Tokyo 153-8505, Japan
{sgroot,kgoda,yokoyama,miyuki,kitsure}@tkl.iis.u-tokyo.ac.jp

ABSTRACT

Data intensive applications such as MapReduce can have large performance degradation from the effects of I/O interference when multiple processes access the same I/O resources simultaneously, particularly in the case of disks. It is necessary to understand this effect in order to improve resource allocation and utilization for these applications. In this paper, we propose a model for predicting the impact of I/O interference on MapReduce application performance. Our model takes basic parameters of the workload and hardware environment, and knowledge of the I/O behavior of the application to predict how I/O interference affects the scalability of an application. We compare the model's predictions for several workloads (TeraSort, WordCount, PFP Growth and PageRank) against the actual behavior of those workloads in a real cluster environment, and confirm that our model can provide highly accurate predictions.

Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems*

General Terms

Performance

Keywords

MapReduce, Cloud Computing, Data Intensive, Cost Model, I/O Interference, I/O Behavior

1. INTRODUCTION

The need to process and analyze large volumes of data is still increasing. The web, social networks, sensors, etc. provide more data than ever before, and cloud computing has put the resources necessary to analyze big data within everyone's reach. Data processing applications are often implemented using MapReduce [5], in particular using its open source implementation Hadoop [1], and can often benefit greatly from multi-core systems.

Modern server nodes typically have many CPU cores, so it is desirable to use parallelism on each node to utilize the CPU power. However, data intensive applications are often dominated by disk I/O operations, and I/O resources are more limited than CPU. Disk storage can range from single local disks to a large storage array attached to a node, but they are often represented as a single logical device. When multiple processes—or even multiple virtual machines running on a node—access such a device simultaneously this causes interference, leading to significant performance losses in real applications.

Figure 1 shows the scalability of two typical MapReduce workloads when per-node parallelism is increased on a cluster where each node has 8 CPU cores and 1 disk I/O device (see Table 2). WordCount [5] is CPU bound because it performs string processing, has a combiner, and very small output data. This allows it to scale from 1 to 8 cores with 95% efficiency. TeraSort is I/O bound because it performs very little processing and has very large input, intermediate and output data. It only achieves 29% efficiency with 8 parallel tasks, so is unable to fully utilize the CPU resources. At more than 5 parallel tasks per node overall performance actually decreases.

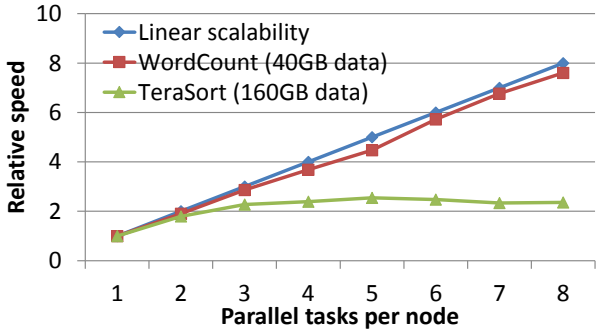


Figure 1: Scalability of WordCount and TeraSort with per-node parallelism.

SAC'13 March 18-22, 2013, Coimbra, Portugal.

This example shows how I/O interference can prevent the systems from being fully utilized. Data intensive applications are becoming more and more pervasive in the cloud computing era, so being able to predict the effects of I/O interference is required to improve scheduling and provisioning decisions. Predicting I/O interference is a complex problem that no existing models for MapReduce address as far as we are aware.

In this paper, we analyze the I/O behavior of MapReduce in detail and propose a comprehensive I/O cost model that combines the application behavior with hardware models derived from micro-benchmarks to predict the effect of I/O interference. This model is evaluated with several representative workloads on a real cluster, and the results show it is able to make predictions with very high accuracy.

2. RELATED WORK

The difficulty of efficiently using MapReduce has been recognized for some time. There have been a number of works that attempt to automatically optimize MapReduce execution [12, 21, 10, 20, 8, 11], typically focusing on optimizing query execution and sharing work between workloads, or automating cluster provisioning and Hadoop configuration. In these works, I/O is only considered when the goal is to reduce the total amount of I/O performed by the application; the impact of I/O interference on existing MapReduce applications is not considered.

Recently, there has been an increasing amount of work in modeling the behavior of MapReduce. Huai et al [9] propose a model that generalizes Map-Reduce and similar frameworks into a matrix-based representation of the data flow. Verma et al [19] propose a model to estimate job completion time based on the observed task completion times measured previously. Jindal et al [11] model the read I/O behavior of map tasks. Yang et al [22] propose a statistical model to evaluate the effect of various configuration parameters.

The model proposed by Herodotou et al [8, 7] is to our knowledge the most complete analytical model of Hadoop, focusing on relative performance between environments and optimizing Hadoop configuration.

Our work focuses on the I/O performance of MapReduce, and differs from these existing models because all of them make simplifying assumptions about I/O or ignore it entirely.

For I/O interference itself, there has been a considerable effort to characterize and predict I/O performance [16, 14, 6], which focuses primarily on the hardware environment.

Chiang et al [4] propose a scheduling system that takes I/O interference into account, but unlike our work it applies to VM scheduling rather than application scheduling, and was only evaluated using simulation.

Related to our work is the work of Shan et al [17], which characterizes the entire I/O behavior of an application using a micro-benchmark; this differs from our work because we use micro-benchmarks only to establish basic hardware I/O performance and use an analytical model to describe the application's behavior. They also target more traditional supercomputers rather than cloud environments.

3. EFFECTS OF I/O INTERFERENCE

A MapReduce application consists of many map and reduce tasks that read and write data on the Distributed File System (DFS) and local disks. In order to utilize multi-core systems, Hadoop schedules multiple tasks to run simultaneously on each node. In this case, the node's I/O resources are divided between these tasks.

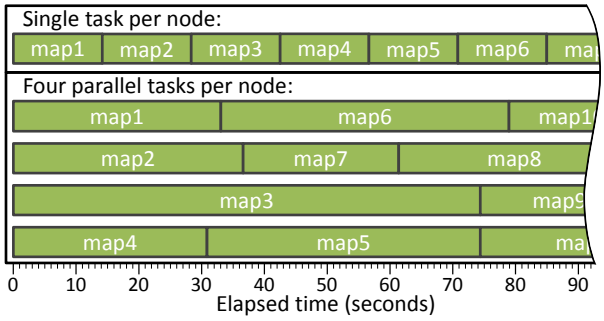


Figure 2: Partial time-line of map task execution of the TeraSort workload.

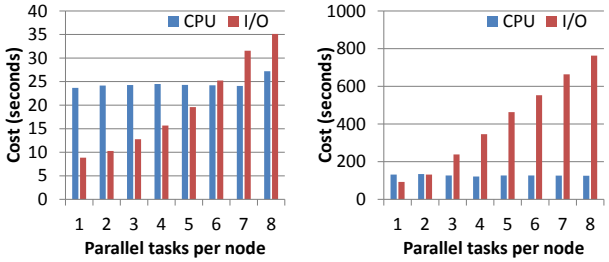


Figure 3: Average CPU and I/O cost of TeraSort tasks.

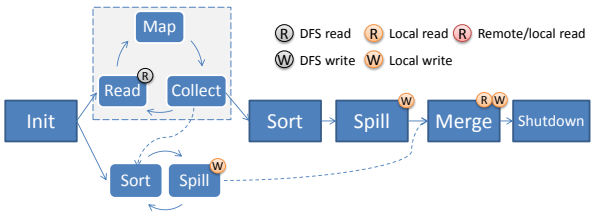
Figure 2 shows a partial time-line for one node in a cluster taken from an actual execution of the TeraSort workload. When the number of parallel tasks per node is increased from 1 to 4 the tasks increase in duration and become highly variable. Although only map tasks on a single node are shown here, the same effect is observed on all nodes and for reduce tasks, and also occurs with different workloads.

The cause of the increased execution time can be seen if we look at the CPU and I/O cost for the tasks. The *CPU cost* is the time it takes for the tasks' CPU processing to complete, and the *I/O cost* is the time it takes for I/O devices to finish servicing requests from the tasks. These costs include the time spent waiting for devices to become available, so they increase under contention. The task execution time is determined by a combination of these two, though it is not a simple sum since I/O operations are done asynchronously.

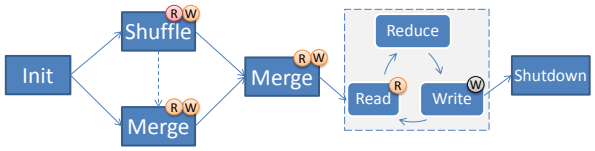
Figure 3 shows how the average CPU and I/O costs of the tasks in the TeraSort workload increase when per-node parallelism is increased. Because we do not run more tasks than the number of CPU cores (8 cores in our case; see Table 2), CPU contention does not occur and these costs remain the same. However, I/O costs clearly increase significantly due to interference.

Although we can reduce I/O costs—and thus the impact of interference—by using compression, this often adversely affects the overall performance of the application [3].

It is necessary to consider the I/O behavior of data intensive applications in further detail to accurately model I/O



(a) Map task



(b) Reduce task

Figure 4: Phases of map and reduce tasks.

costs under interference.

4. I/O BEHAVIOR OF MAPREDUCE

The effects of I/O interference are determined by the I/O patterns and the overall work flow of the application. Since this depends on the specific implementation of MapReduce, this section focuses on Hadoop.

While Section 3 discussed CPU and I/O costs in the context of tasks, in reality we must further break down each task into *phases*: individual steps in the execution of a task. These phases perform distinct operations, and have their own individual costs.

Figure 4 shows the processing work flow of map and reduce tasks, where each box represents a phase. The *read/map/collect* and *read/reduce/write* blocks represent cycles of interleaved operations which are considered as single phases for our purposes. For each phase, the type of I/O performed is indicated.

MapReduce performs primarily sequential I/O. Each map task reads a single DFS block sequentially. Similarly, all write operations write large amounts of data sequentially to a file. Merge operations read multiple input files in an interleaved fashion, but aggressive buffering is used to avoid overhead.

The shuffle phase is the only one that has a different I/O pattern. Reduce task input is divided into *segments* (one segment from each map task), and must be read from all nodes in the cluster. The segments for a particular reduce task are not stored sequentially, which means there is a cost for performing random I/O (latency and seek) involved for each segment.

4.1 I/O Interference Effects on Phases

Every phase can potentially interfere with any other phase, because tasks running on the same node are not guaranteed to execute the same phase at the same time. This is evident from Figure 5, which shows a partial time-line similar to Figure 2. Figure 5 includes additional information about which phase is currently active in each task; the half-height blocks indicate background spills that happen during the *read/map/collect* phase.

The changing overlap between phases means not all tasks

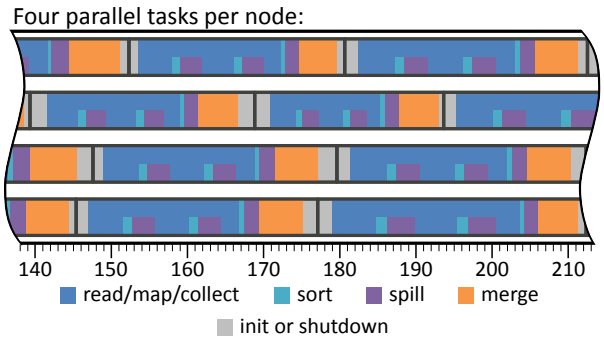


Figure 5: Partial time-line of map task phases of the TeraSort workload.

receive the same interference, which causes the observed variability between the tasks. We cannot predict exactly which phase will receive how much interference at any given time, but when running the same workload multiple times we observed that the overall job execution time and the average costs for each phase varied by at most 3% across executions. This means that while some phases may have more or less interference, the overall effect on the workload does not change substantially.

4.2 Fragmentation

Hadoop stores both DFS blocks and intermediate data in regular files. Physical block allocation on the disks is not controlled by Hadoop, but by the file system and may be subject to file fragmentation.

Reading a fragmented file can be much slower than reading a sequentially stored one, but it does not have a significant impact on interference. However, interference between multiple tasks that are simultaneously writing files can cause heavy fragmentation, adversely affecting performance when those files are read back.

Intermediate files are written and read by the same job, so interference while those files are written affects the performance of that same job. Intermediate file fragmentation was observed to have a significant impact on the overall performance in some cases, so we must be able to predict what effect interference while writing has on the read performance of those files.

5. I/O COST MODEL

Figure 6 shows the work flow of our modeling process. The model consists of a MapReduce model and a hardware model. The latter is used to calculate I/O cost under interference by first assuming I/O bandwidth is split evenly between tasks and then applying hardware specific interference functions. These interference functions are determined for a specific hardware environment by using micro-benchmarks. CPU costs are considered constant, as we never run more tasks than there are CPU cores.

The MapReduce model is used to calculate the *phase cost*, which is made up of the CPU cost and the I/O cost under interference. The larger of the two values is used, because I/O is performed asynchronously. We then calculate the average *task execution time*, the map and reduce *stage execution time*, and the overall *job execution time*.

The model has as input several parameters for each work-

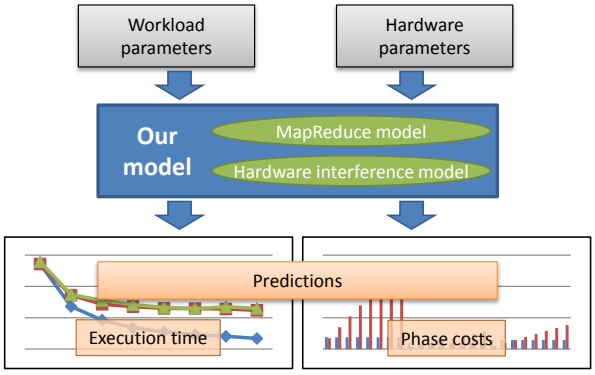


Figure 6: Modeling workflow.

Workload (global)	
N_{tasks}^{map}	Number of map tasks
N_{tasks}^{reduce}	Number of reduce tasks
Workload (phase)	
S_{read}	Data read in bytes
S_{write}	Data written in bytes
CPU_{phase}	CPU cost in seconds
Environment (system)	
N_{nodes}	The number of nodes in the cluster
N_{slots}	Number of parallel tasks per node (map or reduce)
Environment (hardware)	
R_{disk}	Cost of reading 1 byte, in seconds
W_{disk}	Cost of writing 1 byte, in seconds
C_{random}	Cost of a random I/O operation
F	The fragmentation factor
f_i	Hardware-specific interference function

Table 1: Model parameters

load describing the CPU costs and input and output data sizes for each task phase, and some related to the environment. Table 1 lists these parameters. Workload parameters are split into global parameters, which apply to the entire workload, and phase parameters that have different values for each phase. How workload parameters are obtained is described in Section 5.1, and hardware parameters in Section 5.4.

I/O costs can include network transfer costs for I/O operations that are performed on remote hosts. However, in our experiments the disk I/O costs always exceeded the network transfer costs, and since cloud data centers provide generous network bandwidth this is expected to often be the case. Therefore, network transfer costs are omitted from the I/O costs in this section.

The goal of the model is to predict how performance is affected when the system parameter N_{slots} changes, which expresses the degree of parallelism per node. There are actually two separate values for this parameter; one for map and one for reduce tasks. Since most calculations are the same for both types of tasks we do not distinguish this in the formulas, and the appropriate value should be used depending on the type of task the calculation is applied to.

The output of the model is a prediction about the effect of interference for each value of N_{slots} on the average task execution time and overall job execution time; it also provides per-phase predictions that provide insights into which parts of the application are I/O bottlenecks.

5.1 Measuring Workload Parameters

For each phase we need to determine the value of CPU_{cost}, S_{read} and S_{write}.

We have made two changes to Hadoop: 1) record timing information in the task log files, from which CPU costs for each phase are derived; 2) record any additional data sizes needed. An automated tool reads the job history and task log files to extract the values and determine the parameters.

In order to get accurate timing information, the measurement must occur in an environment without interference, which means only 1 map and reduce slot. To save time during measurement, only part of the workload needs to be executed; information for the full workload can be inferred. If the input data or Hadoop configuration changes the approach used in [7] can be used to adjust the values without measuring again.

5.2 Phase Cost Estimation

Estimating the cost of each phase (C_{phase}) is done depending on the I/O pattern of those phases. The basic formula for most phases is as follows:

$$C_{phase} = \max(CPU_{phase}, IO_{phase}^{read} + IO_{phase}^{write}) + f_i$$

For phases that perform sequential I/O, the read and write I/O costs (IO_{phase}^{read} and IO_{phase}^{write}) are calculated by simply multiplying the data sizes with the hardware cost of reading and writing:

$$IO_{phase}^{read} = N_{slots} \cdot S_{read} \cdot R_{disk}$$

$$IO_{phase}^{write} = N_{slots} \cdot S_{write} \cdot W_{disk}$$

We multiply the I/O costs by the value of N_{slots} to indicate bandwidth is divided between parallel tasks. The function f_i accounts for additional hardware specific interference costs (e.g. latency and seek), and is described later. It is not treated as part of the I/O costs because it may interact differently with the CPU costs in some environments; if this is the case it should be reflected by f_i itself.

These equations are used to estimate the phase costs of the map task *init*, *read/map/collect*, *sort*, *spill*, and *merge* phases, and the reduce task *init* and *merge* phases. Note that the *init* and *sort* phases do not perform I/O, so their value is always equal to CPU_{phase}.

For the I/O pattern used by the reduce task *shuffle* phase, we replace the value of IO_{phase}^{read} with an alternate value $IO_{phase}^{read\ \prime}$:

$$IO_{phase}^{read\ \prime} = N_{slots} \cdot (S_{read} \cdot R_{disk} + N_{tasks}^{map} \cdot C_{random})$$

Here, we add an additional cost for each segment that the reduce task reads. The number of segments is equal to the number of map tasks.

To account for intermediate file fragmentation as described in Section 4.2, we must adjust the cost of reading an intermediate file based on the expected level of fragmentation created when writing it under interference. To express this level of fragmentation, we use the fragmentation factor parameter F, whose derivation is described in Section 5.4.3. For phases where fragmentation is an issue, we replace IO_{phase}^{read} with $IO_{phase}^{read\ \prime\prime}$:

$$IO_{phase}^{read\ \prime\prime} = {N_{slots}}^{F} \cdot S_{read} \cdot R_{disk}$$

Although there are a few phases that could in theory ex-

perience intermediate file fragmentation, in our experiments we only observed this to be a factor for the reduce task *read/reduce/write* phase, so we only use this calculation for that phase.

5.2.1 Remote I/O

When a task performs I/O on a remote host, it contributes to the interference on the destination node rather than the one the task is running on. This can occur for map tasks when they are scheduled on a node that does not have the task's input data. However, the scheduler tries to avoid this and it does not happen often enough to significantly affect the execution time.

For reduce tasks, the shuffle phase reads data from both local and remote hosts. We need to know how much data is read on average from that node per task. Each task reads on average $\frac{1}{N_{nodes}} \cdot S_{read}$ bytes from each node (including its local node), so the total amount of data read per node is $\frac{N_{tasks}}{N_{nodes}} \cdot S_{read}$. To get the average per task, we need to divide this by the number of tasks per node, $\frac{N_{tasks}}{N_{nodes}}$, which equals S_{read}. This means no adjustment to the shuffle phase formula is necessary.

5.3 Task and Workload Estimation

Task execution time can be estimated as a sum of all the individual phase costs for a task. The exception is the shutdown phase, since it depends on some wait operations that mean the task execution time must be rounded up to the nearest multiple of the wait interval. Since tasks can only be scheduled on heartbeats, we must again round this value up to the nearest multiple of the heartbeat interval. If out-of-band heartbeats are enabled, this is not necessary.

We separately estimate the time of the *map stage* and *reduce stage*. At the end of each stage, there may not be enough tasks left so that all nodes are running N_{slots} tasks, so there would be less interference on those nodes. Because this is outside the target of this model, we do not include these last few tasks, although they could be added by doing a separate interference calculation for those tasks alone.

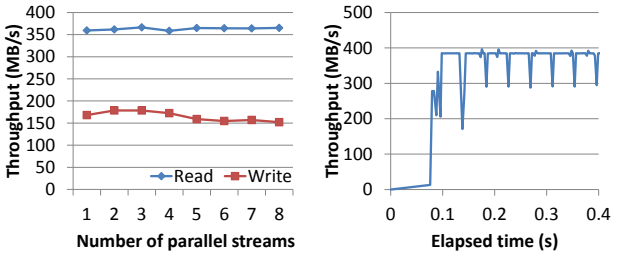
The overall *job execution time* is the sum of the map and reduce stage execution times. Note that although the first wave of reduce tasks can execute during the map tasks, our observations show this does not affect the overall job execution time much.

5.4 I/O Interference Estimation

I/O interference depends on a large number of factors including the hardware, OS and application behavior. Rather than attempt to model all of these, we create a black-box approximation of the hardware behavior based on micro-benchmarks. In this section, we outline the method used to determine hardware parameters and functions for the various I/O patterns employed by Hadoop. The values in this section use the hardware environment specified in Table 2, but the procedure itself is applicable to any environment. The maximum degree of parallelism we are interested in equals the number of CPU cores; 8 in our case.

5.4.1 Sequential I/O

For sequential I/O, we need to measure the average read and write throughput of the I/O device to determine R_{disk} and W_{disk}, and estimate the interference function f_i. We do this using a micro-benchmark that reads or writes several files with varying numbers of parallel streams.



(a) Parallel throughput (b) Read time-line (1 stream)

Figure 7: Sequential I/O performance of our environment.

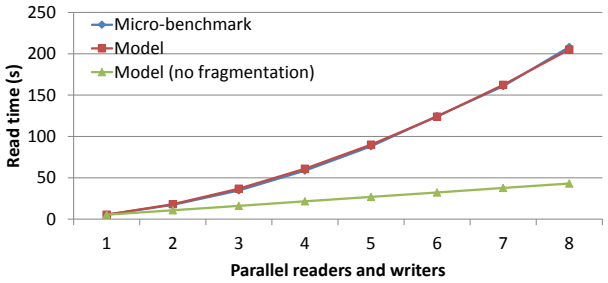


Figure 8: Read performance with fragmentation.

Figure 7(a) shows the read and write throughput of our hardware environment for each number of streams. The value is the aggregate throughput over all streams. Because of efficient read-ahead caching in our storage system, the total throughput remains relatively stable even when parallel streams are increased.

Figure 7(b) shows a time-line of reading a single stream. It shows that there is a random access cost of approximately 100ms, after which stable throughput is reached. The start-up overhead was observed to increase with additional parallel streams. For writes, no such start-up overhead is observed.

Based on these observations, we estimate $R_{disk} = \frac{1}{380MB/s}$ and $W_{disk} = \frac{1}{150MB/s}$, and $f_i = 0.1 \cdot N_{slots}$.

5.4.2 Shuffle I/O

To estimate the random access cost per segment (C_{random}) we use a micro-benchmark that reads a single segment from each file out of a large number of files. We vary the size of the segments and observe the delay between the segments. We observed that the delay matches the earlier observed 100ms, therefore $C_{random} = 0.1$.

5.4.3 Fragmentation

To determine the fragmentation factor F we use a micro-benchmark that writes a varying number of files in parallel, and measures the throughput of reading those files back ($W^{fragmented}_{throughput}$). We then find F by solving $N^{F-1} = \frac{1}{R_{disk} \cdot W^{fragmented}_{throughput}}$ for F. We use the average of the values found as the value of F in the model. For our environment, we found $F = 1.75$.

System	
CPU	2x quad-core Xeon E5530 2.4GHz
Memory	24GB
Storage (RAID)	
Controller	JCS VCRVAX-4F RAID
HBA	QLogic QLE2462
Disks	10x Hitachi HDS721010CLA332

Table 2: Hardware configuration of each node in our cluster

Figure 8 shows how the calculated transfer time of reading the fragmented files compares to the measured values. It also shows an estimation that does not take the fragmentation into account; clearly, fragmentation can have a significant impact on read performance.

6. EXPERIMENTAL EVALUATION

We evaluated our model by running several workloads on a 10 node cluster using Hadoop 0.20.203.0. Table 2 shows the hardware configuration of each node. We obtained the parameters given in Table 1 for the hardware and for each workload using the methods given in Section 5. We then executed the workloads normally, varying the number of tasks per node between 1 and 8, and compared the actual performance to the predictions made by the model.

The map stage and reduce stage are evaluated separately because this provides more interesting insights than just the overall job execution time. To facilitate this, Hadoop was configured to delay reduce task scheduling until all map tasks for a job are finished. We have verified that using background reduce tasks did not affect the overall job execution time, but those results are not shown here for space reasons.

The workloads we used are described in Table 3, and were chosen to provide a representative mix of I/O and CPU intensive behavior. TeraSort [18] is almost entirely I/O intensive, WordCount [5] is entirely CPU intensive, and PFP Growth [13] and PageRank [15, 2] are I/O intensive in some parts and CPU intensive in others. Note that the latter two workloads consist of multiple job; we have verified the model for all jobs but only one for each workload is shown here.

6.1 Results

Figure 9 shows a comparison between the model predictions (the line labeled *Model*) and actual stage execution time (*Actual*). We also compare with a model that ignores interference and assumes linear scalability (*Linear scalability*). For reduce tasks, we show both the prediction with and without accounting for fragmentation (*Model* and *Model (no interference)* respectively). Figure 10 shows the predicted CPU and I/O cost for each individual task phase.

TeraSort: Figure 9(a) and (b) show that ignoring interference is very inaccurate; at 8 slots, the difference between *Actual* and *Linear scalability* is 63% for map tasks and 77% for reduce tasks. The model correctly estimates the execution time, with only 5% difference for map tasks and 6% for reduce tasks (see Section 6.2 for details on the fragmentation result). Figure 10(a) and (b) show that almost all phases of TeraSort are I/O bound after 2 or 3 slots.

WordCount: In Figure 9(c) and (d), *Model* and *Linear scalability* are identical because the workload is CPU bound, and the actual execution time confirms that. The small difference for map tasks is likely caused by memory or cache contention, but this is minor compared to the I/O interference seen in other workloads. Figure 10(c) and (d) also reflect the CPU bound nature of this workload.

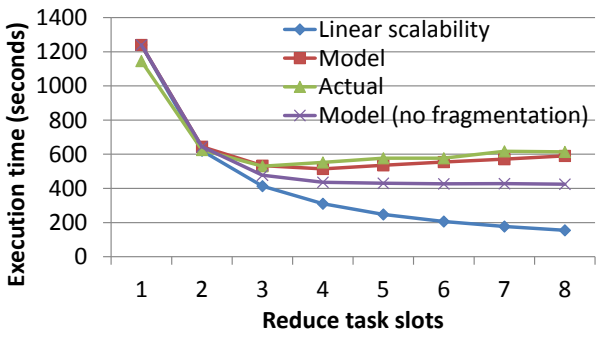


Figure 11: TeraSort reduce stage on a 1 node Hadoop cluster.

PFP Growth: In Figure 9(e) and (f), there is a 22% difference between *Linear scalability* and *Actual* for map tasks at 8 slots, and 59% for reduce tasks. Despite the CPU intensive nature of the map and reduce functions, some parts of the job are still I/O intensive which our model correctly predicts, estimating performance to within 11% for map tasks and 8% for reduce tasks. Figure 10(e) and (f) show that the *spill*, *merge* and *shuffle* phases are the cause of the performance loss.

PageRank: The results shown in Figure 9(g) and (h) are similar to those of PFP Growth, although there is a slightly bigger difference between *Linear scalability* and *Actual* because the intermediate data for this job is bigger. At 8 slots, the model estimates the actual value to within 5% for map tasks, and 6% for reduce tasks. Figure 10(g) and (h) shows the I/O interference originates from the same phases as in PFP Growth.

For each workload, our model was able to estimate the effect of I/O interference to within 5-10% of the execution time of actual execution, and indicate where the I/O interference occurs even for workloads that were CPU intensive in some parts.

6.2 Fragmentation

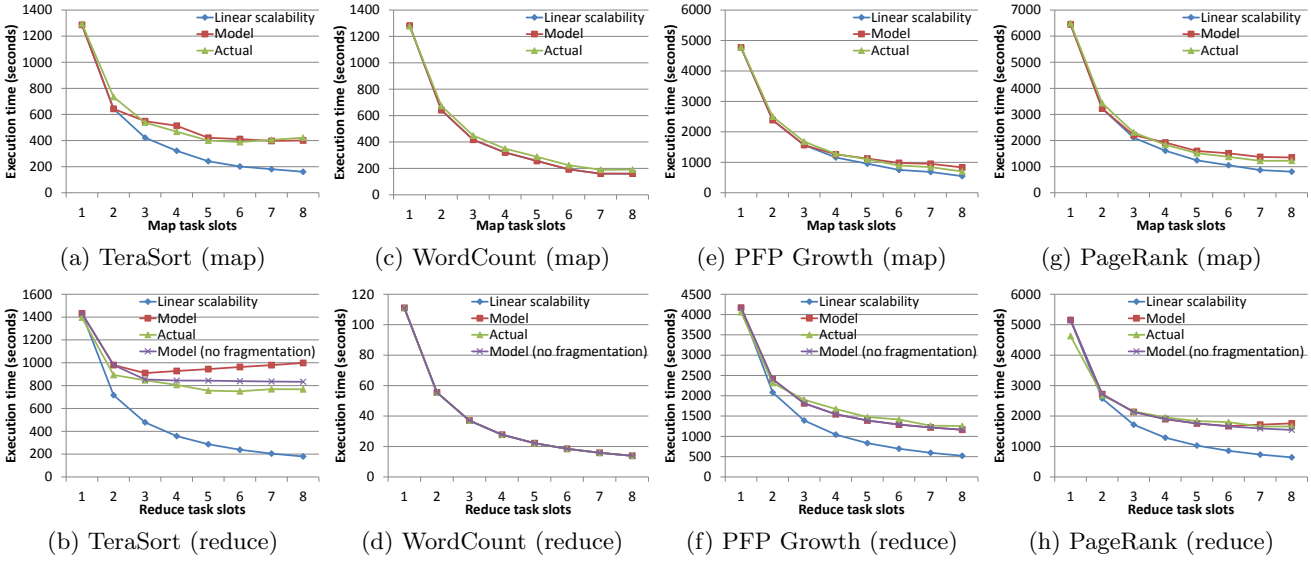
For TeraSort in Figure 9(b) it appears that *Model (no fragmentation)* is more accurate than *Model*; including fragmentation is off by 30% for 8 slots. Fragmentation levels are based on the number of parallel writers, but variability in the shuffle phase means not all tasks are writing at the same time so the fragmentation is less than predicted.

However, in other circumstances we have seen a significant effect from fragmentation. Figure 11 shows the results obtained form running TeraSort on a cluster with only one node. In this case *Model* (with fragmentation) correctly estimates the performance to within 4% for 8 slots, while *Model (no fragmentation)* underestimates the execution time by 34%.

Fragmentation can have a very large impact on performance, but it is difficult to know exactly when it will occur. Improving our handling of fragmentation is part of our ongoing work in this research.

Workload	Data size			Tasks		CPU usage		Combiner
	Input	Intermediate	Output	Map	Reduce	Map	Reduce	
TeraSort	160GB	160GB	160GB	640	80	Low	Low	No
WordCount	40GB	100MB	1MB	400	80	High	High	Yes
PFP Growth	85GB	240GB	10MB	701	80	High	High	No
PageRank	120GB	370GB	120GB	1040	80	High	Medium	No

Table 3: Workloads used in the evaluation.



(a) TeraSort (map) (c) WordCount (map) (e) PFP Growth (map) (g) PageRank (map)

(b) TeraSort (reduce) (d) WordCount (reduce) (f) PFP Growth (reduce) (h) PageRank (reduce)

Figure 9: Comparison of model prediction and actual execution time on a 10 node Hadoop cluster.

7. DISCUSSION

This model can be used to improve scheduling decisions on several levels such as static scheduling policy, mixed workload scheduling, and dynamic task scheduling.

All of the I/O intensive workloads we have observed have a clear point beyond which scheduling additional tasks on a node has little or no performance benefit. Based on the model, we can apply a policy on each workload that limits the number of simultaneous map or reduce tasks on a single node. The scheduler will then give preference to additional nodes if they are available, rather than more tasks per node, improving overall performance. If a job is limited to fewer parallel tasks than the maximum for a node, the remaining task slots can be used for other, more CPU intensive, jobs. For example in Figure 1 it can be seen that TeraSort does not benefit from more than 3 parallel tasks, so the remaining slots could be used for WordCount if that job is active at the same time. This allows the scheduler to determine how to divide cluster resources between heterogeneous workloads to obtain improved utilization, especially when the model is further extended to handle mixed workloads.

The model can also be used for dynamic scheduling. When a new task must be scheduled on a node, information about the currently active tasks and the I/O load of the node will allow an estimation of the performance impact of possible new tasks. The per-phase estimations as shown in Figure 10 can be used to refine scheduling decisions at this level; for example, if the currently active tasks have already finished the I/O intensive part of their work (e.g. the shuffle phase), scheduling an additional I/O intensive task is possible. Due to task variability predictions at this level are necessarily less accurate, but we believe this can still offer a significant benefit. This type of I/O-aware scheduler would be well-suited to the resource-based scheduler system used in YARN for Hadoop 2.0.

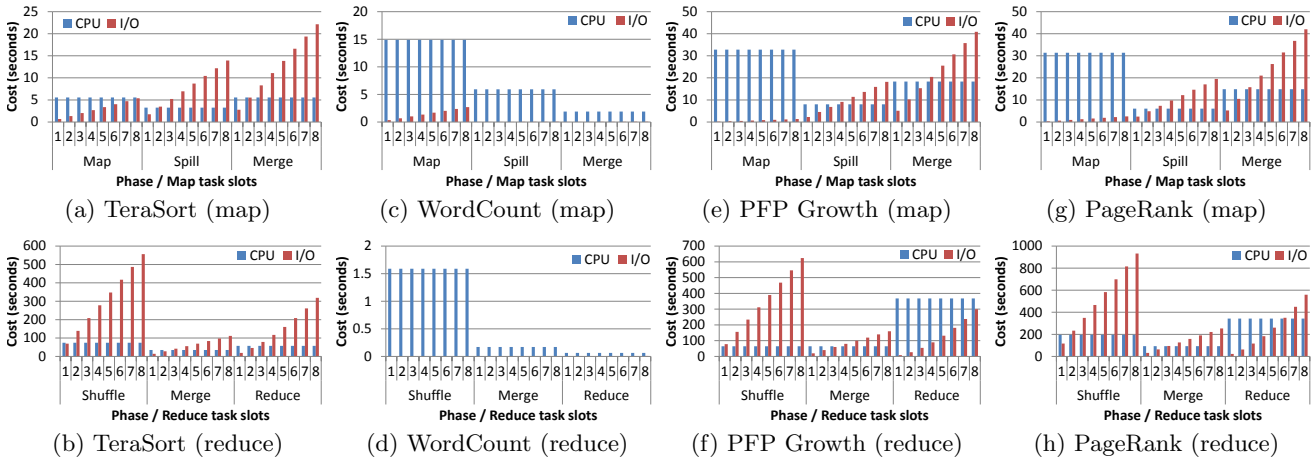
7.1 I/O Interference in Other Applications

The problem of I/O interference does not just affect MapReduce, but is a problem for all data intensive applications. The model we present in this paper is tailored for MapReduce, but the approach itself can apply to other data intensive applications. Our method allows users to break down a complex workload into smaller pieces (for MapReduce, these were tasks and phases; for other applications, it can be derived for example from database logs), model the I/O behavior of each part separately, and use an application model to combine the parts and make statements about the whole application. Once the I/O behavior of an application is known, our approach for determining the impact of interference would be applicable to those applications as well.

8. CONCLUSION

We have analyzed the I/O behavior of MapReduce, and proposed an I/O cost model that considers the hardware environment and this behavior to predict the effect of I/O interference. The model is able to accurately predict the effect of I/O interference for several real workloads. This model can be used to improve resource utilization and workload performance.

For our future work, we intend to extend the model to handle heterogeneous workloads. This will be used to create a tool that can automatically derive workload parameters by



(a) TeraSort (map) (c) WordCount (map) (e) PFP Growth (map) (g) PageRank (map)

(b) TeraSort (reduce) (d) WordCount (reduce) (f) PFP Growth (reduce) (h) PageRank (reduce)

Figure 10: Predicted CPU and I/O cost of each phase.

running a small subset of any new workload in isolation, and then use this information in a custom task scheduler based on our model, providing real performance benefits.

9. ACKNOWLEDGMENTS

This research is partially supported by the project for "R&D on cloud service infrastructure for recovering wide-area disaster (Reliable cloud service platform technology)" of the Ministry of Internal Affairs and Communications.

10. REFERENCES

[1] Apache. Hadoop Core. `http://hadoop.apache.org/core`.

[2] P. Castagna. Having fun with PageRank and MapReduce. Hadoop User Group UK talk, `http://static.last.fm/johan/huguk-20090414/paolo_castagna-pagerank.pdf`.

[3] Y. Chen, A. Ganapathi, and R. H. Katz. To compress or not to compress - compute vs. IO tradeoffs for MapReduce energy efficiency. In *Proc. of Green Networking*, pages 23–28, New York, NY, USA, 2010. ACM.

[4] R. Chiang and H. Huang. TRACON: Interference-aware scheduling for data-intensive applications in virtualized environments. In *Proc. of SC*, pages 1 –12, nov. 2011.

[5] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[6] A. Gulati, G. Shanmuganathan, I. Ahmad, C. Waldspurger, and M. Uysal. Pesto: online storage performance management in virtualized datacenters. In *Proc. of SOCC*, pages 19:1–19:14, New York, NY, USA, 2011. ACM.

[7] H. Herodotou. Hadoop performance models. Technical report, Duke University, 2010. `http://www.cs.duke.edu/starfish/files/hadoop-models.pdf`.

[8] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *Proc. of SOCC*, pages 18:1–18:14, New York, NY, USA, 2011. ACM.

[9] Y. Huai, R. Lee, S. Zhang, C. H. Xia, and X. Zhang. DOT: a matrix model for analyzing, optimizing and deploying software for big data analytics in distributed systems. In *Proc. of SOCC*, pages 4:1–4:14, New York, NY, USA, 2011. ACM.

[10] E. Jahani, M. J. Cafarella, and C. Ré. Automatic optimization for mapreduce programs. *Proc. VLDB Endow.*, 4(6):385–396, Mar. 2011.

[11] A. Jindal, J.-A. Quiané-Ruiz, and J. Dittrich. Trojan data layouts: right shoes for a running elephant. In *Proc. of SOCC*, pages 21:1–21:14, New York, NY, USA, 2011. ACM.

[12] K. Kambatla, A. Pathak, and H. Pucha. Towards optimizing hadoop provisioning in the cloud. In *Proc. of HotCloud*, Berkeley, CA, USA, 2009. USENIX.

[13] H. Li, Y. Wang, D. Zhang, M. Zhang, and E. Y. Chang. PFP: Parallel FP-growth for query recommendation. In *Proc. of RecSys*, pages 107–114, New York, NY, USA, 2008. ACM.

[14] M. P. Mesnier, M. Wachs, R. R. Sambasivan, A. X. Zheng, and G. R. Ganger. Modeling the relative fitness of storage. In *Proc. of SIGMETRICS*, pages 37–48, New York, NY, USA, 2007. ACM.

[15] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[16] X. Pu, L. Liu, Y. Mei, S. Sivathanu, Y. Koh, and C. Pu. Understanding performance interference of I/O workload in virtualized cloud environments. In *Proc. of CLOUD*, pages 51 –58, july 2010.

[17] H. Shan, K. Antypas, and J. Shalf. Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark. In *Proc. of SC*, pages 42:1–42:12, Piscataway, NJ, USA, 2008. IEEE Press.

[18] TeraSort. `http://sortbenchmark.org/`.

[19] A. Verma, L. Cherkasova, and R. H. Campbell. ARIA: automatic resource inference and allocation for mapreduce environments. In *Proc. of ICAC*, pages 235–244, New York, NY, USA, 2011. ACM.

[20] X. Wang, C. Olston, A. D. Sarma, and R. Burns. CoScan: cooperative scan sharing in the cloud. In *Proc. of SOCC*, pages 11:1–11:12, New York, NY, USA, 2011. ACM.

[21] S. Wu, F. Li, S. Mehrotra, and B. C. Ooi. Query optimization for massively parallel data processing. In *Proc. of SOCC*, pages 12:1–12:13, New York, NY, USA, 2011. ACM.

[22] H. Yang, Z. Luan, W. Li, and D. Qian. MapReduce workload modeling with statistical approach. *Journal of Grid Computing*, 10:279–310, 2012. 10.1007/s10723-011-9201-4.