# IBIS: Interposed Big-data I/O Scheduler

Yiqi Xu
Florida International University
11200 SW 8th St
Miami, FL 33199 USA
yxu006@cs.fiu.edu

Ming Zhao
Arizona State University
699 S Mill Ave
Tempe, AZ 85281 USA
mingzhao@asu.edu

## ABSTRACT

Big-data systems are increasingly shared by diverse, data-intensive applications from different domains. However, existing systems lack the support for I/O management, and the performance of big-data applications degrades in unpredictable ways when they contend for I/Os. To address this challenge, this paper proposes *IBIS*, an Interposed Big-data I/O Scheduler, to provide I/O performance differentiation for competing applications in a shared big-data system. IBIS transparently intercepts, isolates, and schedules an application's different phases of I/Os via an I/O interposition layer on every datanode of the big-data system. It provides a new proportional-share I/O scheduler, SFQ(D2), to allow applications to share the I/O service of each datanode with good fairness and resource utilization. It enables the distributed I/O schedulers to coordinate with one another and to achieve proportional sharing of the big-data system's total I/O service in a scalable manner. Finally, it supports the shared use of big-data resources by diverse frameworks and manages the I/Os from different types of big-data workloads (e.g., batch jobs vs. queries) across these frameworks. The prototype of IBIS is implemented in Hadoop/YARN, a widely used big-data system. Experiments based on a variety of representative applications (WordCount, TeraSort, Facebook, TPC-H) show that IBIS achieves good total-service proportional sharing with low overhead in both application performance and resource usages. IBIS is also shown to support various performance policies: it can deliver stronger performance isolation than native Hadoop/YARN (99% better for WordCount and 15% better for TPC-H queries) with good resource utilization; and it can also achieve perfect proportional slowdown with better application performance (30% better than native Hadoop).

## 1. INTRODUCTION

Big data is an important computing paradigm that becomes increasingly used by many science, engineering, medical, and business disciplines for knowledge discovery, decision making, and other data-driven tasks based on processing and analyzing large volumes of data. These applications are built upon computing paradigms that can effectively express data parallelism and exploit data locality (e.g., MapReduce [9]) and storage systems that can provide high scalability and availability (e.g., Google File System [10], Hadoop HDFS [16]). As the needs of data-intensive computing continue to grow in various disciplines, it becomes increasingly common to use shared infrastructure to run such applications. First, big-data systems often require substantial investments on computing, storage, and networking resources. Therefore, it is more cost-effective for both resource users and providers to use shared infrastructure for big-data applications. Second, hosting popular data sets (e.g., human genome data, weather data, census data) on shared big-data systems allows such massive data to be conveniently and efficiently shared by different applications from different users.

Although computing resources (CPUs) are relatively easy to partition, shared storage resources (I/O bandwidths) are difficult to allocate, particularly for data-intensive applications which compete fiercely for access to large volumes of data on the storage. Existing big-data systems lack the mechanisms to effectively manage shared storage I/O resources, and as a result, applications' performance degrades in unpredictable ways when there is I/O contention. For example, when one typical MapReduce application (WordCount) runs concurrently with a highly I/O-intensive application (TeraGen), WordCount is slowed down by up to 107%, compared to when it runs alone with the same number of CPUs.

I/O performance management is particularly challenging for big-data systems because of two important reasons. First, big-data applications have complex I/O phases (e.g., rounds of map and reduce tasks with different amounts of inputs, intermediate results, and outputs for a MapReduce application), which makes it difficult to understand their I/O demands and allocate I/O resources properly to meet their performance requirements. Second, a big-data application is highly distributed across many datanodes, which makes it difficult to coordinate the resource allocations across all the involved nodes needed by the data-parallel application. For example, the performance of a MapReduce application depends on the received total storage bandwidth from all the nodes assigned to its map and reduce tasks.

This paper proposes *IBIS*, an Interposed Big-data I/O Scheduler, to provide performance differentiation for competing applications' I/Os in a shared big-data system. This scheduler is designed to address the above-mentioned two challenges. First, *how to effectively differentiate I/Os from competing applications and allocate the shared storage bandwidth on the individual nodes of a big-data system?* IBIS introduces a new I/O interposition layer upon the distributed file system in a big-data system, and is able to transparently intercept the I/Os from the various phases of applications and isolate and schedule them on every datanode of the system. IBIS also employs a new proportional-share I/O scheduler, SFQ(D2), which can automatically adapt I/O concurrency based on the storage load



*HPDC'16, May 31-June 04, 2016, Kyoto, Japan*
 ISBN 978-1-4503-4314-5/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2907294.2907319

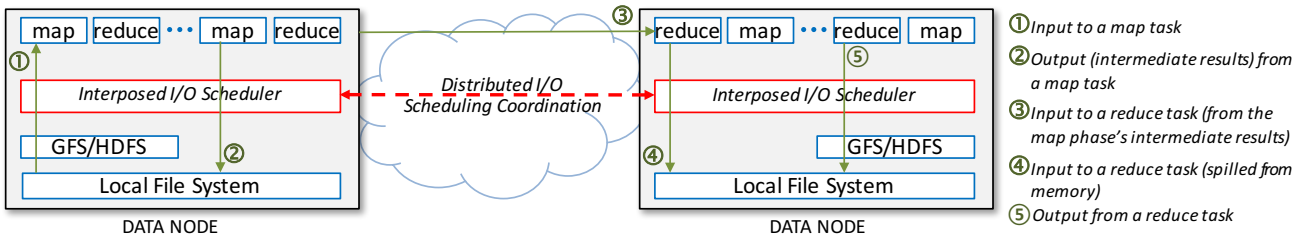



Figure 1: Architecture of MapReduce-type big-data systems and the proposed IBIS-based I/O management

and achieve strong performance isolation with good resource utilization. Second, *how to efficiently coordinate the distributed I/O schedulers across datanodes and allocate the big-data system's total I/O service to the data-parallel applications?* IBIS provides a scalable coordination scheme for the distributed SFQ(D2) schedulers to efficiently coordinate their scheduling across the datanodes. The schedulers then adjust their local I/O scheduling based on the global I/O service distribution and allow the applications to proportionally share the entire system's total I/O service.

The IBIS prototype is implemented in Hadoop/YARN, a widely used big-data system, by interposing HDFS as well as the related local and network I/Os transparently to the applications, and it is able to support the I/O management of diverse applications from different big-data frameworks. It is evaluated using a variety of representative big-data applications (WordCount, TeraSort, TeraGen, Facebook2009 [6], TPC-H queries on Hive [17]). The results confirm that IBIS can effectively achieve total-service proportional bandwidth sharing for diverse applications in the system. They also show that IBIS can support various important performance polices. It achieves strong *performance isolation* for a less I/O-intensive workload (WordCount, Facebook2009, TPC-H) when under heavy contention from a highly I/O-intensive application (TeraGen and TeraSort), which outperforms native Hadoop by 99% for WordCount and 15% for TPC-H queries. This result is accomplished while still allowing the competing application to make good progress and to fully utilize the storage bandwidth ($<4\%$ reduction in total throughput). IBIS can also achieve excellent *proportional slowdown* for competing applications (TeraSort vs. TeraGen) and outperforms native Hadoop by 30%. Finally, the use of IBIS introduces small overhead in terms of both application runtime and resource usages.

Overall, unlike most of the related works which focus on improving the I/O efficiency of big-data systems [8, 11], this paper addresses the problem of I/O interference and performance management in big-data systems, which is not adequately addressed in the literature. Although existing mechanisms such as cgroups [4] can be employed to manage the contention among local I/Os, as the results in this paper will show, they are insufficient due to the lack of control on distributed I/Os which are unavoidable for big-data applications. IBIS therefore complements the existing solutions for CPU and memory management of big-data systems, and provides the missing control knob for I/O management which is much needed by increasingly data-intensive applications. Compared to the few related works [20, 15, 19] that also studied the performance management of big-data storage, IBIS supports applications that are more challenging (with complex computing and I/O demands) and diverse (including both batch and query workloads).

The rest of the paper is organized as follows: Section 2 introduces the background and motivating examples; Sections 3, 4, and 5 describe the I/O interposition framework, SFQ(D2) scheduler, and distributed scheduling coordination of IBIS, respectively; Section 6 discusses the support for I/O management across different big-data frameworks; Section 7 presents the experimental evaluation; Section 8 examines the related work; Section 9 discusses the limitations and future work; and Section 10 concludes the paper.

# 2. BACKGROUND AND MOTIVATIONS

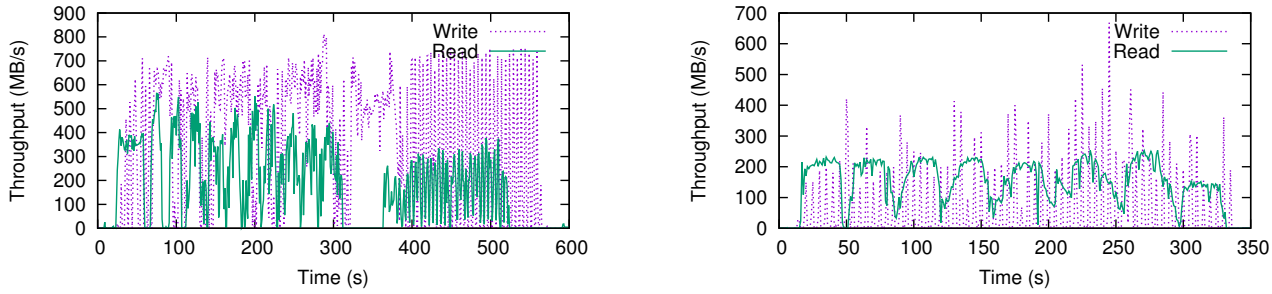
## 2.1 Big-data Systems

Typical big-data computing systems are often built upon a highly scalable and available distributed file system. In particular, Google File System (GFS) [10] and its open-source clone Hadoop Distributed File System (HDFS) [16] provide storage for massive amounts of data on a large number of nodes built with inexpensive commodity hardware while supporting fault tolerance at scale. A big-data application runs many tasks on these datanodes, which process the locally stored data in parallel via the I/O interface provided by such a distributed file system. In particular, the MapReduce programming model and associated run-time system are able to automatically execute user-specified map and reduce functions in parallel and handle job scheduling and fault tolerance [9]. Higher-level storage services such as databases (e.g., Hive [17]) can be further built upon the distributed file system and offer more convenient interfaces (e.g., SQL) for users to process the data. Therefore, this paper focuses on big-data storage systems of the GFS/HDFS kind.

Both the map and reduce phases of a MapReduce application can spawn large numbers of map and reduce tasks on the GFS/HDFS nodes to process data in parallel. They often have complex but well-defined I/O phases (Figure 1). A *map* task is preferably scheduled to the node where its input data is stored. It reads the input from GFS/HDFS (either via the local file system or across the network) and *spills* and *merges* key-value pairs onto the local file system as intermediate result. A *reduce* task starts by *copying/shuffling* its inputs from all the map tasks' intermediate results (either stored locally or across the network). It then *merges* the copied inputs, performs the *reduce* processing, and generates final output to GFS/HDFS. Each of the above phases can have different bandwidth demands for input and output. Moreover, given the same volume of data to a map or reduce task, it can take different amount of time to process the data depending on the application's computational complexity.

## 2.2 Big-data Resource Management

Existing big-data systems offer simple core resource management functions. Hadoop MapReduce [1] allocates CPU resources in terms of *slots* to map or reduce tasks, where the number of available slots is set according to the number of CPU cores in the system. Recent developments such as Mesos [12] and YARN [18] allow the




(a) TeraSort (b) WordCount

Figure 2: I/O demands of two classic MapReduce applications

allocation of both CPU and memory resources to competing big-data applications. The management of shared I/O bandwidth is still missing from existing solutions, which is however crucial to the performance of inherently I/O intensive big-data applications.

The performance management problem for big-data storage is not adequately addressed in the literature. Frosting [20] provides a scheduling layer upon HBase [3], but it treats the entire HBase stack as a single black box and it is thus difficult to achieve strong performance isolation and good resource utilization. PISCES [15] achieves fair-sharing of a key-value store by controlling the dispatching of simple requests to datanodes, and Cake [19] provides QoS support to HBase queries by controlling the queuing of simple requests. In comparison, IBIS is designed to manage the I/O performance for diverse big-data applications including those with much more complex and dynamic I/O demands. A detailed examination of the related work is presented in Section 8.

## 2.3 Motivating Examples

The lack of I/O management in big-data systems presents a serious hurdle for data-intensive applications to get their desired performance. In a MapReduce system, on every single datanode, the tasks from different MapReduce applications compete with one another across all their phases for HDFS, local file system, and network I/Os. Across the whole big-data system, these highly distributed applications also compete on many datanodes and their performance depends on the total amount of I/O services that they can get from all the involved nodes.

As an example of the diverse I/O demands of big-data applications, Figure 2 compares the I/O profiles of two classic MapReduce applications, *TeraSort* and *WordCount*, each running alone with the same allocation of CPU and memory resources. These profiles show that *TeraSort* has a much more intensive I/O workload than *WordCount*. *TeraSort* has intensive HDFS reads and local file system writes in the map phase and intensive HDFS writes in the reduce phase. *WordCount*'s output is much smaller than its input, but there are plenty of intermediate writes throughout the map and reduce phases.

With such diverse big-data applications, the lack of I/O management will lead to severe and unpredictable performance interference between the applications. As an example of the I/O contention's performance impact, Figure 3 compares the performance of WordCount when it runs alone to when it runs with another application (TeraGen, TeraSort, TeraValidate) while keeping its CPU allocation (half of the 96 CPU cores in the system) the same. Details of the experiment setup are provided in Section 7. The results show substantial performance degradation in WordCount, which confirms the significant performance impact caused by I/O contention (CPU cache contention is relatively insignificant to the performance of these data-intensive applications). This paper addresses this serious problem with an interposed big-data I/O scheduling approach, IBIS, which is presented in the rest of the paper.

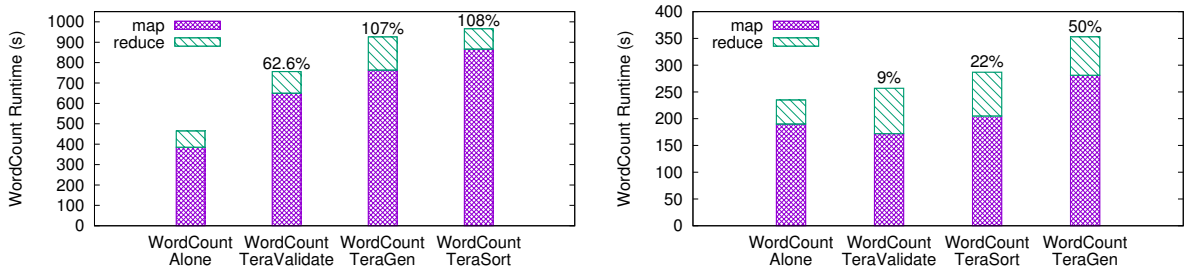
# 3. INTERPOSED I/O SCHEDULING

The first question addressed by IBIS is *how to effectively differentiate the I/Os across the different phases of competing MapReduce applications on every datanode of a big-data system*. The general design of IBIS is based on the *virtualization* principles, where an indirection layer exposes the interfaces already in use by the big-data system to access storage, allowing applications to time-share the storage system without modifications, while enforcing performance isolation and differentiation among them.

A key design decision that needs to be made in a virtualization approach is choosing the proper abstraction to introduce the virtualization layer. In the context of a big-data system, there are multiple layers in the storage hierarchy, from the applications, to HDFS, and to local file system and storage devices. On one hand, introducing virtualization at a higher layer can make use of more application knowledge to help the implementation, but it is more tied to specific applications and loses control of how I/Os are executed by the underlying layers. On the other hand, introducing virtualization at a lower layer of the storage hierarchy allows more control of I/O executions and can support more diverse applications, but it has to deal with more primitive I/O operations and loses application semantics that are useful for I/O differentiation.

Considering this tradeoff, IBIS is introduced upon the GFS/HDFS layer of the MapReduce storage architecture (Figure 1). This design can make effective use of application information to differentiate I/Os, because applications access the shared storage mostly through the HDFS interface. It also has enough low-level I/O control by scheduling the dispatch of I/Os to local file systems. Interposing of the applications' direct local file system I/Os and network I/Os are done at other interfaces at the same level. The rest of this section details the interposition of these different types of I/Os used by a MapReduce application. All the modifications described below for implementing IBIS are made to Hadoop/YARN and do not require any change to applications.

*Persistent I/Os* are I/Os serviced by HDFS, where the inputs for map tasks are read from HDFS, and the outputs from reduce tasks are written to HDFS. Tasks use the *DFSClient* to interface with the *Data Node*, which represents an HDFS daemon, and Data Node converts the data requests, from both local and remote map tasks, to local file system I/Os. To differentiate I/Os from competing applications, the DFSClient interface is modified to carry application-specific information (job identifier and I/O service weight) as part of the header of each data request issued by the map/reduce tasks.




(a) HDD setup

(b) SSD setup

Figure 3: Runtime of WordCount when it runs alone vs. when it runs with another job on native Hadoop. The numbers on top of the bars are the slowdown w.r.t. the standalone runtime. The CPU allocation to WordCount is kept the same in all cases. Two different storage setups are considered, one with all hard disk drives (HDDs) and the other with all solid state drives (SSDs).

These requests are scheduled by the IBIS component implemented in Data Node, which maintains a request queue for its local storage and dispatches the queued requests according to the chosen scheduling algorithm and policy.

*Intermediate I/Os* are I/Os to a datanode's local file system (not HDFS) for storing temporary data. Both map and reduce tasks use the local file system for spilling and merging in-progress data. The intermediate I/Os can also influence an application's performance. For example, a sorting program can generate the same amount of intermediate data as its input. In IBIS, these intermediate I/Os are first tagged with the job identifier and I/O service weight and then routed to the IBIS component implemented within a local I/O scheduler, which can also reside in the Data Node daemon that runs on every datanode. IBIS schedules the intermediate I/Os in the same way as the persistent I/Os, following the same scheduling algorithm and policy.

*Network I/Os* occur during a shuffling phase between all the map tasks and reduce tasks. Because each reduce task's input is a partition of the map phase's outputs, it generally has to request a portion of the outputs from every map task. The data pulling thread launched by a reduce task is initiated with the job identifier and I/O service weight, which are carried over in the header of every HTTP-based data request. These requests are handled by the HTTP servlets which are implemented in the *Node Manager* daemons. Therefore, an IBIS scheduler is also implemented in the *Node Manager* to differentiate the network I/Os and schedule the corresponding local file system I/Os.

Note that IBIS does not rely on any bandwidth control from the network layer, and it is shown to be sufficient in the experiments because of two reasons: *1)* The storage is generally saturated before the network; *2)* By applying bandwidth control at the storage endpoints of the network I/Os, IBIS indirectly influences the contention on the network. However, IBIS can incorporate the network bandwidth control mechanisms such as OpenFlow [5] if they are necessary and available, which will be left for future work.

In all the above I/O phases, concurrent requests from different applications are differentiated by their unique application IDs. An application obtains its ID from the job scheduler, which is carried over to all of its parallel tasks and used by the tasks to tag their I/Os for HDFS, intermediate, and network data. For every shared I/O service, these requests are queued and dispatched by an IBIS scheduler according the algorithm presented in the next section.

# 4. PROPORTIONAL I/O SHARING

The second question addressed by IBIS is *how to allow the tasks from competing applications to proportionally share the I/O service of each datanode in a big-data system.* The interposed I/O scheduling framework in IBIS is flexible enough to support different algorithms. This paper focuses on algorithms that allow applications to proportionally share the I/O bandwidth, in the same way they share the CPU time proportionally (e.g., using the Hadoop Fair Scheduler [2]), so that it can provide the much needed, missing control knob for I/O allocation in big-data systems. Proportional resource sharing is defined as when the total demand is greater than the available resource, each application should get a *share* of the resource proportional to its assigned *weight*. Because only the relative values of weights matter to the bandwidth allocation, in the paper, the weight assignment to applications is often specified in terms of the *ratio* among the weights.

The proposed proportional-share scheduler is built upon the SFQ family of schedulers because of their computational efficiency, work-conserving nature, and theoretically provable fairness. SFQ schedules the backlogged requests from different applications using a priority queue, where each request's priority is positively affected by its application's weight and negatively affected by its cost (often estimated based on the size of the request). The scheduler can dispatch only one outstanding request, and it chooses the one with the earliest *start time* in the queue.

The SFQ(D) scheduler [13] is an extension of SFQ for proportional sharing of storage resources which are commonly capable of handling multiple outstanding requests concurrently. The level of concurrency that the shared storage resource supports is captured by the *depth* parameter $D$ in SFQ(D). The scheduler follows the original SFQ algorithm to dispatch queued requests, but it allows up to $D$ outstanding I/Os to be serviced concurrently by the underlying storage in order to take advantage of the available I/O concurrency.

The choice of $D$ has important implications on both fairness and resource utilization for a real storage system. On one hand, a larger $D$ allows more concurrent I/Os and a higher utilization of the storage, but it may hurt fairness because of the scheduler's work-conserving nature. A more aggressive workload can use up all the storage bandwidth and even overload it, delaying the I/Os from a less aggressive workload. On the other hand, a smaller $D$ gives the scheduler a tighter control on the amount of I/O share that a more aggressive workload can steal from others, and allows the I/Os from a less aggressive workload to be serviced quickly when they arrive. It can thus improve fairness among the compet-

ing workloads but may lead to underutilization of the storage. So it is difficult to determine the optimal value of $D$ statically, and it depends on the characteristics of the storage and workloads, some of which are also dynamic. It was in fact left as future work in the SFQ(D) paper [13].

To address the above problem, this paper introduces a new SFQ-based algorithm, *Dynamic Depth SFQ*, or *SFQ(D2)* in short. It employs a feedback controller to automatically and dynamically adjust the value of $D$ online. The controller works periodically (e.g., every second), and decides the depth $D_{k+1}$ for the next period $k+1$, based on the distance between the observed average I/O latency $L_k$ of the previous period and the reference latency $L_{ref}$:

$$D_{k+1} = D_k + K \times (L_{ref} - L_k) \quad (1)$$

where $K$ is an integral gain factor which determines how aggressively the controller works to reach the target latency. Following this equation, the controller automatically optimizes the value of $D$ as it steers the observed I/O latency towards the reference latency.

The controller chooses I/O latency as the target because its goal is to maximize the storage utilization without compromising the fairness among applications, and I/O latency directly reflects the I/O performance of applications and the I/O load of the underlying storage. The reference latency is decided offline by profiling the storage using a synthetic MapReduce workload with increasing I/O concurrency. Both the I/O latency and throughput are measured during the profiling, and the I/O latency observed before the storage starts to saturate is the reference latency for the controller. Such profiling needs to be done only once for a given storage setup. If the storage's read and write performance are asymmetric such as in SSDs, the profiling can give separate reference latencies for reads and writes. In this case, the $L_{ref}$ and $L_k$ in the controller become the weighted average of the read latencies and write latencies, with the weights being the percentages of reads and writes observed in the previous control period.

This SFQ(D2) scheduler works upon the interposition layer described in Section 3 on every datanode of the big-data system. Each scheduler independently adjusts $D$ based on its local dynamics in the workloads and underlying storage, and dispatches up to $D$ I/Os from its local queue to the storage. This scheduler is used to provide proportional sharing of all the important I/O services offered by a datanode, including HDFS I/Os, temporary data I/Os, and network I/Os.

# 5. DISTRIBUTED I/O SCHEDULING COORDINATION

The third question addressed by IBIS is *how to efficiently coordinate the distributed I/O schedulers across datanodes to support proportional sharing of a big-data system's total I/O service among competing applications.* A limitation of the IBIS scheduler described above is that a local scheduler's decision is made independently at each datanode, without accounting for information from other nodes. Local scheduling, based on only local knowledge, however, is not sufficient to deliver the desired performance differentiation from the perspective of the highly distributed big-data applications. The parallel nature of such an application requires it to get the necessary I/O service from all the nodes where its tasks are scheduled, and its performance depends on the total amount of I/O service that it gets from the system. Therefore, I/O management at the system level should support *total-service proportional sharing*, which means that the applications share the total I/O service from all the datanodes in the system proportionally to their assigned weights.

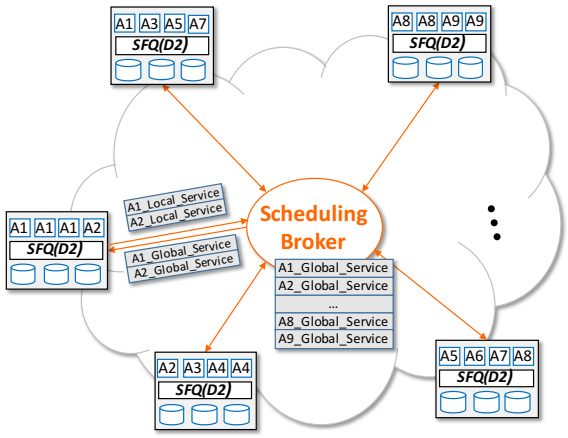



Figure 4: Architecture for distributed I/O scheduling coordination

The challenge to achieving total-service proportional sharing is that applications often get unevenly distributed I/O services from the involved nodes. The exact amount of service that an application gets from a particular node depends on the number of CPU slots that it gets on the node—which decides the I/O demands, and the applications running on the other slots of the same node—which decides the I/O contention. The number of slots that an application gets on a node in turn depends on the combination of, at any moment, the global CPU slot allocation policy, the application's data locality on the node, and the number of slots currently available on the node. Because of such uneven distribution of I/O service across the nodes, simply applying the same sharing ratio to each node and enforcing it using the local SFQ(D2) scheduler will not produce the same ratio of sharing of the total I/O service.

To address this challenge, IBIS enables the distributed SFQ(D2) schedulers to coordinate with one another and enforce total-service proportional sharing collaboratively. Every scheduler shares its local I/O service distribution—the applications that it serves and the amounts of services that they get locally, with the other schedulers. Based on the global I/O service distribution, every scheduler can then adjust its local I/O service distribution so that the total services that the applications get are proportional to their assigned weights. Specifically, IBIS follows the algorithm in DSFQ [21] to adjust local SFQ scheduling for total-service proportional sharing. When an SFQ(D2) scheduler considers the scheduling of a queued request, it delays the request's *start time* by the total amount of service that the corresponding application has received from all the other nodes. In this way, the local scheduler dispatches the requests from different applications according to their received total I/O services, not just the local services.

Another challenge that must be addressed by IBIS is how to efficiently coordinate a large number of distributed schedulers in a big-data system. If every scheduler has to broadcast its information to all the other schedulers, it can easily overwhelm the schedulers and the network as the system scales out. The DSFQ [21] work assumes a traditional remote I/O model, where the clients send their I/Os to remote datanodes and a coordinator can be interposed in between to gather and pass on the global I/O service information. But this approach does not apply to a big-data system, where computing tasks are shipped to the nodes where their data is stored and they process the data using primarily local I/Os.

To solve this problem, IBIS employs a centralized *Scheduling Broker* to facilitate the information exchange among the distributed

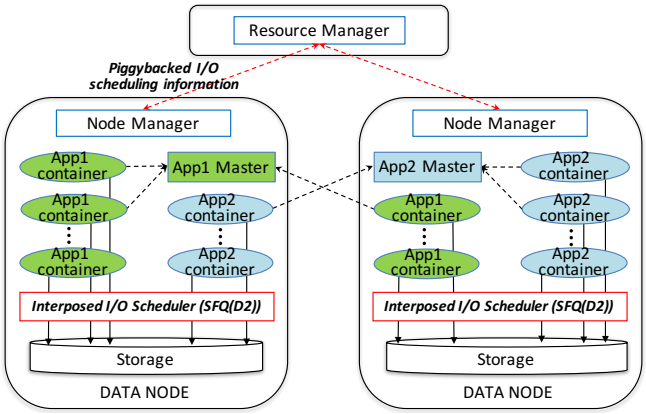



Figure 5: Integration of IBIS with YARN for supporting the I/O management of applications running on different big-data frameworks.

schedulers in a scalable manner (Figure 4). Every local scheduler $j \in \{1,\ldots,m\}$ sends its current I/O service distribution—a vector of *local* I/O service amount $a_{ij}$ for each application $i \in \{1,\ldots,n\}$ that the scheduler $j$ serves—to the broker periodically (e.g., every 1 second). Based on the information received from all the local schedulers, the broker summarizes the total I/O service $A_i = \sum_{j=1}^{m} a_{ij}$ for each application $i$ in the system. It then responds to a local scheduler's message with the total I/O service distribution—a vector of *total* I/O service amount $A_i$ for each application $i$ that the local scheduler currently serves. Based on this total service information, the local scheduler can then adjust its scheduling as discussed above.

The overhead of this scheduling coordination scheme is small. The size of the messages between a local scheduler and the broker is bounded by the number of applications that the scheduler currently serves. The state that the broker needs to maintain is simply a vector of total I/O service amount for all the applications currently in the system. The frequency of coordination can be adjusted based on the desired granularity of fairness and the scale of the system—more frequent coordination reduces transient unfairness but increases the overhead; and vice versa. Hadoop/YARN already employs centralized managers, in particular the *Resource Manager* for coordinating the distributed *Node Managers*, which is shown to be scalable for managing thousands of nodes [18]. In fact, in the IBIS implementation, the I/O Scheduling Broker is embedded as part of the Resource Manager and the I/O scheduling coordination information is piggybacked on the existing communications between the managers to further reduce its overhead.

# 6. MULTI-FRAMEWORK I/O SCHEDULING

Big-data resources are increasingly shared by diverse computing frameworks [9, 17], as users have different data processing requirements as well as different preferences of programming models. No single framework is perfect for all big-data problems and all users. Solutions such as YARN [18] and Mesos [12] allow different frameworks to share the same set of resources and employ mechanisms such as containers [4] to allocate CPU cores and memory capacity to the resource-sharing applications. However, these resource management solutions still cannot provide strong performance isolation, because they do not support the allocation of shared I/O resources which the data-intensive applications have to compete for. As the experiments will show in Section 7.4, although containers do provide some level of I/O isolation, it is not sufficient. Containers can control only the I/Os directly issued to the local file system, e.g., intermediate I/Os from MapReduce, but not the distributed I/Os, e.g., HDFS I/Os, which are serviced by a shared datanode server and cannot be differentiated using the container mechanism.

Thus, existing multi-framework resource management solutions still need IBIS to provide the missing I/O control knob for effective I/O bandwidth allocation. Specifically, in YARN, IBIS is seamlessly integrated in its *Application Master*, *Resource Manager*, *Node Manager*, and *Data Node* components (Figure 5). IBIS allows an application to specify its required total I/O bandwidth (e.g., 300MB/s) to its *Application Master*, in addition to the amount of required CPUs and memory (e.g., 64 CPU cores and 64GB RAM), in order to achieve its desired performance. The centralized *Resource Manager* collects the resource requests from the concurrent *Application Masters* and uses IBIS to determine the global, total-service I/O bandwidth allocation, in addition to allocating the CPUs and memory using an existing scheduler such as the Fair Scheduler. The *Resource Manager* then coordinates with the distributed *Node Managers* to enforce the resource allocations on every datanode. Each *Node Manager* uses the local *Data Node* to schedule the local I/Os according to the global, total-service I/O sharing target, similarly to how it uses containers to enforce the CPU and memory allocations to the local data processing tasks. Finally, the IBIS scheduler in *Data Node* interposes all the I/Os, as discussed in Section 3 and uses SFQ(D2) discussed in Section 4 to schedule the I/Os according to the given bandwidth allocation.

Table 1: The YARN configuration used in the evaluation

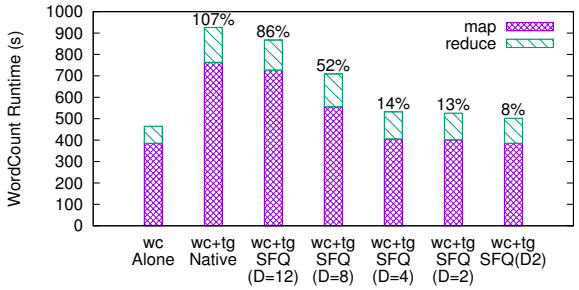
| Key | Value |
|---|---|
| dfs.replication | 3 |
| dfs.block.size | 134,217,728 |
| fairscheduler.preemption | true, 5s |

# 7. EVALUATION

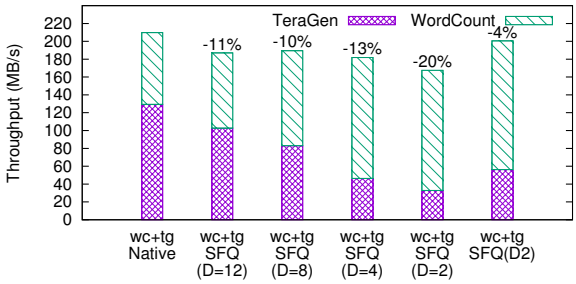
## 7.1 Setup

The experimental evaluation was done on a cluster of nine nodes each with two six-core 2.4GHz AMD Opteron CPUs, 32GB of RAM, and two 500GB 7.2K RPM SAS disks, interconnected by a Gigabit Ethernet switch. All the nodes run the Debian 4.3.5-4 Linux with the 3.2.20-amd64 kernel and use EXT3 as the local file system. The evaluation was performed in YARN 2.7.0 with the IBIS prototype implemented in the *Resource Manager*, *Node Manager*, *Application Master*, and *Data Node* as described in Sections 3 and 6. Eight nodes are dedicated to run applications consuming up to 96 CPU cores and 192GB memory by their tasks, where each map task uses 1 CPU core and 2GB of memory and each reduce task uses 1 CPU core and 8GB of memory. One additional node runs the YARN *Resource Manager* and *Name Node* and the IBIS scheduling broker. The two disks on each node are used to store HDFS data and intermediate data separately. The configuration parameters of YARN and its Fair Scheduler used in the evaluation are listed in Table 1.

The evaluation compares the performance of IBIS to native Hadoop with YARN using a variety of benchmarks, including TeraGen (1TB output), TeraSort (50–400GB input), WordCount (50GB Wikipedia input), Facebook2009 [6], and TPC-H on Hive [17] (53GB input), which are explained in detail in the following experiments. For IBIS with the SFQ(D2) scheduler, the control period is set to 1 second.




(a) Runtime of WordCount. The numbers on top of the bars are the slowdown w.r.t. the standalone runtime. The shuffling time of the first wave of reduce tasks is overlapped with the map phase and not shown in the bars. But the height of the bars reflects the total runtime.




(b) Total throughput of WordCount and TeraGen. The numbers on top of the bars are the throughput loss w.r.t. the native case.

Figure 6: Performance of WordCount (*wc*) when it runs alone and against TeraGen (*tg*) in an HDD-based storage setup

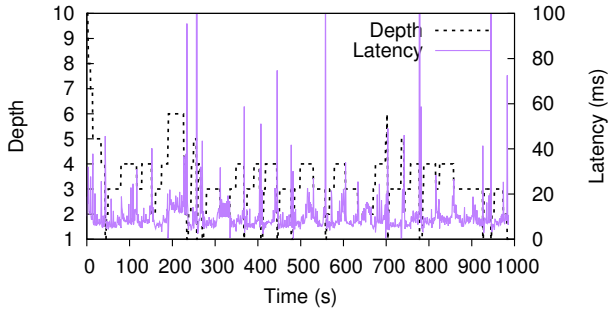



Figure 7: Adaptation of $D$ by SFQ(D2) based on the observed I/O latency on one datanode

## 7.2 Performance Isolation (WordCount)

The first experiment evaluates whether IBIS is able to provide performance isolation to one application while it is under intensive I/O contention from others as in the motivating example discussed in Section 2.3. It is an important policy in many scenarios where the performance of an important big-data application must be guaranteed regardless of the contention from others. As in the motivating example, Figure 6a shows that when WordCount runs with TeraGen, it is slowed down by 107% due to I/O contention, compared to when it runs alone with the same CPU and memory allocation (48 CPU cores and 96GB RAM). Performance isolation is challenging to accomplish for WordCount because its I/O rate is much lower than TeraGen, while a work-conserving I/O scheduler tries not to underutilize the storage.

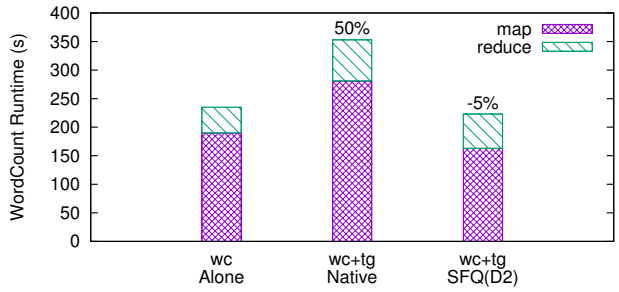
Figure 6a shows the results of IBIS from using both the classic SFQ(D) scheduler with a static value of $D$ and the new SFQ(D2) scheduler which dynamically adjusts $D$. The sharing ratio between WordCount and TeraGen is set to 32:1 to favor WordCount, but TeraGen can always use the spare I/O bandwidth because the schedulers are work-conserving. Comparing the results from SFQ(D) with different $D$ values, it shows that reducing $D$ does give the scheduler a tighter control on I/O scheduling and achieves better performance isolation for WordCount, reducing its slowdown to as low as 13%. Comparing the results from SFQ(D) to SFQ(D2), it shows that the new scheduler achieves the best isolation for WordCount with a runtime that is only 8% slower than when it runs alone, and it does so by automatically adjusting the value of $D$.

Note that the 32:1 sharing ratio is used here because the objective of this experiment is to restore the performance of WordCount without underutilizing the bandwidth. Lower sharing ratios would favor WordCount less and result in worse performance of WordCount while still being much better than the native case. For example, a sharing ratio of 2:1 restores WordCount's performance to 148% of its standalone runtime with SFQ(D=2) and 118% with SFQ(D2).

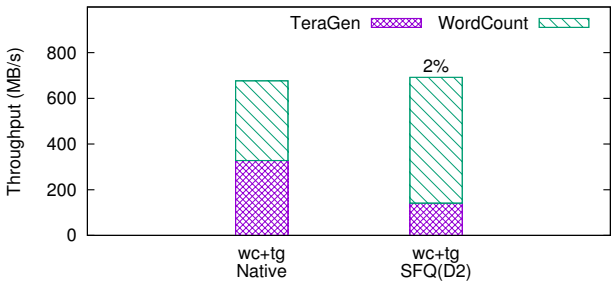
The excellent performance isolation from IBIS is accomplished while still allowing the competing application, TeraGen, to make good progress and fully utilize the underlying storage. To confirm this, Figure 6b compares the total throughput of WordCount and TeraGen when they run on native Hadoop without I/O management vs. when they run on IBIS. The native case has the highest total throughput, because TeraGen's I/Os are sent to storage as soon as they come without any control. In comparison, the number of outstanding I/Os is controlled by $D$ in the schedulers of IBIS. The results show that IBIS can achieve good storage utilization in all configurations, where the best result is also from SFQ(D2) which is only 4% lower than the native case. This result is achieved while reducing WordCount's runtime slowdown from 107% to 8% as discussed above.

To provide a detailed view of how SFQ(D2) works, Figure 7 shows how it adapts $D$ based on the observed I/O latency on one of the datanodes. It follows the equation for the feedback controller described in Section 4. The gain factor is set to $10^{-6}$. The value of $D$ is bounded between 1 and 12. Throughout the run the controller reacts quickly to the observed latency and adapts $D$ quickly to sustain strong performance isolation with good resource utilization. Noticeable that at the 260th second and 790th second, the underlying storage system undergoes foreground flushing of the writes buffered in memory and causes the high spikes in I/O latency, while the controller still responds in a timely manner. Although IBIS does not have direct control of such lower-level dynamics, it can still effectively mitigate their impact by timely adapting the I/O concurrency. It is therefore able to sustain good performance isolation without having to modify the underlying storage layers which would be much more intrusive and expensive.

Although faster storage devices such as SSDs are increasingly considered by big-data systems, they cannot completely replace HDDs due to their limited capacity. Moreover, faster storage does not make the I/O contention problem go away; the increasing volume and velocity of big data will always demand I/O management. To confirm this, the same experiment is repeated on a different storage setup using SSDs (Intel 120GB MLC SATA-interfaced flash devices) to store both HDFS and temporary data on each datanode. The results in Figure 8a first confirm that WordCount is still severely interfered (50% slowdown) by TeraGen on native Hadoop due to I/O contention. They also confirm that IBIS still achieves




(a) Runtime of WordCount




(b) Total throughput of WordCount and TeraGen

Figure 8: Performance of WordCount (*wc*) when it runs alone and against TeraGen (*tg*) in an SSD-based storage setup

strong performance isolation with excellent storage utilization for this faster storage setup. Interestingly, IBIS with SFQ(D2) achieves a better runtime for WordCount than when it runs alone, and a better total throughput for WordCount and TeraGen than native Hadoop. This can be explained by the read/write asymmetry of flash devices and the implicit promotion of reads in SFQ(D2). Writes are much slower than reads on flash devices and they can significantly slow down the reads that are scheduled after them. When intensive writes are received by the scheduler, it automatically reduces *D*, which gives the reads a better chance to establish backlogged requests and be dispatched before some of the writes, therefore achieving better overall performance. This unique characteristic of flash devices will be further studied in the future work to optimize the IBIS scheduler specifically for the use of SSDs in big-data systems.

## 7.3 Performance Isolation (Facebook2009)

The second experiment evaluates whether IBIS is also able to provide performance isolation to the Facebook2009 workload, which is far more diverse than WordCount. A total of 50 jobs are created using the SWIM workload generator [6], by sampling the historical Facebook job logs and emulating their computing and I/O phases. The samples are down-scaled to fit the size of this paper's testbed. The workload consists of diverse MapReduce applications, including both small and large jobs with different levels of I/O demands. Their input-to-shuffle ratio and shuffle-to-output ratio vary between 0.05 to $10^3$ and $2^{-5}$ to $10^2$ respectively. These ratios represent the relative data sizes between map input and shuffle input and between shuffle input and reduce output. Varying these ratios generates different levels of computation and I/O intensities for the various phases of the jobs.

The Facebook2009 jobs are run together with TeraGen on the native Hadoop (*Interfered*) and on IBIS using the SFQ(D2) scheduler with a bandwidth sharing ratio of 32:1 favoring Facebook jobs (*SFQ(D2)*). As a baseline, Facebook2009 is also run alone without I/O contention from others (*Standalone*). The CPU and memory resources allocated to Facebook2009 are kept to half of the total resources for all the cases.

Figure 9 compares the cumulative distribution of the Facebook2009 jobs' runtimes. In the *Standalone* case, 90% of Facebook2009 jobs finish within 120s. When they run together with TeraGen without I/O management in the *Interfered* case, they are impacted drastically by TeraGen, and no job finishes within 50s and 90% of them take up to 230s. In comparison, using *SFQ(D2)*, IBIS is indeed able to provide strong isolation to Facebook2009, and 90% of the jobs can finish within 138s. Comparing the average runtime of Facebook2009, it is reduced from 168s in the *Interfered* case to 115s under *SFQ(D2)*, where the *Standalone* average runtime is 98s. Most of these jobs require only one wave of map and reduce tasks. Without an I/O scheduler, their I/Os are severely interfered by TeraGen and slowed down substantially. With IBIS, they are well isolated from TeraGen and can utilize the allocated storage bandwidth to achieve a performance close to the standalone case.

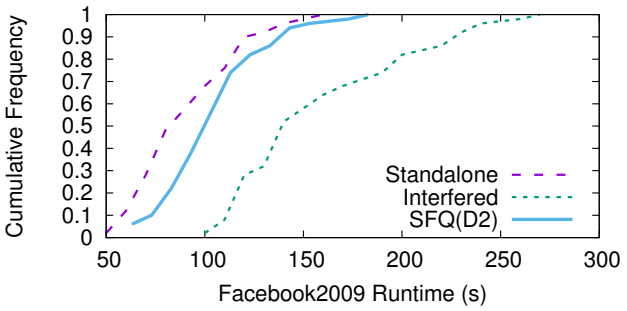


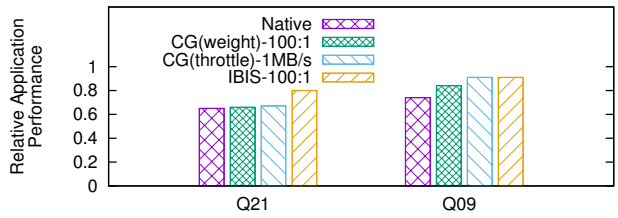

Figure 9: Cumulative distribution of Facebook2009 job runtimes

## 7.4 Multi-framework I/O Scheduling

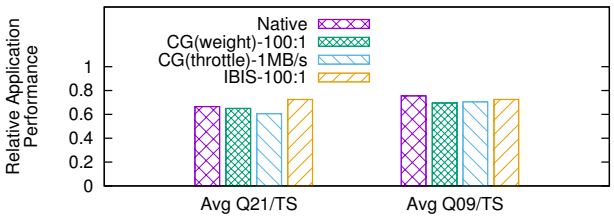
The third experiment evaluates IBIS' ability to schedule I/Os and manage their performance for different big-data frameworks, Hive [17] and MapReduce, that share the same infrastructure. Specifically, this experiment considers TPC-H queries [7] as the benchmark for Hive. TPC-H represents decision support systems scanning large volumes of business data, executing queries with a high degree of complexity, and providing keys to important business questions. Hive is a data warehouse framework built upon Hadoop, to support the SQL query execution for data stored on HDFS. Its execution engine spawns a series of MapReduce jobs for query fulfillment, providing end users with much flexibility in data format adaptation and ease of use in a scalable cluster environment.

The experiment focuses on the TPC-H queries Q9 (*product type profit*) and Q21 (*suppliers who kept orders waiting*) which involve multiple intensive I/O phases including both HDFS and intermediate I/Os. Q9 reads 53GB of initial input from five tables stored on HDFS and generates 120GB of intermediate I/Os. Q21 reads 45GB of initial input from four tables on HDFS, and generates 40GB of intermediate I/Os. Both queries launch up to 15 sequential Hadoop jobs. Q9's final output is 5KB and Q21's final output is 2.6GB.

The TPC-H queries on Hive and TeraSort on MapReduce are run concurrently, each with half of the CPU cores and memory. Although the native YARN does not provide any support for I/O management, it is conceivable to extend it to use cgroups [4], which




(a) Performance of TPC-H queries (Q9 and Q21) relative to their standalone runtimes




(b) The average relative performance of TPC-H and TeraSort

Figure 10: The performance interference and effectiveness of I/O scheduling for TPC-H on Hive and TeraSort on MapReduce that share the same infrastructure

YARN already uses to allocate CPUs and memory, to also manage I/O bandwidth allocation. To compare to this cgroups-based approach, YARN is extended to use the cgroups mechanisms to allocate shared I/O bandwidth between the two frameworks. This extended YARN can use both the proportional-sharing and throttling modes of cgroups to manage I/Os. In the proportional-sharing mode, the shared bandwidth is allocated to competing applications according to their assigned weights. In the throttling mode, a specific cap can be set to an application's bandwidth usage. Note that as discussed in Section 6, this approach as well as other similar ones can manage only the intermediate I/Os, but not HDFS I/Os; in contrast, IBIS is able to differentiate both local and distributed I/Os and schedule them according to the given performance policy.

Figure 10a shows the relative performance of the two TPC-H queries when running against TeraSort w.r.t. their standalone runtimes. For Q21, on *Native* YARN, the query experiences a 35.2% performance loss when compared to its standalone runtime. When using cgroups' two different modes with aggressive parameters to favor TPC-H—100:1 bandwidth sharing ratio in *CG weighted 100:1* and 1MB/s bandwidth cap on TeraSort in *CG throttled 1MB/s*, it can only improve the query performance by 1.2% and 2.5% respectively. In comparison, IBIS is able to improve the query performance to within 80% of its standalone performance, which is 15.2% better than native YARN and 12.7% better than cgroups. For Q9, the query experiences a 26% performance loss when running against TeraSort on *Native* YARN. Both cgroups' throttling policy and IBIS can restore the query performance to 91% of its standalone runtime, which is better than cgroups' proportional-share policy by 8%. The cgroups-based I/O throttling works better for Q9 than Q21, because Q9 has a higher level of intermediate I/Os which can be throttled by cgroups. However, throttling causes underutilization of storage and unnecessary slowdown of the competing application, TeraSort. Consequently, the performance of TeraSort is up to 16% worse when using cgroups throttling, compared to IBIS which is work-conserving.

To evaluate the overall system performance considering both competing frameworks, the experiment considers the *average relative performance* of the two applications, i.e., the average of each application's relative performance w.r.t. its own standalone performance. Figure 10b shows that when Q21 runs with TeraSort, the two applications experience a 26% performance loss in average on *Native* YARN, and the use of cgroups-based proportional bandwidth sharing does not improve it. The I/O throttling policy of cgroups makes it even worse because it is non-work-conserving and causes underutilization of the I/O bandwidth. In comparison, IBIS is able to achieve an average relative performance of 80%. For Q9, cgroups and IBIS achieve similar average relative performance, which is about 4% lower than *Native* because this query is more I/O intensive and incurs a higher overhead in I/O scheduling.

Considering the results from both figures, a multi-framework resource management solution such as YARN cannot provide strong performance isolation among competing applications due to the lack of I/O management. In comparison, IBIS is able to provide I/O isolation and when used in combination with YARN's CPU and memory management, it is able to protect the performance of a vulnerable application such as TPC-H while still allowing the competing, intensive application such as TeraSort to make good progress by using the available storage bandwidth.

## 7.5 Proportional Slowdown

The previous three experiments are designed to show IBIS' ability to support the performance isolation policy. Another important and commonly used policy is *proportional slowdown*, i.e., the relative performance of competing applications, w.r.t. their standalone performance, is proportional to their assigned weights. This policy is often used to achieve fairness for applications in terms of their performance, not their resource allocations. A big-data application's performance depends on both the available CPU cores and I/O bandwidth, and its use of CPU and I/O resources are correlated. Without control on the I/O bandwidth, it is possible to achieve proportional slowdown by limiting the CPU slots allocated to the more I/O-intensive application and indirectly throttling its I/O rate, so that the less I/O-intensive one can get more I/O bandwidth. Nonetheless, such a configuration leads to storage underutilization and suboptimal performance of the applications.

With IBIS, system administrators can tune both CPU slot and I/O bandwidth allocations together, and achieve proportional slowdown without wasting the resources. Ideally, this tuning should be done automatically without human intervention, which would require performance models of the big-data applications that can capture their CPU and I/O resource demands given different performance targets. How to create such models and use them to automatically tune the resource allocations are interesting research problems on their own and will be considered in the future work. This paper focuses on the problem of providing the necessary I/O control mechanisms to support a variety of performance policies such as performance isolation and proportional slowdown, which comes with a set of unique challenges discussed earlier and is tackled by the proposed IBIS framework. Without such control knobs enabled by IBIS, it would be difficult for either administrators or autonomic software to achieve the desired performance policy with efficient resource utilization.

In this experiment, *equal slowdown* of both TeraSort and TeraGen is the target policy, meaning that both applications should be

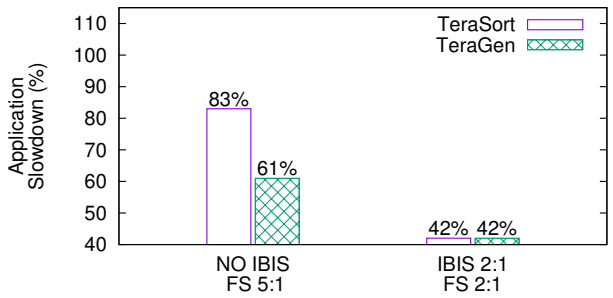



Figure 11: Performance slowdown of TeraSort and TeraGen using Hadoop Fair Scheduler (*FS*) based CPU slot allocations and IBIS-based I/O bandwidth allocations. The FS and IBIS ratios indicate the CPU and I/O shares, respectively, between TeraSort and TeraGen.

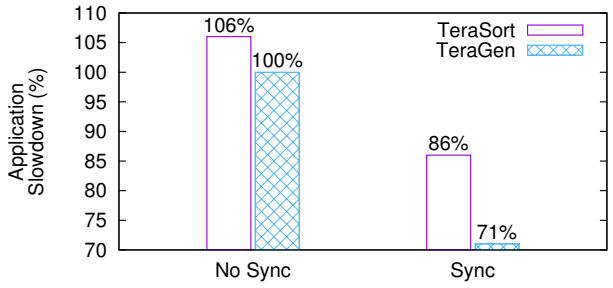



Figure 12: Performance slowdown of TeraSort and TeraGen when using IBIS *without* distributed scheduling coordination (*No Sync*) and *with* distributed scheduling coordination (*Sync*). The CPU sharing ratio of TeraSort vs. TeraGen is 1:1 and the I/O bandwidth sharing ratio is 32:1.

slowed down by the same percentage relative to their respective standalone runtime. Figure 11 shows the performance slowdown of these two applications. By adjusting only the CPU allocation using the Hadoop Fair Scheduler, the best equal slowdown that it can get is 83% slowdown for TeraSort and 61% for TeraGen. By using Fair Scheduler and IBIS to tune both CPU and I/O allocations together, it is able to get a perfect equal slowdown of 42%, which is 30% better than the average slowdown of the two applications when using Fair Scheduler only. These results therefore confirm that IBIS is also able to support the proportional slowdown policy and optimize the application performance under this policy.

## 7.6 Coordinated Scheduling

As discussed in Section 5, many factors decide the I/O service that an application gets from each datanode in a big-data system, including data distribution, slot allocation, task assignment, and competing applications, which all contribute to the uneven distribution of I/O services across the nodes. Without a mechanism for coordinating the distributed I/O schedulers and an algorithm to adjust local sharing ratios based on the global sharing policy, the total service that an application gets from the entire big-data system will diverge from the given target. This experiment evaluates the effectiveness of the proposed distributed scheduling coordination mechanisms (Section 5) for achieving total-service proportional sharing.

The experiment is conducted similarly to the previous one for achieving equal slowdown for TeraSort and TeraGen, but it considers two different IBIS setups where the distributed scheduling

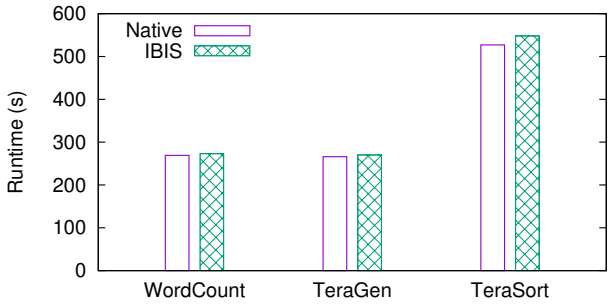



Figure 13: Runtime of WordCount, TeraGen, and TeraSort when each benchmark runs on native Hadoop (*Native*) vs. on *IBIS*

Table 2: CPU and memory usages of the YARN and IBIS daemons including the Resource Manager, Node Manager, and Data Node

| Benchmark | Resource | Native | IBIS |
|---|---|---|---|
| WordCount | CPU | 0.4% | 0.5% |
| TeraGen | CPU | 1.7% | 5.1% |
| TeraSort | CPU | 0.55% | 0.65% |
| WordCount | Memory | 1.2% | 8.2% |
| TeraGen | Memory | 2.0% | 8.1% |
| TeraSort | Memory | 1.6% | 10.6% |

coordination is disabled (*No Sync*) and enabled (*Sync*). The latter case should allow IBIS to find better equal slowdown because it can dynamically adjust local I/O service distribution based on global service distribution, which leads to better CPU and I/O resource utilization and better performance for both applications. Figure 12 shows the performance slowdown of TeraSort and TeraGen with respect to their own standalone runtimes. The average performance slowdown with *Sync* is 25% better than from *No Sync*, confirming the improvement made by the coordinated I/O scheduling.

## 7.7 Overhead

The last experiment evaluates the overhead of IBIS from several aspects. First, it studies the performance impact to a big-data application from IBIS-based I/O interposition and scheduling. WordCount, TeraGen, and TeraSort are all considered because each of them has distinct I/O patterns and demands. They are run separately with all the 96 CPU cores in the system. Figure 13 shows that the overhead of using IBIS is 1%, 2%, and 4% for WordCount, TeraGen, and TeraSort, respectively, in terms of runtime.

Second, the resource usages of IBIS are evaluated by tracking the total CPU and memory utilizations of the YARN Resource Manager, Node Manager, and Data Node, where the IBIS implementation is located. Table 2 lists the per-core CPU utilization and per-node memory utilization, which are reasonable compared to native YARN's resource usages.

Third, Table 3 summarizes the code development complexity in terms of lines of code categorized by the IBIS components. IBIS provides a flexible big-data I/O scheduling framework, and allows users to conveniently create new schedulers for different objectives. The amount of work required to develop a sophisticated scheduler on IBIS is only at the level of a thousand lines of code.

# 8. RELATED WORK

As storage performance becomes increasingly important to big-data applications, several recent works have studied this problem.

Table 3: Development cost of IBIS

| Component | Lines of Code |
|---|---|
| Interposition | 2593 |
| SFQ(D) Scheduler | 734 |
| SFQ(D2) Scheduler | 1520 |
| Scheduling Coordination | 1705 |
| Total | 6552 |

Frosting [20] provides a scheduling layer upon HBase which dynamically controls the number of outstanding requests and proportionally shares the HBase storage among competing clients. However, it treats the entire distributed HBase storage stack as a single black box, and may underutilize the individual datanodes in order to provide any performance guarantee. In contrast, IBIS manages I/Os at the lower big-data file system layer and in a distributed manner, which can provide more effective I/O performance differentiation while making efficient use of the underlying storage resources.

PISCES [15] provides fair sharing of key-value storage by controlling requests dispatched to storage nodes according to the shares. In comparison, in a MapReduce-type big-data system an application's task distribution is driven by both CPU slot requirement and data locality, and its I/O demands are much more complex—including multiple phases of local and network I/Os, and diverse—with different intensities on the various types of I/Os. Hence, I/O management in a MapReduce system cannot be achieved by merely controlling task dispatching, and has to rely on both local I/O scheduling and global coordination which are part of the IBIS solution.

Cake [19] presents a two-level scheduling approach to meeting the performance goal of latency-sensitive applications (HBase) when they are consolidated with throughput-oriented applications (MapReduce), but it cannot provide any performance guarantee to the latter. In comparison, IBIS supports both types of applications.

Finally, a preliminary study of IBIS [23] presented the results from employing the traditional SFQ(D) scheduler for MapReduce applications only. This paper extends upon the initial results and includes a new scheduling framework that supports different types of big-data applications (both batch jobs and queries) and a new I/O scheduler SFQ(D2) that substantially outperforms SFQ(D).

Several works studied other orthogonal aspects of big-data storage: PACMan [8] manages memory-based caching of map task inputs to improve application performance; iShuffle [11] improves the performance of intermediate I/Os. However, they would still need a storage management solution like IBIS to provide performance isolation for the intermediate I/Os and the I/Os that trickle down the memory cache layer among concurrent applications.

I/O interposition is a technique often used to manage a shared storage resource that does not provide native knobs for controlling its competing I/Os. It has been employed in the related work to realize a proportional bandwidth scheduler for a shared file service [13], to create application-customized virtual file systems upon a shared network file system [25], and to manage the performance of a parallel file system based storage system [22]. Big-data systems present unique challenges to I/O management because of the complexity (different types of I/Os), diversity (different levels of intensity), and scale (many datanodes) of the I/O contention. These are addressed by the techniques embodied in IBIS, including holistic interposition of HDFS, local file system, and network I/Os, an adaptive proportion-share I/O scheduler, and scalable coordination of distributed I/O scheduling.

There are also related works on the performance management of other types of storage systems. Horizon [14] can provide global minimum throughput guarantee for a RAID storage system, but it requires a centralized controller to assign deadlines to requests, which is difficult to apply to a big-data system where I/Os are issued directly by local tasks on each datanode. A two-level scheduler [24] was proposed for meeting I/O latency and throughput targets, but it supports only local I/O scheduling, whereas a big-data system requires the distributed storage management provided by IBIS.

# 9. DISCUSSIONS

Overall IBIS can provide good performance isolation to various big-data applications. For WordCount, IBIS reduces the slowdown to merely 8% when it is under heavy I/O contention from a much more intensive job; but for a more latency-sensitive TPC-H query (Q21), it still shows a 25% slowdown despite the significant improvement over native YARN. Note that these results are achieved without underutilizing the shared storage bandwidth. Further improvement is possible by trading resource utilization for performance isolation. IBIS enables this tradeoff by adjusting its scheduler parameters ($D$ in SFQ(D) and $L_{ref}$ in SFQ(D2)) and choice of schedulers—in the extreme case, a non-work-conserving scheduler can provide strict performance isolation but may severely underutilize the storage.

This paper focuses on providing the missing control knob for managing I/Os in big-data systems, but it does not answer the question of how to automatically tune this new knob to meet an application's desired performance target—the results in Section 7 are from manually adjusting the scheduler parameters. A possible solution, which will be explored in the future work, is to build performance models for big-data applications that can map an application's resource allocations, including IBIS-based I/O bandwidth allocation, to its performance. Based on such models, admission control and resource allocation can be then done automatically given the desired application performance.

Although IBIS uses a centralized scheduling broker, its lightweight design promises good scalability. The broker handles only the forwarding of the global I/O service information to the local schedulers and let them decide how to schedule their local flows based on this information. In comparison, an alternative design that uses the broker to decide the scheduling of all the local flows in the system would not scale. The amount of information to be communicated between the broker and each local scheduler is also small, and it is piggybacked on the big-data system's existing heartbeat messages to further reduce its overhead. Therefore, the cost of global scheduling coordination does not grow significantly as the system size increases. Future work will consider the evaluation of IBIS on larger-scale testbeds to quantify its scalability.

Finally, the paper assumes that the big-data storage system is physical. A virtualized environment where the datanodes are hosted on virtual machines would present some new challenges. For example, several virtual datanodes may be sharing the same physical storage, and the scheduling decisions made by the IBIS schedulers running on these datanodes may be conflicting. As virtualized big-data systems gain wider adoption, this is also an interesting direction for future work.

# 10. CONCLUSIONS

Big-data systems are increasingly important platforms for efficient processing of large amounts of data for knowledge learning and sharing in various disciplines. Big-data applications are by nature I/O intensive, and their performance strongly depends on the I/O services that they get from the system. Existing big-data systems support only the allocation of CPUs and memory, which

however does not provide any performance isolation on the shared storage. Moreover, it is challenging to achieve I/O performance management in a big-data system because of the intrinsic complexity of the application I/Os and the distributed nature of the system.

This paper presents IBIS, an Interposed Big-data I/O Scheduler, to address the above challenges and provide the much needed I/O performance differentiation to diverse big-data applications, possibly from different frameworks. IBIS is designed to transparently differentiate and schedule application I/Os on every datanode by interposing upon the distributed file system commonly used in big-data systems. It includes a new proportional-share I/O scheduler that can dynamically adjust I/O concurrency to optimize the trade-off between application fairness and resource utilization. It provides efficient coordination for the I/O schedulers distributed across the datanodes to cooperate and achieve proportional sharing of the big-data system's total I/O service. The results from an extensive evaluation confirm that IBIS can effectively address the severe I/O interference problem that existing big-data systems have and provide strong performance isolation with efficient resource usage.

## 11. ACKNOWLEDGMENTS

The authors thank the anonymous reviewers and the shepherd, Jon Weissman, for their helpful comments on the paper. This research is sponsored by National Science Foundation CAREER award CNS-125394 and Department of Defense award W911NF-13-1-0157.

## 12. REFERENCES

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

[2] Hadoop Fair Scheduler. https://hadoop.apache.org/docs/r2.7.1/hadoop-yarn/hadoop-yarn-site/FairScheduler.html.

[3] HBase. http://hbase.apache.org.

[4] Linux containers. https://linuxcontainers.org.

[5] OpenFlow. https://www.opennetworking.org/sdn-resources/openflow.

[6] Statistical workload injector for MapReduce (SWIM). https://github.com/SWIMProjectUCB/SWIM/wiki.

[7] TPC-H Benchmark Specification. http://www.tpc.org/tpch.

[8] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. PACMan: Coordinated memory caching for parallel jobs. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, 2012.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design and Implementation (OSDI'04)*, Berkeley, CA, USA, 2004. USENIX Association.

[10] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 29–43, New York, NY, USA, 2003. ACM.

[11] Y. Guo, J. Rao, and X. Zhou. iShuffle: Improving Hadoop performance with shuffle-on-write. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC'13)*, pages 107–117, San Jose, CA, 2013. USENIX.

[12] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. 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*, 2011.

[13] W. Jin, J. S. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'04)*, pages 37–48, New York, NY, USA, 2004. ACM.

[14] 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 (HPDC'10)*, pages 1–12, New York, NY, USA, 2010. ACM.

[15] D. Shue, M. J. Freedman, and A. Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, pages 349–362, 2012.

[16] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop distributed file system. In *Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10. IEEE, 2010.

[17] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive - A petabyte scale data warehouse using Hadoop. In *Proceedings of the 26th IEEE International Conference on Data Engineering (ICDE'10)*, pages 996–1005, March 2010.

[18] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, and S. Seth. Apache Hadoop YARN: Yet another resource negotiator. In *Proceedings of the Fourth ACM Symposium on Cloud Computing*, 2013.

[19] A. Wang, S. Venkataraman, S. Alspaugh, R. Katz, and I. Stoica. Cake: Enabling high-level SLOs on shared storage systems. In *Proceedings of the Third ACM Symposium on Cloud Computing (SOCC'12)*, pages 14:1–14:14, New York, NY, USA, 2012. ACM.

[20] A. Wang, S. Venkataraman, S. Alspaugh, I. Stoica, and R. Katz. Sweet storage SLOs with Frosting. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Computing (HotCloud'12)*, Berkeley, CA, USA, 2012. USENIX Association.

[21] Y. Wang and A. Merchant. Proportional-share scheduling for distributed storage systems. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST'07)*, Berkeley, CA, USA, 2007. USENIX.

[22] Y. Xu, D. Arteaga, M. Zhao, Y. Liu, R. Figueiredo, and S. Seelam. vPFS: Virtualization-based bandwidth management for parallel storage systems. In *Proceedings of the 28th IEEE Conference on Massive Data Storage (MSST)*, April 2012.

[23] Y. Xu, A. Suarez, and M. Zhao. IBIS: Interposed big-data I/O scheduler. In *Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing (HPDC'13)*, pages 109–110, New York, NY, USA, 2013. ACM.

[24] J. Zhang, A. Sivasubramaniam, Q. Wang, A. Riska, and E. Riedel. Storage performance virtualization via throughput and latency control. *ACM Transactions on Storage*, 2(3):283–308, Aug. 2006.

[25] M. Zhao and R. J. Figueiredo. Application-tailored cache consistency for wide-area file systems. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)*, 2006.