Pfimbi: Accelerating Big Data Jobs Through Flow-Controlled Data Replication

Simbarashe Dzinamarira*, Florin Dinu†, T. S. Eugene Ng*
Rice University*, EPFL†

***Abstract*—The performance of HDFS is critical to big data software stacks and has been at the forefront of recent efforts from the industry and the open source community. A key problem is the lack of flexibility in how data replication is performed. To address this problem, this paper presents Pfimbi, the first alternative to HDFS that supports both synchronous and flow-controlled asynchronous data replication. Pfimbi has numerous benefits: It accelerates jobs, exploits under-utilized storage I/O bandwidth, and supports hierarchical storage I/O bandwidth allocation policies.**

We demonstrate that for a job trace derived from a Facebook workload, Pfimbi improves the average job runtime by 18% and by up to 46% in the best case. We also demonstrate that flow control is crucial to fully exploiting the benefits of asynchronous replication; removing Pfimbi's flow control mechanisms resulted in a 2.7x increase in job runtime.

I. Introduction

For years, developers have been constantly introducing new big data processing tools (e.g., Pig, Mahout, Hive, SparkR, GraphX, SparkSQL) into big data stacks such as Hadoop [1] and Spark [2] to address an increasing number of use cases. Figure 1 illustrates the Hadoop and the Spark ecosystems today. On this fast-changing landscape, the open-source Hadoop Distributed File System (HDFS) [3], which is modeled after the Google File System [4], has remained the preeminent distributed storage solution. Given the ubiquity of HDFS, a significant improvement to HDFS will have a sizable real-world impact.

However, we observe that several recent efforts at improving HDFS are revealing an emerging need to handle data replication in HDFS more flexibly. As a first example, HDFS developers have recently added heterogeneous storage support [5] in HDFS. This addition allows HDFS to explicitly place one or more data copies on faster media (RAM Disk or SSD) for faster future data reads, but still leveraging slower and cheaper media (HDD) for maintaining backup copies. However, this feature offers no substantial performance improvement for data writes. The fundamental reason is that whenever data replication is enabled, HDFS writes synchronously through a pipeline of DataNodes to create copies at those DataNodes. Consequently, the performance of such writes is dictated by the slowest portion of the pipeline, usually an HDD. As a second example, HDFS developers have also added a feature to allow a DataNode to initially write data in a small RAM Disk [6] and then lazily persist the data to the underlying non-volatile storage. While this feature improves the performance of jobs that write a small amount of data, it offers no substantial performance improvement for jobs that write a large amount of data for which improvements are arguably most needed. The fundamental reason is the same as above. When the data size is larger than a DataNode's RAM Disk, data writes slow to the speed of that of the underlying non-volatile storage, and the overall write performance is again dictated by the slowest DataNode in the pipeline.

An additional flexibility to allow a mix of synchronous and asynchronous data replication can go a long way to addressing these write performance problems. For instance, a job can choose to perform the primary data copy's writes synchronously, and the replicas' writes asynchronously. Such a job completes quickly as soon as the primary copy has been written, and the job's lifespan is not explicitly tied to the speed of replication at potentially slow DataNodes. However, realizing such improvements will require additional I/O flow control mechanisms. Without such mechanisms, asynchronous replication writes may contend arbitrarily with primary writes of executing jobs and among themselves, harming storage I/O efficiency and negating the benefit of asynchronous replication.

This paper presents Pfimbi, a replacement for HDFS that effectively supports both synchronous and flow-controlled asynchronous data replications.

Pfimbi can isolate primary writes from the interference caused by asynchronous replication writes and leverage storage I/O under-utilization to effectively speed up jobs. To achieve this, Pfimbi has to overcome a challenge, namely that in real workloads storage I/O under-utilization is plentiful but individual intervals of under-utilization are often brief and interleaved with periods of peak utilization. Moreover, these periods are not necessarily correlated between different DataNodes. Therefore, to ensure good performance, data blocks must be delivered to a DataNode in a timely manner for the DataNode to take advantage of moments of storage I/O inactivity. However, these transmissions cannot be overly aggressive or they might overwhelm the DataNode and take away its ability to fully control usage of storage I/O bandwidth. These requirements imply that DataNodes must coordinate their actions. To provide this coordination, Pfimbi employs a protocol that is distributed and scalable, yet can achieve very high I/O utilization while avoiding interference.

Pfimbi supports hierarchical flow control which enables highly flexible resource management policies. Asynchronous replication fundamentally changes how flexibly data blocks can be handled. While in synchronous replication every data block is a part of a rigid and *continuous* data stream, in asynchronous replication, each data block is discrete and stands alone. Therefore, a flow control mechanism can freely

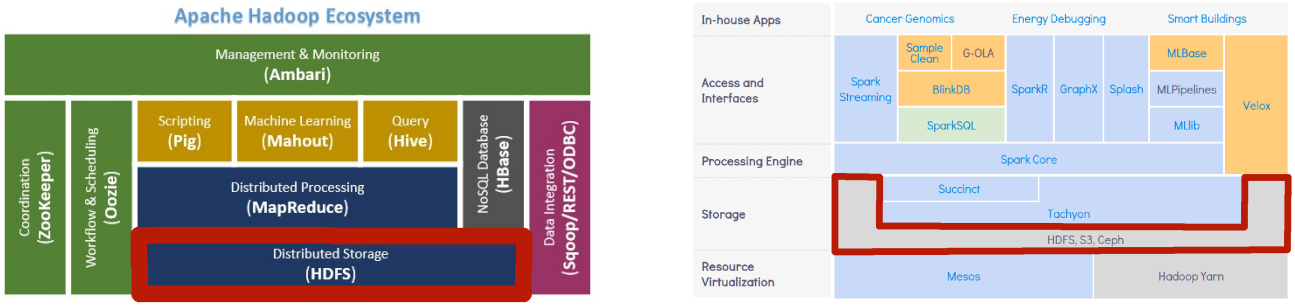


Fig. 1: HDFS is the foundation for large big-data-analytics efforts. Left: The Hadoop ecosystem. Right: The Spark ecosystem. Image sources: http://www.mssqltips.com/tipimages2/3260_Apache_Hadoop_Ecosystem.JPG and https://amplab.cs.berkeley.edu/software/

consider various attributes of a data block (e.g., job ID, task ID, replica number, block size, source DataNode ID, etc.) in making resource allocation decisions. However, the challenge lies in being able to flexibly express and enforce policies that involve multiple attributes. Pfimbi addresses this challenge by supporting a hierarchical model for flow control, where different attributes are used at different levels in the hierarchy to control resource allocation. Many natural policies can be expressed in this hierarchical manner.

Pfimbi cleanly separates mechanisms and policies. The length of the synchronous portion of a pipeline can be set by users on a job by job basis. Therefore, users can choose whether replication is fully asynchronous, synchronous like it is in HDFS, or a hybrid of the two. The weights assigned to different replication flows can also be set individually, allowing users to dictate how jobs share the available bandwidth. This separation of mechanisms and policies makes Pfimbi flexible and extensible.

Our experimental evaluation in Section VI shows that for a job trace derived from a Facebook workload, Pfimbi improves the job runtime by 18% on average, up to 46% for small jobs (writing under 1GB), and up to 28% for the large jobs (writing 80GB). Pfimbi improves the runtime of DFSIO, a Hadoop micro-benchmark, by 52% as compared to HDFS on a cluster with all HDDs, and finishes all replication work in a time similar to HDFS. When the first position of the replication pipelines is switched to SSD, the runtime improvement goes up to 73%. Finally, for a job running alongside the asynchronous replication of an earlier job, we observe a 2.7x increase in job duration if flow control is turned off, highlighting the importance and effectiveness of Pfimbi's flow control mechanisms.

The rest of this paper is organized as follows. In Section II, we review basic concepts in HDFS. In Section III, we motivate the feasibility and potential benefits of asynchronous replication. The details of Pfimbi's design are presented in Section IV and we discuss key properties of Pfimbi in Section V. Performance of Pfimbi is evaluated in Section VI. Finally, we discuss related work in Section VII and conclude this paper.

II. Background

A. Terminology

For a job to complete, its output needs to be written once. We call this the primary copy (primary write). Data replication

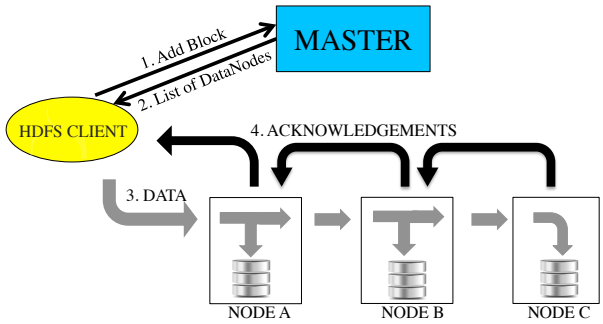


Fig. 2: The anatomy of a synchronous pipelined write in HDFS

creates additional replicas. A client is application code that reads and writes to the file system. By synchronous replication we mean replication is on the critical path of client writes. The client write will not complete until replication is done. We use the term asynchronous replication to refer to data replication that is decoupled from client writes. We use the term "normal traffic" to refer to all reads and writes excluding asynchronous replication.

B. HDFS architecture

HDFS is used to distribute data over a cluster composed of commodity computer nodes. It uses a master-slave architecture. The master (called NameNode) handles metadata, answers client queries regarding data locations and directs clients to write data to suitable locations. The slave nodes (called DataNodes) handle the actual client read and write requests. Data is read and written at the granularity of blocks which are typically tens to hundreds of MB in size (64MB, 256MB are popular). Clusters often co-locate storage with computation such that the same set of nodes that run compute tasks also run HDFS.

C. Synchronous pipelined writes in HDFS

Block writes are performed in a synchronous pipelined fashion. Figure 2 presents in more detail the anatomy of a synchronous pipelined block write. This process is repeated for every new block. For every block write, the client contacts the master ① and receives a list of nodes ② that will host the block copies. The size of the list depends on the replication

factor of the file (i.e., number of copies). Commonly, the first node in the list is the same node that executes the task writing the data. The client then organizes the provided nodes into a pipeline, ordering them to minimize the total network distance from the client to the last node in the pipeline [4].

Once the pipeline is setup up, the block's data is sent over the pipeline at the granularity of application-level packets ③. When a node receives a packet from upstream, it forwards the packet downstream and writes the data locally. The last node in the pipeline, once it has successfully received the packet, generates an acknowledgment ④ which is then propagated through the pipeline, upstream, all the way to the client. A window-based scheme is used to limit the maximum number of un-acknowledged packets. A packet is considered successfully written after the client receives the corresponding acknowledgment. A block is considered successfully written after all its packets have been acknowledged.

III. Motivation

This section motivates the feasibility and potential benefits of asynchronous replication. First, we discuss the drawbacks of synchronous replication. Next, we discuss why asynchronous replication may still provide sufficient data locality for many jobs. Then, we highlight the fact that consistency can also be guaranteed by asynchronous replication. Finally, we show that disk I/O under-utilization can be frequently encountered which can be exploited to perform asynchronous replication.

A. Drawbacks of synchronous replication

Synchronous replication couples replication with primary writes, thus putting replication on the critical path of the writes. This design has important negative implications for both performance and resource efficiency.

Synchronous replication causes contention between primary writes and replica writes even within a single job. This contention leads to a disproportional increase in job completion times. Take sorting 1TB of data on 30 nodes in Hadoop for example. We find that adding 1 replica causes a slowdown of 23%, and disproportionally, adding 2 replicas causes a slowdown of 65%.

Synchronous replication can also lead to inefficient cluster-wide resource usage. Since replication prolongs task execution times, the more replicas are being created, the longer tasks hold on to their allocated CPU and memory resources. Overall cluster-wide job throughput can be increased if these resources are released promptly and allocated to other jobs sooner.

Slow DataNodes (e.g., caused by a temporary overload) greatly compound the problems described. Since one node can serve multiple replication pipelines simultaneously, one single slow node can delay several tasks at the same time. Finally, synchronous replication increases a task's exposure to network congestion. Any slow network transfer can also slow down the task.

B. Many jobs can have good data locality without synchronous replication

A task is said to have data locality if it executes on the same node from which it reads its input data. Data locality may improve a task's runtime when reading input data locally is faster than reading it remotely. Synchronous replication can help improve data locality of a subsequent job by ensuring many replicas of such job's input data are available when the job starts. This increases the probability that a node that is selected to run a task of the job also hosts an input block of that job.

Data locality can also be obtained in the absence of synchronous replication. Many jobs that process large quantities of data naturally have their input blocks spread over a large portion of a cluster so their tasks will be data-local with high probability even without any replication. For these jobs, with respect to data locality, whether the input was replicated synchronously or asynchronously is unimportant.

C. Consistency guarantees can be obtained with asynchronous replication

A distributed file system requires that replicas of the same block are consistent with each other. HDFS allows a restricted set of operations on files – (1) only a single application can modify a file at any given time; (2) written data is immutable and new data can only be added at the end of a file. This restricted set of operations is powerful enough to satisfy the needs of the targeted jobs. Section V-A describes how consistency is guaranteed under both synchronous and asynchronous replication. For the two ways in which a file can be modified, write and append, we show that consistency is maintained for both reads following writes and writes following writes.

D. Exploitable storage I/O under-utilization

We now argue that disk I/O under-utilization can be frequently encountered. This presents an opportunity for performing efficient and timely asynchronous replication. We also analyze the pattern of disk I/O under-utilization as this directly impacts Pfimbi's design.

Disk I/O under-utilization is plentiful but irregular. This stems from fundamental job properties. Different jobs have different dominant resources and put different pressure on the storage subsystem. Some jobs are storage I/O bound (e.g., Terasort [7], NutchIndexing [8] and Bayesian Classification [9]) while others are CPU bound (e.g., Kmeans Clustering [10]). Even a single task may use the storage subsystem differently throughout its lifetime. For example, a Hadoop reducer is more storage I/O bound during the write phase compared with the shuffle phase if the reducer has been configured with enough memory.

To illustrate the pattern of disk I/O under-utilization, we run the SWIM workload injector [11] with the first 100 jobs of a 2009 Facebook trace provided with SWIM on a 30-node cluster. We compute the average disk throughput (reads + writes) by reading OS disk activity counters every 100ms. Every one second we log a computed sample. Each node has one 2TB HDD used solely by the workload. Figure 3a shows a cluster-wide (all 30 nodes) view of the pattern of disk I/O utilization for a representative time window. Figure 3b illustrates the same pattern at the level of a single node.

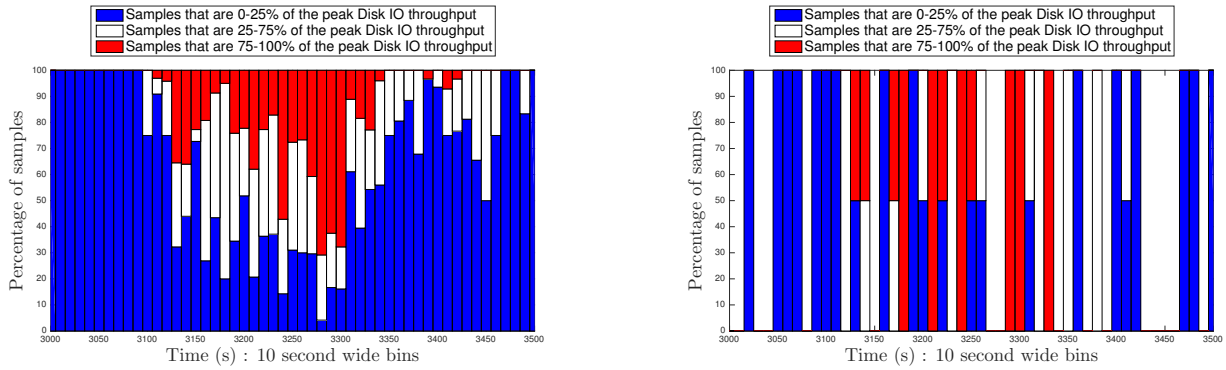


Fig. 3: Disk I/O (write throughput + read throughput) for the SWIM workload on a 30-node cluster, over a 10 minute period. IO measurements in 10 second intervals are binned together. (a) Distribution of IO samples for all 30 nodes. (b) Distribution of IO samples for a single node.

Figure 3a suggests that a significant part of the disk bandwidth is not utilized. Even when compute slots are actively being used, the disk can be idle. Such under-utilization can be exploited to opportunistically and asynchronously replicate data. Figure 3b shows the irregular pattern of disk activity for a single node. Periods of idleness or reduced activity are frequently interleaved with periods of peak activity. This observation suggests the need for a solution that quickly reacts to periods of under-utilization.

IV. Pfimbi

In this section, we describe the design and implementation of Pfimbi. To implement Pfimbi we augmented the HDFS DataNode with a module called the Replication Manager (RM). We use RM to refer to this per-node component implementing the Pfimbi design.

A. Pfimbi design overview

Two basic decisions have guided our design of Pfimbi. First, we must recognize the diverse needs of different jobs and allow jobs to choose flexibly between synchronous replication, asynchronous replication or a hybrid of the two. Second, we must design for scalability, and as such DataNodes should locally make all decisions related to flow control (e.g. deciding when to write data and which data to write). We do not allow centralized collection of statistics nor centralized flow control decision making due to scalability concerns.

Figure 4 presents a logical view of the Replication Manager (RM) which is the component that implements Pfimbi's new functionality. Arriving data is first de-multiplexed. The RM writes synchronous data to the file system immediately, but asynchronously received replication data is buffered in memory, in the buffer pictured in Figure 4. The Flow Regulator decides when to write the buffered data to disk, based on activity information reported by the Activity Monitor. The Communication Manager exchanges messages with other nodes. It receives block notifications and forwards them to the scheduler. When the Communication Manager detects free space in the local buffer, it invokes the scheduler to select

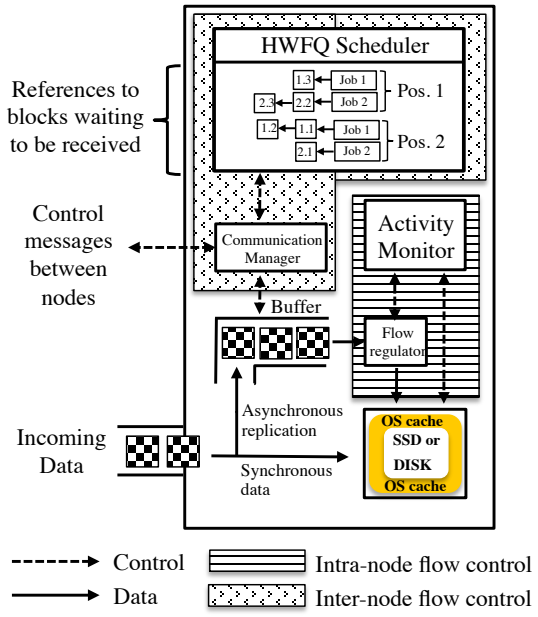


Fig. 4: Components of the Replication Manager in a Pfimbi node.

one of the pending blocks and requests the upstream node to transfer it. The policy enforced by the scheduler dictates which block the DataNode requests.

The main components of Pfimbi's design can be separated into two groups, the ones that enable inter-node flow control vs. those for intra-node flow control. The intra-node flow control components deal with writing asynchronous replication data locally. The inter-node flow control components deal with inter-node communication and facilitate the forwarding of replication data between nodes. Section IV-B explains in detail how intra-node flow control works while Figure 5 and Section IV-C explain inter-node flow control.

B. Intra-node flow control

This section describes how Pfimbi decides when to write buffered replication blocks to stable storage. The challenge is to monitor local IO in a way that permits the rapid and efficient use of disk under-utilization periods while minimizing the interference that replication causes to normal traffic. We start by discussing a number of intuitive approaches that we had to dismiss after thorough analysis and experimentation. Finally, we present the approach taken in Pfimbi.

1) Monitoring local IO - discarded alternatives: Our first alternative was to measure IO activity at the DFS level. The advantage is that a DataNode can differentiate synchronous writes from asynchronous writes based on the position of blocks in their pipelines. Unfortunately, this solution is oblivious to non-DFS writes and can lead to replication writes interfering with reducer and mapper spills, as well as mapper output writes. In Pfimbi, we want to avoid such interference.

As a second alternative, we considered two disk-level metrics: aggregate disk throughput (reads + writes) and IO request latency. Replication writes are allowed to proceed whenever the current metric measurements drop below a predefined threshold. These two metrics are accurate and timely. However, they have weaknesses when used for flow control. First, disk-level metrics cannot differentiate between normal and asynchronous writes. When only asynchronous writes are in the system, they would still be throttled whenever activity is above the predefined threshold, leading to inefficient disk utilization. Second, picking a suitable threshold is difficult. Setting too high a threshold results in interference with normal data while a low threshold leads to decreased utilization. Lastly, when using disk throughput or IO latency, Pfimbi can only detect and react to idleness after it has occurred. The system repeatedly cycles between detecting idleness, writing more asynchronous data, pausing when activity is above the threshold, then detecting idleness again. Small delays between these steps add up, resulting in lower disk utilization.

To avoid low utilization, we considered a third alternative: the sum between disk write throughput and the rate of change of the dirty data in the buffer cache. Using this aggregate metric, we can detect when the cache is being flushed without being replenished (i.e. the sum is zero). This case is important because it means the buffer cache is draining and the disk will become idle. We can, therefore, write asynchronous data before idleness occurs. However, when experimenting with this aggregate metric, we discovered that a lot of asynchronous data would build up in memory. The build up of asynchronous data in the buffer cache reduces the amount of free space for future normal writes to burst into, without blocking due to a full buffer cache. Furthermore, the cache churn rate fluctuates faster, and over a greater range than disk throughput thus making setting a threshold for the aggregate difficult.

2) Monitoring local IO - Pfimbi's approach: Pfimbi's approach is cache-centric. The idea is to allow the least amount of asynchronous replication data to be written to the cache that is necessary to ensure the disk is kept fully utilized. As a result, the impact of this least amount of asynchronous data in the cache on any normal writes is limited.

Pfimbi tracks the amount of dirty data in the buffer cache using standard OS mechanisms and tries to maintain the level of dirty-data above the threshold T at which the OS continuously flushes dirty data to disk (e.g., /proc/sys/vm/dirty_background_bytes in Linux). To make sure that the amount of dirty data never falls below T, Pfimbi aims to keep $T + \delta$ dirty bytes in memory. δ is set to be at least the maximum disk throughput multiplied by the monitoring interval. This guarantees that in-between Pfimbi's measurements, even if the buffer cache is drained at the maximum disk throughput, the amount of data will not fall below T before Pfimbi can write more data.

C. Inter-node flow control

Section III-D showed that periods of disk under-utilization are often interleaved with periods of peak utilization. When disk under-utilization is detected, a node has to write asynchronous data with little delay. Triggering a remote network read after detecting disk under-utilization incurs much too high a latency to be viable. Instead, Pfimbi achieves fast reaction by having a number of asynchronous blocks buffered and ready in memory.

The RM controls the flow of asynchronous replication blocks to ensure that receiver side buffers, as shown in Figure 4, are always replenished. Pfimbi uses a credit-based flow control scheme. The receiver notifies the sender when there is space in the buffer to receive a block. Only when such a notification is received will a sender start a block transfer. However, a receiver does not initially know which blocks are destined for it. The Communication Manager in Figure 4 uses Remote Procedure Calls (RPCs) to let senders notify downstream nodes of blocks destined for them.

Figure 5 demonstrates the use of flow control for replication in Pfimbi. This example assumes that two replicas (A and B) are written synchronously and two asynchronously (C and D). The client sends the data to node A which forwards it synchronously to node B ①. Node B is the last node in the synchronous pipeline so it sends acknowledgments which are propagated upstream to the client ②. The client's write operation completes when all the data it sent is acknowledged by nodes A and B. As they receive data, nodes A and B write the data locally. This data is initially absorbed by the local OS buffer cache and finally ends up in stable storage. After receiving a data block and writing it locally, node B notifies node C that it has a block available for it ③. Node C has plenty of room in its buffer so it immediately requests the block ④ and then begins receiving it ⑤. Since node C is on the asynchronous part of the pipeline it will treat the incoming block as an asynchronous block ⑥. After receiving the block and writing it locally ⑦ it notifies node D that it has a block available for it ⑧. Unfortunately, node D has no room in its buffer at this point ⑨. It will first have to write some of the buffered blocks locally to disk to make room for other blocks. Only then will it request the block from node C.

1) Hierarchical flow control: A node is typically part of multiple pipelines at the same time. These pipelines can be from different clients of the same job (e.g., different reducer

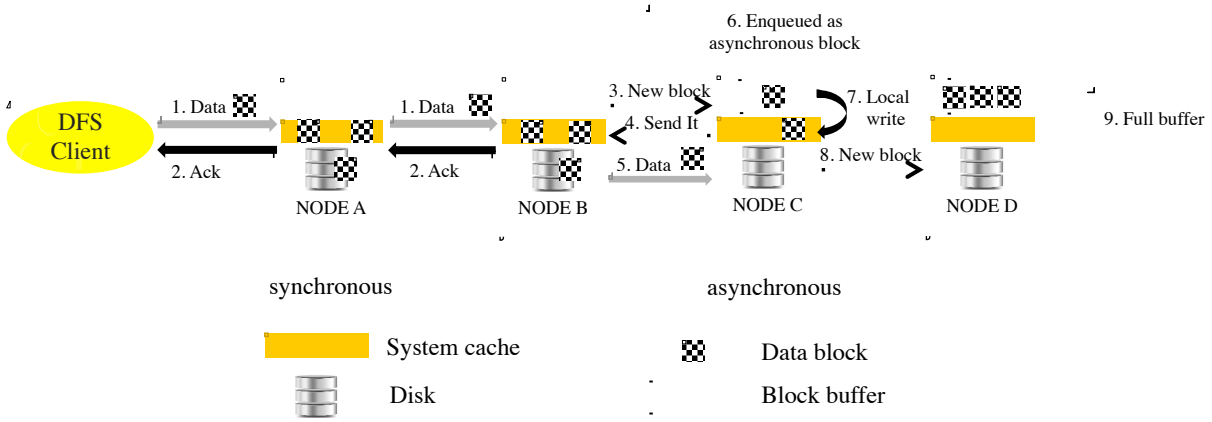


Fig. 5: Inter-node flow control in Pfimbi. Two replicas are written synchronously (A, B) and two asynchronously (C, D).

tasks) or clients from multiple jobs. Moreover, for any of these pipelines, the node can be in different positions (such as first, second and third). In general, each data block to be replicated is associated with multiple attributes such as job ID, task ID, replica number, block size, source DataNode ID, etc. Pfimbi employs Hierarchical Weighted Fair Queueing [12] for block scheduling to provide a highly flexible mechanism for expressing and enforcing flow control policies that involve multiple attributes.

To explain how the hierarchical model works, we give an example using a two level hierarchy. The idea generalizes to any number of levels in the hierarchy. In our example, we define a root class to which all blocks belong. Then in the first level of the hierarchy, we define a class for each replica position and maintain a virtual queue for each replica position class (e.g., Pos.1 and Pos.2 in Figure 4), and each virtual queue is associated with a different weight to express an allocation proportion (e.g., an earlier replica gets a larger weight). In the second level, under each replica position class, there is a physical queue for each job (e.g., Job 1 and Job 2 in Figure 4), and each queue is also associated with a different weight to expressed an allocation proportion. When a block notification is received from an upstream node, the RM creates a reference to that block, classifies that block first by replica position, then by job ID, and inserts this reference into the corresponding queue. For instance, in Figure 4, each block reference is annotated by "jobID.sequenceNumber". Block 1.3 means the third block from job 1. Because block 1.3's pipeline happens to use this DataNode in position 1 in the pipeline, it is queued under the Pos.1 Job 1 queue. Similarly, block 1.1's pipeline uses this node in position 2, its reference is queued under the Pos.2 Job 1 queue.

The scheduling algorithm recursively decides from which position class, and then from which job to initiate the next block reception. The algorithm maintains a system virtual time function $v^s(\cdot)$ for each internal class in the hierarchy. When a block reference to the k-th block of queue i at the bottom level reaches the head of the queue, it is assigned a virtual start time s_i^k and a virtual finish time f_i^k at the bottom level queue as well as all ancestor classes' virtual queues in the hierarchy. The algorithm then applies the smallest start time first policy recursively down the hierarchy to choose the block to initiate the transfer for. The system virtual time function is given by $v^s(\cdot) = (s_{min} + s_{max})/2$, where s_{min} and s_{max} are the minimum and maximum start times among all active head of queue blocks under a class. This ensures that the discrepancy between the virtual times of any two active queues is bounded [13]. Furthermore, $s_i^k = max(v^s, f_i^{k-1})$, and $f_i^k = s_i^k + l_i^k/w_i$, where l_i^k is the length of the block.

This example illustrates how Pfimbi can enforce weighted bandwidth sharing between jobs, and also between replicas using only local decisions. This obviates the need for heavy centralized collection of statistics, and coordination of nodes.

2) Pfimbi can guarantee a bandwidth share for asynchronous replication: For many workloads, there is enough idleness for pending replication work to be performed. However, there may exist some I/O intensive workloads that keep the disk busy for extended intervals. This would result in replication being throttled indefinitely. To address this case, Pfimbi allows a share of the bandwidth to be guaranteed for asynchronous replication by using a weighted round robin approach. That is, after receiving a predefined number of synchronous blocks, Pfimbi can flush an asynchronous block regardless of the current activity to ensure asynchronous replication is not starved.

V. Discussion

A. Consistency

In this section, we argue that Pfimbi maintains the same consistency guarantees as HDFS.

We use the same read consistency definition as in the HDFS append design document: a byte read at one DataNode can also be read at the same moment in time at any other DataNode with a replica [14]. We define write consistency as: writes proceed in the same order at all replicas.

We separately discuss regular writes and appends. The difference is that a regular write creates a new file while an append adds data to an existing file. For reads, writes, and appends, we discuss how Pfimbi maintains consistency for both synchronous and asynchronous pipelines. The mechanisms used to ensure consistency leverage properties of

the write and append operations: (1) a file can only have one client adding data to it (writing or appending); (2) written data is immutable; (3) new content can only be added at the end of a file which implies that only the last block in a file can be under modification at any given time. This means for writes and appends, we only need to focus on the last block in a file.

Definitions: For the current block at node i, let R^{i} be the number of bytes received, and A^{i} be the number of bytes that have been acknowledged by downstream nodes to node i. If no synchronous node follows node i then $A^{\mathrm{i}} = R^{\mathrm{i}}$. A *generation stamp* is the version number for a block. A block is first assigned a generation stamp when it is created, and before a block is appended to, its generation stamp is updated.

1) **Synchronous pipelines***:*

Read consistency after a write or an append: Data flows down the pipeline while acknowledgements go upstream meaning $R^0 \geq R^1 \geq \ldots \geq R^{\mathrm{N\text{-}1}} \geq A^{\mathrm{N\text{-}1}} \geq \ldots \geq A^1 \geq A^0$. This implies that the following condition always holds:

C1: $\max_{\forall i}(A^{\mathrm{i}}) \leq \min_{\forall i}(R^{\mathrm{i}})$.

When a read operation starts, the reader checks how many bytes it is allowed to read. The reader is allowed to read up to A^{i} bytes, where $i \in \{0,1,\ldots,N-1\}$. Because of **C1**, all A^{i} bytes will have been received at (and can be served from) all other nodes in the pipeline. This guarantees read consistency, in that a byte read at one DataNode can also be read at the same moment in time at any other DataNode with a replica. Appends are similar to writes in how bytes become visible to readers.

Write and append consistency: HDFS only allows one client to be writing or appending to a file at a time. This is enforced centrally at the NameNode. Having a single writer or appender ensures that writes from different clients are properly ordered.

2) **Asynchronous pipelines***:*

Though Pfimbi changes when replicas are created, the following two principles ensure Pfimbi maintains both read and write consistency as defined above.

P1: Pfimbi does not make incomplete asynchronously forwarded replicas visible to clients.
P2: When an append operation overlaps with asynchronous replication, Pfimbi aborts ongoing forwarding of replicas for the current block being created asynchronously, and only restarts it after the append operation is complete.

Read consistency after a write: For nodes in the asynchronous portion of a pipeline $A^{\mathrm{i}} = R^{\mathrm{i}}$. When a block is in the process of being forwarded asynchronously, the number of bytes acknowledged, A^{i}, at the upstream nodes will be larger than the bytes received, R^{j}, at the downstream node because Pfimbi only asynchronously forwards a block after it has been completed at the upstream node. This violates the condition **C1** we used above. However, we can invoke **P1** to ensure that the replicas where the condition is violated are not visible. When a block is being forwarded, the replica being created at the destination is in a temporary state. Such a replica is only made visible to the NameNode when forwarding is complete. When forwarding is complete, R^{j} at the downstream node becomes equal to A^{i} at the upstream node. This guarantees that all visible replicas contain at least $\max_{\forall i}(A^{\mathrm{i}})$ bytes so **C1** holds for all visible replicas.

Read consistency after an append: When an append operation starts, data is appended synchronously to all currently completed replicas of the block. So if a block has been fully replicated (whether synchronously or asynchronously) the append proceeds as it does in HDFS. If the block is only partially replicated, we invoke **P2**. When a node starts servicing an append request it aborts ongoing block forwarding for the block being appended to. The downstream node will delete partially created files in the local file system. The node servicing the append also disregards requests to asynchronously forward the block. The append operation then synchronously adds data to all currently completed (and thus visible) replicas, guaranteeing that the appended data can be read from all of them and so maintaining condition **C1**. After the append finishes, a node can restart asynchronous block forwarding. Subsequently forwarded replicas will have post-append data and the updated generation stamp.

When the append starts after a DataNode completes forwarding an asynchronous replica, but before that new replica is visible at the NameNode, the new replica will not be included in the append pipeline. If this replica becomes visible to clients, **C1** would be violated. The use of generation stamps prevents this from happening. When the downstream DataNode notifies the NameNode of the replica's completion, the NameNode will check if the generation stamp of the replica is equal to the one the NameNode holds. In this case, it will not be, so the pre-append replica will not be added to the NameNode's map, and therefore will not become visible to clients. The NameNode also instructs the DataNode to delete this replica.

Guaranteeing a single writer: HDFS guarantees only one client is writing or appending to a file by issuing leases at the NameNode. In addition to the guarantees provided by leases, we invoke **P2** to ensure that asynchronous forwarding is never concurrent with a client's append.

B. Failure handling

Failure handling in Pfimbi requires no new mechanisms on top of those already used in HDFS and the big data frameworks running on top of it. The only difference is when these mechanisms need to be invoked.

If the node hosting the client (i.e., the task that writes) fails, then the pipelined write stops. Tasks failures are handled by the computation framework (Hadoop, Spark). A new copy of the task is typically launched on another node.

If the node that fails is not at the first position in the pipeline but it is within the synchronous segment of the pipeline, then

the pipelined write also stops. In this case the client times out the write and selects another pipeline to retry.

If the node that fails is within the asynchronous part of the pipeline, then the asynchronous pipeline will be severed at that node. All synchronous or asynchronous copies upstream of the failed node can complete normally. The node immediately upstream of the failed node also continues normally. If this upstream node sends a block notification to the failed node, then it will receive an error and stop retrying. If the block notification went through before the failure but there is no block request arriving from the failed node, the upstream node is still unaffected. This is because the upstream node does not maintain any extra state after sending the block notification since the remaining data transfer is entirely driven by the downstream node. However, in this failure scenario, if no further action is taken, the data will remain under-replicated. To get the desired number of replicas, Pfimbi falls back to the block-loss recovery mechanisms already supported by the master node. The master node periodically checks for under-replicated blocks within the file system and starts their replication in an out-of-band manner until the desired number of replicas is reached.

Lastly, all copies of a block could be lost after the job writing the block has finished. Recovery from such failures entails re-starting previously finished jobs. This requires tracing data lineage across jobs. Such mechanisms can be found in Tachyon [15], Spark [16] and in RCMP [17] for Hadoop.

C. Scalability

Pfimbi is just as scalable as HDFS and can leverage all scalability enhancements for HDFS (e.g. distributed master nodes) because the overall file system architecture remains the same. Importantly, the centralized master performs the exact set and number of operations. Thus, the load on the master remains unchanged. Pfimbi also retains the pipelined design to replication. Pfimbi only changes the manner in which the pipeline is managed. Finally, the coordination mechanism introduced by Pfimbi to enable flow control is lightweight and local to a pair of upstream-downstream nodes. Thus, the coordination mechanisms in Pfimbi scale with the size of the DFS.

D. Choosing between synchronous and asynchronous replication

Pfimbi leaves it to the application programmer to decide whether to use synchronous replication, asynchronous replication or a hybrid of the two. Each job chooses whether to use synchronous or asynchronous replication as part of its configuration. The application programmer can include this setting in the source code, or as a simple command line parameter when launching the job. The number of DataNodes in the synchronous portion of the pipeline is similarly specified as part of the job configuration. Automating this decision is beyond the scope of this work, and is left to future work.

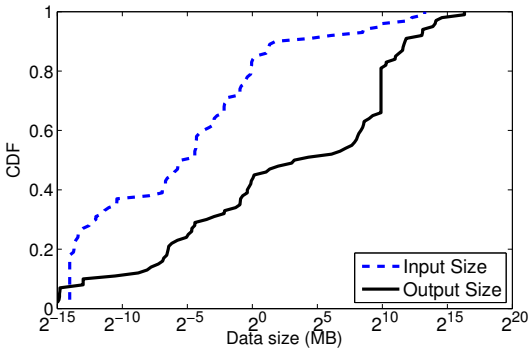


Fig. 6: CDF of job input and output sizes for the SWIM workload

VI. Evaluation

In this section, we report experimental results to quantify the performance of Pfimbi.

Software setup: We have implemented Pfimbi by extending Hadoop 3.0.0. We compare against default Hadoop 3.0.0. We run Hadoop and Pfimbi on top of the YARN resource allocator.

Hardware setup: We use 30 worker nodes and one master node. The worker nodes host HDFS DataNodes and YARN NodeManagers. Hence, computation is collocated with the data storage. The master and worker nodes have the same hardware configuration. Each node has two 8-core AMD Opteron 6212 CPUs and a 10GbE connection. Nodes have 128GB of RAM, a 200GB SSD and a 2TB hard disk drive. There are no other jobs running on the nodes during our experiments.

Configurations: We represent the tested configurations using the format DFS(#copies, #synchronous copies). Pfimbi(3,1) means using Pfimbi to write 3 copies, with only 1, the primary, being created synchronously. Two replicas are created asynchronously. HDFS(3,3) means using HDFS to create 3 copies, all synchronously. For HDFS, the two numbers will always be the same since HDFS uses synchronous replication.

Default Pfimbi parameters: The per-node buffer used for storing asynchronous replication blocks can hold 16 block. Replication flows use a flow weight of 1.0, unless otherwise specified, and by default, we prioritize replicas that are at earlier pipeline positions.

Metrics: We use four metrics to measure the performance of Pfimbi:

- Job runtime: the time it takes a job from start until it finishes all synchronous copies.
- The time between job start and the completion of all first replicas. This gives resilience against any single node failure.
- The time between job start and the completion of all replication work.
- The number of block write completions each second.

Job runtime measures job performance whilst the other metrics give information about the efficiency of the system. A job reports completion to the user after it finishes writing all synchronous copies. In HDFS, all copies are created syn-

chronously, so the time to write the primary will be identical to the time taken to write all the copies. However, when using Pfimbi these metrics will be different. This is why we consider them separately.

Workload: We use three workloads to test our system: a SWIM workload derived from a Facebook trace [18], Sort jobs and DFSIO jobs [19].

SWIM [11], [20] is a MapReduce workload generator. Given an input trace, SWIM synthesizes a workload with similar characteristics (input, shuffle and output data size, job arrival pattern). The SWIM workload is derived from a 2009 Facebook trace [18] and contains the first 100 jobs generated by SWIM. Figure 6 illustrates the distribution of job input and output sizes in this workload. Most jobs are small and read and write little data while the few large jobs write most of the data. Such a heterogeneous workload is popular across multiple data center operators [21], [22]. Including replication, roughly 900GB of data is written into the DFS. In our experiments we gradually scale up the workload by up to 8× (i.e., 7TB). This allows us to observe the behaviour of Pfimbi under light and heavy load conditions.

To analyze Pfimbi's benefits in a more controlled environment, we use Sort jobs. In all our Sort experiments, we sort 1TB of randomly generated data. Sort experiments are repeated 3 times for each configuration.

Lastly, we use DFSIO [19] as a pure write intensive workload. DFSIO is routinely used to stress-test Hadoop clusters. Unlike Sort, DFSIO is storage I/O bound for its entire duration, and its only writes are to the DFS. This enables us to analyze the behavior of DFS writes in isolation from the non-DFS disk writes (task spills, mapper writes) encountered in other Hadoop jobs.

A. Pfimbi improves job runtime

We start by analyzing Pfimbi's benefits on an I/O intensive workload by running a DFSIO job. We also analyze the ability of HDFS and Pfimbi to leverage heterogeneous storage by varying the location of primary writes (SSD or HDD). The two replicas always go to HDDs. Thus, we analyze the following pipeline configurations: SSD→HDD→HDD and HDD→HDD→HDD. Since in the SSD case the primary writes do not conflict with replica writes, for a fair comparison with the HDD case. We partition the nodes into two groups. Half of the nodes handle the replicas while the other half handle the primary writes and the tasks. *Partitioning is only necessary for this experiment. For all our other experiments, we do not partition our nodes.*

Figure 7 illustrates the results for the two DFSIO jobs. Pfimbi significantly lowers the completion time for primary writes whilst maintaining high disk utilization. Pfimbi improves job runtime by 52% and 73% for the HDD→HDD→HDD and SSD→HDD→HDD configurations, respectively. The time to obtain the first replica is also reduced by 32% for both configurations. These large gains are obtained with only a 5% penalty on the time it takes for all data to be written to the non-volatile storage. The small bar above the 2nd replica shows the time it takes for all dirty data to be flushed to non-volatile storage after an operating system sync call. The bar is much smaller for Pfimbi, since Pfimbi restricts the amount of asynchronous data that can be in the buffer cache.

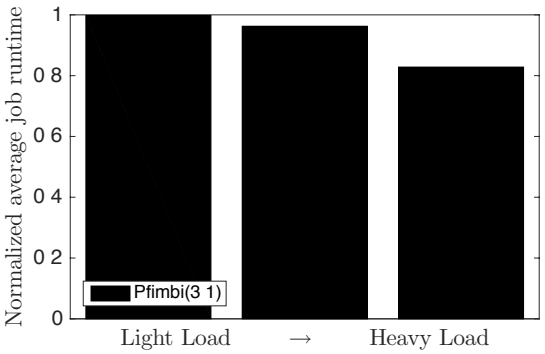


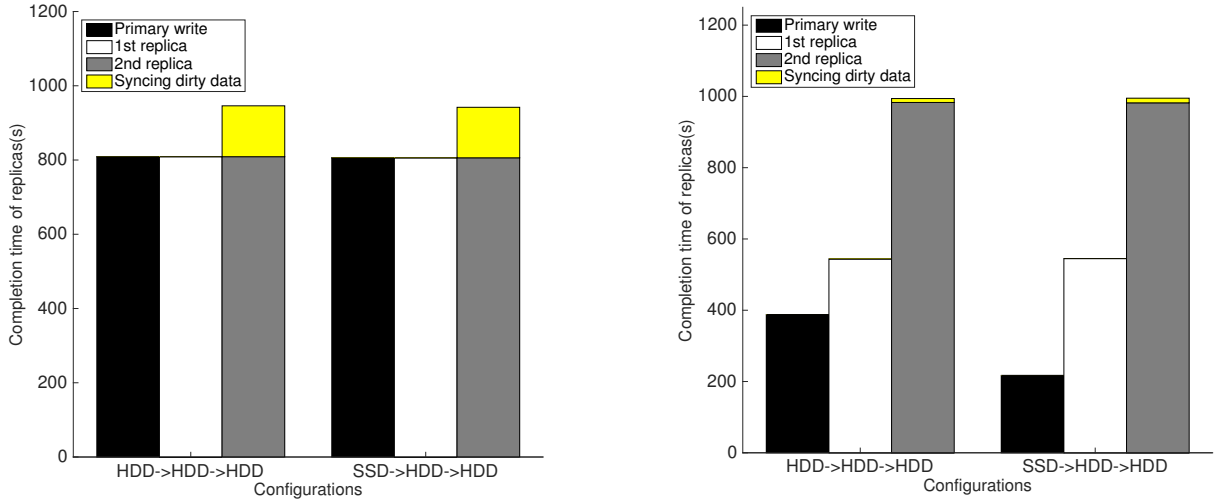
Fig. 8: Average SWIM job runtime for Pfimbi(3,1) normalized to HDFS(3,3) under varying workload scaling factors. Pfimbi outperforms HDFS by up to 18% on average under heavy I/O load; performance gains for individual jobs can be much higher (see Section VI-A). For light load there is very little contention so Pfimbi and HDFS perform similarly.

In Figure 7a, HDFS(3,3) cannot benefit from moving the first copy write to SSD because it is using synchronous replication. The pipelines are bottlenecked by the slower HDDs. With Pfimbi, the primary write duration improves by 44% when we move from the HDD→HDD→HDD to the SSD→HDD→HDD configuration. Pfimbi is better able to exploit storage heterogeneity.

We ran the SWIM workload to evaluate Pfimbi's benefits under different load conditions. The load on the storage subsystem is varied by scaling up the workload data size. Figure 8 shows the results. It plots the average job runtime under Pfimbi(3,1) normalized to the average job runtime under HDFS(3,3). Under a heavy workload (8x scaling) Pfimbi shows an 18% improvement in average job runtime. The per-job improvements (not illustrated) are up to 46% for small jobs (writing less than 1GB), and up to 28% for the most data intensive jobs (writing 80GB). We also ran lighter workloads (2x and 4x scaling). Pfimbi does not show significant benefits for these. When the workload is light there is very little contention caused by replication so there is little room for optimization.

Pfimbi shows improvements also when a Sort job is run in isolation. Figure 9a shows that as we decrease the length of the synchronous portion of the pipeline, the job runtime decreases. The runtime of Pfimbi(3,1), which is purely asynchronous is 36% less than HDFS(3,3), which is purely synchronous. This improvement is because Pfimbi reduces contention between normal traffic and asynchronous traffic.

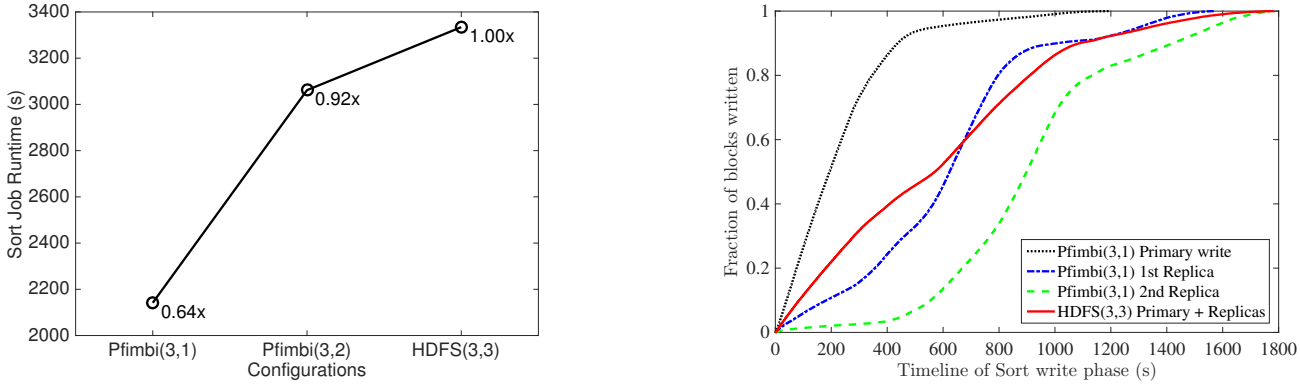
We next analyze how efficiently Pfimbi performs replication. We use the single Sort job to decouple the results from the influence of concurrent jobs. Figure 9b is CDF showing the proportion of replicas created with time. We measure time from when the first block of the Sort job is written. Pfimbi completes replication just as quickly as HDFS, all whilst reducing the duration of the write phase of the job by almost



(a) HDFS(3,3)

(b) Pfimbi(3,1)

Fig. 7: Completion time of different write stages for a DFSIO job. Primary writes go to either HDD or SSD. (a) HDFS(3,3) cannot benefit from SSDs because of synchronous replication. (b) Pfimbi finishes primary writes much faster due to the decoupled design and also benefits from switching to SSD. The small bar above the 2nd replica shows the time it takes for all dirty data to be flushed to disk after a sync call.



(a)

(b)

Fig. 9: Sort job under HDFS and Pfimbi. (a) Pfimbi improves Sort runtime. (b) The primary writes (black) and first replicas (blue) complete faster in Pfimbi. The second replicas (green) complete on par with HDFS(3,3).

30%.

For HDFS(3,3) blocks at all stages in the pipeline are created at the same rate since replication is synchronous. For Pfimbi, the blocks for the primary writes proceed at a much faster rate because they do not have to contend with replication writes. As the number of primary block completions decreases, Pfimbi starts creating more and more replicas.

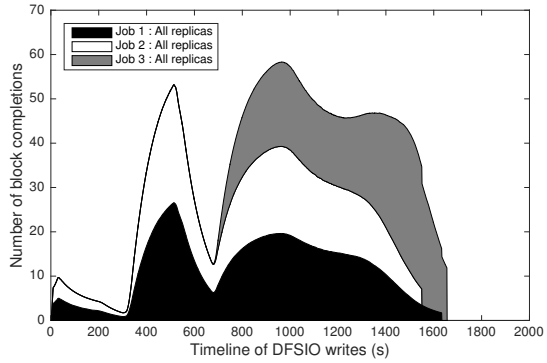
B. Pfimbi isolates primary writes from asynchronous replication

Asynchronous replication *without* flow control can be achieved using the setRep mechanism in HDFS. After a file has been written, the setRep command is invoked to increase the replication factor of the file's blocks. In Figure 10, we can see that such asynchronous replication *without* flow control causes the second of two back to back DFSIO jobs to run 2.7x slower than under Pfimbi. Flow control in Pfimbi minimizes the interference that the asynchronous replication data for the first job has on the runtime of the second. The duration of first job should be the same under both Pfimbi and HDFS. The slight difference in Figure 10 is within the expected natural variation between multiple runs of the same job.

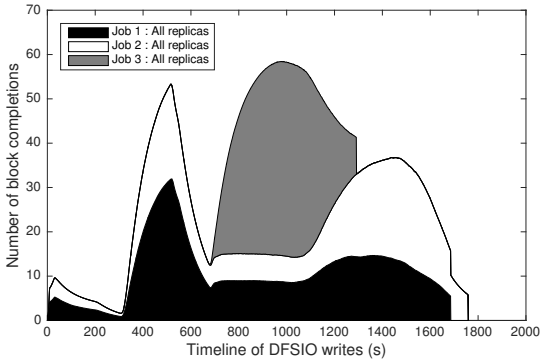
C. Pfimbi can divide bandwidth flexibly

Flexibly dividing disk bandwidth between jobs: We ran three concurrent DFSIO jobs and varied their flow weights. The third job is launched 500 seconds after the first two. Figures 11a and 11b show the rate of block completions for the three jobs. Time point zero second on the plot is when the first block is written. Each job writes 600GB of data. For this experiment, we fairly share bandwidth between replicas at different positions in pipelines. This enables us to clearly see the effects of dividing bandwidth between jobs.

In Figure 11a the flow weights are equal, so the three jobs'



(a) Flow weights ratio 1:1:1



(b) Flow weights ratio 1:4:16

Fig. 11: Three DFSIO jobs are run concurrently with different replication flow weights. Each job has a replication factor of 3. We measure time from when the jobs start writing blocks. Job 3 is started 500 seconds after the first two. When flow weights are different (in (b)) we observe bandwidth sharing at proportions according to the ratio of the flow weights, thus Job 3 is able to achieve failure resilience much sooner.

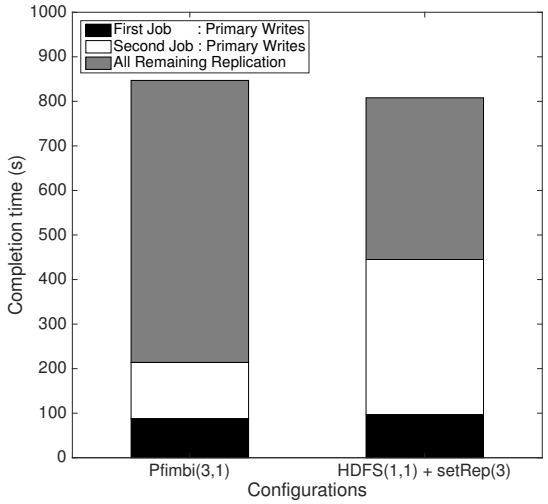


Fig. 10: Pfimbi(3,1) vs. HDFS(1,1)+setRep(3) for two back to back DFSIO jobs. setRep is called immediately after a job is completed. Asynchronous replication by itself (in setRep) is not enough to obtain large performance benefits. Without flow control setRep's asynchronous replication severely interferes with the second job.

replication writes share the available bandwidth equally, giving similar block completion rates. For Figure 11b we use different weights for the jobs in the ratio 1:4:16. From 300-500 seconds, the ratio of Job 1 and 2's block completions match the 1:4 ratio of the flow weights. Likewise, from 800-1000 second, the block completions of all three jobs reflect the assigned flow weights. By assigning Job 3 a higher replication weight, its replication finishes much sooner in Figure 11b than in Figure 11a. Throttling replication to avoid interfering with normal data results in dips in the block completion rates at the beginning of the experiment and at 600 seconds.

Prioritizing earlier pipeline positions within a job: Next, we show how prioritizing replicas that occur at earlier positions in their pipelines can help Pfimbi progressively achieve increasing levels of failure resilience. To better make the case we configured one DFSIO job to write 4 copies of its output data (3 replicas).

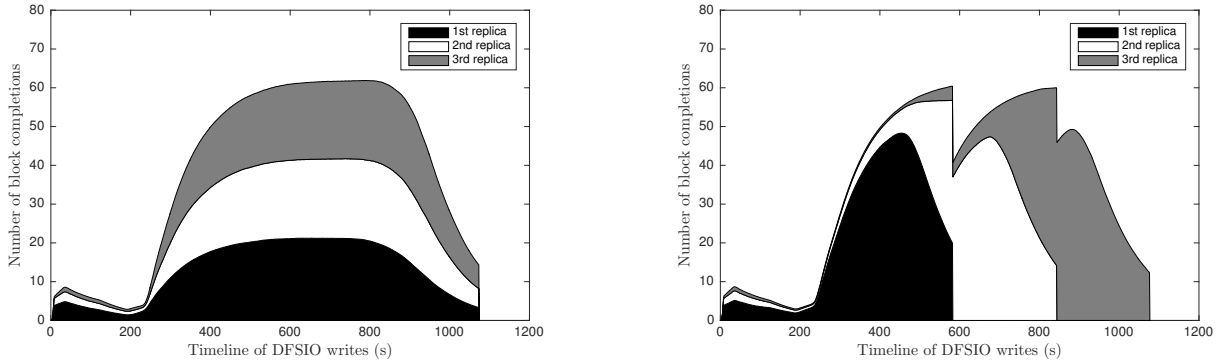
Figure 12 shows the number of block completions versus time for replicas at different positions in pipelines. Similar to Figure 11 time point zero second on the plot is when the first block is written. In Figure 12a we do not give priority to earlier positioned replicas, and we observe the 2nd replicas and 3rd replicas being created at the same rate as the 1st replicas. A user may prefer all the 1st replicas to be given priority. In Figure 12b, we set Pfimbi to give priority to replicas at earlier positions in the pipelines when selecting the block to be received next. We set a ratio of 100:10:1. This reduces the overlap between the writes of blocks at different positions in the pipeline, and replicas at earlier positions are finished sooner.

VII. Related work

In Section I, two related works [5][6] by HDFS developers have already been mentioned. The rest of this section will discuss additional related work in the literature.

Sinbad [23] addresses network contention as a potential performance bottleneck in DFSes. Sinbad leverages the flexibility of big-data jobs in choosing the locations for their replicas. Sinbad chooses replica locations to reduce the risk of network hot-spots. Sinbad and Pfimbi are highly complementary. The block placement strategy in Sinbad and Pfimbi's flexibility to perform flow-controlled asynchronous replication can potentially be applied simultaneously to achieve the best of both worlds.

TidyFS [24] is a simple distributed file system. It performs by default lazy, asynchronous replication. To motivate TidyFS, the authors find that in a Microsoft cluster, less than 1% of the data is read within the first minute after creation, and not more than 10% within the first hour after creation. These data access patterns also provide additional motivation for Pfimbi. TidyFS fundamentally differs from Pfimbi in that there is no management of asynchronous replication traffic. Once an asynchronous replication thread finds replicas to be created, it starts creating them immediately regardless of the system



(a) Fair sharing

(b) Prioritizing earlier replicas

Fig. 12: A single DFSIO job with replication factor 4 under Pfimbi. (a) The three replicas share the available bandwidth fairly at a 1:1:1 ratio. (b) The ratio is set to 100:10:1. This results in earlier replicas finishing sooner, achieving progressively increasing levels of failure resilience. Even if a failure were to occur at 700s, the data is still preserved.

load, similar to the setRep trick we considered in Section VI-B. Thus, in TidyFS, local clients writes will experience contention from asynchronous replication traffic leading to poor performance as Section VI-B shows. In contrast, Pfimbi uses flow control techniques to allow replication to be performed during periods of disk under-utilization, thus minimizing interference.

Pfimbi's flow control is distributed and its flow control decisions are made locally at each DataNode. Intelligence is also distributed since each DataNode runs its own hierarchical block scheduler to implement different resource sharing policies. This design enables Pfimbi to make fine-grained flow control decisions to exploit momentary IO under-utilization. In contrast, Retro [25] is a centralized resource management framework. In the case of HDFS, Retro would centrally compute and set the data rates of different synchronous replication pipelines to implement a resource sharing policy. The intelligence is centralized in Retro, DataNodes simply perform data rate control as instructed. It would be impractical to use Retro to implement the fine-grained flow control that is necessary to enable efficient asynchronous replication.

Parallel replication streams sharing a single source has been proposed as an alternative to pipelined replication to reduce write latency [26]. However, parallel replication does not address the overhead due to contention between replication and normal data. In reality, the overhead of I/O contention either in the network or on the disk can have a much larger effect on job performance than the overhead of pipeline latency.

VIII. Conclusion

Over the past five years, since its initial release, HDFS has continued to gain adoption. Many new features and optimizations have since been introduced, but until recently, little has changed about how data replication is handled. The nascent effort on adding in-memory storage with lazy-persist to HDFS has highlighted a need for a more flexible approach to data replication that does not sacrifice performance. In this paper, we have explored the idea of asynchronous replication within a design called Pfimbi. Our key contribution is that we have demonstrated that asynchronous replication when combined carefully with flow control mechanisms can provide very substantial improvement in job runtime over a range of workloads and heterogeneous storage configurations. Moreover, the flow-control mechanisms in Pfimbi can be used to realize a rich set of IO resource sharing policies. Finally, Pfimbi is readily deployable in existing big data software ecosystems.

Acknowledgement

We would like to thank the anonymous reviewers for their thoughtful feedback. This research is sponsored by the NSF under CNS-1422925, CNS-1305379 and CNS-1162270, an IBM Faculty Award, and by Microsoft Corp. Simbarashe Dzinamarira is also supported by a 2015/16 Schlumberger Graduate Fellowship.

References

[1] "Apache hadoop," http://hadoop.apache.org/, 2015.

[2] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010, pp. 10–10.

[3] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. IEEE, 2010, pp. 1–10.

[4] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03.

[5] "Archival Storage, SSD & Memory," http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/ArchivalStorage.html.

[6] "Memory Storage Support in HDFS," http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/MemoryStorage.html.

[7] "Terasort," https://hadoop.apache.org/docs/r2.7.1/api/org/apache/hadoop/examples/terasort/package-summary.html.

[8] "Nutch," https://wiki.apache.org/nutch/NutchTutorial.

[9] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*. IEEE, 2010, pp. 41–51.

[10] "k-Means clustering - basics," https://mahout.apache.org/users/clustering/k-means-clustering.html.

[11] Y. Chen, S. Alspaugh, and R. Katz, "Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1802–1813, 2012.

[12] J. C. Bennett and H. Zhang, "Hierarchical packet fair queueing algorithms," *Networking, IEEE/ACM Transactions on*, vol. 5, no. 5, pp. 675–689, 1997.

[13] I. Stoica, H. Zhang, and T. Ng, "A hierarchical fair service curve algorithm for link-sharing, real-time, and priority services," *IEEE/ACM Transactions on Networking (TON)*, vol. 8, no. 2, pp. 185–199, 2000.

[14] "Append/hflush/read design," https://issues.apache.org/jira/secure/attachment/12445209/appendDesign3.pdf, 2009.

[15] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: Reliable, memory speed storage for cluster computing frameworks," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC '14. New York, NY, USA: ACM, 2014, pp. 6:1–6:15.

[16] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12.

[17] F. Dinu and T. S. E. Ng, "Rcmp: Enabling efficient recomputation based failure resilience for big data analytics," in *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, ser. IPDPS '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 962–971. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2014.102

[18] "SWIM Workloads repository," https://github.com/SWIMProjectUCB/SWIM/wiki/Workloads-repository.

[19] "Running DFSIO mapreduce benchmark test," https://support.pivotal.io/hc/en-us/articles/ 200864057-Running-DFSIO-mapreduce-benchmark-test.

[20] "SWIM," https://github.com/SWIMProjectUCB/SWIM/wiki.

[21] Y. Chen, S. Alspaugh, and R. Katz, "Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads," in *Proc. VLDB*, 2012.

[22] "J. Wilkes - More Google cluster data," http://googleresearch.blogspot.ch/2011/11/more-google-cluster-data.html.

[23] M. Chowdhury, S. Kandula, and I. Stoica, "Leveraging endpoint flexibility in data-intensive clusters," in *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*. ACM, 2013, pp. 231–242.

[24] D. Fetterly, M. Haridasan, M. Isard, and S. Sundararaman, "Tidyfs: A simple and small distributed file system." in *USENIX annual technical conference*, 2011.

[25] J. Mace, P. Bodik, R. Fonseca, and M. Musuvathi, "Retro: Targeted resource management in multi-tenant distributed systems," in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*.

[26] "Ability for hdfs client to write replicas in parallel," https://issues.apache.org/jira/browse/HDFS-1783, 2015.