Managing Tail Latency in Datacenter-Scale File Systems Under Production Constraints

Pulkit A. Misra★ María F. Borge‡ Íñigo Goiri†
Alvin R. Lebeck★ Willy Zwaenepoel‡ δ Ricardo Bianchini†
★*Duke University* ‡*University of Sydney* δ*EPFL* †*Microsoft Research*

Abstract

Distributed file systems often exhibit high tail latencies, especially in large-scale datacenters and in the presence of competing (and possibly higher priority) workloads. This paper introduces techniques for managing tail latencies in these systems, while addressing the practical challenges inherent in production datacenters (*e.g.*, hardware heterogeneity, interference from other workloads, the need to maximize simplicity and maintainability). We implement our techniques in a scalable distributed file system (an extension of HDFS) used in production at Microsoft. Our evaluation uses 70k servers in 3 datacenters, and shows that our techniques reduce tail latency significantly for production workloads.

1 Introduction

Motivation. Large-scale distributed systems exhibit unpredictable high-percentile (tail) latency variations, which harm performance predictability and may drive users away. Many factors may cause these variations, such as component failures, replication overhead, load imbalance, and resource contention [18, 21, 33, 35, 49]. The problem is exacerbated when systems run on harvested resources. Resource-harvesting datacenters colocate latency-sensitive services (*e.g.*, search engines) with batch jobs (*e.g.*, data analytics, machine learning) to improve resource utilization [24, 50, 58, 61]. In these datacenters, performance isolation mechanisms [20, 24, 29, 36, 50, 56, 58, 61] throttle or even deny resources to the batch jobs when the services need them.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
EuroSys '19, March 25–28, 2019, Dresden, Germany
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6281-8/19/03...$15.00
https://doi.org/10.1145/3302424.3303973

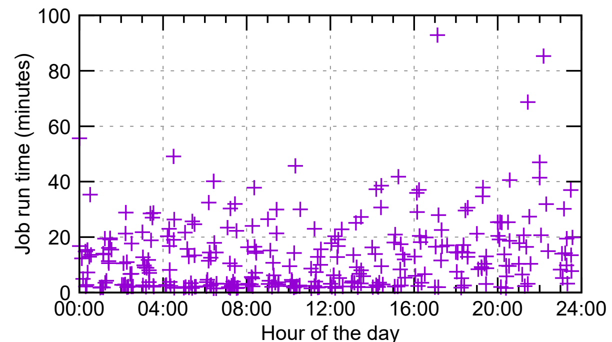


Figure 1. Run times of 300 copy jobs over 2 weeks. All days are overlayed over their 24 hours.

Many papers have addressed tail latency management in distributed systems. For example, they have tackled straggler tasks in data analytics frameworks [9, 10, 43, 59] and requests in multi-tier services [31], tail latencies in distributed file/storage systems [34, 44], and enforced service-level objectives (SLOs) for compute or storage [33, 37, 47, 51–53, 62].

Unfortunately, the prior works do not account for some important challenges and constraints of *real production systems* (Sections 2 and 7). For example, the server hardware in datacenters is heterogeneous in terms of resources and storage configurations (static heterogeneity). The performance isolation mechanisms of resource-harvesting datacenters produce another form of (dynamic) heterogeneity. Thus, tail latency management techniques evaluated in the absence of such static and dynamic heterogeneity may miss important effects. Datacenters are also very large; evaluating ideas on small systems sidesteps many scalability challenges, *e.g.* difficulties with using centralized components. Perhaps most importantly, production systems must be simple and maintainable. Complex techniques (*e.g.*, relying on sophisticated performance modeling) are undesirable, as they require expertise and skills that most engineers are not trained on.

Our work. In this paper, we focus on tail latencies in distributed file systems under these production constraints. In particular, we address the challenging scenario where the distributed file system only stores data for the batch workloads, but the latency-sensitive services have full priority over the shared resources (*e.g.*, CPUs, local disk

bandwidth) on the same machines. Thus, *we seek the best possible file system tail latency for the batch workloads;* service performance is protected by isolation techniques (*e.g.*, [29, 36]) and are *not* a topic of this paper.

Though the batch jobs have more relaxed performance constraints than the services, lowering their tail latency is important because: (1) completing them faster frees up capacity for other jobs to run; (2) lower tail latency improves performance predictability and user satisfaction; and (3) batch jobs may be rendered useless if they take excessively long. As an illustration of the performance variation we observe, Figure 1 shows the run times of 300 executions of a simple I/O-bound job (distributed copy of a 200-GB file) on 4k servers in a production datacenter over 2 weeks. As we can see, job performance may vary 60× or more, due to tail data access latencies.

These performance variations occur despite the fact that speculative execution (SE) is enabled in our jobs. SE tracks the tasks' durations and issues duplicate tasks for managing tail latencies in batch workloads [9, 10, 59]. Unfortunately, SE cannot tackle all sources of storage-level tail latencies without a stronger coupling between compute and storage layers [43]. Worse, SE may harm performance when resources are scarce [8, 39, 41, 43, 45], a common scenario in resource-harvesting datacenters. Thus, managing storage-level tail latencies independently is more attractive, leads to simpler systems (less coupling), and reduces the need for speculative tasks.

Along these lines, we first characterize the sources of storage-level tail latency impacting the run times of I/O-bound jobs shown in Figure 1 (Section 3). The characterization shows that data read and write accesses exhibit the most performance variation, not metadata operations. In addition, long disk queues and server-side throttling are the main culprits, not the utilization of the network, CPU or memory.

We then propose two client-side techniques for managing tail data access latencies: "fail fast" for writes, and "smart hedging" for reads (Section 4). Both techniques are oblivious to the source of variations, and rely on simple server-side performance reporting and careful reactive policies, while leveraging the existing replication and fault-tolerance mechanisms in distributed file systems. Many systems use chain replication [48] for data writes, where each write request (possibly broken into chunks called "packets") is pipelined serially across the servers that store a replica of the block [1, 2, 11, 13, 14, 22, 23, 46]. In such systems, a slow server in the pipeline can significantly degrade write latency. However, prior works have almost always assumed that writes happen "in the background" (buffered writes). In contrast, our characterization shows that, in practice, several concurrent block writes on the same server can overflow the write buffer and significantly impact tail latencies.

To reduce data write tail latencies, our fail fast technique detects and replaces the slowest server in a pipeline. The new server must receive all packets that have already been written before writing can resume. However, aggressive server replacements may overload the system. In fact, since replacements are expensive, fail fast estimates the cost of replacing a server and performs a replacement only if it would indeed reduce write latencies overall.

Our smart hedging technique for data reads monitors the server's performance on a per-packet basis, and starts a "hedge" (duplicate) request [18] when performance starts to degrade. Since aggressive hedging may cause overload, we hedge adaptively and (exponentially) back-off from a server that does not complete a hedge before the original request.

Implementation and results. To experimentally evaluate our techniques, we implement them in Hadoop Distributed File System (HDFS) [1], a popular file system for frameworks such as MapReduce [19] and Spark [12], and call the result "CurtailHDFS" (Section 5).

Our evaluation uses (1) synthetic workloads on a 4k-server testbed; and (2) production workloads in 3 datacenters with a total of 70k servers (Section 6). We compare CurtailHDFS to baselines that include typical server-side techniques for managing latency, such as tracking server performance. The results show that CurtailHDFS can reduce 99th-percentile latency by 19× compared to HDFS for I/O-bound jobs. For the more balanced production workloads, CurtailHDFS reduces the 99.9^{th}-percentile write latency by 2× compared to HDFS, and the average read latency by 1.4× compared to state-of-the-practice hedging. These are significant improvements, especially given the limited scope of our changes, on top of existing techniques (*e.g.*, speculative task execution).

Though we evaluate our techniques in highly heterogeneous resource-harvesting datacenters, they are general and applicable in other contexts as well. In fact, we are contributing our techniques to open-source HDFS, so they will immediately benefit frameworks and workloads that currently use it in any datacenter.

Summary. Our contributions are:

- We characterize the tail latency of batch jobs running on production HDFS in resource-harvesting datacenters.
- We propose general client-side techniques for managing tail latencies in distributed file systems, and implement them in HDFS. Our write technique uses an entirely new approach to shortening tail latencies, whereas our read technique improves on prior works.
- We evaluate our techniques extensively, and show that they lower tail latencies significantly.

2 Production challenges and constraints

There are many challenges to manage tail latency in production distributed file systems, here we enumerate several we encountered.

Massive scale. At datacenter scale, systems are more prone to transient misbehaviors or failures that cause latency tails. Moreover, they must avoid centralized components. So, systems with a single primary metadata manager, such as HDFS, must be extended. Our production HDFS federates multiple HDFS sub-clusters by placing multiple software "routers" in front of the sub-cluster metadata managers, as proposed by Misra *et al.* [38].

High heterogeneity. Datacenter hardware is often heterogeneous, with at least a few server generations with different performance characteristics. Even servers of the same generation may use multiple types (SSD, HDD, or both) or configurations of storage media (*e.g.*, raided with different numbers of devices). There are 12 server configurations in the datacenters we study in this paper.

Resource harvesting. Our production HDFS uses two isolation mechanisms to protect the co-located services: (1) throttling of the data access throughput of the server (at 60MB/sec), and (2) a "busy" flag that informs the corresponding metadata manager and clients that the server cannot take more access load. We find that ~20% of servers may concurrently become unavailable because of such activity.

Background replication. On server or disk failures, the lost replicas must be re-created in the background, which increases server load. The problem is exacerbated in our datacenters because the operators of the latency-sensitive services may reformat disks at any time. To prevent data loss, our production HDFS stores the replicas of each block on servers that run different services, as proposed by Zhang *et al.* [61]. Despite doing this, it is common to have to re-create 25TB monthly, with peaks of 128TB.

Load imbalance. At scale, data servers and/or metadata managers may become overloaded and provide poor performance. Our production HDFS deployments create an average of 2 blocks per second with spikes of more than 100. During these spikes, some servers are idle while others show more than 20 concurrent data accesses.

High hidden costs. Datacenter operators are careful to limit complexity in their systems, to lower labor and maintenance costs. Similarly, new system components and data pipelines are expensive to produce, operate, and maintain. Complexity and new infrastructure must produce significant benefits to justify their costs.

3 Characterizing performance variation

This section characterizes the performance variations we observe in real datacenters and their sources. We base our study on HDFS, but our observations directly apply to other well-known systems, such as Cosmos Store [17] and GFS [23], that have similar structure.

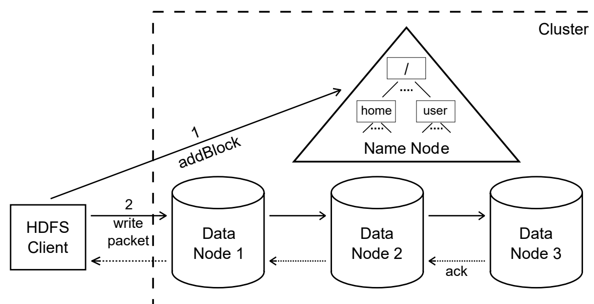


Figure 2. Pipeline to write a block in HDFS.

3.1 HDFS overview

HDFS is a popular open-source system that is used in companies like Yahoo, Microsoft, and Twitter. It comprises a primary metadata manager called the "Name Node" (NN), and its per-server block storage node called "Data Node" (DN). The NN manages the namespace and maps files to their blocks. HDFS replicates each block (across three DNs, by default) for redundancy. Next, we describe write and read operations on a file.

Writes. Writes append data to an existing file, *i.e.* there are no writes-in-place. The client first asks the NN for the creation of a new block (256MB in our deployments) and specifies the replication factor for it. The NN returns an ordered set of DNs that should store the block's replicas; it selects the DNs based on a pre-configured policy (*e.g.*, balance free space). The DNs form a *write pipeline*.

HDFS writes the data in a pipeline at a packet granularity (64kB). Figure 2 shows the steps involved in writing a packet. The client sends the packet to the first DN, which writes it to its local file system and propagates it to the next downstream DN, and so on. Upon receiving the packet, the last DN sends an ack that propagates all the way back to the client, thereby completing the operation.

To speed up the process, the client sends multiple (64) packets concurrently. In the event of failures, the client asks the NN for replacements and uses an existing DN in the pipeline for transferring all committed data to the new DNs. Writing can resume after the new DNs have been brought to a consistent state.

Reads. Reads involve fewer steps. First, the client asks the NN for the block's locations, and the NN returns a list of DNs that store the replicas; the list is sorted based on proximity to the client. Then, the client establishes a connection with the first DN on the list and reads one packet (1MB) at a time. In case of errors (*e.g.*, DN failure or data corruption), the client tries reading the remaining packets from the next DN on the list.

Concurrency. HDFS uses a single-writer, multiple-reader model. An HDFS client that opens a file for writing is given *write ownership* of the file by the NN and no other client can write to the file until the writer relinquishes ownership. However, write ownership does not prevent other clients from reading the file; a file can have many concurrent readers.

Consistency. Although HDFS allows multiple readers and a writer to access the same file concurrently, it provides no guarantees about the visibility of writes to a file under modification. However, HDFS does guarantee that all writes become immediately visible to any reader after the writer closes the file and relinquishes the write ownership. Thereafter, HDFS provides strong consistency: a client can read the latest data from any replica (DN) that stores the recently created/appended file block(s) [**?**].

Extensions. As we mention in Section 2, our production HDFS uses two main extensions to the standard open-source version: (1) the replica placement algorithm from Zhang *et al.* [61]; and (2) the sub-cluster federation techniques from Misra *et al.* [38]. Writes and reads happen as we describe above, below the routers at the federation layer. *Neither extension has a relevant impact on tail latency or on our techniques to manage it.*

3.2 Methodology

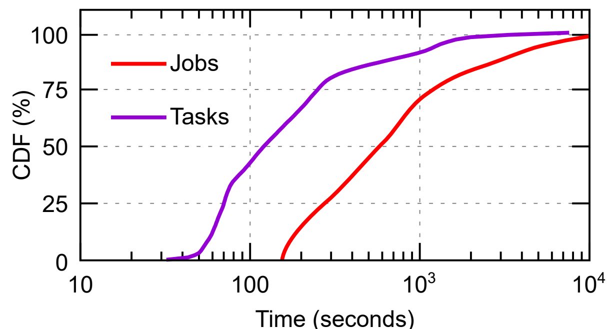
Experimental setup. For our next experiments, we deploy our production HDFS on 4k servers across 4 sub-clusters in a resource-harvesting datacenter. The servers have 12-32 cores, 32-128GB of memory, 10-Gbps NICs, and a 6TB RAID consisting of 4 15k-RPM HDDs. In some servers, we also use a 1TB RAID of 2 SSDs to store the file system data. The NN in each sub-cluster replicates each block across 4 DNs in the same sub-cluster; replicas of a block may be stored on HDDs, SSDs or some combination of the two storage mediums.

Workload. Our experiment profiles a DistCP [4] job that spawns 200 tasks for copying a 200GB file between 2 sub-clusters. We run this experiment every 2 hours, for a week. Each run spans a few thousand servers; speculative task execution is enabled in each run. We use TeraGen [5] to re-create the source file before each run. See Section 6 for our study of other workloads.

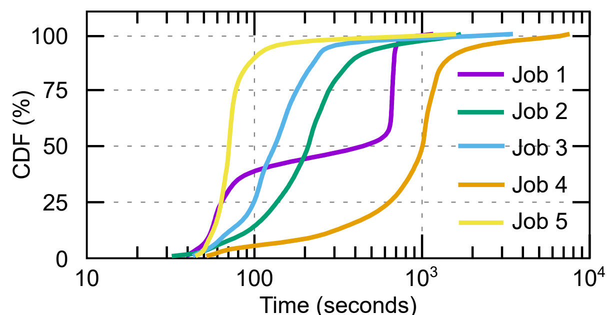
Monitoring infrastructure. We extended HTrace [3] to profile I/O operations at a packet granularity and collect system load metrics. We are contributing our extensions to open-source HDFS [6, 7].

3.3 Job/task performance variation

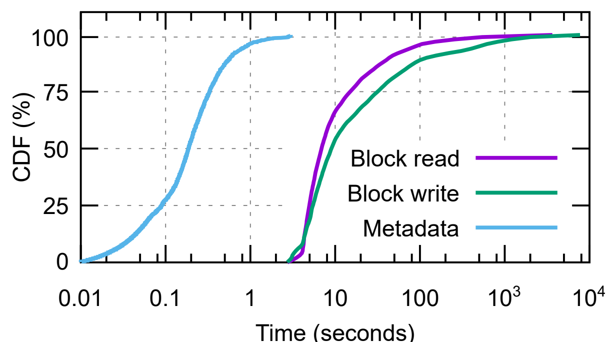
Figure 3a presents the CDF of the DistCP job run times. The average job run time is 20 minutes and the standard deviation is 27 minutes. More strikingly, *the fastest job takes 2 minutes whereas the* 99th *percentile is 120 minutes.* To understand this huge variation, we consider the running times of tasks within a job.



(a) Job and task running times.



(b) Task running times within sampled jobs.



(c) Latencies per operation type.

Figure 3. Performance of DistCP jobs and file accesses.

Figure 3a also shows a CDF of the task run times across all DistCP jobs. Though each task does the same amount of work (copies 1 GB), their run times vary significantly. The average task run time is 334 seconds and the standard deviation is 730 seconds. The fastest task takes 32 seconds whereas the 99th percentile is ~1 hour.

To explore whether there are common job slowdown patterns, we consider the task run times within a job. Figure 3b shows the results for a few DistCP jobs. Clearly, a few hotspots slow down jobs in some cases and heavy load on the entire system has an impact in others. For example, a few stragglers slow down Job 5. In contrast, more than 75% of the tasks are an order of magnitude

Table 1. Correlating latency and DN resource utilization.

Resource type	Spearman coefficient (ρ)	
	Reads	Writes
CPU utilization (%)	0.23	0.31
Memory utilization (%)	0.29	0.25
Disk queue length	0.7	0.65
Network bandwidth (MB/s)	0.14	0.11

slower than the fastest task in Job 4, indicating that the system is under heavy load. This is also corroborated by the fact that the fastest task in Jobs 4 and 5 has similar run times (~50 seconds), but the median task run time in Job 4 (1000 seconds) is 20× slower than the fastest task and 15× slower than the median task run time in Job 5 (70 seconds). The other jobs show similar behavior.

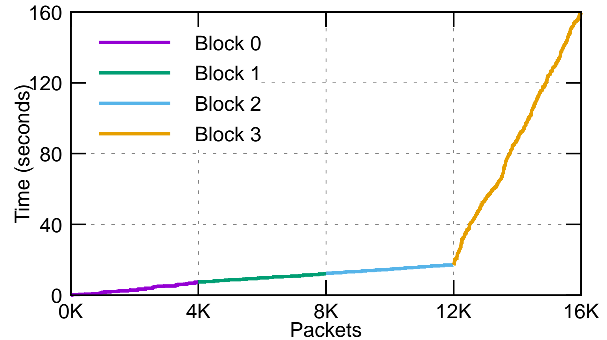
These extreme variations in run times are less likely to occur in other settings. For example, in smaller clusters lacking server heterogeneity [26, 43, 44, 57], or when experiments (*e.g.*, background load) are tightly controlled. Datacenters exhibit more static and dynamic heterogeneity, non-uniform server load distributions [29], interference from co-located applications, and server reimaging/restart by cluster management systems.

3.4 Sources of performance variation

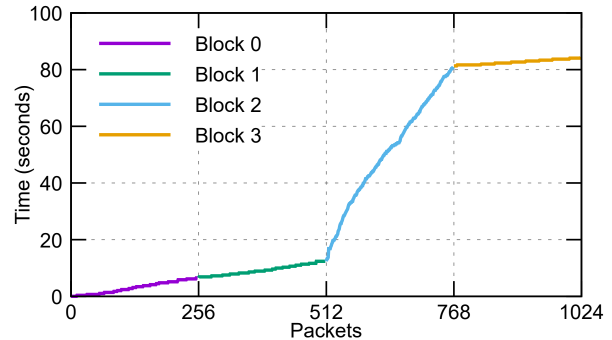
In this section, we analyze the traces of slow tasks to determine operations that cause tasks to slowdown. Figure 3c shows a CDF of the operation latencies. We classify operations into block read, block write and block metadata. Reads and writes suffer major slowdowns: the 99th percentile latencies are an order of magnitude greater than the median, but metadata operations have little impact since the 99th percentile metadata operation latency is similar to the fastest reads and writes.

Next, we look at individual traces of writes and reads to determine the causes for slowdown.

Writes. We correlate between block write latencies and the DNs' utilization metrics in Table 1. The table shows that there is a strong correlation between write latencies and disk queue length, indicating that disk contention (either from latency-sensitive services or other batch jobs) and the resulting queuing delays have a substantial impact on performance variation. Interestingly, this correlation is strong even though a packet write at each server completes as soon as it is written to the local buffer cache. Several HDFS blocks concurrently being written to the same server are enough to overflow the buffer cache. (HDFS block reads and I/O load from the local latency-sensitive service may also put pressure on the cache.) Again, *this effect is less likely to occur in smaller scale systems with less contention/imbalance or in the absence of co-located applications.* The correlation between write latencies and other DN resources is weaker.



(a) Write four 256MB blocks to a 1GB file.



(b) Read four 256MB blocks from a 1GB file.

Figure 4. Sample write and read timelines. Each color represents a block. Each point depicts a set of bytes transferred during a write or read. Time is on the Y axis.

Figure 4a illustrates the write performance variation. It shows the observed latencies while writing the packets of the four blocks in a file, each represented with a different color. Block 3 suffers a huge slowdown because of high write latencies through its pipeline, taking more than 2 minutes to complete. Two servers exhibit large disk queues during the writing of this block. The 99th percentile of packet write latencies is almost 100ms for this block, but less than 16ms for the others.

Reads. Like writes, our correlation analysis of read latencies and DN resource utilization shows that disk contention is a primary cause for performance variation (Table 1). Figure 4b shows the packet latencies while reading four blocks. As we can see, block 2 takes significantly longer (more than 1 minute) to read. However, long disk queues are not the only reason. Another factor is server-side throttling that limits the overall DN throughput at 60MB/s (instead of the many hundreds of MB/s we can get from the RAIDed storage devices), regardless of how many clients are accessing the server. *This effect would not have occurred in dedicated clusters or other setups where there is no resource harvesting.*

Other sources. We find cases where contention for other resources (*e.g.*, CPU, network bandwidth) has an impact on performance, but they are rare.

3.5 Summary

We find that: (1) job running times are highly variable and without clear slowdown patterns, such as time of day; (2) load from both latency-critical services and other batch jobs impacts performance; (3) data read and write accesses exhibit the most performance variation, not metadata operations; and (4) long disk queues and server-side throttling introduce the most variation, not the utilization of the network, CPU or memory. These effects are more likely to occur in large production systems.

As Figures 4a and 4b show, long disk queues and throttling can last for minutes, whereas other block accesses take seconds. In addition, although not seen in our characterization study, contention for other resources can cause similar performance variation for data accesses, *e.g.*, contention for CPU in erasure-coded systems and/or due to high demand from co-located CPU-bound applications. This means that, *regardless of the source of the delay,* it is important to react, *i.e.* direct accesses away from these slow servers. However, we need to react carefully so (1) transient effects do not cause unneeded reactions; (2) older, less-performant servers can still be used; and (3) we avoid reactions when all servers are overloaded.

4 Managing tail latencies

In this section, we present our techniques for tackling the sources of performance variability from Section 3. Throughout the section, we use the general terms "metadata server" and "block server", as our techniques apply to distributed file systems beyond just HDFS.

4.1 Basic principles

Both techniques leverage information available at the distributed file system client library to make decisions. At a high level, they proceed through the following phases: *track latency*, *check for slowness*, *decide whether to act*, and *take action*. In the *track* phase, the client stores the packet and block latencies for each block server with which it is communicating. It then uses these tracked latencies in the *check* phase to determine whether a server is slow. This check needs to be agnostic of the source of the delay and, simultaneously, loose enough to avoid reacting to transient conditions or older, less-performant servers. The client runs the check phase for the first time after it has collected enough latencies to make a meaningful decision. If the check determines that a server is slow, the client *decides* whether to take a mitigation action (*i.e.*, a fail fast for a write or a hedge for a read). This decision includes backing off from overloaded servers (reads) and avoiding actions that would not produce a clear latency benefit (writes). If the client decides that it will be beneficial to act, it *takes the action* and goes back to the tracking phase.

Our approach adapts to dynamic load changes by only considering recent requests in the check phase (load may have changed recently at the servers), and by comparing the server's request latency against those of other servers (all servers may be highly loaded). Moreover, the approach is agnostic to the reason for slowness, *e.g.* delays due to other file system requests, latency-sensitive service activity, or throttling.

Importantly, these basic principles conform to the constraints of production environments. First, they enable a client to make local decisions and do not require any centralized infrastructure in the decision-making process. Therefore, they do not pose any impediment to scalability. Second, they adapt to dynamic changes and do not make any assumptions about the system, which makes them amenable to run on harvested and heterogeneous resources. Third, they can be easily deployed and integrated into existing systems, since our techniques are simple and do not require invasive server-side changes.

Below, we detail each technique and discuss the alternate approaches we considered in Section 4.4.

4.2 Fail fast for writes

Fail fast dynamically replaces the slowest server in a data write pipeline. The client treats the slowest server as having failed (hence the name fail fast), and leverages the existing replication and fault-tolerance mechanisms in the file system for replacing the server.

In more detail, the client records the packet latencies from the block servers in a pipeline (track). Next, it uses the recorded latencies to determine whether the slowest server is substantially slower than the fastest server (check). Specifically, the check compares a high percentile latency of the slowest server against a multiple of the same high percentile latency of the fastest server to determine if the slowest server is too slow. (In our HDFS implementation, the default high percentile is the 95th and the multiplicative factor is 3×.) If the slowest server is deemed too slow, the client uses a cost function to decide whether replacing it would be beneficial, given the high cost of reconstructing the already-written data of the block on a different server (decide). If it deems the replacement to be beneficial, the client asks the metadata server for a replacement, disconnects the current pipeline, rebuilds the block on the replacement server, and finally, resumes writing the remainder of the block with the new pipeline (action). Figure 5 illustrates the action phase of fail fast operation: the client observes that Block Server 2 is exhibiting high packet write latencies and replaces it in the pipeline. Different distributed file systems may

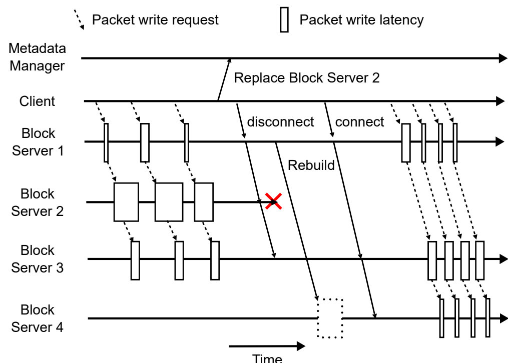


Figure 5. Fail fast of write pipeline.

implement the replacement protocol slightly differently, but the principle is the same.

The cost function deserves more explanation. It predicts two completion times for the write: without replacement and with replacement of the slowest server. Assuming no replacement, we multiply the average write latency of the slowest server by the number of packets remaining to be written. For predicting completion time with replacement, we first predict the time to reconstruct the committed data in the block on a new server. We compute this value using the latencies of the fastest server in the pipeline as the source for reconstruction. Next, we compute the time to complete the write with the new pipeline using the average of the write latencies of the remaining servers in the pipeline. The time to complete the write with replacement is the sum of these two predicted values. The client replaces the slowest server if the predicted completion time with replacement is less than the projected completion time for the current pipeline.

Under heavy load, the time for reconstructing the block on the replacement server might substantially exceed the expected time. In such cases, the action phase aborts the current reconstruction and restarts with a new replacement server. It also changes the source server and increases the expected reconstruction time linearly with each abort. Finally, we stop attempting a replacement after a threshold number of unsuccessful tries. (By default, our HDFS deployments try 5 times before giving up.)

4.3 Smart hedging for reads

Request hedging [18] leverages the multiple copies of data for cutting read tail latencies. A popular approach is to issue a hedge request, if the latency of the current request is greater than a static threshold. Figure 6 illustrates read hedging, where a client switches to another replica (server 2) after observing a high read latency (from server 1). Although this static approach can cut tail latencies, its effectiveness depends on the value of the threshold: an excessively high value may result in infrequent hedging, whereas an excessively low value could overload

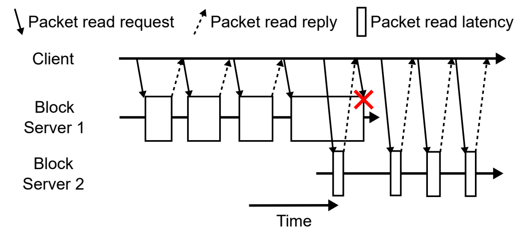


Figure 6. Read hedging.

the system. We actually observed the latter behavior in production (Section 6.4). Moreover, even a finely-tuned threshold value may require readjustment under varying load conditions (dynamic heterogeneity).

Thus, our smart hedging technique uses a *per-client* dynamic threshold along with a retry policy to control the hedging rate. Each client constantly records the latency of the packets it reads from servers (track). Initially, it uses a static threshold to trigger hedging and over time leverages the tracked latencies to adjust the threshold (check). The check compares the latency of the server against a multiple of a high percentile of the latencies the client has experienced from other servers. In this case, the multiplicative factor compensates for variability in request service times [44] or queueing delays [27]. (Like for fail fast, the default high percentile is the 95th and the multiplicative factor is 3×.) As the value of this high percentile latency changes, the hedging threshold for each client changes as well.

If a request is taking too long, the client considers sending a hedge request to the next server on the replica list for the block (decide), say server 2. Before issuing the hedge, it checks whether the last hedge to server 2 completed after the corresponding non-hedge request (*i.e.*, the last hedge to server 2 was unsuccessful). If so, it exponentially backs off from server 2 and considers the next one on the list. Otherwise, it issues the hedge to server 2 (action). Exponential backoff ensures that we do not attempt to access an already overloaded server, and works well in practice. Specifically, our smart client waits for an exponentially increasing number of packets before re-attempting a hedge request to a server who failed a hedge in the past attempt.

4.4 Alternative techniques we discarded

Eventually-consistent writes. To reduce write tail latencies, we considered making a client write to all replicas in parallel and completing the write as soon as a preconfigured subset of the replicas acknowledge the write. The remaining replicas would receive the write in the background (from the client or other consistent replicas) and become consistent eventually. However, this approach would have required a client to read from multiple replicas to provide strong consistency for reads.

The extra reads would increase the system load and could degrade the read tail latency. Moreover, with fewer consistent replicas, consistency and durability could also become concerns, especially in a resource-harvesting datacenter where a service operator can inadvertently wipe any replica before background replication is complete. For these reasons, we discarded this approach.

Fail fast without replacement. To eliminate recreation of committed data, we considered just removing the slowest server from a pipeline without adding any replacement and let the file system recreate the missing replica in the background. This approach differs from eventually-consistent writes because all remaining replicas are in a consistent state, so a client can read the latest write from any replica. However, like eventually-consistent writes, durability was a concern with this approach because of the reduced number of replicas.

Cloning writes. We also considered writing to multiple pipelines at the same time and discarding those that complete the write late. We decided against this approach because it would generate a large amount of potentially unnecessary disk and metadata traffic.

Variable-sized blocks. We considered using variable-sized blocks to avoid recreating committed data. In this approach, a client, upon observing slow packet writes, could seal the current block and start writing to a new one with a new pipeline. Though this feature exists in some distributed file systems, we decided against it for two reasons: it could dramatically increase the metadata space requirement if premature sealing is frequent, and move the bottleneck from data writes to metadata accesses.

Centralized controller. Instead of making independent, local decisions about contention, the clients could access this information in a central repository maintained by the metadata manager. However, this approach would be highly dependent on the freshness of the load estimations. If the estimation window is too large, the manager would provide stale data and may not reduce tail latencies. On the other hand, a small window could provide accurate information but may overwhelm the manager. As others have observed in production [38], centralized components often harm scalability.

OS-based improvements. A design goal of many large-scale distributed systems, such as HDFS or Spark, is to provide portability across platforms. Thus, we focus on improving file system performance in the user space, rather than modifying the underlying software or requiring features that few operating systems provide.

5 Implementation in HDFS

We implement our techniques in HDFS and call the resulting system "CurtailHDFS". We create a new client library that applications can use. The library allows each technique to be enabled independently, which is key for assessing their benefits in production. Moreover, adoption can be incremental as users slowly deploy our client.

Our client keeps a configurable amount of the recent history of packet and block latencies. For fail fast, we use the existing pipeline latency reporting mechanism to piggyback each DN packet processing latency with each ack. On the client side, we determine the slowest DN and leverage the pipeline error-handling mechanism to perform a DN replacement. We enhance the error handling to select the fastest DN for reconstruction and use timeouts for the reconstruction process. For smart hedging, we extend HDFS's static hedging option [54, 55]. In both implementations, we avoid using any DN that has flagged itself as "busy" (a latency-sensitive service needs the server's resources). We discuss the impact of the techniques' parameters in Section 6.2.

Finally, our code is modular and easy to add/remove/-configure, as needed for maintainability in production.

6 Evaluation

In this section, we describe our experimental methodology and results. We conclude with a discussion of our experience bringing CurtailHDFS to production.

6.1 Methodology

Experimental setup. We evaluate CurtailHDFS in two environments: the 4k-server experimental testbed we describe in Section 3.2; and 3 large production deployments, each with 20k to 30k servers spread across 6 to 10 federated sub-clusters. These deployments use the same server configurations as the testbed, but exhibit more sharing with latency-sensitive services. The replication factor is 4 in the testbed and production deployments.

Evaluating our techniques in production deployments is difficult, because both the batch and latency-sensitive service loads are constantly changing. To do so accurately, we take an "A/B testing" approach where we run multiple techniques simultaneously. Specifically, we configure some clients to run the technique(s) we are evaluating while other clients run the baseline(s) for comparison. To simplify log processing, all clients on the same machine run the same technique(s). In addition, the deployments experience periods of relative inactivity that are uninteresting. Thus, we collect file system telemetry in 10-minute periods, and focus our evaluation on those periods that experience both more than 1000 full-block reads and 1000 full-block writes. These are the minimum number of accesses we need to compute 99.9^{th} percentiles for each access type. Table 2 lists the percentage of time these intervals represent.

Table 2. Characteristics of production deployments.

Metric	DC0	DC1	DC2
#servers	25.2k	32.4k	19.0k
#server configs	4	4	4
#local storage configs	3	3	3
%servers shared with services	66.5%	29.4%	28.7%
%time in 10-min intervals	29.2%	72.0%	36.0%
#reads	2.6M	43.3M	38.6M
#writes	2.3M	8.3M	9.0M
#compute frameworks	4	4	4

Workloads. We use either synthetic or production workloads. For our synthetic workload, we extend DistCP in three ways. First, we allow one of the sides of the copy to be main memory, so that we can either only read file data or only write file data. Second, we allow the setting of the number of tasks that should use each technique (*e.g.*, 1000 map tasks should read blocks using smart hedging and 1000 map tasks should use static hedging). Third, we instrument the code to collect key metrics from the client, such as latency and number of hedges. We call the resulting program ExtendedCP. We also implement a MapReduce job that creates a pre-defined amount of background file system load (*i.e.*, amount of HDFS open connections) in our testbed. The job monitors the load in the cluster and generates/terminates read/write operations to maintain a constant load over time. Moreover, we ensure that the distribution of load it generates across DNs is the same as in the production workloads.

In contrast, the production workloads are organic to our deployments and come from multiple engineering teams. They comprise various data analytics frameworks and applications. Spark and machine learning training, respectively, being the most common. The number of tasks in each job varies from a few to tens of thousands, whereas the tasks' lifespans vary from a few seconds to hours. Another common application is moving data between systems (*e.g.*, Cosmos to HDFS) for other applications (*e.g.*, training of machine learning models) to use. Section 6.3 details the patterns of these workloads. A large percentage ($> 90\%$) of the jobs access the distributed file system with our extended client, but others still use older clients.

Monitoring infrastructure. In Section 3.2, we describe our HTrace-based fine-grained monitoring infrastructure. The data provided by this infrastructure is extremely detailed, which leads to high collection, aggregation and processing times. So, it is only suitable for characterization and troubleshooting. Our synthetic and production workloads do not need this much detail. For our synthetic workloads, we use the metrics we collect from ExtendedCP. For the production workload, we extend the client to log per-block statistics (*i.e.*, total bytes, time of the operation, and throughput) for each read/write operation [7].

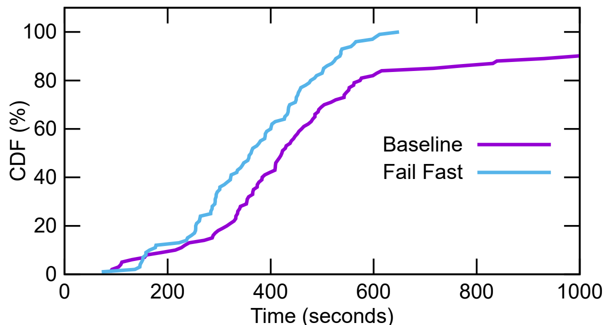


Figure 7. Latency of 1000 writes under heavy load.

We rely on the existing Hadoop infrastructure to collect and store these logs in a repository (in HDFS). To speed up our analysis process, we implement a MapReduce job that parses all application logs and gets statistics related to (1) the throughput of each access, (2) hedges (tried and successful), and (3) fail fast replacements. In production, we parse these logs, aggregate the data, and expose the statistics as performance counters.

6.2 Synthetic workload results

We first evaluate CurtailHDFS with our ExtendedCP job in the 4k-server experimental testbed.

Writes. We evaluate fail fast writes against our production HDFS baseline. We run experiments where an ExtendedCP job writes 1TB of data using 1000 clients (each client writes 1GB) in parallel. For comparison, we run 10 experiments without background file system load and 10 with background file system load produced by 5k clients (similar to background load in production). Each experiment produces a distribution of write times.

In the absence of background load, both fail fast and the baseline exhibit a median 99th-percentile write latency (*i.e.*, the median value of the 99th-percentile write latencies across the experiments) of 155 seconds, as there are no DN replacements. The benefits of fail fast become significant with background load. In this case, the medians of the 99th-percentile write latencies are 2826 seconds for the baseline, but only 751 seconds for fail fast. Figure 7 shows one of these experiments, where the 99th-percentile write latency for the baseline is more than 22 minutes whereas that for fail fast is only 611 seconds.

Writes sensitivity analysis. There are three parameters for fail fast: (1) the latency percentile that we use to compare server performance; (2) the multiplicative factor by which a server needs to be slower than others to be considered slow; and (3) the number of recent requests we consider in making the replacement decision. Lower percentiles increase the tail latency when writing; according to our exploration of this parameter, the 95th percentile produces the best trade-off: low latency with small number of DN replacements. Similarly, we find the best slowness factor to be a difference of 3× between

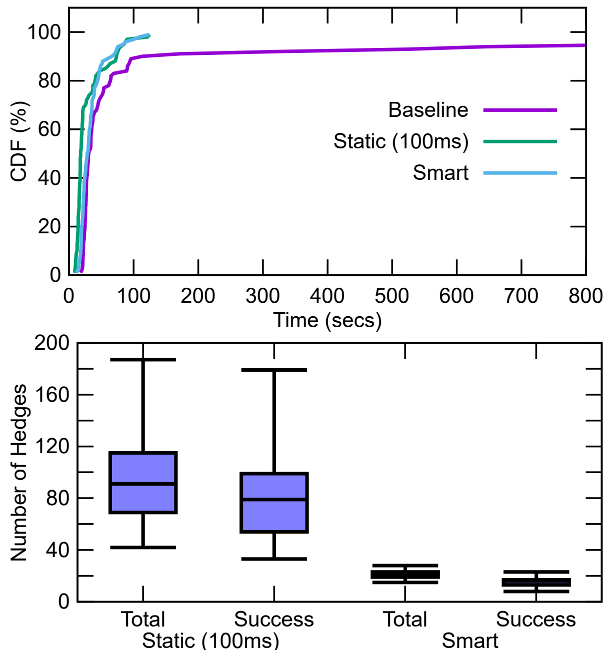


Figure 8. Latency (top) and hedges (bottom) of 1000 reads under heavy load.

the fastest and slowest DNs. The number of requests required for a replacement decision has a strong impact. Small values can produce high overhead and include too few samples of actual disk accesses (recall that writes are buffered), whereas large values may take too long to produce a decision. Our sensitivity analysis for this parameter shows that 100 samples provides enough history for informed reactions.

Reads. We now evaluate smart hedging against the baseline and the baseline with static hedging (100ms threshold), using the same setup as above but reading 1GB files. Without background file system load, the medians of the 99th-percentile read latencies are 36 seconds for the baseline, 22 seconds for static hedging, and 32 seconds for smart hedging. The static version triggers a median of 56 hedges whereas the smart approach uses only 22. This trade-off becomes starker as the load increases. We run 10 more experiments with 30k clients generating high background load by reading from HDFS. (Reads are less expensive than writes, so they need more clients for the same amount of load.) In this case, the medians of the 99th-percentile latencies are more than 22 minutes for the baseline, 81 seconds for static hedging, and 89 seconds for smart hedging. In addition, static hedging triggers a median of 97 hedges whereas smart triggers only 21. Figure 8 shows the latency (top) and the hedging (bottom) statistics for one of these experiments. In the hedging graph, the vertical ranges go from the minimum to the maximum number of total and successful hedges, whereas the blue boxes range from the 25th to the 75th percentiles. The horizontal line across the boxes is the median value. The figure shows that smart hedging is effective at lowering the latency with a much lower number of hedges. Overall, the tail latency for static and smart hedging is similar at higher loads, while the number of hedges is much smaller under smart hedging.

Static hedging depends on a single parameter: the time to wait for a read before triggering a hedge. This threshold can be too conservative thereby offering little benefit, or too aggressive thereby increasing the load on the system. For example, we have observed scenarios where, under heavy load, the common-case read latency was greater than the timeout in the static technique. In such cases, the clients kept bouncing between replicas.

In general, we observe that static hedging thresholds longer than 100ms degrade latency, especially under low background load. However, in a production environment with many co-located latency-sensitive services, it makes sense to use a higher value (*e.g.*, 500ms as in our deployments) to reduce the amount of hedging load. Smart hedging does not depend strongly on this parameter as it only uses the static value for bootstrapping.

Reads sensitivity analysis. Smart hedging has the same three parameters as fail fast, as well as the retry policy. Higher percentiles and higher slowness factors do not impact latency significantly, but do reduce the number of hedges. As for writes, the best trade-off is to use the 95th percentile and a factor of 3×. The number of recent requests to consider does not have a significant impact and medium numbers (*e.g.*, 20 samples) provide a good trade-off. Finally, the retry policy has a large impact on both latency and number of hedges. The best policy is exponential with 3 retries, which provides a latency comparable to static hedging with fewer hedges. Thus, it provides low latency without excessively increasing the system load. In contrast, a highly restrictive retry policy (*e.g.*, never retrying a failed hedge) reduces significantly the number of hedges, but latency increases substantially.

Reads and writes. We now evaluate the combined effect of our read and write techniques by running ExtendedCP with the same settings as above. In the absence of background load, the medians of the 99th-percentile latencies are 376 seconds for the baseline, 194 seconds for smart hedging, 227 seconds for fail fast, and 189 seconds when using both techniques. In this setup, smart hedging reduces tail latency the most, and combining the techniques produces 2× lower tail latency than the baseline system. In contrast, under high background load, fail fast is more much beneficial than smart hedging. We increase the load by having 10k clients generate reads and writes. In this case, the medians of the 99th-percentile latencies are 5708 seconds for the baseline, 4994 seconds for smart hedging, 685 seconds for fail fast, and 297 seconds when

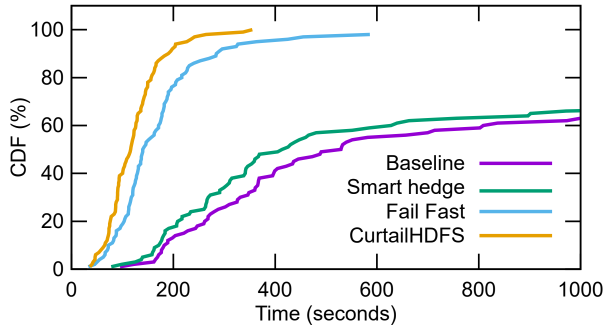


Figure 9. Latency of 1000 copies under heavy load.

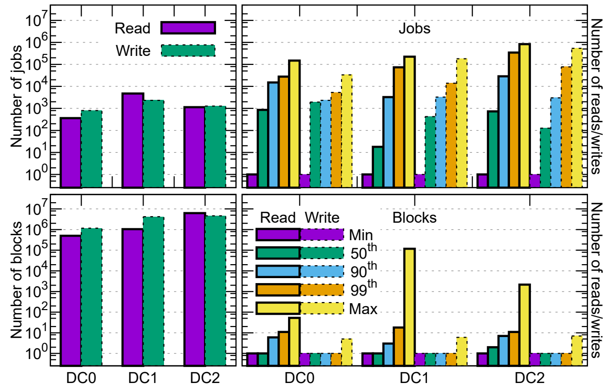


Figure 10. Characteristics of production workloads. Note the same log scales in the left and right Y-axes.

the two techniques are combined. Under these settings, *CurtailHDFS reduces the median tail latency by* 19×. Figure 9 shows one of these experiments. Overall, the combination of both techniques in CurtailHDFS substantially reduces tail latency, under various load conditions.

6.3 Production results

Table 2 lists the main characteristics of our production deployments and workloads for one month. The workload data includes only the 10-minute intervals for which we report results. Recall that our evaluation focuses on time intervals with enough file system load to meaningfully compute 99.9^{th}-percentile latencies. Most rows in the table are self-explanatory. The three local storage configurations are SSD RAID only, HDD RAID only, and both SSD and HDD RAIDs. The "%time in 10-min intervals" row lists the percentage of the month that we are reporting about in the 10-minute intervals. The four compute frameworks include Spark and MapReduce. As we can see from the table, our deployments are large and heterogeneous in terms of hardware and software.

Figure 10 illustrates the characteristics of the production workloads (again only accounting for the 10-minute intervals) in each deployment. The graphs on the left show the numbers of jobs (top) and blocks accessed (bottom) in log scale. We split the job data into jobs with read accesses and write accesses; the two sets overlap, but not every job reads data and not every job writes data in our intervals. The graphs on the right show the distribution of reads and writes across jobs (top) and blocks (bottom) in log scale. For example, the leftmost bar in the top right graph shows the number of reads that the job with the minimum number of reads performed in DC0. Similarly, the leftmost bar in the bottom right shows the minimum number of reads that a block received in DC0. On both sides, the bars with solid and dashed contours represent reads and writes, respectively.

Unlike our synthetic jobs, the figure shows significant skew in the distributions of the number of accesses across jobs, and even greater skew in the distribution of the popularity of blocks for reading. Blocks only get written more than once when the first write does not write the entire block. Our intervals include any access to 256MB or more, but in some cases users define their blocks to be larger than 256MB. That is why the maximum number of writes per block is more than 2 in our deployments.

Although we do not illustrate this, the production workloads are more balanced than our ExtendedCP job, using much more compute cycles per file system access.

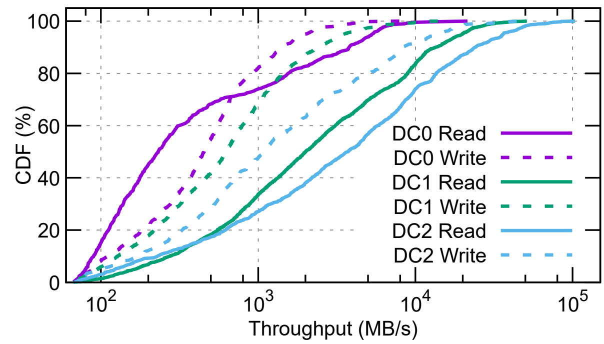


Figure 11. Throughput in the production deployments.

Figure 11 shows the aggregate client write and read access throughputs (in MB/second) in our intervals. In all deployments, throughputs vary significantly (up to 3 orders of magnitude) over time, and read throughput tends to be higher than write throughputs.

In the context of these deployments, Figure 12 summarizes the latency results in *log scale* for writes (top) and reads (bottom) across many percentiles; the leftmost bars show the average results. The write bars compare fail fast with regular writes in HDFS, whereas the read bars compare smart hedging against static hedging (we cannot use regular reads in production, as users are already accustomed to the lower tail latency of hedging). The height of the bars is the average value for the percentile/average across the 10-minute intervals, whereas the vertical ranges go from the 5th to the 95th percentiles of the distributions of the 10-minute intervals.

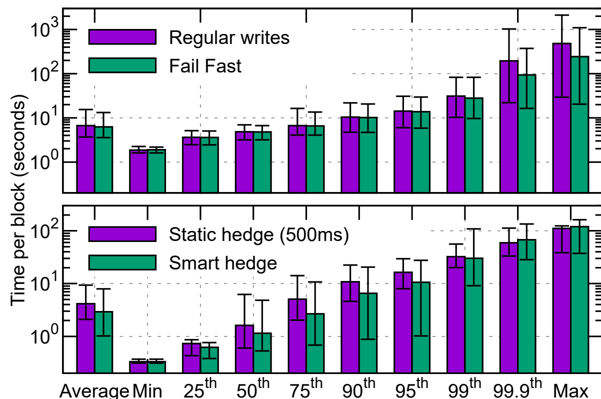


Figure 12. Summary of production results for a month. Note the log scales in the Y-axes.

The results show that fail fast significantly reduces latency in the highest percentiles, 99.9^{th} and max. *At those percentiles, the reduction is* $\sim 2\times$ *on average and the entire distributions improve.* In contrast, smart hedging reduces latency in the middle percentiles (25th to 95th), as we are comparing it to static hedging configured for low traffic with a threshold of 500ms. (As we mention in Section 6.4, the initial setting of 100ms for static hedging was generating too much traffic in production, so it had to be increased to 500ms.) For these percentiles, the improvements to the low end of the distributions (bottom end of the vertical ranges) is particularly pronounced. *On average, smart hedging reduces latency by* $1.4\times$ *compared to static hedging.*

These improvements are remarkable for two main reasons: (1) our baselines already include techniques that help mitigate tail latencies (*e.g.*, speculative task execution, static hedging, servers warn clients and metadata managers when they are busy); and (2) they embody all challenges of production systems (*e.g.*, need to be simple, high heterogeneity, extreme scale, real workloads, interference from services).

To understand these results at a deeper level, we next detail the impact of our techniques for writes and reads.

Writes. Figure 13 shows the distribution of the 99.9^{th}-percentile latencies *in log scale* (left) and of the average number of fail fast operations per request (right), across the intervals. The left figure shows improvements across nearly all intervals. The right figure shows that the average number of fail fast operations per block is very small for most requests. Across all intervals, the average is 0.15 and the worst case is 5 operations per block.

Reads. Figure 14 shows the distribution of the average read latencies (left) and the distribution of the average number of hedges per block request (right), across the 10-minute intervals. Clearly, the average read latency is lower in nearly all intervals. Hence, smart hedging reduces tail latencies without negatively impacting the average read latency.

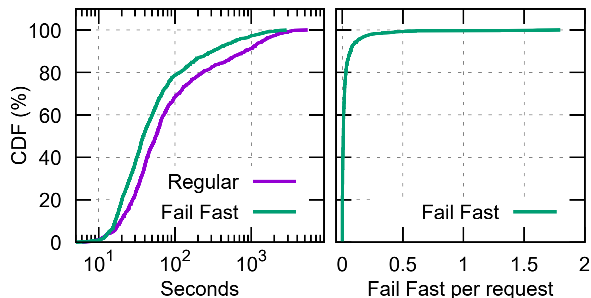


Figure 13. 99.9^{th}-percentile write latency in log scale (left) and average fail fast operations per block (right).

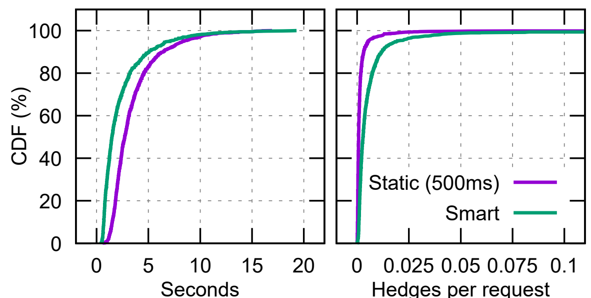


Figure 14. Average read latency (left) and number of hedges per block (right).

Figure 14 also shows a side-effect of smart hedging: slightly more hedges per request on average compared to static hedging. The static hedging threshold in production deployments is 500ms, up from the 100ms threshold in the 4k-server testbed (Section 6.2). A higher threshold in production reduces the number of hedges, which is especially important under heavy load (Section 6.4). In contrast, smart hedging uses the 500ms threshold only for bootstrapping and later adapts the threshold based on observed read latencies. This approach reduces tail latencies, but can also cause more hedges in scenarios where the load across replicas is non-uniform (*e.g.*, high for a few and low across the remaining replicas).

Coming full circle. We motivated the paper by showing that I/O-bound jobs using our production HDFS (plus static hedging) exhibit large performance variations in one of our production deployments (Figure 1). Figure 15 shows the same data, and CurtailHDFS results for these jobs in the same deployment; speculative task execution is enabled in jobs on both systems. CurtailHDFS significantly improves performance across most of the spectrum, but especially in the high percentiles (by $\sim 3\times$).

Unfortunately, we cannot perform the same analysis with production jobs, as we do not know whether two executions of a job use the same data (we expect that they do not) or even the same code.

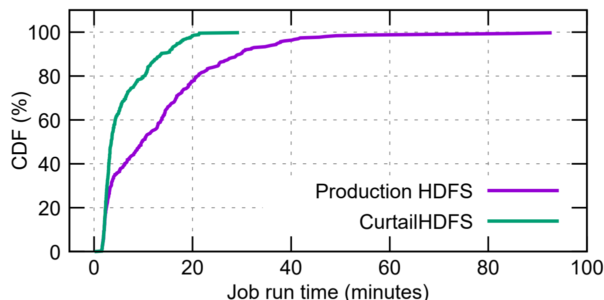


Figure 15. CDF of copy job run times over 2 weeks.

6.4 Experiences in production

Our experience taking CurtailHDFS from the 4k-server testbed to production illustrates issues that are often overlooked in the literature. We first experimented extensively with I/O-bound jobs in the testbed, always observing significant improvements from our techniques.

In February 2018, we brought fail fast to production in one datacenter, but the initial results were disappointing: write latency did not improve in the common-case load and degraded significantly under heavy load. So, we quickly turned it off. After some investigation, we realized that the larger percentage of servers shared with latency-sensitive services in our production deployments (Table 2) was causing much more throttling than in the testbed. We also realized that the write latency reporting was not detailed enough for the client to identify the throttling cases. With the change in reporting, we deployed the version we study in this paper in March 2018.

In June 2017, we deployed static hedging in production in one datacenter with a threshold of 100ms. Read latency improved significantly and we enabled it in all deployments after 3 months. However, after a few months, in November 2017, we started to observe periods when some servers were becoming overloaded. After some investigation, we realized that this behavior was caused by the high number of hedges under high load. To mitigate this issue, we increased the threshold to 500ms. This significantly reduced the load, but it also reduced the effectiveness of the technique. As a result, tail latencies were still reduced significantly, but the average latency increased. Clearly, static hedging is heavily dependent on its threshold and load. So, we introduced smart hedging in February 2018 to reduce read tail latencies, without negatively impacting the average latency.

We concluded the A/B testing in June 2018 and enabled our two tail-mitigation techniques by default in the modified HDFS client. The techniques have improved the user experience significantly and many users have already switched to our client.

7 Related work

To our knowledge, no prior work has addressed all the practical challenges of production systems (Section 2). Nevertheless, our work is related to the following efforts.

Tail latency in distributed file/storage systems. Most prior works have assumed that writes to persistent storage occur in the background (buffered writes), so write tail latency has not received much attention. An exception is CosTLO [53], which issues redundant writes (and reads) in key-value stores. CosTLO's approach is similar to the cloning write pipelines technique (Section 4.4) that we considered but discarded. We opted for fail fast, because it operates at a finer granularity (slow server replacement in a pipeline). In addition, fail fast is much simpler to implement as it leverages existing replication and fault-tolerance mechanisms.

In contrast, several works addressed read tail latencies [16, 40, 44, 46, 54, 55]. C3 [44] proposes a replica selection algorithm that uses server-side information and rate limits. However, C3 is not scalable as it needs to store server load data on ephemeral clients. Rein [40] targets key-value stores and schedules concurrent reads of multiple keys. CRAQ [46] improves read throughput and latency under chain replication by enabling reading from any replica in the chain, while still providing strong consistency. Smart hedging is orthogonal to CRAQ and can reduce its read tail latencies. Other works propose techniques that run at pre-defined intervals. For example, Cassandra uses dynamic snitching [16], which periodically ranks servers based on observed latencies and server-reported loads to select the best replica for a read. HDFS's static hedging starts a hedge request after a threshold amount of time. Unfortunately, these techniques cannot tackle dynamic heterogeneity since the interval/threshold might be inappropriate at run time. In smart hedging, each client determines when to react at run time, and uses back-off to prevent overload.

Earlier works [25, 32, 42, 47, 51, 60, 62] manage resources to meet tail latency SLOs and ensure fair sharing. They are orthogonal to our techniques, and often rely on centralized components and/or complex modeling.

Tail latency in local file systems. Other works have characterized storage latencies (*e.g.*, [15, 27]), and proposed techniques for cutting latency tails (*e.g.*, [28]). Such techniques are complementary to our work.

Tail latency in data analytics frameworks. Mantri [10], LATE [59], Dolly [9], and PBSE [43] improve running times via SE. Our techniques handle storage-level bottlenecks independently from computation, while avoiding greater compute-storage coupling.

8 Conclusion

We introduced techniques for managing tail latencies in production distributed file systems, implemented them in a popular system, and evaluated them extensively. Our results demonstrate large latency improvements with I/O-bound jobs, and smaller improvements with more balanced production workloads. The results also illustrate that it is easy to overlook important effects in non-production systems. We conclude that it is possible to devise effective techniques while considering the challenges and constraints of real datacenters.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Kang Chen, for their comments on our paper. We also thank Virajith Jalaparti, Bianca Schroeder, Sameh Elnikety, Jonathan Mace and Marco Canini for their comments on an earlier draft. We are indebted to Lukas Majercak for his help in deploying and testing in production. This work was supported in part by the National Science Foundation (CNS-1616947).

References

[1] HDFS Architecture Guide, 2008. http://hadoop.apache.org/docs/current/hdfs_design.html.

[2] HDFS Architecture Guide, 2008. https://docs.mongodb.com/manual/tutorial/manage-chained-replication/.

[3] Apache HTrace: A tracing framework for use with distributed systems, 2017. http://htrace.incubator.apache.org/.

[4] DistCp Guide, 2017. http://hadoop.apache.org/docs/current/hadoop-distcp/DistCp.html.

[5] TeraGen, 2017. https://hadoop.apache.org/docs/current/api/org/apache/hadoop/examples/teragen/package-summary.html.

[6] Track time to process packet in Datanode, 2017. https://issues.apache.org/jira/browse/HDFS-13053.

[7] Track speed in DFSClient, 2018. https://issues.apache.org/jira/browse/HDFS-12861.

[8] Amvrosiadis, G., Park, J. W., Ganger, G. R., Gibson, G. A., Baseman, E., and DeBardeleben, N. On the diversity of cluster workloads and its impact on research results. In *USENIX ATC* (2018).

[9] Ananthanarayanan, G., Ghodsi, A., Shenker, S., and Stoica, I. Effective Straggler Mitigation: Attack of the Clones. In *NSDI* (2013).

[10] Ananthanarayanan, G., Kandula, S., Greenberg, A., Stoica, I., Lu, Y., Saha, B., and Harris, E. Reining in the Outliers in Map-reduce Clusters Using Mantri. In *OSDI* (2010).

[11] Andersen, D. G., Franklin, J., Kaminsky, M., Phanishayee, A., Tan, L., and Vasudevan, V. Fawn: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP '09, ACM, pp. 1–14.

[12] Apache Software Foundation. Apache Spark.

[13] Balakrishnan, M., Malkhi, D., Prabhakaran, V., Wobbler, T., Wei, M., and Davis, J. D. CORFU: A shared log design for flash clusters. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (San Jose, CA, 2012), USENIX, pp. 1–14.

[14] Calder, B., Wang, J., Ogus, A., Nilakantan, N., Skjolsvold, A., McKelvie, S., Xu, Y., Srivastav, S., Wu, J., Simitci, H., Haridas, J., Uddaraju, C., Khatri, H., Edwards, A., Bedekar, V., Mainali, S., Abbasi, R., Agarwal, A., Haq, M. F. u., Haq, M. I. u., Bhardwaj, D., Dayanand, S., Adusumilli, A., McNett, M., Sankaran, S., Manivannan, K., and Rigas, L. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 143–157.

[15] Cao, Z., Tarasov, V., Raman, H. P., Hildebrand, D., and Zadok, E. On the Performance Variation in Modern Storage Stacks. In *FAST* (2017).

[16] Carpenter, J., and Hewitt, E. *Cassandra: The Definitive Guide: Distributed Data at Web Scale*. " O'Reilly Media, Inc.", 2016.

[17] Chaiken, R., Jenkins, B., Larson, P.-Å., Ramsey, B., Shakib, D., Weaver, S., and Zhou, J. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *Proceedings of the VLDB Endowment 1*, 2 (2008).

[18] Dean, J., and Barroso, L. A. The Tail at Scale. *Communications of the ACM 56*, 2 (2013).

[19] Dean, J., and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI* (2004).

[20] Delimitrou, C., and Kozyrakis, C. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *ASPLOS* (2014).

[21] Dinu, F., and Ng, T. Understanding the Effects and Implications of Compute Node Related Failures in Hadoop. In *HPDC* (2012).

[22] Escriva, R., Wong, B., and Sirer, E. G. Hyperdex: A distributed, searchable key-value store. *SIGCOMM Comput. Commun. Rev. 42*, 4 (Aug. 2012), 25–36.

[23] Ghemawat, S., Gobioff, H., and Leung, S.-T. The Google File System. In *SOSP* (2003).

[24] Goder, A., Spiridonov, A., and Wang, Y. Bistro: Scheduling Data-Parallel Jobs Against Live Production Systems. In *USENIX ATC* (2015).

[25] Gulati, A., Ahmad, I., and Waldspurger, C. A. PARDA: Proportional Allocation of Resources for Distributed Storage Access. In *FAST* (2009).

[26] Hao, M., Li, H., Tong, M. H., Pakha, C., Suminto, R. O., Stuardo, C. A., Chien, A. A., and Gunawi, H. S. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *SOSP* (2017).

[27] Hao, M., Soundararajan, G., Kenchammana-Hosekote, D. R., Chien, A. A., and Gunawi, H. S. The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments. In *FAST* (2016).

[28] He, J., Nguyen, D., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Reducing File System Tail Latencies with Chopper. In *FAST* (2015).

[29] Iorgulescu, C., Azimi, R., Kwon, Y., Elnikety, S., Syamala, M., Narasayya, V., Herodotou, H., Tomita, P., Chen, A., Zhang, J., et al. Perfiso: Performance isolation for commercial latency-sensitive services. In *USENIX ATC* (2018).

[30] Isard, M. Autopilot: Automatic Data Center Management. *ACM SIGOPS Operating Systems Review 41*, 2 (2007).

[31] Jalaparti, V., Bodik, P., Kandula, S., Menache, I., Rybalkin, M., and Yan, C. Speeding up Distributed Request-Response Workflows. In *SIGCOMM* (2013).

[32] Jin, W., Chase, J. S., and Kaur, J. Interposed Proportional Sharing for a Storage Service Utility. In *SIGMETRICS* (2004).

[33] Jyothi, S. A., Curino, C., Menache, I., Narayanamurthy, S. M., Tumanov, A., Yaniv, J., Mavlyutov, R., Goiri, Í., Krishnan, S., Kulkarni, J., et al. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *OSDI* (2016).

[34] Li, H., Ghodsi, A., Zaharia, M., Shenker, S., and Stoica, I. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *SoCC* (2014).

[35] Li, J., Sharma, N. K., Ports, D. R., and Gribble, S. D. Tales of the Tail: Hardware, OS, and Application-Level Sources of Tail Latency. In *SoCC* (2014).

[36] Lo, D., Cheng, L., Govindaraju, R., Ranganathan, P., and Kozyrakis, C. Heracles: Improving Resource Efficiency at Scale. In *ISCA* (2015).

[37] Mace, J., Bodik, P., Fonseca, R., and Musuvathi, M. Retro: Targeted Resource Management in Multi-tenant Distributed Systems. In *NSDI* (2015).

[38] Misra, P., Goiri, I., Kace, J., and Bianchini, R. Scaling Distributed File Systems in Resource-Harvesting Datacenters. In *USENIX ATC* (2017).

[39] Ouyang, X., Garraghan, P., Primas, B., Mckee, D., Townend, P., and Xu, J. Adaptive speculation for efficient internetware application execution in clouds. *ACM Trans. Internet Technol. 18*, 2 (Jan. 2018), 15:1–15:22.

[40] Reda, W., Canini, M., Suresh, L., Kostić, D., and Braithwaite, S. Rein: Taming Tail Latency in Key-Value Stores via Multiget Scheduling. In *EuroSys* (2017).

[41] Ren, X., Ananthanarayanan, G., Wierman, A., and Yu, M. Hopper: Decentralized speculation-aware cluster scheduling at scale. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 379–392.

[42] Shue, D., Freedman, M. J., and Shaikh, A. Performance Isolation and Fairness for Multi-tenant Cloud Storage. In *OSDI* (2012).

[43] Suminto, R. O., Stuardo, C. A., Clark, A., Ke, H., Leesatapornwongsa, T., Fu, B., Kurniawan, D. H., Martin, V., Uma, M. R. G., and Gunawi, H. S. Pbse: A robust path-based speculative execution for degraded-network tail tolerance in data-parallel frameworks. In *Proceedings of the 2017 Symposium on Cloud Computing* (New York, NY, USA, 2017), SoCC '17, ACM, pp. 295–308.

[44] Suresh, P. L., Canini, M., Schmid, S., and Feldmann, A. C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection. In *NSDI* (2015).

[45] Tang, S., Lee, B., and He, B. Dynamicmr: A dynamic slot allocation optimization framework for mapreduce clusters. *IEEE Transactions on Cloud Computing 2*, 3 (July 2014), 333–347.

[46] Terrace, J., and Freedman, M. J. Object storage on craq: High-throughput chain replication for read-mostly workloads. In *USENIX Annual Technical Conference* (2009), San Diego, CA.

[47] Trushkowsky, B., Bodík, P., Fox, A., Franklin, M. J., Jordan, M. I., and Patterson, D. A. The SCADS Director: Scaling a Distributed Storage System Under Stringent Performance Requirements. In *FAST* (2011).

[48] van Renesse, R., and Schneider, F. B. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6* (Berkeley, CA, USA, 2004), OSDI'04, USENIX Association, pp. 7–7.

[49] Veeraraghavan, K., Meza, J., Chou, D., Kim, W., Margulis, S., Michelson, S., Nishtala, R., Obenshain, D., Perelman, D., and Song, Y. J. Kraken: Leveraging Live Traffic Tests to Identify and Resolve Resource Utilization Bottlenecks in Large Scale Web Services. In *OSDI* (2016).

[50] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., and Wilkes, J. Large-Scale Cluster Management at Google with Borg. In *EuroSys* (2015).

[51] Wang, A., Venkataraman, S., Alspaugh, S., Katz, R., and Stoica, I. Cake: Enabling High-Level SLOs on Shared Storage Systems. In *SoCC* (2012).

[52] Wu, Z., Butkiewicz, M., Perkins, D., Katz-Bassett, E., and Madhyastha, H. V. SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services. In *SOSP* (2013).

[53] Wu, Z., Yu, C., and Madhyastha, H. V. CosTLO: Cost-Effective Redundancy for Lower Latency Variance on Cloud Storage Services. In *NSDI* (2015).

[54] Xe, L. Support Hedged Reads in DFSClient, 2014. https://issues.apache.org/jira/browse/HDFS-5776.

[55] Xe, L., and McCabe, C. P. Support Non-Positional Hedged Reads in HDFS, 2017. https://issues.apache.org/jira/browse/HDFS-6450.

[56] Yang, H., Breslow, A., Mars, J., and Tang, L. Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In *ISCA* (2013).

[57] Yang, S., Harter, T., Agrawal, N., Kowsalya, S. S., Krishnamurthy, A., Al-Kiswany, S., Kaushik, R. T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Split-level i/o scheduling. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), ACM, pp. 474–489.

[58] Yang, Y., Kim, G.-W., Song, W. W., Lee, Y., Chung, A., Qian, Z., Cho, B., and Chun, B.-G. Pado: A Data Processing Engine for Harnessing Transient Resources in Datacenters. In *EuroSys* (2017).

[59] Zaharia, M., Konwinski, A., Joseph, A. D., Katz, R. H., and Stoica, I. Improving MapReduce Performance in Heterogeneous Environments. In *OSDI* (2008).

[60] Zhang, J., Riska, A., Sivasubramaniam, A., Wang, Q., and Riedel, E. Storage Performance Virtualization via Throughput and Latency Control. In *MASCOTS* (2005).

[61] Zhang, Y., Prekas, G., Fumarola, G. M., Fontoura, M., Goiri, I., and Bianchini, R. History-Based Harvesting of Spare Cycles and Storage in Large-Scale Datacenters. In *OSDI* (2016).

[62] Zhu, T., Tumanov, A., Kozuch, M. A., Harchol-Balter, M., and Ganger, G. R. PriorityMeister: Tail Latency QoS for Shared Networked Storage. In *SoCC* (2014).