Leveraging Endpoint Flexibility in Data-Intensive Clusters

Mosharaf Chowdhury[1], Srikanth Kandula[2], Ion Stoica[1]
[1]UC Berkeley, [2]Microsoft Research
{mosharaf, istoica}@cs.berkeley.edu, srikanth@microsoft.com

ABSTRACT

Many applications do not constrain the destinations of their network transfers. New opportunities emerge when such transfers contribute a large amount of network bytes. By choosing the endpoints to avoid congested links, completion times of these transfers as well as that of others without similar flexibility can be improved. In this paper, we focus on leveraging the flexibility in replica placement during writes to cluster file systems (CFSes), which account for *almost half* of all cross-rack traffic in data-intensive clusters. The replicas of a CFS write can be placed in any subset of machines as long as they are in multiple fault domains and ensure a balanced use of storage throughout the cluster.

We study CFS interactions with the cluster network, analyze optimizations for replica placement, and propose Sinbad – a system that identifies imbalance and adapts replica destinations to navigate around congested links. Experiments on EC2 and trace-driven simulations show that block writes complete $1.3\times$ (respectively, $1.58\times$) faster as the network becomes more balanced. As a collateral benefit, end-to-end completion times of data-intensive jobs improve as well. Sinbad does so with little impact on the long-term storage balance.

Categories and Subject Descriptors

C.2 [**Computer-communication networks**]: Distributed systems—*Cloud computing*

Keywords

Cluster file systems, data-intensive applications, datacenter networks, constrained anycast, replica placement

1 Introduction

The network remains a bottleneck in data-intensive clusters, as evidenced by the continued focus on static optimizations [7, 31] and data-local task schedulers [10, 34, 43, 44] that reduce network usage, and on scheduling the exchanges of intermediate data [10, 20]. The endpoints of a flow are assumed to be fixed: network schedulers [8, 13, 20, 28] can choose between different paths, vary rates of flows, and prioritize one flow over another, but they cannot change where a flow originates from or its destination.

However, many inter-machine transfers do not require their destinations to be in specific locations as long as certain constraints are satisfied. An example of such transfers in data-intensive clusters is the writes to cluster file systems (CFSes) like GFS [27], HDFS [15], Cosmos [19], Amazon S3 [2], or Windows Azure Storage [17]. These systems store large pieces of data by dividing them into fixed-size blocks, and then transferring each block to three machines (for fault-tolerance) in two different racks (for partition-tolerance) using chain replication [42]. The replicas can be in any subset of machines that satisfy the constraints.

Analysis of traces from production clusters at Facebook and Microsoft reveals that such replicated writes (referred to as *distributed writes* from hereon) account for *almost half of all cross-rack traffic* in both clusters (§4). Moreover, recent data suggests a growing trend in the volume of distributed writes with hundreds of terabytes of data being ingested everyday into different CFS installations [5, 6, 17]. We also find that while the network is often underutilized, there exists substantial imbalance in the usage of bottleneck links. Such imbalance leads to congestion and performance degradation. Common causes of imbalance include skew in application communication patterns[1] [14, 35] and a lack of control on the background traffic in multi-tenant datacenters.

Even though replication takes place over the network, its interactions with the network have so far been ignored. In this paper, we present Sinbad, a system that leverages the flexibility in endpoint placement during distributed writes to steer replication transfers away from network hotspots. Network-balanced placement has two implications. First, it improves CFS write throughput by avoiding network contention; response times of tasks that write improve as well. Second, by steering CFS' traffic away from hotspots, throughput of non-CFS traffic on those links increase; in data-intensive clusters, this speeds up tasks that shuffle intermediate data and jobs with large shuffles.

Exercising the freedom in endpoint selection in our approach is akin to that of overlay anycast, which is typically employed by latency-sensitive request-response applications to select the "best" source (server) from where to retrieve the content [18, 25]. However, in the case of distributed writes, we exploit this flexibility for picking the "best" set of destinations to maximize the throughput of replicating large blocks. In addition, we consider constraints like the number of fault domains and aim to minimize the storage imbalance across machines.

Storage imbalance during replica placement is harmful, because machines receiving too many replicas can become hotspots for future tasks. Existing CFSes employ a uniform replica placement

[1]For example, when counting the occurrences of DISTINCT keys in a dataset, the amount of data in each partition (to be received by corresponding reducer) can be very skewed.

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 or the author must be honored. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGCOMM'13, August 12–16, 2013, Hong Kong, China.
Copyright 2013 ACM 978-1-4503-2056-6/13/08 ...$15.00.

policy and perform periodic re-balancing to avoid such imbalance. We show that a network-balanced placement policy does not trigger additional storage balancing cycles. While network hotspots are stable in the short term to allow network-balanced placement decisions, they are uniformly distributed across all bottleneck links in the long term ensuring storage load balancing.

Optimizing distributed writes is NP-hard even in the offline case, because finding the optimal solution is akin to optimally scheduling tasks in heterogeneous parallel machines without preemption [9, 26]. We show that if hotspots are stable while a block is being written and all blocks have the same size, greedy placement through the least-loaded bottleneck link is optimal for optimizing the average block write time (§5). Under the same assumptions, we also show that to optimize the average file write time, files with the least remaining blocks should be prioritized.

Sinbad employs the proposed algorithms and enforces necessary constraints to make network-aware replica placement decisions (§6). It periodically measures the network and reacts to the imbalance in the non-CFS traffic. An application layer measurement-based predictor performs reasonably well in practice due to short-term (few tens of seconds) stability and long-term (hours) uniformness of network hotspots. We find this approach attractive because it is not tied to any networking technology, which makes it readily deployable in public clouds.

We have implemented Sinbad as a pluggable replica placement policy for the Facebook-optimized HDFS distribution [4]. HDFS is a popular open-source CFS, and it is the common substrate behind many data-parallel infrastructures [3, 32, 45]. We avoid the many known performance problems in Hadoop [3] by running jobs using an in-memory compute engine (e.g., Spark [45]). We have evaluated Sinbad (§7) by replaying the scaled-down workload from a Facebook trace on a 100-machine Sinbad deployment on Amazon EC2 [1]. We show that Sinbad improves the average block write time by $1.3\times$ and the average end-to-end completion time of jobs by up to $1.26\times$ with limited penalties due to its online decisions. In the process, it decreases the imbalance across the network with little impact on storage load balancing. For in-memory storage systems, the improvements can be even higher. Through trace-driven simulations, we also show that Sinbad's improvement ($1.58\times$) is close to that of an optimistic estimation ($1.89\times$) of the optimal.

We discuss known issues and possible solutions in Section 8, and we consider Sinbad in light of relevant pieces of work in Section 9.

2 CFS Background

This section provides a brief architectural overview of cluster file systems (CFSes) focusing primarily on the end-to-end write pipeline. Examples of CFSes include distributed file systems (DFS) like GFS at Google [27], HDFS at Facebook and Yahoo! [4, 15], and Cosmos [19] at Bing. We also include public cloud-based storage systems like Amazon S3 [2] and Windows Azure Storage (WAS) [17] that have similar architecture and characteristics, and are extensively used by popular services like `dropbox.com`.

2.1 System Model

A typical CFS deployment consists of a set of storage slaves and a master that coordinates the writes to (reads from) CFS slaves. Files (aka objects/blobs) stored in a CFS are collections of large blocks. Block size in production clusters varies from 64 MB to 1 GB.[2] The block size demonstrates a trade-off between disk I/O throughput vs. the benefit from parallelizing across many disks. Most CFS designs provide *failure recovery guarantees* for stored files through replication and ensure *strong consistency among the replicas*.

[2]Blocks are not padded, i.e., the last block in a file can be smaller.

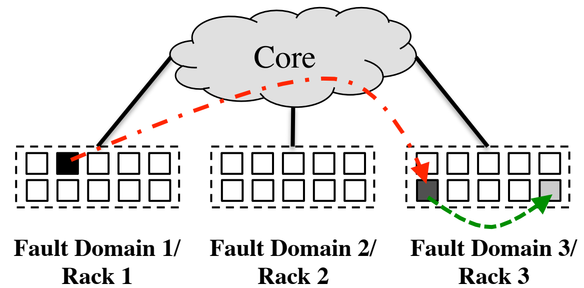


Figure 1: Distributed write pipeline. Each block has three copies in two racks and three different machines.

Write Workflow When writing a file to the CFS, the client provides a *replication* (r) factor and a *fault-tolerance* (f) factor to ensure that each block of that file has r copies located in at least $f(< r)$ fault domains. The former is for load balancing (blocks in popular files have more replicas [11]), while the latter ensures availability in spite of failures. Machines in different racks are typically considered to be in independent fault domains. Typically, $r = 3$ and $f = 1$; meaning, each block is stored in three machines in two racks and can survive at most one rack failure (Figure 1). Thus, writing a block copies it at least once across racks.

The replica placement policy in the CFS master independently decides where to place each block irrespective of their parent files. Blocks from the same file and their replicas need not be collocated. The goal is to *uniformly* place blocks across all machines and fault domains so as to

- minimize the imbalance in storage load across disks, and
- balance the number of outstanding writes per disk.

Both these constraints assume that writes are bottlenecked only by disks. This assumption, however, is not always true since the extent of oversubscription in modern datacenter networks (typically between the core and racks) can cause writes to bottleneck on the oversubscribed links. Even on topologies with full bisection bandwidth, writes can bottleneck on the servers' network interfaces for high in-degrees or when the cumulative write throughput of a server is larger than its NIC speed. For example, a typical server with six to eight commodity disks [33, 41] has sequential write throughput that is several times the typical NIC speed (1 Gbps).

Once replica locations have been determined, the CFS slave transfers the block to the selected destinations using chain replication [42]. Distributed writes are synchronous; to provide strong consistency, the originator task will have to wait until the last replica of the last block has been written. Hence, write response times influence task completion times as well.

Read Workflow Reading from the CFS is simpler. Given a file, the CFS master reports the locations of all the replicas of all the blocks of that file. Given these locations, task schedulers try to achieve data locality through a variety of techniques [10, 34, 43, 44].

Although reads are separate from writes, read performance is still influenced by the placement of blocks. By minimizing storage imbalance, a CFS strives to minimize the performance impact of reads in future tasks.

2.2 Network Model

CFS deployments in modern clusters run on topologies that often have a full-bisection bandwidth core (e.g., fat-tree [38], VL2 [28]) with some oversubscription in core-to-rack links (Figure 1). We consider a network model, where downlinks to storage racks can be skewed. This is common in typical data-intensive clusters with

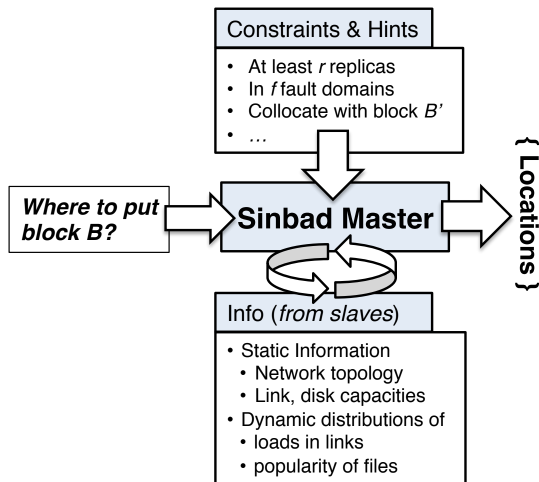


Figure 2: Decision process of Sinbad master.

collocated compute and storage. For dedicated storage racks (i.e., when compute and storage are *not* collocated), skew can still exist due to random placement decisions made by the CFS master.

3 Sinbad Overview

Sinbad is a measurement-based system to perform network-balanced replica placement during distributed writes. In this section, we present a brief overview of Sinbad to help the reader follow the measurements (§4), analysis (§5), and design (§6) presented in subsequent sections.

3.1 Problem Statement

Given a replica placement request – with information about the location of the writer, size of the block, and the replication factor – Sinbad must return a set of locations for the CFS master to replicate that block to (Figure 2). All information about a block request is unknown prior to its arrival.

One can think of this problem as constrained overlay anycast in a throughput-sensitive context. However, instead of retrieving responses from the best sources, we have to replicate large blocks to multiple machines in different fault domains without introducing significant storage imbalance across the cluster.

The problem of placing replicas to minimize the imbalance across bottleneck links is NP-hard even in the offline case (§5). Sinbad employs algorithms developed by exploiting observations from real-world clusters (§4) to perform reasonably well in realistic settings.

3.2 Architectural Overview

Sinbad is designed to replace the default replica placement policy in existing CFSes. Similar to its target CFSes, Sinbad uses a centralized architecture (Figure 3) to use global knowledge while making its decisions. Sinbad master is collocated with the CFS master, and it makes placement decisions based on information collected by its slaves.

Sinbad slaves (collocated with CFS slaves) periodically send measurement updates to the master by piggybacking on regular heartbeats from CFS slaves. They report back several pieces of information, including incoming and outgoing link utilizations at host NICs and current disk usage. Sinbad master aggregates the collected information and estimates current utilizations of bottleneck links (§6.2). Sinbad uses host-based application layer techniques for measurements. This comes out of practicality: we want Sinbad to be usable in public cloud offerings with little or no access to in-network measurement counters. Sinbad can interact nicely with existing network-level load balancers (e.g., Hedera [8], MicroTE [13], or ECMP [28]). This is because network-level techniques balance load among paths given source-destination pairs, whereas Sinbad dictates destinations without enforcing specific paths.

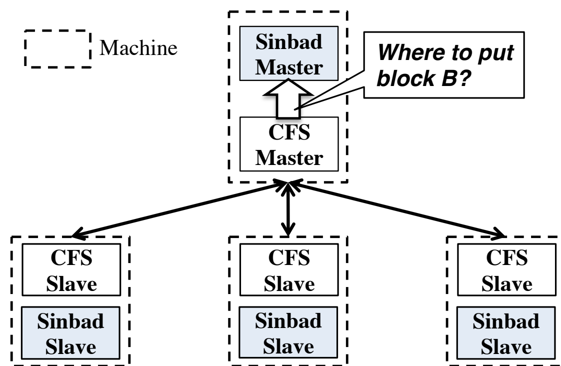


Figure 3: Sinbad architecture. Sinbad agents are collocated with the corresponding agents of the parent CFS.

Table 1: Details of Facebook and Microsoft Bing Traces

	Facebook	**Microsoft Bing**
Date	Oct 2010	Mar-Apr 2012
Duration	One week	One month
Framework	Hadoop [3]	SCOPE [19]
Jobs	175, 000	Tens of Thousands
Tasks	30 million	Tens of Millions
CFS	HDFS [4]	Cosmos [19]
Block Size	256 MB	256 MB
Machines	3, 000	Thousands
Racks	150	Hundreds
Core:Rack Oversubscription	10 : 1	Lower (i.e., Better)

Fault Tolerance and Scalability Since Sinbad agents are collocated with that of the parent CFS, host failures that can take Sinbad slaves offline will take down corresponding CFS agents as well. If Sinbad master dies, the CFS master can always fall back to the default placement policy. Because most CFSes already have a centralized master with slaves periodically reporting to it – Sinbad does not introduce new scalability concerns. Furthermore, piggybacked measurement updates from Sinbad slaves introduce little overhead.

4 Measurements and Implications

In this section, we analyze traces from two production data-parallel clusters – Facebook's Hadoop-HDFS cluster and Microsoft Bing's SCOPE-Cosmos cluster. Table 1 lists the relevant details. In both clusters, core-to-rack links are the most likely locations for network bottlenecks.

Our goal behind analyzing these traces is to highlight characteristics – the volume of distributed writes, the impact of writes on job performance, and the nature of imbalance in bottleneck link utilizations – that motivate us to focus on distributed writes and enable us to make realistic assumptions in our analysis and evaluation.

4.1 Network Footprint of Distributed Writes

Available data points indicate a growing trend of the volume of data ingested into data-intensive clusters. Recent data from Facebook

Table 2: Sources of Cross-Rack Traffic

	Reads	Inter.	Job Writes	Other Writes
Facebook	14%	46%	10%	30%
Microsoft	31%	15%	5%	49%

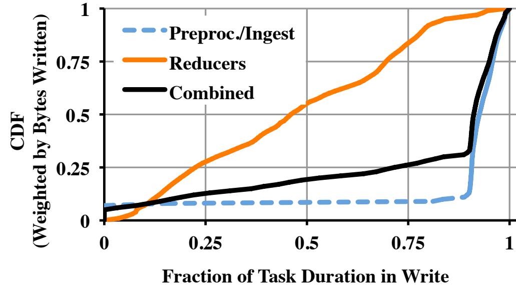


Figure 4: Weighted CDF of the fraction of task durations spent in writing to HDFS by tasks with write phases.

claims that it ingests more than 500 TB *every day* (Aug. 2012) [5]. To provide fresh ads and content recommendations, ingestion happens at regular intervals including peak hours [16, 41]. The ingestion engine of Windows Azure Storage (WAS) keeps around 350 TB of Facebook and Twitter data to provide near-realtime search results within 15 seconds [17]. Finally, Amazon S3 has experienced a sharp rise in the number of objects it stores since 2006; it currently stores more than 1.3 trillion objects (Jan. 2013) [6], many of which are likely to be large blobs due to S3's performance and pricing models.[3]

The impact of the increasing volume of writes was evident in our traces. Although intermediate transfers are known to have significant impact on application performance [20], they are far from being the dominant source of cross-rack traffic! We found that intermediate transfers account for 46% and 15% of all cross-rack traffic in Facebook and Microsoft clusters (Table 2). As expected, both clusters achieved good data locality – only 10% of all tasks read input from non-local machines [10, 34, 44].

Contrary to our expectations, however, we observed that cross-rack replication due to distributed writes accounted for 40% and 54% of the network traffic in the two clusters. In addition to the final output that jobs write to the CFS, we identified two additional sources of writes:

1. *Data ingestion* or the process of loading new data into the cluster amounted for close to 50% of all cross-rack bytes in the Microsoft cluster.
2. *Preprocessor outputs.* Preprocessing jobs only have map tasks. They read data, apply filters and other user-defined functions (UDFs) to the data, and write what remains for later consumption by other jobs. Combined with data ingestion, they contributed 30% of all cross-rack bytes in the Facebook cluster.

4.2 Impact of Writes on Job Performance

To understand the impact of writes on task durations, we compare the duration of the write phase with the runtime of each writer. For a reduce task, we define its "write phase" as the time from the completion of shuffle (reading its input from map outputs over the network) until the completion of the task. For other writers, we define the write phase as the timespan between a task finishing reading its entire input (from local disk) and its completion time.

[3] S3 charges for individual `PUT` requests, and `PUT` response times are empirically better for larger objects.

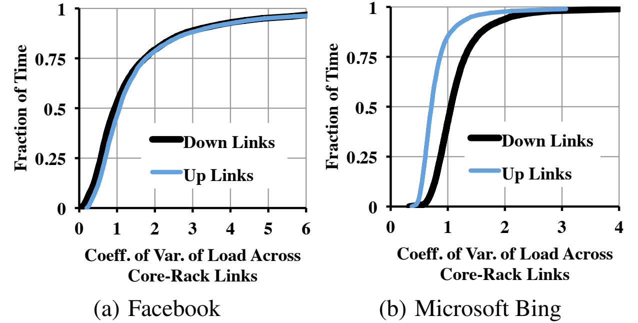


Figure 5: Imbalance in 10s average utilizations of up and downlinks between the core and racks in Facebook and Microsoft clusters due to CFS and non-CFS traffic.

We found that 37% of all tasks write to the CFS. A third of these are reducers; the rest are other writers. We observed that 42% of the reduce tasks and 91% of other writers spent at least half their durations in the write phase (Figure 4). This suggests that faster writes can help many tasks to complete earlier.

4.3 Characteristics of Network Imbalance

We found that the cumulative traffic from intermediate transfers, distributed writes, cluster management workload, and the traffic from collocated services can be substantially imbalanced across the bottleneck links in the short term (e.g., tens of seconds). Causes of imbalance include skew in application communication patterns [14, 35], imbalance in the number of mappers and reducers, and cluster management events such as rolling updates.

We measured network imbalance using the coefficient of variation[4] (C_v) of the average link utilization in each 10s interval across up and downlinks (Figure 5). With perfect balance, these values would be zero. However, in both traces, we found that the downlinks had $C_v > 1$ for almost half the time. Uplinks were equally imbalanced in the Facebook trace, but the imbalance was somewhat lower in the Bing trace. We do not yet know the reason for this.

Although skewed, link utilizations remained stable over short intervals. Such stability is important to enable predictable online placement decisions for relatively short block transfers. To analyze the stability of link utilizations, we calculated average utilization over different time intervals in all core-to-rack links in the Facebook cluster. We consider a link's utilization $U_t(l)$ at time t to be stable for the duration T if the difference between $U_t(l)$ and the average value of $U_t(l)$ over the interval $[t, t+T)$ remains within StableCap% of $U_t(l)$. We observed that average link utilizations remained stable for smaller durations with very high probabilities. For the most unpredictable link, the probabilities that its current utilization from any instant will not change by more than 5% for the next 5, 10, 20, and 40 seconds were 0.94, 0.89, 0.80, and 0.66, respectively. Compare these with the 256 MB blocks used in these clusters. It will take around 5s to write such a block at a disk throughput of 50 MBps, which is small enough to exploit utilization stability periods. We found that 81% of all bytes written to the CFS come from 32% of the blocks that are 256 MB in size.

Imbalance without congestion may not impact performance. We observed in the Facebook trace that the 95th percentile load across bottleneck links was more than 75% of link capacity 25% of the time. However, the effect of congested links is magnified – a single

[4] Coefficient of variation, $C_v = \frac{\sigma}{\mu}$, shows the extent of variability in relation to the mean of the population.

bottleneck link can impact a large number of jobs if they have tasks communicating through that congested link.

Finally, despite significant short-term imbalance, link usages become more balanced over longer intervals (e.g., over hours). During normal operation (i.e., in absence of failures), we observed that each bottleneck link is almost equally likely to become a hotspot.

4.4 Summary

We make the following observations in this section.

- Replication accounts for almost half the cross-rack traffic in data-intensive clusters, and its magnitude is rapidly increasing.
- Network congestion in such clusters has significant skew across bottleneck links in the short term, but the skew is minimal over longer durations.
- Durations of write tasks (e.g., ingestion and preprocessing tasks) are dominated by time spent in writing.
- Most bytes written to the CFS belong to the maximum sized (256 MB) blocks.

5 Analytical Results

The distributed writing problem is NP-hard. In this section, we provide insights into the complexity of this problem, consider assumptions that make the problem tractable, and present two optimal algorithms under the simplifying setting. We also discuss the potential for improvements using a network-aware solution.

Detailed analysis and proofs can be found in the appendix.

5.1 Optimizing Block Writes

The primary objective of a CFS is to minimize the average block write time, which also results in maximizing the system utilization of the CFS. The optimal placement algorithm must select the best destinations (through suitable bottleneck links) for all block requests as they arrive.

Complexity Optimizing for block writes is NP-hard, even when all block requests and link capacities are known beforehand. This can be proven by a reduction from the NP-hard job-shop scheduling problem (**Theorem A.1**).

The online distributed writing problem is even harder because of the following reasons:

1. Links have different available capacities.
2. Lack of future knowledge about
   (a) bottleneck link utilizations throughout the duration of replication, and
   (b) new replica placement requests (sizes and arrival times) to arrive while a block is being replicated.

Variations of job shop scheduling with one or more of the above-mentioned characteristics are known to be NP-hard and hard to approximate as well [9, 36, 37].

Simplified Model For the ease of analysis, we make the following assumptions based on our observations in Section 4.

1. *Blocks have a fixed size.* The size of each block is already bounded. Additionally, we assume all blocks to have the same size (i.e., blocks are padded). Because most bytes are generated by a third of all blocks written, we ignore the impact of the rest during analysis.
2. *Link utilization is fixed while a block is being written.* Since link utilizations remain reasonably stable in the short term, we assume that changes in bottleneck links are precisely known throughout the duration of writing a block. Changes are expected only due to traffic introduced by replication.
3. *Potential bottleneck links are easily discernible.* Given the oversubscribed (logical) tree-based topologies of data-intensive clusters, links leading in and out of the racks are likely to be the bottlenecks. We assume that the potential network bottlenecks are known, which allows us to abstract away the topology.[5]
4. *Decision intervals are independent.* We assume that block requests arrive at the beginning of decision intervals, and they are small enough so that their replication finishes within the same interval. All decision intervals have the same length, q.

Given the aforementioned assumptions, *greedy assignment of blocks to the least-loaded link first is optimal* for minimizing the average block write time (see **Theorem A.2**). We refer to this algorithm as OPT. OPT's improvements over the uniform placement policy increases with the increasing imbalance in the network and as the number of off-rack replicas increases (§A.2).

5.2 Optimizing File Writes

CFSes store data by dividing it into fixed-sized blocks. Hence, a request to write a large file/object generates a sequence of block write requests. For a writer, the objective then to minimize the average file write time. Optimizing the average file write time is no easier than optimizing the average block write time, and, it is NP-hard as well.

OPT is optimal when all blocks have the same size. However, files can have different sizes, which can result in different numbers of equal-sized blocks. Using a simple counter-example it can be shown that OPT is not optimal in this case (**Lemma B.1**).

Given the assumptions in Section 5.1 and with OPT in place, *greedy assignment of blocks through links in the least-remaining-blocks-first order is optimal* for minimizing the average file write time (see **Theorem B.2**). We refer to this algorithm as OPT'.

OPT' favors smaller files. However, larger files will not completely starve as long as the arrival rate of block requests does not exceed the simultaneous serving capacity of the system.

OPT' requires the decision interval to be longer than zero (i.e., $q > 0$) so that it can order blocks from different files by the number of blocks remaining in their parent files. In contrast, $q = 0$ refers to a pure online solution, where OPT' reduces to OPT. The length of the decision interval (q) presents a tradeoff. A larger q potentially provides better opportunities for OPT', but it introduces additional delay to the write time of each block in that interval.

6 Design Details

This section discusses the expected operating environment of Sinbad (§6.1), how Sinbad estimates bottleneck link utilizations across the cluster (§6.2), and how it incorporates (§6.3) the algorithms introduced in Section 5.

6.1 Operating Environment

We make the following assumptions about the characteristics of write requests and on the availability of certain pieces of information (e.g., oversubscription in the topology).

Because most bytes are generated by a third of all blocks written, we consider only fixed-sized blocks in Sinbad. This allows Sinbad to ignore the arrival order of block requests when making a decision for the current block request. In a sense, this problem is akin to an online load balancing problem. A frequent stream of roughly equal-sized entities is quite easy to balance. Contrast this with the case when block sizes are unbounded; whether or not large blocks will arrive in the future crucially impacts placement,

[5] Generalizing to arbitrary topologies adds overhead. For example, we would have to run max-flow/min-cut to determine which of the many bottlenecks are tight given a placement.

Pseudocode 1 Request Dispatching Algorithm

```
1:  procedure GETREPLICALOCATIONS(Request B)
2:      if B.size < THRESHOLD then                 ▷ Ignore small blocks
3:          return Default.getReplicaLocations(B)
4:      end if
5:
6:      if q = 0 then                              ▷ Instantaneous decision
7:          return selectLinks(B, Nil)
8:      end if
9:
10:     ℚ.addToQueue(B)         ▷ Queue up the request. Order by policy.
11: end procedure

12: procedure DISPATCHREQUEST(Link l)            ▷ Called at q intervals.
13:     for all B ∈ ℚ do
14:         return selectLinks(B, Nil)
15:     end for
16: end procedure
```

since one needs to keep room on both network links and disks for such blocks. The rest of the blocks, which are many but contribute insignificant amount of bytes, are placed using the default policy.

Sinbad uses optional information provided by the operator including the topology of the cluster, oversubscription factors at different levels of the topology, and fault domains with corresponding machines. We populate the bottleneck links' set ($\mathbb{L}$) with links that are likely to become bottlenecks; on the topologies used in datacenters today, these are links that have a high oversubscription factor (e.g., host to top-of-rack switch and top-of-rack-switch to core). In the absence of this information, Sinbad assumes that each machine is located in its own fault domain, and $\mathbb{L}$ is populated with the host-to-rack links.

6.2 Utilization Estimator

Sinbad master receives periodic updates from each slave at Δ intervals containing the receiving and transmitting link utilizations at corresponding NICs. After receiving individual updates, Sinbad estimates, for each potential bottleneck link l, the downlink ($Rx(l)$) and uplink ($Tx(l)$) utilizations using exponentially weighted moving average (EWMA):

$$v_{\text{new}}(l) = \alpha\, v_{\text{measured}}(l) + (1-\alpha)\, v_{\text{old}}(l)$$

where, α is the smoothing factor (Sinbad uses $\alpha = 0.2$), and $v(.)$ stands for both $Rx(.)$ and $Tx(.)$. EWMA smooths out the random fluctuations. $v(l)$ is initialized to zero. Missing updates are treated conservatively, as if the update indicated the link was fully loaded.

When $\mathbb{L}$ contains internal links of the topology (i.e., links between switches at different levels), $v_{\text{new}}(l)$ is calculated by summing up the corresponding values from the hosts in the subtree of the farthest endpoint of l from the core.

The update interval (Δ) determines how recent the $v_{\text{new}}(l)$ values are. A smaller Δ results in more accurate estimations; however, too small a Δ can overwhelm the incoming link to the master. We settled for $\Delta = 1$s, which is typical for heartbeat intervals (1 to 3 seconds) in existing CFSes.

Hysteresis After a Placement Decision Once a replica placement request has been served, Sinbad must temporarily adjust its estimates of current link utilizations in all the links involved in transferring that block to avoid selecting the same location for subsequent block requests before receiving the latest measurements. We use an increment function $I(B, \delta)$ that is proportional to the size of the block and inversely proportional to the amount of time remaining until the next update (denoted by δ). At the beginning of an update period, we set $\widehat{v}(l) = v_{\text{new}}(l)$, and upon each assignment

Pseudocode 2 Link Selection Algorithm

```
1:  procedure SELECTLINKS(Request B, Link l)
2:      if l is an edge link then                  ▷ Terminating condition
3:          return {Machine attached to l}
4:      end if
5:
6:      𝕄 = {}
7:      if l = Nil then                            ▷ Called with the tree root
8:          𝕃cur = 𝕃                               ▷ Consider all bottleneck links
9:          𝕄 = {B.localMachine}
10:         B.r = B.r − 1
11:     else                                       ▷ Called recursively
12:         𝕃cur = {l′ : l′ ∈ subtree of l}
13:     end if
14:
15:     𝕃cur = 𝕃cur.filter(B.constraints)          ▷ Filter (§6.4)
16:
17:     if |𝕃cur| < B.r then
18:         return Nil                             ▷ Not enough locations
19:     end if
20:
21:     SORT_DESC 𝕃cur by expectedCapacity(l)
22:     for all l ∈ {First B.r links from 𝕃cur} do
23:         Add hysteresis to l                    ▷ Only to the selected links
24:         Set B.r = 1                            ▷ One replica from each subtree
25:         𝕄 = 𝕄 ∪ {selectLinks(B, l)}
26:     end for
27:     return 𝕄
28: end procedure

29: procedure EXPECTEDCAPACITY(Link l)
30:     return min(Cap(l) − R̂x(l), DiskWriteCap)
31: end procedure
```

of a block B to link l, we add hysteresis as follows:

$$\begin{aligned} I(B,\delta) &= \min(Cap(l) - \widehat{v}(l), \frac{Size(B)}{\delta}) \\ \widehat{v}(l) &= \widehat{v}(l) + I(B,\delta) \end{aligned}$$

When an updated measurement arrives, $\widehat{v}(l)$ is invalidated and $v_{\text{new}}(l)$ is used to calculate the weighted moving average of $v(l)$. Here, $Cap(l)$ represents the capacity of link l and $Size(B)$ is the size of block B.

6.3 Network-Balanced Placement Using Sinbad

Sinbad employs the algorithms developed in Section 5 to perform network-aware replica placement. It involves two steps: ordering of requests and ordering of bottleneck links.

Sinbad queues up (Pseudocode 1) all block requests that arrive within a decision interval of length q and orders them by the number of remaining blocks in that write request (for OPT'). For $q = 0$, Sinbad takes instantaneous decisions. The value of THRESHOLD determines which blocks are placed by Sinbad. Recall that to lower overhead, Sinbad causes most of the smaller blocks, which are numerous but contribute only a small fraction of cluster bytes, to be placed using the default policy.

Given an ordered list of blocks, Sinbad selects the machine with the highest available receiving capacity, i.e., the one that is reachable along a path with bottleneck link l_{sel}, which has the largest remaining capacity among all potential bottlenecks:

$$l_{\text{sel}} = \underset{l \in \mathbb{L}}{\operatorname{argmax}} \min(Cap(l) - \widehat{Rx}(l), \text{DiskWriteCap})$$

where, $\widehat{Rx}(l)$ is the most recent estimation of $Rx(l)$ at time $Arr(B)$ and DiskWriteCap is the write throughput of a disk. This holds because the copy has to move in the transmit direction regardless. The choice of placement only impacts where it ends up, and hence, which other links are used on their receive direction.

Further, Sinbad can generalize to the case when there are multiple bottleneck links along such receiving paths. Hysteresis (described above) lets Sinbad track the ongoing effect of placement decisions before new utilization estimates arrive.

Pseudocode 2 shows how Sinbad proceeds recursively, starting from the bottleneck links, to place the desired number of replicas. The entry point for finding replicas for a request B with this procedure is SELECTLINKS(B, Nil). Calling it with an internal link restricts the search space to a certain subtree of the topology.

Because a replication flow cannot go faster than DiskWriteCap, it might seem appealing to try to match that throughput as closely as possible. This choice would leave links that have too much spare capacity for a given block free, possibly to be used for placing another larger block in near future. However, in practice, this causes imbalance in the number of replication flows through each bottleneck link and in the number of concurrent writers in CFS slaves, and hurts performance.

6.4 Additional Constraints

In addition to fault tolerance, partition tolerance, and storage balancing, CFS clients can provide diverse suitability constraints. Collocation of blocks from different files to decrease network communication during equi-joins is one such example [23]. Because such constraints decrease Sinbad's choices, the improvements are likely to be smaller. Sinbad satisfies them by filtering out unsuitable machines from $\mathbb{L}_{cur}$ (line 15 in Pseudocode 2).

7 Evaluation

We evaluated Sinbad through a set of experiments on a 100-machine EC2 [1] cluster using workloads derived from the Facebook trace. For a larger scale evaluation, we used a trace-driven simulator that performs a detailed replay of task logs from the same trace. Through simulation and experimentation, we look at Sinbad's impact when applied on two different levels of the network topology: in the former, Sinbad tries to balance load across links connecting the core to individual racks; whereas, in the latter, it aims for balanced edge links to individual machines (because the EC2 topology and corresponding bottlenecks are unknown). Our results show the following:

- Sinbad improves the average block write time by $1.3\times$ and the average end-to-end job completion time by up to $1.26\times$, with small penalties for online decisions (§7.2).
- Through simulations, we show that Sinbad's improvement ($1.58\times$) is close to that of an optimistic estimation ($1.89\times$) of the optimal (§7.2).
- Sinbad decreases the median imbalance across bottleneck links (i.e., median C_v) by up to 0.33 in both simulation and experiment (§7.3).
- Sinbad has minimal impact on the imbalance in disk usage (0.57% of disk capacity), which remains well within the tolerable limit (10%) (§7.4).
- If disks are never the bottlenecks, Sinbad improves the average block write time by $1.6\times$ and the average end-to-end job completion time by up to $1.33\times$ (§7.6).

7.1 Methodology

Workload Our workload is derived from the Facebook trace (§4). During the derivation, we preserve the original workload's write characteristics, including the ratio of intermediate and write traffic, the inter-arrival times between jobs, the amount of imbalance in communication, and the input-to-output ratios for each stage. We scale down each job proportionally to match the size of our cluster.

Table 3: Jobs Binned by Time Spent in Writing

Bin	1	2	3	4	5
Write Dur.	$< 25\%$	25–49%	50–74%	75–89%	$\geq 90\%$
% of Jobs	16%	12%	12%	8%	52%
% of Bytes	22%	5%	3%	3%	67%

We divide the jobs into bins (Table 3) based on the fraction of their durations spent in writing. Bin-5 consists of *write-only* ingestion and preprocessing jobs (i.e., jobs with no shuffle), and bins 1–4 consist of typical MapReduce (*MR*) jobs with shuffle and write phases. The average duration of writes in these jobs in the original trace was 291 seconds.

Cluster Our experiments use extra large EC2 instances with 4 cores and 15 GB of RAM. Each machine has 4 disks, and each disk has 400 GB of free space. The write throughput of each disk is 55 MBps on average.

The EC2 network has a significantly better oversubscription factor than the network in the original Facebook trace – we observed a bisection bandwidth of over 700 Mbps/machine on clusters of 100 machines. At smaller cluster sizes we saw even more – up to 900 Mbps/machine for a cluster of 25 machines. Note that the virtual machine (VM) placement in EC2 is governed by the provider and is supposed to be transparent to end users. Further, we did not see evidence of live migration of VMs during our experiments.

We use a computation framework similar to Spark [45] and the Facebook-optimized distribution of HDFS [4] with a maximum block size of 256 MB. In all experiments, replication and fault-tolerance factors for individual HDFS files are set to three ($r = 3$) and one ($f = 1$), respectively. Unless otherwise specified, we make instantaneous decisions ($q = 0$) in EC2 experiments.

Trace-Driven Simulator We use a trace-driven simulator to assess Sinbad's performance on the scale of the actual Facebook cluster (§4) and to gain insights at a finer granularity. The simulator performs a detailed task-level replay of the Facebook trace that was collected on a 3000-machine cluster with 150 racks. It preserves input-to-output ratios of all tasks, locality constraints, and presence of task failures and stragglers. We use the actual background traffic from the trace and simulate replication times using the default uniform placement policy. Disk write throughput is assumed to be 50 MBps. Unless otherwise specified, we use 10 second decision intervals ($q = 10s$) in simulations to make them go faster.

Metrics Our primary metric for comparison is the improvement in average completion times of individual block replication, tasks, and jobs (when its last task finished) in the workload, where

$$Factor\ of\ Improvement = \frac{Modified}{Current}$$

We also consider the imbalance in link utilization across bottleneck links as well as the imbalance in disk usage/data distribution across the cluster. The baseline for our deployment is the uniform replica placement policy used in existing systems [4, 19]. We compare the trace-driven simulator against the default policy as well.

7.2 Sinbad's Overall Improvements

Sinbad reduces the average completion time of write-heavy jobs by up to $1.26\times$ (Figure 6) and jobs across all bins gain a $1.29\times$ boost in their average write completion times. Note that varying improvements in write times across bins are not correlated with the characteristics of jobs in those bins; blocks in these experiments were placed without considering any job-specific information. MR jobs in bin-1 to bin-4 have lower overall improvements than bin-5

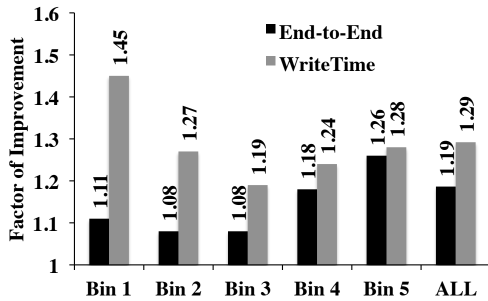


Figure 6: [EC2] Improvements in average job completion times and time spent in distributed writes.

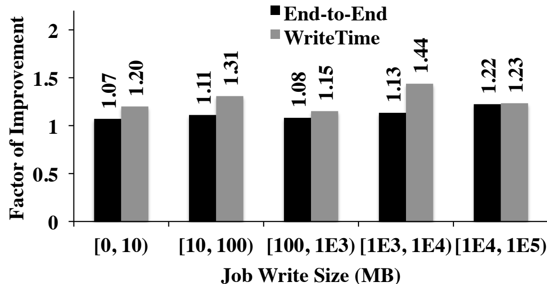


Figure 7: [EC2] Improvements in jobs categorized by amounts of distributed writes.

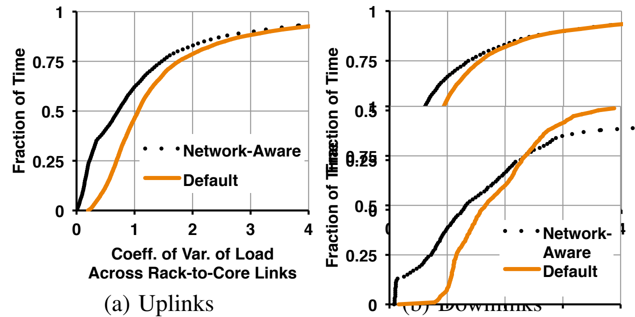


Figure 8: [Simulation] Network awareness (Sinbad) decreases load imbalance across racks in the Facebook trace.

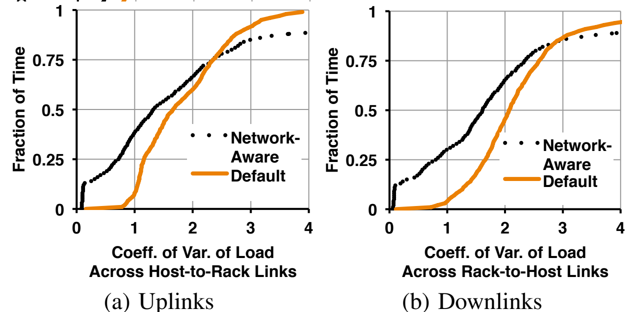


Figure 9: [EC2] Network awareness (Sinbad) decreases load imbalance across machines.

because they are computationally heavy and because shuffles see negligible change in these set of experiments. The average end-to-end completion times of jobs improved by $1.19\times$ and block write time across all jobs improved by $1.3\times$.

Being an online solution, Sinbad does not always perform well. We found that 15% of the jobs either had no improvements or experienced slightly higher overall completion times. In contrast, the top 15% jobs improved by at least $1.4\times$. Part of Sinbad's inefficiency can be attributed to its limited view of the network in the virtualized EC2 environment.

A breakdown of improvements in jobs by their write sizes (Figure 7) does not show any significant difference between categories. This is because we optimize for blocks in the experiments using $q = 0$ which does not differentiate between the different amounts that each task or job writes.

Trace-Driven Simulation Unlike the EC2 deployment, we precalculated the background traffic due to shuffle in the simulation and assumed it to be unaffected by placement decisions. Hence, we do not distinguish between jobs in different bins in the simulation. We used OPT' in the simulations with $q = 10s$.

We found that 46% of individual block replication requests improved, 47% remained the same, and 7% became slower. The average improvement across all requests was $1.58\times$, and 16% completed at least $2\times$ faster. Average completion times of writers and communication times of jobs (weighted by their sizes) improved by $1.39\times$ and $1.18\times$, respectively.

How Far are We From the Optimal? While it was not possible to exhaustively enumerate all the possible orderings of block requests to find the optimal, we tried to find an optimistic estimation of improvements. First, we increased q to up to $100s$. But, it did not result in significant increase in improvements.

Next, we tried a drastic simplification. Keeping $q = 10s$, we assumed that there are no bottlenecks at sources; i.e., sources can also be placed at suitable locations before writes start. Note that unlike destination placement, this is hard to achieve in practice because it is hard to predict when a task might start writing. With this simplification, we found that 58% requests improved, 39% remained the same, and 3% were worse off using Sinbad. The average improvements in block write time, task completion time, and weighted job communication time were $1.89\times$, $1.59\times$, and $1.36\times$, respectively.

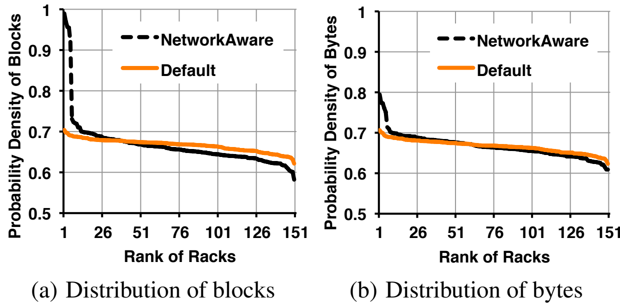
On Batched Decisions We did not observe significant improvement for $q > 0$ using OPT'. This is because the effectiveness of batching depends on the duration of the window to accumulate requests, which in turn depends on the request arrival rate and the average time to serve each request. Batching can only be effective when the arrival rate and/or the service time are substantial (i.e., larger blocks).

In the Facebook trace, on average, 4 large block requests arrive every second. With 50 MBps disk write speed, writing a 256 MB block would take 5 seconds. However, the number of suitable bottleneck links is several times larger than the arrival rate. Consequently, batching for a second will not result in any improvement, but it will increase the average write time of those 4 blocks by 20%.

7.3 Impact on Network Imbalance

We found that Sinbad decreases the network imbalance in both simulation and deployment. Figure 8 plots the change in imbalance of load in the bottleneck links in the Facebook cluster. We see that Sinbad significantly decreases load imbalance (C_v) across up and downlinks of individual racks – decrease in median C_v being 0.35 and 0.33, respectively. We observed little improvement on the low end (i.e., C_v close to zero) because those rack-to-core links had almost equal utilizations to begin with.

Figure 9 presents the imbalance of load in the edge links connecting individual machines to the EC2 network. We notice that irrespective of the placement policy, the average values of imbal-



(a) Distribution of blocks (b) Distribution of bytes

Figure 10: [Simulation] PDFs of blocks and bytes across racks. Network-aware placement decisions do not significantly impact data distribution.

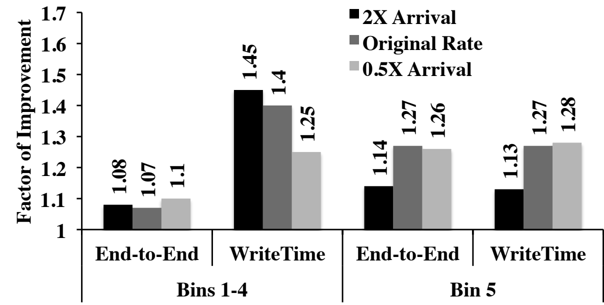


Figure 11: [EC2] Improvements in average completion times with different arrival rates of jobs.

ance (C_v) are higher than that observed in our simulation. This is because we are calculating network utilization and corresponding imbalance at individual machines, instead of aggregating over 20 machines in each rack. We find that Sinbad decreases imbalance across edge links as well – decreases in median values of C_v for up and downlinks are 0.34 and 0.46, respectively. For high C_v values ($C_v \geq 2.5$), Sinbad experienced more variations; we found that most of these cases had low overall utilization and hence had little impact on performance.

7.4 Impact on Future Jobs (Disk Usage)

Since replica placement decisions impact future jobs,[6] does Sinbad create significant imbalance in terms of disk usage and the total number of blocks placed across bottleneck links?

We found that after storing 9.24 TB data (including replicas) in an hour-long EC2 experiment, the standard deviation of disk usage across 100 machines was 15.8 GB using Sinbad. For the default policy it was 6.7 GB. Hence, the imbalance was not so much more that it would trigger an automated block rebalancing, which typically happens when imbalance is greater than some fixed percentage of disk capacity.

Simulations provided a similar result. Figure 10 presents the probability density functions of block and byte distributions across 150 racks at the end of a day-long simulation. We observe that Sinbad performs almost as good as the default uniform placement policy in terms of data distribution.

Sinbad performs well because it is always reacting to the imbalance in the background traffic, which is uniformly distributed across all bottleneck links in the long run (§4.3). Because Sinbad does not introduce noticeable storage imbalance, it is not expected to adversely affect the data locality of future jobs. Reacting to network imbalance is not always perfect, however. We see in Figure 10 that some locations (less than 4%) received disproportionately large amounts of data. This is because these racks were down[7] throughout most of the trace. Sinbad can be extended with constraints on maximum imbalance in data placement to avoid such scenarios.

7.5 Impact of Cluster/Network Load

To understand the impact of network contention, we compared Sinbad with the default placement policy by changing the arrival rate of jobs. In this case, we used a shorter trace with the same job mix, scaled down jobs as before, but we spread the jobs apart over time.

We notice in Figure 11 that Sinbad performed well for different levels of network contention. However, changing the arrival rate by a factor of N does not change the network load by $N\times$ which depends on many factors including the number of tasks, the input-output ratio, and the actual amount of data in the scaled-down scenario. By making jobs arrive $2\times$ faster, we saw around $1.4\times$ *slower* absolute completion times for both policies (not shown in Figure 11); this suggests that the network load indeed increased by some amount. We observe that Sinbad's improvements decreased as well: as the network becomes more and more congested, the probability of finding a less loaded destination decreases.

Changing the arrival rate to half of the original rate (i.e., $0.5\times$ arrival rate) decreased the overall time very little (not shown in Figure 11). This suggests that perhaps resource contention *among* tasks from the same job is more limiting compared to that across tasks from different jobs. For the write-only jobs, Sinbad's improvements did not change either. However, for the MR jobs, shuffle time improved at the expense of corresponding writes.

7.6 Impact on In-Memory Storage Systems

So far we have considered Sinbad's performance only on disk-based CFSes. However, improvements in these systems are bounded by the write throughput of disks and depend on how CFSes schedule writes across multiple disks. To avoid disk constraints, several in-memory storage systems and object caches have already been proposed [12]. To understand Sinbad's potentials without disk constraints, we implemented an in-memory object cache similar to PACMan [12]. Blocks are replicated and stored in the memory of different machines, and they are evicted (in FIFO order) when caches become full. Note that the network is the only bottleneck in this setup. We used the same settings from §7.2.

From EC2 experiments, we found that jobs across all bins gained boosts of $1.44\times$ and $1.33\times$ in their average write and end-to-end completion times, respectively. The average block write time improved by $1.6\times$ using Sinbad in the in-memory CFS.

A trace-driven simulation without disk constraints showed a $1.79\times$ average improvement across all block write requests. We found that 64% of individual block replication requests improved, 29% remained the same, and 7% became slower. Average completion times of writers and communication times of jobs (weighted by their sizes) improved by $1.4\times$ and $1.15\times$, respectively.

8 Discussion

Network-Assisted Measurements In the virtualized EC2 environment, Sinbad slaves cannot observe anyone else's traffic in the network including that of collocated VMs. This limits Sinbad's effectiveness. When deployed in private clusters, with proper instrumentation (e.g., queryable SDN switches), Sinbad would observe the impact of all incoming and outgoing traffic. We expect that correcting for more imbalances will increase Sinbad's gains.

Sinbad is also sensitive to delayed and missing heartbeats, which

[6] Placing too many blocks in some machines and too few in others can decrease data locality.

[7] Likely reasons for downtimes include failure or upgrade, but it can be due to any number of reasons.

can introduce inaccuracy in usage estimations. By placing the heartbeats in a higher priority queue [24], the network can ensure their timely arrival.

Flexible Source Placement By carefully placing the sources of write traffic, it is possible to make the entire write pipeline network aware. However, this is harder and less useful than choosing destinations. Because, a task must finish everything else before writing, e.g., computation for a preprocessing task and shuffle for a reducer, it is hard to estimate the start time or the size of a write operation. Moreover, all the blocks to be written by a source share the same upstream bottleneck. This further constrains source placement, because effectively they look like one large, variable-sized block.

Optimizing Parallel Writers The completion time of a data-parallel job depends on the task that finishes last [20, 21]. Hence, to minimize the job completion time, one must minimize the *makespan* of all the concurrent writers of that job. This, however, calls for a cross-layer design with additional job-specific details in the placement algorithm.

9 Related Work

Datacenter Traffic Management Hedera [8] uses a centralized scheduler to infer the demands of *elephant* flows and assigns them to one of the several paths through the core that exist in a fat-tree topology between the same endpoints. MicroTE [13] adapts to traffic variations by leveraging short-term predictability of the traffic matrix. Orchestra [20] focuses on certain forms of network traffic in data-intensive applications (shuffles and broadcasts) but does not consider the network impact of cluster storage. All of the above manage flows with already-fixed endpoints and do not leverage the flexibility in endpoint placement.

Endpoint-Flexible Transfers Overlay anycast comes the closest to Sinbad in exploiting flexibility in choosing endpoints [18, 25]. However, anycast has typically been used in low latency request-response traffic (e.g., DNS requests); on the contrary, we use endpoint flexibility in a throughput-sensitive context for replica placement. Our constraints – number of replicas/fault domains and overall balance in storage – are different as well.

Full Bisection Bandwidth Networks Recent datacenter network architectures [28–30, 38] aim for full bisection bandwidth for better performance. This, however, does not imply no network contention. In presence of (skewed) data-intensive communication, some links often become more congested than others [35]. Selection of appropriate destinations during replica placement is necessary to pick less congested destinations.

Distributed File Systems Distributed file systems [15, 19, 27] often focus on the fault-tolerance and consistency of stored data without considering the network. Replica placement policies emphasize availability in presence of network failures. We focus on performance improvement through network-balanced replica placement *without* changing any of the fault-tolerance, availability, or consistency guarantees. Unlike traditional designs, Flat Datacenter Storage (FDS) separates computation and storage: data is always read over the network [39]. FDS does not have a centralized master and uses randomization for load balancing. Network-awareness can potentially improve its performance as well.

Erasure-Coded Storage Systems To mitigate the storage crunch and to achieve better reliability, erasure coding of stored data is becoming commonplace [33, 40]. However, coding typically happens *post facto*, i.e., each block of data is three-way replicated first, then coded lazily, and replicas are deleted after the coding finishes. This has two implications. First, network awareness accelerates the end-to-end coding process. Second, in presence of failures and ensuing rebuilding storms, the network experiences even more contention; this strengthens the need for Sinbad.

Data Locality Disk locality [22] has received more attention than most concepts in data-intensive computing, in designing both distributed file systems and schedulers for data-intensive applications [10, 34, 44]. Data locality, however, decreases network usage only during reads; it does not affect the network consumption of distributed writes. Sinbad has little impact on data locality, because it keeps the storage balanced. Moreover, Sinbad can help systems like Scarlett [11] that increase replica count to decrease read contention. Recent calls for memory locality [12] will make network-aware replication even more relevant by increasing network contention during off-rack reads.

10 Conclusion

We have identified the replication traffic from writes to cluster file systems (CFSes) as one of the major sources of network communication in data-intensive clusters. By leveraging the fact that a CFS only requires placing replicas in *any* subset of feasible machines, we have designed and evaluated Sinbad, a system that identifies network imbalance through periodic measurements and exploits the flexibility in endpoint placement to navigate around congested links. Network-balanced replica placement improves the average block write time by $1.3\times$ and the average completion time of data-intensive jobs by up to $1.26\times$ in EC2 experiments. Because network hotspots show short-term stability but they are uniformly distributed in the long term, storage remains balanced. We have also shown that Sinbad's improvements are close to that of the optimal, and they will increase if network imbalance increases.

Acknowledgments

We thank Yuan Zhong, Matei Zaharia, Gautam Kumar, Dave Maltz, Ali Ghodsi, Ganesh Ananthanarayanan, Raj Jain, the AMPLab members, our shepherd John Byers, and the anonymous reviewers for useful feedback. This research is supported in part by NSF CISE Expeditions award CCF-1139158 and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, SAP, Blue Goji, Cisco, Clearstory Data, Cloudera, Ericsson, Facebook, General Electric, Hortonworks, Huawei, Intel, Microsoft, NetApp, Oracle, Quanta, Samsung, Splunk, VMware, Yahoo!, and a Facebook Fellowship.

11 References

[1] Amazon EC2. http://aws.amazon.com/ec2.
[2] Amazon Simple Storage Service. http://aws.amazon.com/s3.
[3] Apache Hadoop. http://hadoop.apache.org.
[4] Facebook Production HDFS. http://goo.gl/BGGuf.
[5] How Big is Facebook's Data? 2.5 Billion Pieces Of Content And 500+ Terabytes Ingested Every Day. TechCrunch http://goo.gl/n8xhq.
[6] Total number of objects stored in Amazon S3. http://goo.gl/WTh6o.
[7] S. Agarwal et al. Reoptimizing data parallel computing. In *NSDI*, 2012.
[8] M. Al-Fares et al. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, 2010.
[9] N. Alon et al. Approximation schemes for scheduling on parallel machines. *Journal of Scheduling*, 1998.
[10] G. Ananthanarayanan et al. Reining in the outliers in mapreduce clusters using Mantri. In *OSDI*, 2010.
[11] G. Ananthanarayanan et al. Scarlett: Coping with skewed popularity content in mapreduce clusters. In *EuroSys*, 2011.

[12] G. Ananthanarayanan et al. PACMan: Coordinated memory caching for parallel jobs. In *NSDI*, 2012.

[13] T. Benson et al. MicroTE: Fine grained traffic engineering for data centers. In *CoNEXT*, 2011.

[14] P. Bodik et al. Surviving failures in bandwidth-constrained datacenters. In *SIGCOMM*, 2012.

[15] D. Borthakur. The Hadoop distributed file system: Architecture and design. Hadoop Project Website, 2007.

[16] D. Borthakur et al. Apache Hadoop goes realtime at Facebook. In *SIGMOD*, 2011.

[17] B. Calder et al. Windows Azure Storage: A highly available cloud storage service with strong consistency. In *SOSP*, 2011.

[18] M. Castro et al. Scalable application-level anycast for highly dynamic groups. *LNCS*, 2003.

[19] R. Chaiken et al. SCOPE: Easy and efficient parallel processing of massive datasets. In *VLDB*, 2008.

[20] M. Chowdhury et al. Managing data transfers in computer clusters with Orchestra. In *SIGCOMM*, 2011.

[21] M. Chowdhury and I. Stoica. Coflow: A networking abstraction for cluster applications. In *HotNets-XI*, 2012.

[22] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[23] M. Y. Eltabakh et al. CoHadoop: Flexible data placement and its exploitation in hadoop. In *VLDB*, 2011.

[24] A. D. Ferguson et al. Hierarchical policies for Software Defined Networks. In *HotSDN*, 2012.

[25] M. Freedman, K. Lakshminarayanan, and D. Mazières. OASIS: Anycast for any service. In *NSDI*, 2006.

[26] M. Garey and D. Johnson. "Strong" NP-completeness results: Motivation, examples, and implications. *Journal of the ACM*, 25(3):499–508, 1978.

[27] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, 2003.

[28] A. Greenberg et al. VL2: A scalable and flexible data center network. In *SIGCOMM*, 2009.

[29] C. Guo et al. DCell: A scalable and fault-tolerant network structure for data centers. In *SIGCOMM*, 2008.

[30] C. Guo et al. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. *ACM SIGCOMM*, 2009.

[31] Z. Guo et al. Spotting code optimizations in data-parallel pipelines through PeriSCOPE. In *OSDI*, 2012.

[32] B. Hindman et al. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*, 2011.

[33] C. Huang et al. Erasure coding in Windows Azure Storage. In *USENIX ATC*, 2012.

[34] M. Isard et al. Quincy: Fair scheduling for distributed computing clusters. In *SOSP*, 2009.

[35] S. Kandula et al. The nature of datacenter traffic: Measurements and analysis. In *IMC*, 2009.

[36] J. Lenstra, D. Shmoys, and É. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46(1):259–271, 1990.

[37] R. Motwani, S. Phillips, and E. Torng. Non-clairvoyant scheduling. In *SODA*, 1993.

[38] R. N. Mysore et al. PortLand: A scalable fault-tolerant layer 2 data center network fabric. In *SIGCOMM*, 2009.

[39] E. Nightingale et al. Flat Datacenter Storage. In *OSDI*, 2012.

[40] M. Sathiamoorthy et al. XORing elephants: Novel erasure codes for big data. In *PVLDB*, 2013.

[41] A. Thusoo et al. Data warehousing and analytics infrastructure at Facebook. In *SIGMOD*, 2010.

[42] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, 2004.

[43] M. Zaharia et al. Improving mapreduce performance in heterogeneous environments. In *OSDI*, 2008.

[44] M. Zaharia et al. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, 2010.

[45] M. Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

APPENDIX

A Optimizing Block Writes

A.1 Problem Formulation and Complexity

Assume that a replica placement request for a block B of size $Size(B)$ arrives at time $Arr(B)$ with replication factor r and fault-tolerance factor f. The CFS client will write a local copy wherever it is located, and the replica placement policy must find locations for $(r-1)$ off-rack copies in f other fault domains across rack boundaries. We choose $r=2$ and $f=1$ to simplify the analysis: the case of larger r is ignored, because an increase in r does not increase the number of off-rack copies; the case of larger f is discussed in §A.2. Assume that there are no other constraints besides physical limits such as link capacity and disk throughput.

Let $\mathbb{L}$ denote the set of possible bottleneck links in the network. Also, let $Cap(l)$ and $U_t(l)$ denote the capacity and the estimation of the utilization of link $l \in \mathbb{L}$ at time t. Placement decisions are instantaneous.

For a given time period T (discretized into equal-sized decision interval or quanta, q) from t, the objective ($\mathsf{U}(.)$) can then be represented by the following equation.

$$\text{Minimize} \sum_{\{B \mid Arr(B) \in [t, t+T)\}} Dur(B) \tag{1}$$

where $Dur(B)$ is the time to write a block B from $Arr(B)$.

Distributed writing (i.e., optimizing U) is NP-hard, even when all block requests and link capacities are known beforehand.

Theorem A.1 *Distributed Writing is NP-hard.*

Proof Sketch We reduce job shop scheduling, which is NP-hard [26], to distributed writing. Consider m identical machines, and n jobs $(J_1, J_2, \ldots, J_n)$ of varying lengths that arrive over time. Let $dur(J_k)$ denote the time to process J_k in any of the machines. Now, consider a network with m bottleneck links, where all links have the same capacity. For each job J_k, create a block B_k, such that B_k takes exactly $dur(J_k)$ time to be written through any of the bottleneck links. Then, optimally placing these n blocks through m links will provide the optimal schedule of jobs. ∎

A.2 Optimal Block Placement Algorithm

If all blocks have the same size, decision intervals are independent, and link utilizations do not change within the same decision interval, greedy assignment of blocks to the least-loaded link will maximize U.

Theorem A.2 *Under the assumptions in Section 5.1, greedy assignment of blocks to the least-loaded link is optimal (OPT).*

Proof Sketch Assume that the bottleneck links in $\mathbb{L}$ at time t are sorted in the non-decreasing order of their current utilizations $\left(U_t(l_1) \leq U_t(l_2) \leq \ldots \leq U_t(l_{|\mathbb{L}|})\right)$, and multiple block requests ($\mathbb{B} = \{B_j\}$) of equal size arrived at t. Since all blocks have the same size, they are indistinguishable and can be assigned to corresponding destinations through some bottleneck links one by one.

Assume block B_j is placed through a bottleneck link l_i. If there is another link l_{i-1} with more available capacity, then we can simply place B_j through l_{i-1} for a faster overall completion time. This process will continue until there is no more such links, and B_j has been placed through l_1 – the least-loaded bottleneck link. ∎

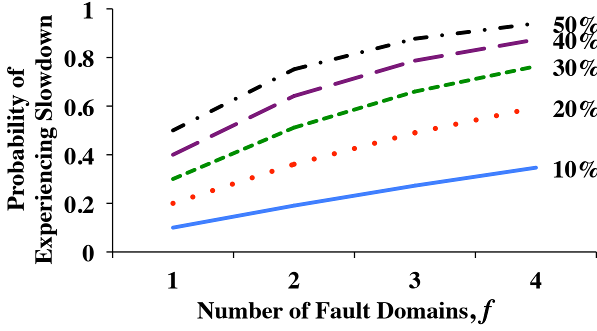


Figure 12: Analytical probability of at least one off-rack replica experiencing a slowdown with uniform placement, as the number of off-rack replicas (f) increases. Each line corresponds to the fraction of contended links (out of 150) that can cause slowdown.

Improvements Over the Default Policy Let $AC(l_i)$ denote the available capacity to write block B through l_i,

$$AC(l_i) = \min(Cap(l_i) - U_t(l_i), \mathsf{DiskWriteCap})$$

where DiskWriteCap is the write throughput of a disk. Then the completion time of OPT for writing $|\mathbb{B}|$ blocks that must be placed during that quantum is

$$\mathsf{U}(OPT) = |\mathbb{B}|Size(B)\sum_{i=1}^{|\mathbb{L}|}\frac{f_i}{AC(l_i)} \tag{2}$$

where f_i is the fraction of blocks allocated through l_i. Because OPT greedily places blocks starting from the least-loaded link, higher $AC(l_i)$ will result in higher f_i.

Current CFS replica placement algorithms (UNI) place replicas uniformly randomly across all bottleneck links. The expected time for UNI to write $|\mathbb{B}|$ blocks that must be placed during a single quantum is

$$\mathsf{U}(UNI) = \frac{|\mathbb{B}|Size(B)}{|\mathbb{L}|}\sum_{i=1}^{|\mathbb{L}|}\frac{1}{AC(l_i)} \tag{3}$$

where each link l_i will receive equal fractions ($\frac{1}{|\mathbb{L}|}$) of blocks.

Given (2) and (3), the factor of improvement ($\mathcal{I}$) of OPT over UNI at any decision interval is

$$\mathcal{I} = \frac{\mathsf{U}(UNI)}{\mathsf{U}(OPT)} = |\mathbb{L}|\frac{\sum_{i=1}^{|\mathbb{L}|}\frac{f_i}{AC(l_i)}}{\sum_{i=1}^{|\mathbb{L}|}\frac{1}{AC(l_i)}} \tag{4}$$

The overall improvement during the discretized interval $[t, t+T)$ is

$$\frac{\sum_{[t,t+T)}\mathsf{U}(UNI)}{\sum_{[t,t+T)}\mathsf{U}(OPT)} \tag{5}$$

Improvements for $f > 1$ We have assumed that only one copy of each block crosses the bottleneck links (i.e., $f = 1$). However, for a given placement request, the probability of experiencing contention using UNI increases with f; this should increase $\mathcal{I}$, because OPT will never experience more contention than UNI.

While creating replicas in f fault domains through $|\mathbb{L}|(\gg f)$ bottleneck links, if one of the f replicas experience contention, the entire write will become slower. Using the uniform placement policy, the probability of at least one of them experiencing contention is

$$P(\text{UniSlowdown}) = \Big(1 - \prod_{g=1}^{f}\frac{|\mathbb{L}| - G - g + 1}{|\mathbb{L}| - g + 1}\Big)$$

where G is the number of congested links in $\mathbb{L}$. As f increases, so does $P(\text{UniSlowdown})$; Figure 12 shows this for different values of G in a 150-rack cluster. The extent of UNI slowdown (thus $\mathcal{I}$) depends on the distribution of imbalance.

B Optimizing File Writes

B.1 Problem Formulation and Complexity

Assume that a file $W = (B_1, B_2, \ldots, B_{|W|})$ with $|W|$ blocks arrives at $Arr(W)$. Requests for its individual blocks B_j arrive one at a time, each at the beginning of a discrete decision interval. For a given time period T (discretized into equal-sized quanta, q) from t, the objective ($\mathsf{V}(.)$) can then be represented by the following equation.

$$\text{Minimize} \sum_{\{W \mid Arr(W)\in[t,t+T)\}} Dur(W) \tag{6}$$

where $Dur(W) = \max_{B_j \in W} Dur(B_j)$ denotes the time to finish writing all the blocks of W from $Arr(W)$. Optimizing (6) is no easier than optimizing (1), and it is NP-hard as well.

B.2 Optimal File Write Algorithm

Lemma B.1 *OPT is not optimal for end-to-end write operations with multiple blocks (V).*

Proof Sketch We prove this using a counterexample. Let us take a simple example with two bottleneck links, l_1 and l_2, through which a single block can be written in d_1 and d_2 time units ($d_1 < d_2$ and $d_1, d_2 \le 1$ time unit), respectively. Also assume that two write requests $W_1 = (B_{11}, B_{12})$ and $W_2 = (B_{21})$ arrive at the same time t; more specifically, B_{11} and B_{21} arrive at t and B_{12} arrives at $(t + 1)$.

Because OPT treats requests interchangeably, it can either write B_{11} through l_1 and B_{21} through l_2 or in the inverse order (B_{11} through l_2 and vice versa) at t. In either case, OPT will write B_{12} through l_1 at $(t+1)$ because d_1 is smaller. Irrespective of the order, $\mathsf{U}(OPT) = 2d_1 + d_2$.

However, end-to-end response times are different for the two cases: in the former, $\mathsf{V}^1 = (1 + d_1) + d_2$, while in the latter, $\mathsf{V}^2 = (1 + d_1) + d_1$. Because OPT can choose either with equal probability, its expected total response time is $\frac{1}{2}(\mathsf{V}^1 + \mathsf{V}^2)$, which is more than the actual optimal V^2. ■

Theorem B.2 *Given Theorem A.2, greedy assignment of writes through links using the least-remaining-blocks-first order is optimal (OPT').*

Proof Sketch Given OPT, the bottleneck links $l_i \in \mathbb{L}$ at time t are sorted in the non-decreasing order of their expected times to write a block, i.e., $d_1 \le d_2 \le \ldots \le d_{|\mathbb{L}|}$. Assume that multiple equal-sized block requests ($\mathbb{B} = \{B_j\}$) from unique write operations ($\mathbb{W} = \{W_j\}$) have arrived at t, and let us denote the number of remaining blocks in a write operation W_j by $Rem(W_j)$.

Consider two blocks B_j and B_{j+1} from writes W_j and W_{j+1} that will be considered one after another and will be assigned to destinations through links l_i and l_{i+1} such that $d_i \le d_{i+1}$. Assume, $Rem(W_j) > 1$ and $Rem(W_{j+1}) = 1$. Now, the total contributions of B_j and B_{j+1} to $\mathsf{V}(OPT')$ is $(1 + d_{i+1})$. We can decrease it to $(1 + d_i)$ by swapping the order of B_j and B_{j+1}. By continuously performing this pairwise swapping, we will end up with an order where the block from the write with the fewest remaining blocks will be considered before others. ■