Blizzard: Fast, Cloud-scale Block Storage for Cloud-oblivious Applications

James Mickens, Edmund B. Nightingale, Jeremy Elson, Krishna Nareddy, Darren Gehring
Microsoft Research

Bin Fan[*]
Carnegie Mellon University

Asim Kadav[†] Vijay Chidambaram[‡]
University of Wisconsin-Madison

Osama Khan[§]
Johns Hopkins University

Abstract

Blizzard is a high-performance block store that exposes cloud storage to cloud-oblivious POSIX and Win32 applications. Blizzard connects clients and servers using a network with full-bisection bandwidth, allowing clients to access any remote disk as fast as if it were local. Using a novel striping scheme, Blizzard exposes high disk parallelism to both sequential and random workloads; also, by decoupling the durability and ordering requirements expressed by flush requests, Blizzard can commit writes out-of-order, providing high performance *and* crash consistency to applications that issue many small, random IOs. Blizzard's virtual disk drive, which clients mount like a normal physical one, provides maximum throughputs of 1200 MB/s, and can improve the performance of unmodified, cloud-oblivious applications by 2x–10x. Compared to EBS, a commercially available, state-of-the-art virtual drive for cloud applications, Blizzard can improve SQL server IOp rates by seven-fold while still providing crash consistency.

1 Introduction

As enterprises leverage cloud storage to process big-data workloads, there is increasing pressure to migrate traditional desktop and server applications to the cloud as well. However, migrating POSIX/Win32 applications to the cloud has historically required users to select from a variety of unattractive options. Databases like MySQL and email servers like Exchange can be trivially migrated to the cloud by running the server binaries inside of VMs that reside on cloud servers. Unfortunately, the storage abstractions exposed to those VMs lack the high performance and transparent scaling that are enjoyed by non-POSIX applications written for scale-out cloud stores like HDFS [6] and FDS [30]. For example, Azure and EC2 provide virtual disks that are backed by remote storage and that unmodified POSIX/Win32 applications can mount. However, each virtual drive only provides 50–250 MB/s of throughput [1, 28]; in contrast, a raw cloud store can provide more than a thousand MB/s to clients.

Datacenter operators provide "cloud-optimized" versions of a few popular applications like SQL and ActiveDirectory [3, 4, 15, 26, 27], implicitly acknowledging the difficulty of extracting cloud-scale performance from unmodified POSIX[1] applications. These cloud-optimized programs directly interface with the network storage using raw cloud APIs. This strategy provides higher performance than a naïve VM port, but such cloud-optimized applications offer fewer customization options than traditional POSIX/Win32 versions, making it difficult for users to tweak performance for individual workloads. More importantly, for the long tail of the application distribution, there are no pre-built, cloud-optimized versions. Large or technologically savvy companies may have the resources to write cloud-optimized versions of their applications, but for safety reasons, datacenter operators do not provide external developers with full access to the raw cloud APIs that are needed to maximize performance. Even if customers had such access, many customers would prefer to simply deploy their standard binaries to the cloud and automatically receive fast, scalable IO.

Unfortunately, desktop and server applications have significantly different IO patterns than traditional cloud-

[*] Work completed as a Microsoft intern; now at Google.

[†] Work completed as a Microsoft intern; now at NEC Labs.

[‡] Work completed as a Microsoft intern.

[§] Work completed as a Microsoft intern; now at Twitter.

[1] For conciseness, we use "POSIX" to mean "POSIX/Win32" in the rest of this paper.

scale applications. A typical MapReduce-style workload issues large, sequential IOs, but POSIX applications issue small, random IOs that are typically 32–128 KB in size [25, 41]. POSIX applications also require finer-grained consistency than many big-data workloads. For an application like an Internet-scale web crawl, append-at-least-once semantics [14] are often reasonable, and losing MBs of append-only data may be an acceptable risk for higher throughput; in contrast, for a POSIX database, compiler, or email server that is randomly accessing small blocks, the loss or duplication of just a few adversarially-chosen blocks can result in metadata inconsistencies and catastrophic data loss. To enforce fine-grained consistency, POSIX applications use disk flushes to order writes [8, 33, 39], but these flushes introduce write barriers. In the context of cloud storage, these barriers make it difficult for POSIX applications to issue large numbers of parallel writes to remote cloud disks. Lower disk parallelism leads to lower performance.

In this paper, we introduce Blizzard, a high-performance block store that exposes unmodified, cloud-oblivious applications to fast, scalable cloud storage. From the perspective of a client application, Blizzard's virtual disk looks like a standard SATA drive. However, Blizzard translates block reads and writes to parallel IOs on remote cloud disks, transparently handling the removal, addition, or failure of those remote disks.

Blizzard is not the first system to propose a virtual disk backed by remote storage [2,11,24]. However, Blizzard's virtual drive has several unique characteristics:

Locality-oblivious, full-bisection bandwidth block access: Blizzard is built atop a CLOS network with no oversubscription [17], i.e., arbitrary client/server pairs can exchange data at full NIC speeds without inducing network congestion. Blizzard also pairs each server disk with enough network bandwidth to read and write that disk at full sequential speed. These properties mean that clients can stripe their data across arbitrary remote disks in a locality-oblivious manner. This simplifies Blizzard's striping algorithm and permits very aggressive sharding (which results in better performance, since spreading data over more disks increases IO parallelism).

Nested striping: POSIX applications typically issue small, random IOs, but even when their IOs are large, client file systems often break such large operations into smaller pieces. A naïve stripe of a virtual drive across remote disks will cause *IOp convoy dilation*—as the small requests belonging to a single large operation travel from the client to a remote disk, the inter-request spacing will increase due to network jitter and scheduling vagaries on the client and the server. The longer the dilation, the less likely that the remote disk can use a single seek to handle all of the adjacent disk requests in the convoy. Blizzard uses a novel striping scheme called *nested striping* which avoids these problems. Nested striping ensures that convoy blocks are spread across multiple disks in parallel. This amortizes the seek costs for the individual blocks, and globally acts to prevent disk hotspots.

Fast flushes with prefix write commits: POSIX applications use disk flushes to order writes and provide crash consistency. However, such flushes restrict IO parallelism, and massive IO parallelism is the primary technique that clients must leverage to unlock cloud-scale IO performance. When Blizzard's virtual disk receives a flush request, it immediately acknowledges the flush to the client application, even though Blizzard has not made writes from that flush epoch durable. Asynchronously, the virtual drive issues writes in a way that respects *prefix epoch semantics*. If the client or the virtual disk crashes, the disk will always recover to a consistent state in which writes from different flush epochs will never be intermingled—all writes up to some epoch $N-1$ will be durable; some writes from epoch N may be durable; and all writes from subsequent epochs are lost. Blizzard's asynchronous writes lengthen the window for potential data loss, but they permits much higher levels of write performance. This approach also reduces the penalty for N-way data replication, since acknowledging a write no longer proceeds at the pace of the slowest replica for that write. Prior work has shown how prefix semantics can be added to the ext4 file system [7], but Blizzard shows how such semantics can be added at the disk level, *in a file-system agnostic manner*, and in a way that also provides high performance to applications that bypass the file system entirely and issue raw disk IOs.

Blizzard has several additional features, like support for disconnected operation[2], and tunable levels of disk parallelism (§2.4).

We have built a Blizzard prototype consisting of 1,200 disks and 150 servers. Using this prototype, we demonstrate that Blizzard can improve the performance of unmodified IO-intensive applications by 2x–10x. Importantly, our Blizzard prototype coexists alongside our FDS [30] deployment, using the same servers, disks, and networking equipment. FDS is optimized for large, business-scale computations. Thus, using Blizzard, cloud providers can leverage a single set of cluster hardware for both big-data computations and POSIX

[2] Not discussed further due to space constraints.

applications, reducing hardware outlays while consolidating administrative costs and providing fast, scalable IO to both types of workloads. From the perspective of developers, Blizzard allows POSIX applications to receive cloud-scale performance and availability without requiring datacenter operators to expose their raw, unsafe cloud APIs—instead, developers simply write applications under the assumption that (virtual) disk drives are extremely fast.

2 Design

Blizzard has two high-level goals. First, it has to run unmodified, cloud-oblivious POSIX applications on the same cloud infrastructure used by traditional big-data applications. Second, it must provide those POSIX applications with the storage performance, scalability, and availability that big-data programs receive. By "cloud-level performance," we mean that a single client should be able to issue hundreds of MBs of IO requests every second. By "cloud-level scalability and availability," we mean that client storage should transparently improve as the cloud operator adds more remote disks or better networking capacity. Also, administrative efforts that help big-data applications should also improve unmodified POSIX applications.

To satisfy these design goals, Blizzard must efficiently handle two aspects of POSIX workloads:

- POSIX applications typically generate small, random IOs between 32 KB and 128 KB in size [25,41]. To offer high performance to such seek-bound workloads, Blizzard needs to expose applications to massive disk parallelism. This allows applications to issue multiple operations simultaneously and overlap the seek costs.
- POSIX applications use the `fsync()` system call to control the ordering and durability of writes, ensuring consistency after crashes [7, 8, 33, 39]. An `fsync()` call generates a disk flush, and the disk flush acts as a write barrier, preventing writes issued after the flush from completing before all previous writes are durable. These write barriers limit disk parallelism, which results in poor performance. Thus, Blizzard needs to handle `fsync()` calls in a way that preserves notions of write ordering, but does not require clients to wait for synchronous disk events before issuing new writes.

In Section 2.3, we describe how Blizzard leverages full-bisection bandwidth networks to aggressively stripe each client's data across a large number of disks. In the absence of flush requests, this scheme would suffice to provide clients with massive disk parallelism. However, POSIX applications commonly issue flush requests. Thus, as described in Section 2.5, Blizzard uses eventual durability semantics [7] to remove synchronous disk operations from the flush path. When a client issues a flush, Blizzard records ordering information about pending writes; then, Blizzard immediately acknowledges the flush request. Asynchronously, Blizzard writes data to remote disks in a way that respects the client's ordering constraints, and allows the client to recover a consistent view after a crash. In the extreme, Blizzard can issue writes from multiple flush periods completely out-of-order (§2.5.3), removing all synchronization constraints involving writes, but still preserving consistency.

2.1 Blizzard's Storage Abstraction

Frameworks like pNFS [38] and BlueSky [42] expose cloud storage to cloud-oblivious applications by translating (say) NFS operations into operations on cloud disks. However, these systems lock applications into a particular set of file semantics which will not be appropriate for all applications. For example, some applications desire NFS's close-to-open consistency semantics, but other applications require POSIX-style consistency in which a newly written block is immediately visible to readers of the enclosing file [20]. Since all file systems eventually issue reads and writes to a block device (and since some applications issue raw disk commands), we decided to implement Blizzard as a virtual block device which stripes data across remote disks. This allowed us to support heterogeneous POSIX and Win32 file systems like ext3 and NTFS; it also allowed us to expose fast storage to applications like databases that issue raw block-level IOs.

2.2 The Low-level Storage Substrate

Blizzard stripes each virtual drive across several remote physical disks. As the striping factor increases, the virtual drive benefits from greater spindle parallelism (and thus higher IO performance). However, a traditional oversubscribed network constrains how aggressively Blizzard can stripe. In an oversubscribed network, the available cross-rack bandwidth is lower than the available intra-rack bandwidth; thus, for a system with medium-to-high utilization, clients access rack-local disks faster than they access disks in external racks. If Blizzard restricted each virtual drive to use rack-local disks, this would limit spindle parallelism, constrain the total capacity of the virtual drive, and makes job allocation more difficult, since a single job could not harness idle disks spread across multiple racks. However, if Blizzard allowed a virtual disk to span racks, contention in the oversubscribed cross-rack links would prevent the client from fully utilizing rack-external disks.

To avoid this dilemma, Blizzard uses Flat Datacenter Storage (FDS) as its low-level storage substrate [30].

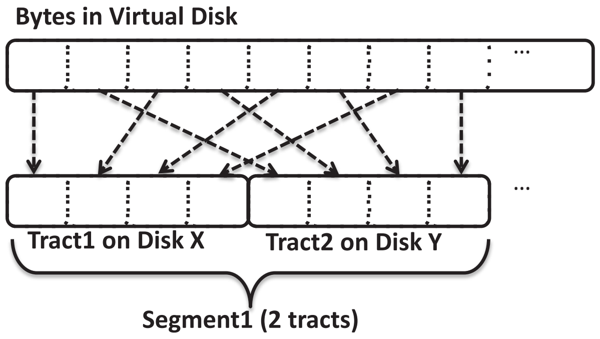


Figure 1: An example of nested striping with a segment size of 2. Virtual disk blocks are striped across tracks, and tracks are scattered across disks.

FDS is a datacenter-scale blob store that connects all clients and disks using a network with full-bisection bandwidth, i.e., no oversubscription [17]. FDS also provisions each storage server with enough network bandwidth to match its aggregate disk bandwidth. For example, a single physical disk has roughly 128 MB/s of maximum sequential access speed. 128 MB/s is 1 Gbps, so if a storage server has ten disks, FDS provisions that server with a 10 Gbps NIC; if the server has 20 disks, it receives two 10 Gbps NICs. The resulting storage substrate provides a locality-oblivious storage layer—any client can access any remote disk at maximum disk speeds[3].

A Blizzard virtual disk is backed by a single FDS blob (although Blizzard does not use a linear mapping between a virtual disk address and the corresponding byte in the FDS blob (§2.3 and §2.5)). FDS breaks each blob into 8 MB segments called tracts, and uses these tracts as the striping unit. Blizzard typically instructs FDS to stripe a blob across 64 or 128 physical disks; optionally, FDS performs N-way replication of each tract.

2.3 Data Placement

FDS provides Blizzard with some nice properties out-of-the-box, and allows Blizzard to satisfy the design goal of running on the same infrastructure that supports traditional big-data applications. However, FDS is optimized for large, sequential IOs. In particular, FDS divides each FDS blob into a series of large tracts. A tract resides in a contiguous location on a particular disk, but FDS scatters a blob's tracts across a large number of different disks. Tracts are 8 MB in size, and FDS's primary read/write interface works at the granularity of tracks. POSIX IOs are typically much smaller than 8 MB. Thus, we had to devise a way for Blizzard's virtual disk to map these small, random IOs onto FDS tracts (which FDS would then map to remote disks).

FDS natively supports a raw, low-level interface that lets applications read and write data in chunks that are smaller than a tract. This interface allows applications to process small pieces of tract metadata without having to read or write entire 8 MB tracts. Our first prototype of Blizzard's virtual disk used this interface, and a very simple block mapping, to translate virtual disk addresses to tract-level offsets. In this initial prototype, the virtual disk split its linear address space into contiguous, tract-sized chunks, and assigned each chunk to a separate FDS tract. A virtual disk IO with offset `byteOffset` would go to tract `byteOffset/TRACT_SIZE_BYTES`. This mapping was straightforward, but it led to disappointing performance. A convoy of sequential IOs often hit the same tract (and thus the same remote disk), eliminating opportunities for disk parallelism. Even worse, sequential convoys often experienced *dilation*. Even if the client used a tight loop to issue writes to adjacent offsets, the temporal spacing between those operations often grew as the operations traveled from client to remote disk. Client-side scheduling jitter increased the spacing, as did random network delays and scheduling jitter on the remote server. Thus, a sequential convoy that initially had little inter-request spacing at the client often arrived at the remote hard disk with larger inter-request gaps. In many cases, this prevented the remote disk from efficiently servicing the entire convoy with a single seek. Instead, the convoy operations were handled with multiple seeks and rotational delays, increasing the total IO latency for *all* operations on that disk.

Given these observations, we designed a new mechanism called *nested striping* that maps linear byte ranges to FDS tracts. We define a *segment* as a logical group of one or more tracts; a segment of N bytes contains striped data for a linear byte range of N bytes. Figure 1 demonstrates nested striping when each segment contains two tracts. Intuitively, increasing the segment size allows Blizzard to provide greater disk parallelism for sequential workloads. For example, a segment size of one restricts sequential IOs to one disk. As shown in Figure 1, a segment size of two spreads sequential IOs across two disks. Figure 1 also demonstrates why we call this striping scheme "nested": Blizzard stripes blocks across FDS tracts, and FDS distributes the tracts in a blob across many remote disks. By default, our Blizzard implementation uses a segment size of 128 tracts.

Experiments show that nested striping dramatically decreases the impact of convoy dilation. Using a segment size of 128 provides over 1000 MB/s of sequential read throughput (§4.1). In contrast, using a segment size of one tract results in sequential read throughput of 222 MB/s.

[3] This assumes that RTTs between clients and storage servers are much smaller than a seek time. We explore the impact of network latency in Section 4.6.

2.4 Role-based Striping

Up to this point, we have assumed that Blizzard is deployed in a shared, public cloud that is utilized by multiple customers. However, Blizzard can also be used in private, enterprise-local clouds. Indeed, one deployment model for Blizzard is a *building-scale deployment*—all of the rooms in the building are connected via a full-bisection bandwidth network, and all desktop and server machines use Blizzard virtual drives to store and manipulate data. Datacenters already deploy full-bisection networks at the scale of thousands of machines [17], so this deployment model is technically feasible, and it would allow an enterprise to consolidate storage and IT effort for a particular building.

In such a building-scale deployment, different users will have different performance needs. For example, a programmer that runs large compilations requires better storage performance than a receptionist who mainly sends emails and performs word processing. For building-wide Blizzard deployments, segment sizes offer a convenient knob that administrators can use to control the amount of disk parallelism that Blizzard exposes to different users with different roles.

2.5 Write Semantics

Blizzard's virtual disk provides three types of data consistency: write-through commits, flush epoch commits with fast acknowledgments, and out-of-order commits with fast acknowledgments. We describe each approach below. Note that, when we say that Blizzard "acknowledges" an operation to a "client," the client is either a file system or an application that issues raw disk IOs. A write request or a flush request is "acknowledged" when that operation returns to the client.

2.5.1 Write-through

In this mode, Blizzard does not acknowledge a virtual disk write until the associated FDS operation has become durable on the relevant remote disk. This approach provides the smallest window of potential data loss if a crash occurs. However, write-through consistency provides the lowest performance, since a thread which issues a blocking write must wait for that write to become durable before issuing additional IOs.

2.5.2 Flush epoch commits with fast acks

Let a *flush epoch* be a period of time between two flush requests from the client. Each flush epoch contains one or more writes. A flush epoch is *issued* if all of its writes have been sent to remote disks. The epoch is *retired* if all of those writes have been reported as durable by the remote disks.

Setup: Blizzard maintains a counter called `epochToIssue`; this counter starts at 0 and represents which writes Blizzard can send to FDS. Blizzard maintains another counter called `currEpoch` that also starts at 0. This counter represents the total number of flush requests that the client has issued. As explained below, `currEpoch` will often be greater than `epochToIssue`.

Acknowledging writes: When Blizzard receives a write request or a flush request, it immediately acknowledges that operation, allowing the client to quickly issue more IO requests. If the incoming operation was a flush, Blizzard increments `currEpoch` by 1. If the operation was a write, Blizzard tags the write with the `currEpoch` value and places the write request in a queue that is ordered by epoch tags.

Draining the write queue: Once a new write is enqueued and acknowledged to the client, Blizzard tries to issue enqueued writes to remote disks. Blizzard iterates from the front of the write queue to the end, i.e., from the oldest unretired epoch to the newest. If the currently examined write is from `epochToIssue`, Blizzard dequeues the write and issues it immediately; otherwise, Blizzard terminates the queue traversal. Later, when a write from epoch N completes, Blizzard checks whether epoch N has now retired. If so, Blizzard increments `epochToIssue` and tries to release new writes from the head of the write queue.

When Blizzard issues a write, it removes it from the write queue. However, Blizzard keeps the write in a separate cache until the write is durable. Meanwhile, if a read arrives for the write's byte range, Blizzard services the read using the cached, fresh data, instead of issuing a read to the underlying remote disk and possibly getting old data.

Consistency semantics: In this consistency scheme, Blizzard treats a flush as an ordering constraint, but not a durability constraint; using the terminology of optimistic crash consistency, Blizzard provides "eventual durability" [7]. This means that Blizzard issues writes in a way that respects flush-order durability, but a flush epoch may retire at an arbitrarily long time after the flush was acknowledged to the client. Indeed, the epoch may never retire if the client crashes before it can issue the associated writes. However, the rebooted client is guaranteed to see a consistent prefix of all writes that were acknowledged as flushed; this suffices for many applications [9].

2.5.3 Out-of-order commits with fast acks

To maximize the rate at which writes are issued, Blizzard defines a scheme that allows writes to be acknowledged immediately *and* issued immediately, regardless of their flush epoch. This means that writes may become durable out-of-order. However, Blizzard enforces prefix consistency using two mechanisms. First, Blizzard abandons nested striping and uses a log structure to avoid updating blocks in place; thus, if a particular write fails to become durable, Blizzard can recover a consistent version of the target virtual disk block. Second, even though Blizzard issues each new write immediately, Blizzard uses a deterministic permutation to determine which log entry (i.e., which `<tract,offset>`) should receive the write. To recover to a consistent state after a crash, the client can start from the last checkpointed epoch and permutation position, and roll the permutation forward, examining log entries and determining the last epoch which successfully retired.

Setup: Let there be V blocks in the virtual disk, where each block is of equal size, a size that reflects the average IO size for the client (say, 64 KB or 128 KB). The V virtual blocks are backed by $P > V$ physical blocks in the underlying FDS blob. Blizzard treats the physical blocks as a log structure. Blizzard maintains a `blockMap` that tracks the backing physical block for each virtual block. Blizzard also maintains an `allocationBitMap` that indicates which physical blocks are currently in use. When the client issues a read to a virtual block, Blizzard consults the `blockMap` to determine which physical block contains the desired data. Handling writes is more complicated, as explained below.

Blizzard maintains a counter called `currEpoch`; this counter is incremented for each flush request, and all writes are tagged with `currEpoch`. Blizzard also maintains a counter called `lastDurableEpoch` which represents the last epoch for which all writes are retired.

The virtual-to-physical translation: When Blizzard initializes the virtual disk, it creates a deterministic permutation of the physical blocks. This permutation represents the order in which Blizzard will update the log. For example, if the permutation begins 18, 3,..., then the first write, *regardless of the virtual block target*, would go to physical block 18, and the second write, regardless of the virtual block target, would go to physical block 3. Importantly, Blizzard can represent a permutation of length P in $O(1)$ space, not $O(P)$ space. Using a linear congruential generator [44], Blizzard only needs to store three integer parameters (a, c, and m), and another integer representing the current position in the permutation. As we will describe later, the serialized permutation will go into the checkpoints that Blizzard creates.

Handling reads is simple: when the client wants data from a particular virtual block, Blizzard uses the `blockMap` to find which physical block contains that data; Blizzard then fetches the data. Handling writes requires more bookkeeping. When a write arrives, Blizzard iteratively calls the deterministic permutation function, and immediately sends the write to the first physical block that is not marked in the `allocationBitMap` as used. However, once the write is issued, Blizzard does *not* update the `allocationBitMap` or the `blockMap`—those structures are reflected into checkpoints, so they can only be updated in a way that respects prefix semantics. So, after Blizzard issues the write, it places the write in a queue. Blizzard uses the write queue to satisfy reads to byte ranges with in-flight (but possibly non-durable) writes. When a write becomes durable, Blizzard checks whether, according to the permutation order, the write was the oldest unretired write in `lastDurableEpoch`$+1$. If so, Blizzard removes the relevant write queue entry, and updates `blockMap` and `allocationBitMap`. Otherwise, Blizzard waits for older writes to commit first. Once all writes in the associated epoch are durable, Blizzard increments `lastDurableEpoch`.

When Blizzard issues a write to FDS, it actually writes an *expanded block*. This expanded block contains the raw data from the virtual block, as well as the virtual block id, the write's epoch number, and a CRC over the entire expanded block. As we explain below, Blizzard will use this information during crash recovery.

If the client issues a write that is smaller than the size of a virtual block, Blizzard must read the remaining parts of the virtual block before calculating the CRC and then writing the new expanded block. This read-before-write penalty is similar to the one suffered by RAID arrays that use parity bits. This penalty is suffered for small writes, or for the bookends of a large write that straddles multiple blocks. For optimal performance, Blizzard's virtual block size should match the expected IO size of the client. For example, POSIX applications like databases and email servers often have a configurable "page size"; these applications try to issue reads and writes that are integral multiples of the page size, so as to minimize disk seeks. For these applications, Blizzard's virtual block size should be set to the application-level page size. For other applications that 1) frequently generate writes that are not an even multiple of Blizzard's block size, or 2) generate writes that are not aligned on Blizzard's block boundaries, Blizzard should be configured to use write-through mode, or fast acknowledgment mode with nested striping (§4.7).

Checkpointing: Periodically, the client checkpoints the `blockMap`, the `allocationBitMap`, the four permutation parameters, `lastDurableEpoch`, and a CRC over the preceding quantities. For a 500 GB virtual disk, the checkpoint size is roughly 16 MB. Blizzard does not update the checkpoint in place; instead, it reserves enough space on the FDS blob for two checkpoints, and alternates checkpoint writing between the two locations.

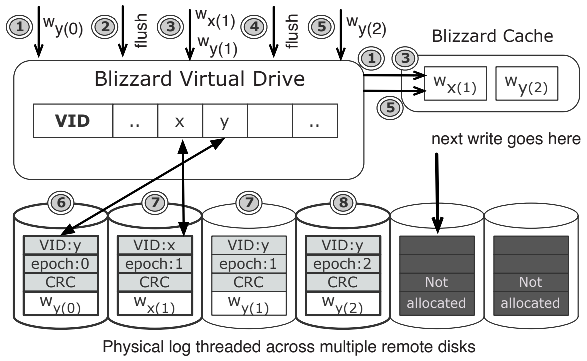


Figure 2: An example of Blizzard's log-based, out-of-order commit scheme. Disks with thick borders contain durable writes. See the main paper text for an explanation of what happens at each time step.

Recovery: To recover from a crash, Blizzard loads the two serialized checkpoints and initializes itself using the checkpoint with the highest lastDurableEpoch and a valid CRC. Blizzard then rolls forward from the permutation position in the checkpoint, scanning physical blocks in permutation order. Since Blizzard issues log writes in epoch order, the recovery scan will process writes in epoch order. If the allocationBitMap says that the current physical block is in use, Blizzard inspects the next physical block in the permutation. If the allocationBitMap says that the current physical block is *not* in use, Blizzard inspects the physical block's epoch number. If the number is less than lastDurableEpoch, Blizzard terminates roll-forward. If the CRCs from all of the physical block's replicas are inconsistent or do not match each other, Blizzard terminates roll-forward. Otherwise, Blizzard updates the allocationBitMap to mark the physical block as used (and the old physical block for the virtual block as unused); Blizzard also updates the relevant blockMap entry to point to the physical block. Finally, Blizzard sets lastDurableEpoch to the epoch value in the physical block. The permutation position at which roll-forward terminates will be the position to which Blizzard sends the next write.

Example: Figure 2 provides an example of Blizzard's log-based, out-of-order commits. For simplicity, this example uses the permutation generator (writeNumber % logSize), such that physical log entries are scanned in linear order, from left to right. At time (1), a write arrives for virtual block Y. Blizzard issues the write to remote storage immediately, and places the write in an in-memory cache, so that reads for Y's data can access the fresh data without having to wait for the remote write to finish. A flush request arrives at (2), which Blizzard acknowledges immediately. The current flush epoch is now 1. At (3), writes to virtual blocks X and Y arrive. Blizzard acknowledges those writes immediately, issuing them in parallel to the next positions in the log, and updating the write cache entries for blocks X and Y (in the latter case, overwriting the old cache value for Y). At (4), another flush arrives, and Blizzard increments the flush epoch to 2. At (5), another write arrives for Y, causing Blizzard to issue a new write to the next position in the log, and updating Y's write cache value. At (6), the first write to Y becomes durable on a remote disk, causing Blizzard to update the blockMap entry for Y to point to that log entry. At time (7), the write to X becomes durable, and Blizzard updates the blockMap appropriately. However, at time (7), the second write to Y has not committed. At time (7), the client takes a checkpoint (note that the last permutation index that the client knows is durable is the second log entry). At time (8), the client learns that the third write to Y is durable. However, since the second write to Y is not durable yet, the client does not change the blockMap–thus, virtual block Y still points to the write from epoch 0.

Suppose that the client crashes immediately after it makes a checkpoint at (7). Further suppose that this crash prevents the write to the third physical block from becoming durable, e.g., because the client needed to retry the write, but crashed before it could do so. After the client reboots, it looks at the checkpoint and extracts the last permutation index known to be durable (log entry two). The client then rolls forward through the log in permutation order. The client examines the third physical log entry and sees that it is marked as unused by the allocationBitMap. The client examines the entry's epoch number and CRC. Since the associated write failed, one or both of those quantities will have invalid values. At this point, the client stops the roll-forward. Even though the write to the fourth log block completed, that write is lost to the client. However, the client has recovered to a prefix-consistent view of the virtual block Y (and the rest of the disk).

IOp dilation: Even though log-based consistency does not use nested striping, the linear congruential generator produces striping patterns that "jump around" enough to prevent convoy dilation. Adversarial write patterns can still result in dilation, but such patterns are rare in practice.

2.6 Application-perceived Consistency

In write-through mode, Blizzard minimizes application-perceived data loss. Since all writes are synchronous and go directly to remote disks, writes can only be lost if the application transfers write data to the virtual drive, but the drive or the client machine crashes before Blizzard can write the data to the remote disks. Once Blizzard has acknowledged a write operation to the client, the client knows that the write data is durable.

For the fast acknowledgment schemes, Blizzard trades higher performance for an increased possibility of data loss. In these schemes, if a crashed client issued writes belonging to flush epochs $F_0, F_1, \ldots, F_N$, the client will recover to a virtual disk which contains all writes from epochs $F_0 \ldots F_R$; some, all, or no writes from epoch F_{R+1}; and no writes from epochs $F_{R+2} \ldots F_N$. Denote these three write segments as the preserved epochs, the questionable epoch, and the disavowed epochs, respectively. With traditional flush semantics for a local physical disk, there are no disavowed epochs—clients cannot issue new writes across a flush barrier if prior writes are outstanding, so, at worst, a client can lose some or all writes from its last, questionable epoch. With Blizzard's fast acknowledgment schemes, $N-R$ may be greater than zero, i.e., there may be a questionable epoch and disavowed epochs.

When operating in fast acknowledgment mode, Blizzard can minimize data loss (i.e., $N-R$) by issuing writes as quickly as possible; the log-based write scheme does precisely this. However, for all three of Blizzard's write schemes, unmodified POSIX file systems and applications will always recover to a prefix-consistent version of the Blizzard drive. Using fast acknowledgments, applications are more likely to share acknowledged (but currently non-durable) writes to external parties, meaning that, if the application crashes, it may not be able to recover those externalized writes from the Blizzard drive. In practice, we do not believe that this is a problem, since many users are willing to receive high performance and crash consistency in exchange for potentially externalizing non-durable data [9]. Users that wish to never externalize non-durable data can run Blizzard in write-through mode.

2.7 Server-side Failure Recovery

Blizzard relies on FDS to recover failed tracts belonging to a virtual drive. However, FDS clients are responsible for retrying aborted writes caused by remote disk failures. Thus, if Blizzard detects that an FDS write was unsuccessful, it must contact the FDS metadata server, download the new mapping between tracts and remote disks, and retry the write. If replication is enabled, Blizzard also ensures that each write to a virtual block results in R successful writes to the R replica disks.

2.8 Coexisting Workloads

Blizzard is built atop FDS, and both systems use a shared physical infrastructure of servers, disks, and network equipment. The network provides full-bisection bandwidth, and FDS uses a request-to-send/clear-to-send mechanism which ensures that senders cannot overrun receivers [30]. Thus, neither Blizzard workloads nor native FDS applications can induce network congestion at the core or the edge. This lack of congestion does not guarantee any notion of application-level "network fairness," so operators that desire such properties must use client-side mechanisms like admission control or rate limiting.

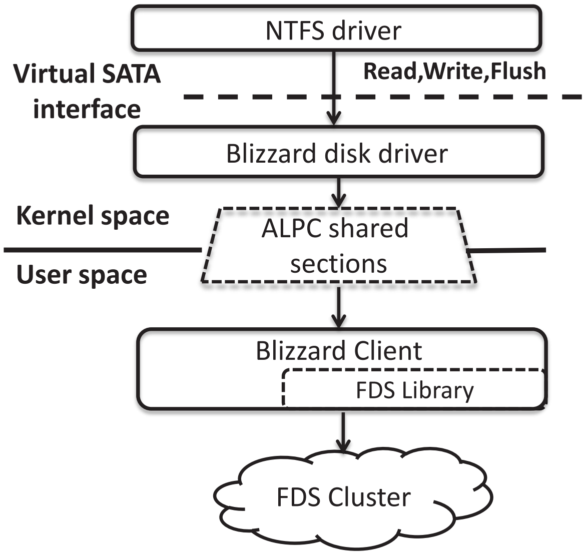


Figure 3: A Blizzard virtual disk on Windows.

Both Blizzard and FDS use aggressive, randomized striping across disks. As the aggregate client IO pressure increases, service times at each disk degrade gracefully, since the workload is spread evenly across each disk (§4.5). In principle, a single disk can store data for both Blizzard applications and native FDS applications. In practice, we typically allocate a single disk to either Blizzard or native FDS applications, but not both. POSIX applications often require low-latency IO in addition to high throughput, so our allocation scheme prevents small POSIX IOs from getting queued behind the much larger IOs from big-data applications.

3 Implementation

Figure 3 shows the architecture for our Blizzard implementation on Windows. The virtual disk contains two pieces: a kernel-mode SATA driver, and a user-mode component which links to the FDS client library. File systems (and applications which issue raw disk IO) send IO request packets (IRPs) to the SATA driver. The driver forwards these requests to the user-mode client, which translates the requests into the appropriate FDS operations. For IRPs that correspond to reads and writes, Blizzard issues reads or writes to the appropriate remote disks. Once the client receives a response from a remote disk, it hands the response to the kernel mode driver, which then informs the requesting application that the IRP has completed. To minimize the overhead of exchanging data across the user-kernel boundary, Blizzard uses Window's Advanced Local Procedure Calls (ALPC); ALPC provides zero-copy IPC using shared memory pages.

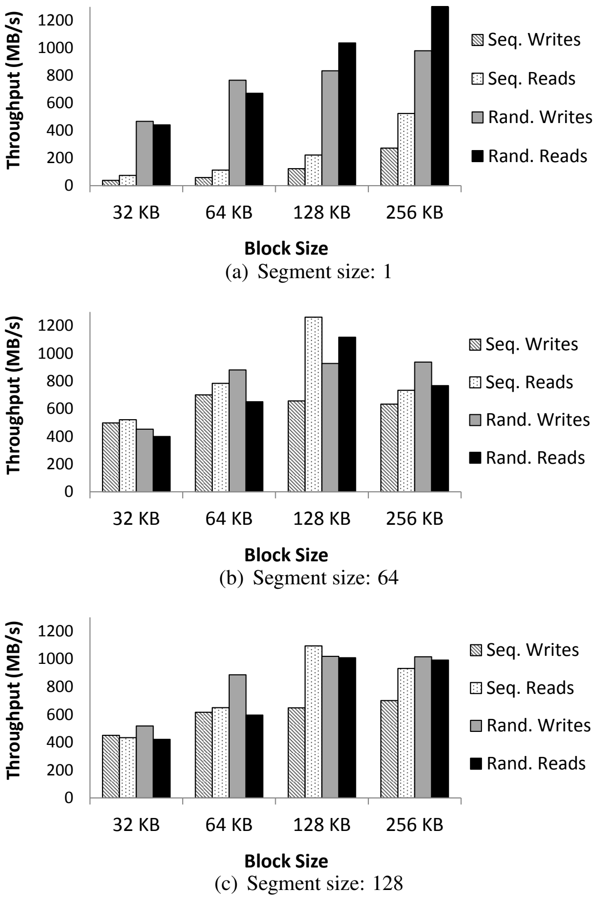
To maximize performance, Blizzard uses asynchronous interfaces to exchange data between the kernel driver, the FDS client, and the FDS cluster. The user-mode component of the virtual disk is multi-threaded, and it maximizes throughput by handling multiple kernel requests and FDS operations in parallel. Such high levels of concurrency and asynchrony make it tricky to preserve the prefix semantics discussed in Section 2.5. In terms of wall-clock time, reads and writes arrive at the user-mode component in the order that the kernel issued them. However, the user-mode component handles each read and write in a separate thread; lacking guidance from the kernel, writes might issue to FDS in a nondeterministic fashion, based on whichever user-mode write threads happened to grab more CPU time. To allow the user-mode component to implement prefix semantics, the kernel driver tags each write request with its flush epoch, its sequence number within that epoch, and the maximum sequence number for writes from the *previous* epoch. This way, different user-mode threads can use lightweight interlocked-increment operations on integers to track the number of writes that have issued for each epoch. Blizzard does eventually require heavyweight locking to add writes to the write queue (§2.5), but this locking takes place in the latter part of the write path, leaving the first part contention-free.

4 Evaluation

In this section, we use a variety of experiments to demonstrate that Blizzard provides low-latency, high-throughput IO to unmodified, cloud-oblivious applications. Unless stated otherwise, when we refer to "Blizzard in fast acknowledgment mode," we refer to the second consistency scheme in Section 2.5, not the log-based approach.

4.1 Microbenchmarks

Figure 4 depicts the raw performance of a Blizzard virtual disk backed by 128 remote disks and using single replication. To generate these results, we ran a custom client program that issued asynchronous, block-level reads and writes to the virtual disk as quickly as possible. Blizzard was configured in write-through mode, to identify the steady-state performance that Blizzard could provide to a completely IO-bound client. The results show that, depending on the block size and the segment size, Blizzard can provide throughputs of 700 MB/s for sequential writes, and over 1000 MB/s for sequential reads, random reads, and random writes.



(a) Segment size: 1

(b) Segment size: 64

(c) Segment size: 128

Figure 4: Throughput microbenchmarks.

From the perspective of a client, increasing the segment size is roughly analogous to adding disks to a private RAID-0 array—it increases the number of disks that a client can access in parallel, improving both performance and load balancing. As shown in Figure 4(a), small segment sizes lead to poor sequential IO performance due to convoy dilation effects (§2.3). However, even for a segment size of one, Blizzard services *random* IOs at 400 MB/s or faster. This is because, at any given time, a random workload accesses more disks than a sequential workload, improving disk parallelism (and thus aggregate throughput). In the rest of this section, unless otherwise specified, all experiments use a segment size of 128, and 128 backing disks.

Increasing the block size beyond 32 KB improves performance, since disks can fetch more data per seek. However, increasing the block size beyond 128 KB leads to diminishing throughput returns.

Figure 5 compares Blizzard's write latency under several consistency settings and replication levels. For this experiment, we intentionally included some old, slow

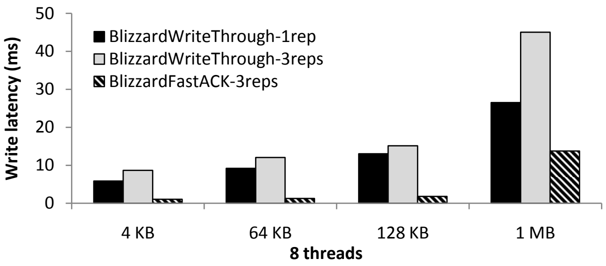


Figure 5: Write latency: Blizzard in write-through mode (1 and 3 replicas) and Blizzard in fast acknowledgment mode (3 replicas).

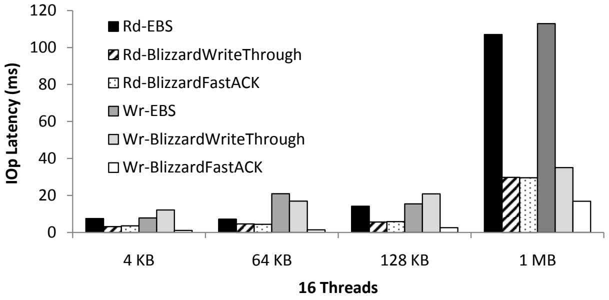


Figure 6: IOp latency: EBS and Blizzard.

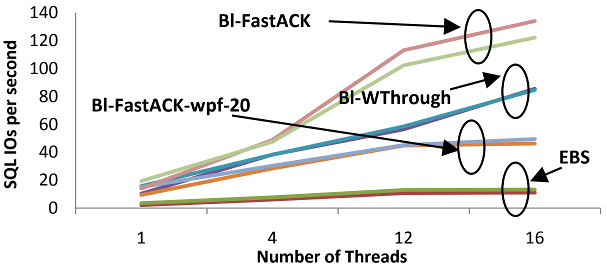


Figure 7: SQL read and write IOs per second: EBS, Blizzard in write-through mode, Blizzard in fast acknowledgment mode, and Blizzard in fast acknowledgment mode with forced epochs every 20 writes. Each pair of circled lines represents the read and write speeds for a particular configuration.

disks that we typically exclude from production workloads. Figure 5 shows that fast acknowledgments dramatically reduce the cost of data redundancy—despite the presence of known-slow disks, the write latency of triple replication with fast acknowledgments is 2x–5x lower than the latency of single replication with write-through semantics. Blizzard clients obviously cannot issue an infinite number of low-latency IOs, since, at some point, some resource (e.g., the network or a remote disk) will become saturated. However, our results show that, until that point is reached, Blizzard's fast acknowledgments provide very low-latency IOs.

4.2 Blizzard vs. EBS

In this section, we compare Blizzard's virtual drive to Amazon's Elastic Block Store (EBS) drive. Our EBS deployment used one Amazon EC2 instance (an EBS-optimized m1.xlarge) which had 4 CPUs and 15 GB of memory, and which ran 64-bit Windows 2008 R2 SP1 Datacenter Edition. The virtual EBS drive was backed by 12 disks, each of which was provisioned for 100 IOps and 10 GB of storage; we constructed a RAID-0 striped volume as the backing storage for the EBS drive. The disks and the EC2 instance were connected by a 1 Gbps network connection. To provide a fair comparison to Blizzard, we configured Blizzard's virtual drive to use 12 backing disks, and we used FDS's built-in rate-limiter to restrict the Blizzard client to 1 Gbps of network bandwidth. The Blizzard client had 4 CPUs and 12 GB of RAM, similar to the EBS client.

Using a multithreaded synthetic load generator, we tested IO latency in EBS, Blizzard in write-through mode, and Blizzard with fast acknowledgments enabled. Since the load generator did not generate flush requests, the latter Blizzard configuration provided good performance, but not prefix consistency; this configuration is still a useful one to investigate, because many people disable disk flushes for the sake of performance [7, 31]. Figure 6 shows that the read latencies for both Blizzard schemes were 2x–4x lower than EBS's latency. Both Blizzard schemes had similar read latencies because delayed durability tricks do not directly affect the completion times of reads (although a client may generate more reads per second if writes require less time to complete).

For write latencies, Blizzard with fast acknowledgments was 7x–14x faster than EBS. Blizzard in write-through mode was essentially equivalent to EBS for small to medium operations, but much faster for 1 MB writes. It is unclear to us why EBS was so much slower for large writes. Throughput tests (which we elide due to space constraints) showed that, even for large IO sizes, EBS only utilized about 80% of the available network bandwidth; thus, the EBS client may be exchanging control traffic with other nodes that we cannot see.

Figure 7 shows the performance of `sqliosim`, a popular SQL benchmark tool, on EBS and several different configurations of Blizzard. We used `sqliosim` in a configuration that had several threads doing random IO to 4 databases in parallel. Each database was 4 GB in size, with its own log file. In the background, read-ahead and bulk updates occurred. Note that `sqliosim` uses write-through IO, not flushes, to provide consistency, so Blizzard with fast acknowledgment simply writes data to remote disks as quickly as possible and provides no crash consistency. We also ran a variant of fast ac-

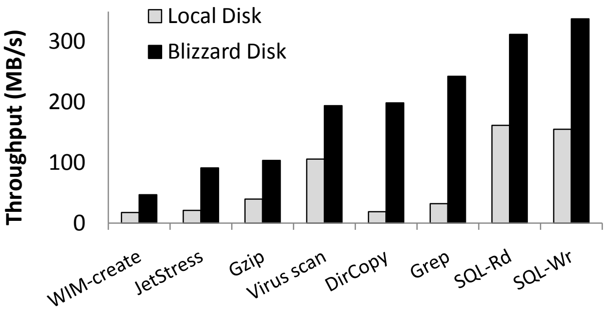


Figure 8: Macrobenchmarks: Blizzard's virtual drive (write-through, single replication) versus a local physical drive.

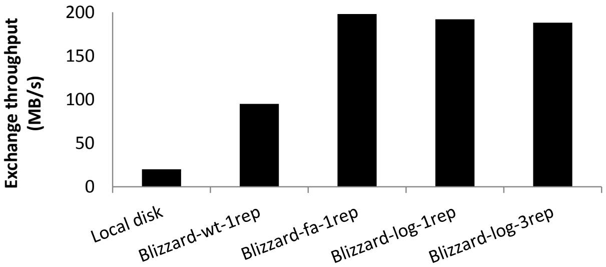


Figure 9: Exchange throughput. For the log-based commit results, Blizzard's block size was set to 64 KB, to match the size of Exchange's transactional IOs. Other Blizzard configurations used a block size of 128 KB.

knowledgment mode which inserted a fake flush every 20 writes. That variant, which immediately acknowledges writes but bounds data loss to 20 writes, performs slower than Blizzard with write-through or vanilla fast acknowledgments; this is because the fake flushes reduced the amount of write parallelism. Nonetheless, all three versions of Blizzard issued significantly more IOps than EBS.

4.3 Macrobenchmarks

Figure 8 shows Blizzard's performance for several IO-intensive Win32 workloads; Blizzard was configured in write-through mode and used single replication. The `WIM-create` workload represents the time needed to generate a bootable WinPE [29] .iso file using the Windows Automated Installation Kit (a WinPE image is a minimal Windows installation that is useful for creating recovery CDs and other diagnostic tools). `JetStress` [23] is a seek-intensive application with 16 threads that emulates the IO load of an Exchange email server. `Gzip` is a file compression program that uses four IO threads. `Virus scan` represents the throughput of a full system analysis performed by System Center 2012 Endpoint. `DirCopy` is derived from the built-in Windows `robocopy` tool, and it recursively copies directories using eight threads. `Grep` is an eight-way threaded program that evaluates regular expressions over file data. `SQL` refers to the `sqliosim` tool that database administrators use to characterize disk performance. By default, the tool launches eight threads that perform random database queries, eight threads that perform sequential queries, and eight threads that perform bulk database updates.

In Figure 8, throughput numbers refer to application-level performance, not disk-level performance. For example, `Gzip` throughput refers to how many MBs of file data the program can compress per second. Similarly, `SQL` throughput refers to how quickly the SQL engine can read or write the database. Figure 8 shows that Blizzard can improve unmodified application's performance by a factor of 2x–10x. Blizzard provides the greatest boost to programs like `DirCopy` and `Grep` which spend very little time on CPU operations.

4.4 Log-based Commit

To test the performance of Blizzard's out-of-order, log-based commit scheme (§2.5), we ran several tests involving the `JetStress` tool, which emulates the workload for an Exchange email server. Unlike `sqliosim`, which issues write-through operations to implement consistency, `JetStress` uses disk flushes. As shown in Figure 9, Blizzard in write-through mode with single replication provides a 4x throughput improvement over a local physical disk, and Blizzard in single replicated, fast acknowledgment mode provides a 9x improvement. Using single replication, Blizzard's log-based, out-of-order commit scheme was only 3% slower than single-replicated fast acknowledgment, despite occasionally needing to perform read-before-writes (§2.5.3). The triple-replicated log scheme was only 5% slower, since Blizzard could hide much of the latency associated with slow replicas (§4.1).

Log-based commits do not provide faster throughput or lower IO latency than simple fast acknowledgments because both schemes acknowledge writes and flushes immediately. However, the log-based scheme *issues* all writes immediately, whereas the simple scheme issues writes in epoch order, waiting for writes from epoch N to commit before issuing writes from epoch $N+1$. Thus, the simple scheme is more prone to data loss in the case of a client crash, since it buffers more writes in-memory than the log-based scheme (§4.7). The increased buffering requirement will also cause client-submitted IO requests to block more often, until Blizzard can deallocate memory belonging to newly retired epochs.

4.5 Multiple Active Clients

Blizzard clients stripe their data across a shared set of disks. As the number of active clients grows, the aggre-

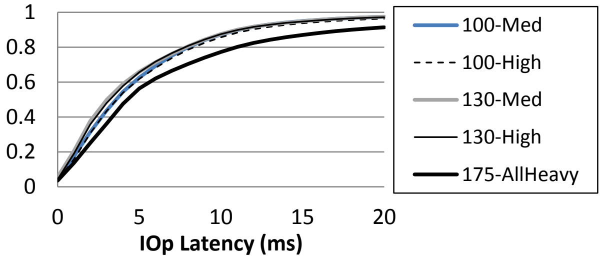


Figure 10: CDF of IOp latency (VDI workloads). The legend format is XXX-YYY, where XXX is the number of clients and YYY describes the client IOp rates. There were 130 remote disks.

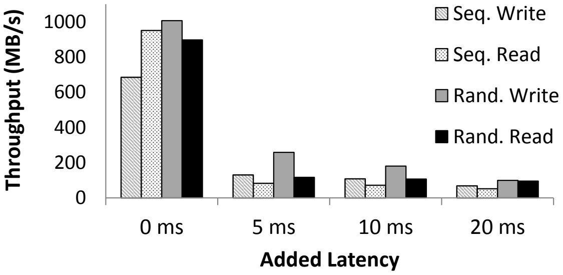


Figure 11: Comparing Blizzard's performance in our deployed network (far left) and our deployed network with synthetic latencies added.

gate request pressure on the disks increases. We designed nested striping (§2.3) and log-based permutation maps (§2.5) so that clients would spread their requests across multiple disks, preventing hotspots from emerging and leading to graceful degradation of disk IO queues. To test our design, we issued IO requests using a synthetic load generator that simulated a variable number of clients with varying levels of IOs per second (IOps). Each emulated client received 10 Gbps of network connectivity, and each client's virtual user was marked as "Light," "Normal," "Power," or "Heavy" based on its IOps rate (5, 10, 20, or 40 respectively). The size of each IOp ranged from 512 bytes to 1 MB, with the statistical distribution of IOp sizes and read/write ratios governed by empirical studies of VDI workloads [12, 13]. Note that a single high-level IOp resulted in multiple FDS operations if the IOp was larger than a Blizzard virtual block.

Figure 10 provides a CDF of IOp latencies for several different client deployments; to measure true IOp latencies, Blizzard was operated in write-through mode. The "Medium" deployment was 10% Light, 50% Normal, 25% Power, and 15% Heavy. The "High" client split was 10%/30%/40%/20%, and the pessimistic "AllHeavy" split was 0%/0%/0%/100%. In all cases, there were 130 remote disks. Figure 10 shows that, with 100 clients (i.e., more than one disk per client) and 130 clients (exactly one disk per client), Blizzard provides IOp latencies that are competitive with those of a local physical disk: at least 62% of IOps had 5 ms of latency or lower, and 85% of IOps had 10 ms of latency or lower.

Interestingly, latencies in the 130 client test were slightly *lower* than those in the 100 client test, e.g., in the "High" IOps test, 65.7% of IOps in the 130 node deployment had 5 ms or less of latency, but this was true for only 61.9% of IOps in the 100 node test. The reason is that nested striping distributes the aggregate client load evenly across all disks. Thus, each disk sees a random stream of seek offsets. When the number of clients increased from 100 to 130, disk queues got deeper, but this was *better* for disks that were serving random workloads—as each disk arm swept across its platters, there were more opportunities to service queued IO requests. Of course, longer disk queues will eventually increase IOp latency, as demonstrated by the pessimistic "AllHeavy" deployment which had 175 clients (i.e., 1.34 clients for each disk) and 40 IOps per client. Even in this case, 56% of IOs had latencies no worse than 5 ms, and 77% had latencies no worse than 10 ms.

4.6 Latency Sensitivity

Blizzard is designed for full-bisection networks in which clients have fast, low-latency connections to remote disks. For example, in our current deployment, clients and storage nodes communicate via links with 500 microseconds of latency. This is an order of magnitude smaller than the seek time for a disk, allowing Blizzard to make network-attached disks as fast to access as local ones. Figure 11 depicts Blizzard's performance with synthetic network latencies added, and with write-through semantics enabled (i.e., the virtual drive does not acknowledge writes to clients until those writes are durable on remote disks). With five milliseconds of additional latency, Blizzard's throughput drops by a factor of 5x–10x, and with twenty milliseconds of additional latency, performance is essentially equivalent to that of a single local disk.

These experiments highlight the importance of Blizzard's congestion-free, full-bisection bandwidth network. In such a system, network delays are negligible, and the client-perceived latency for a write-through IO is governed by the time needed to perform a single seek on a remote storage server (see Figure 10). Figure 11 shows that if the storage network lacks a fast interconnect, then millisecond-level network latencies effectively double or triple the seek penalty for accessing a remote disk. In these scenarios, Blizzard's fast acknowledgment schemes are crucial for eliminating remote access penalties from the critical path of writes.

Write workload	Crash-consistent recoveries		
	Write-through	FastACK	FastACK + log
Only new physical blocks targeted	50/50	50/50	50/50
50% new targets, 50% overwrites	50/50	50/50	50/50
JetStress	50/50	50/50	50/50

Figure 12: For all 450 injected crashes, Blizzard recovered to a prefix-consistent version of the virtual disk.

4.7 Reliability

In this section, we demonstrate two things. First, Blizzard always recovers a crashed virtual drive to a prefix-consistent state. Second, if a Blizzard client primarily issues block-aligned writes for entire blocks of data, the client should use log-based commit to reduce data loss and decrease memory pressure. If clients frequently issue writes that are misaligned, or not an even multiple of Blizzard's block size, clients should use the simple fast acknowledgment scheme (if they wish to maximize performance), or write-through mode (if they wish to minimize data loss).

Recovering to consistent virtual drives: To test Blizzard's reliability in the presence of crashes, we modified the user-mode portion of the virtual driver so that it randomly injected emulated failures. At each emulated failure point, the driver logged the set of writes that were buffered in memory, as well as the subset of those writes that had been issued to remote disks, but not yet acknowledged as being durable. Writes that are buffered but unissued at crash time are lost. Writes that are issued but unacknowledged at crash time may or may not be durable—their durability depends on whether the client OS had actually sent the writes over the network before the crash, and whether remote disks crashed while handling the writes, and so on.

We used three synthetic workloads to explore Blizzard's reliability. Our first workload issued 40 writes per second, such that each write targeted a previously unwritten portion of the virtual disk. Our second workload also issued 40 writes per second, but 50% of the writes targeted new locations, and 50% targeted previously written blocks. Note that a write-only workload of 40 IOps is intense for a POSIX application [12, 13]. We configured Blizzard to use a block size of 128 KB, with half of the writes being 64 KB in size, and the other half being 128 KB, ensuring that, when Blizzard was run in log-based commit mode, the read-after-write code paths would be stressed. Our final workload was JetStress [23], a seek-intensive workload that simulates an Exchange email server. The JetStress tests also used a Blizzard block size of 128 KB, and in all experiments, the virtual drive used 128 backing disks.

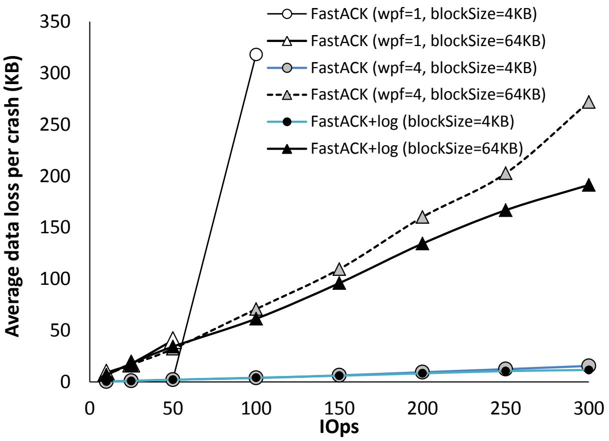


Figure 13: Blizzard loss rates when the size of each write is aligned with Blizzard's block size. "wpf" means "writes per flush."

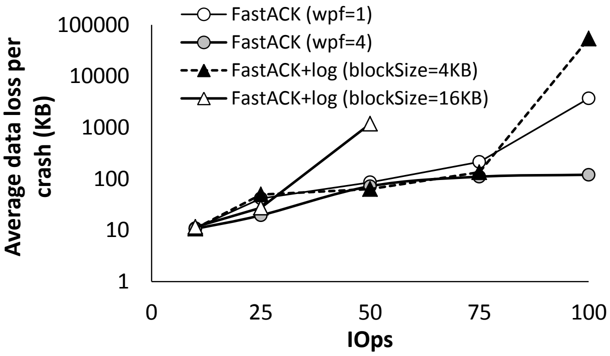


Figure 14: Blizzard loss rates for a simulated write-only VM workload (writes vary between 4KB and 1 MB in size). Note that the y-axis is log-scale.

Figure 12 shows the results of our experiments. For all 450 injected crashes, Blizzard recovered a prefix-consistent version of the virtual drive. To validate whether the recovered drive was prefix-consistent, we ran Blizzard's recovery code, and then used the write log to verify that the recovered disk contained a prefix-consistent representation of the write stream.

Bounding data loss: In fast acknowledgment mode and log-based commit mode, Blizzard exchanges performance for the risk of data loss. A recovered virtual drive is always consistent, but the drive may not contain a trailing set of writes that Blizzard acknowledged to the client but failed to make durable before the crash occurred. To measure this data loss, we ran two additional experiments.

In the first experiment, we generated a synthetic stream of write operations. Each write was the exact size of Blizzard's block size, and it was aligned with a block boundary, ensuring that, when Blizzard used log-based commits, there was no read-before-write penalty (§2.5.3). Using our instrumented Blizzard driver, we picked random moments to simulate crashes, and logged the amount of buffered, unacknowledged data that would be lost in the simulated crash. Our load generator also injected a configurable number of flush requests. Figure 13 shows the results. Each data point represents 100 simulated crashes.

As expected, the loss rate increases as the IO rate increases, because Blizzard must buffer more data. With one write per flush, i.e., with a total ordering over all writes, the simple fast acknowledgment scheme can only have one outstanding write to a remote disk. This severely limits the rate at which writes can retire, and it increases the memory pressure needed to buffer writes. With a write size of 4 KB, the fast acknowledgment scheme can only handle 100 IOps before too many writes queue up, and the virtual drive throttles the client to bound potential data loss; with a write size of 64 KB, the scheme can only handle 50 IOps before it throttles the client. The log-based commit scheme can issue writes immediately, regardless of the flush rate, so this scheme can gracefully scale up to 300 IOps.

For a flush rate of once-every-four writes, the fast acknowledgment scheme does much better, scaling all the way to 300 IOps. With a wider flush epoch, the fast acknowledgment scheme can issue more writes in parallel, and when a write from a new epoch arrives, old writes from the prior epoch are likely to already be committed, or at least issued; thus, the new write is unlikely to be delayed by a full seek time on a remote disk. With a write size of 4 KB, the difference in average loss rates between the two schemes is large in relative percentage, but small in absolute value—at 300 IOps, the fast acknowledgment scheme loses 4.1 writes representing 16.4 KB of data, and the log-based scheme loses 2.4 writes representing 11.5 KB of data. For a larger write size of 64 KB, the data loss per dropped write increases, and the two schemes show bigger differences in absolute amounts of data loss. For IOp rates above 100, the log-based scheme decreases loss rates by 12–30%. For example, at 300 IOps with 64 KB writes, the log-based scheme loses 191 KB per crash, whereas the simple fast acknowledgment scheme loses 272 KB.

Blizzard's log-based commit scheme pays a read-before-write penalty for writes that are smaller than Blizzard's block size. If writes are frequently misaligned (or not even multiples of the block size), the log-based scheme will force many writes to wait for synchronous reads. This will increase buffering requirements and the issue latencies for writes, causing loss rates after a crash to increase. Figure 14 shows this effect. In this example, the writes are aligned with Blizzard's block boundaries, but they range in size from 4 KB to 1 MB, as determined by empirical distributions of VM write sizes [12, 13]. In these experiments, the log-based scheme with a block size of 16 KB could only handle up to 50 IOps—beyond that, the read-before-write penalty forced Blizzard to throttle the client's write rate. Decreasing the block size to 4 KB resulted in fewer read-before-writes, allowing the log-based scheme to scale better. At 75 IOps, the log-based scheme beat the fast acknowledgment scheme by 37%, with a data loss of 135 KB instead of 214 KB. However, at 100 IOps, the log-based scheme can no longer hide the read-before-write penalties, and it has an average data loss of 54 MB, an order of magnitude worse than fast acknowledgments with a wpf of 1, and two orders of magnitude worse than fast acknowledgments with a wpf of 4.

5 Related Work

Block-level interfaces: A variety of protocols use a block interface to expose a single disk to remote clients. Examples of such protocols include ATA-over-Ethernet [22] and iSCSI [36]. Blizzard extends the simple block interface, mapping each virtual drive to multiple backing disks, and providing high-level software abstractions like replication and failure recovery across thousands of disks.

Like Blizzard, Petal [24] defines a distributed, software-implemented virtual disk that is backed by remote storage. However, Blizzard can coexist with traditional big-data workloads, and Blizzard leverages a full-bisection bandwidth network to stripe data more aggressively than Petal; the latter exposes Blizzard clients to higher levels of disk parallelism. Blizzard also leverages delayed durability semantics to increase the rate at which clients can issue writes while still achieving crash consistency.

Salus [43] is another example of a virtual block store. Salus is built atop HDFS/HBase [5, 6], and it provides ordered-commit semantics during normal operation, and prefix semantics when failures occur. Salus achieves these properties with pipelined commit, a protocol that resembles two-phase commit. In contrast, Blizzard achieves consistency with only a single round of communication between the client and the remote data stores. This reduces both network traffic and software complexity.

Mapping schemes: Using techniques like nested striping (§2.3) and deterministic permutations (§2.5), Blizzard translates virtual block accesses to FDS-level block accesses. This is similar to how SSDs use a

Flash Translation Layer (FTL) to map virtual blocks to physical ones. SSDs employ a variety of optimizations to minimize the size of the mapping table that is kept in the SSD's small, on-board SRAM (e.g., [18, 45]). While these optimizations could be applied to Blizzard's mapping structures, we have not found a need for such approaches, since Blizzard's tables are small ($\sim$20 MB) and they easily fit within main memory. FTLs must also implement compaction and garbage collection, since SSD writes and SSD erases have different data sizes. Blizzard's log-based commit scheme avoids compaction and garbage collection by using equivalent sizes for writes and erases in the log.

Cloud-scale storage systems: BlueSky [42] uses NFS or CIFS proxies to expose commercial cloud storage like Azure to enterprise clients. BlueSky allows an enterprise to offload the administrative costs of the storage cluster to a third party. However, compared to systems in which clients and servers reside in the same cloud, BlueSky introduces several new sources of overhead. Pulling data from commercial cloud servers injects wide-area latencies into the IO path. BlueSky proxies also use the chatty, text-based HTTP protocol to communicate with remote servers. The HTTP overhead is magnified if the enterprise has asymmetric upload/download speeds to the cloud—slower upload speeds mean that even small HTTP headers can add significant transfer delays [16].

Unlike BlueSky's focus on WAN access to cloud storage, Parallel NFS (pNFS) exposes clients to local (i.e., on-premise) cloud storage [38]. Storage servers can export a block interface, an object interface, or a file interface; pNFS clients transparently convert NFS requests to the appropriate lower-level access format. pNFS requires applications to adhere to NFS semantics, and both pNFS and BlueSky lack key Blizzard features like role-based striping and asynchronous epoch-based commits for writes.

Desktop/server file systems: OptFS demonstrated how to decouple durability from ordering in the context of a journaling file system [7]. Blizzard shows that these ideas can be applied at the disk level, providing OptFS-style performance improvements in a *file system-agnostic way* that does not depend on knowledge of the file system's consistency scheme (e.g., journaling [40] or shadow paging [21, 35]). OptFS requires disks to be modified to provide asynchronous durability notifications; in contrast, Blizzard's virtual disk implements prefix consistency using standard, asynchronous write-through operations on the backing remote disks.

When Blizzard uses log-based commit, it leverages expanded blocks to enable crash recovery and disconnected operation. Transactional Flash [34] and backpointer-based consistency [8] also embed extra metadata in out-of-band areas.

BPFS [10] is a file system for use with byte-addressable, persistent memory hardware (e.g., Phase Change Memory). BPFS introduces an abstraction, called an epoch barrier, that allows ordering guarantees to be expressed without requiring an immediate flush of dirty data in the CPU cache. Epoch barriers provide data consistency while preserving the ability of the memory controller to reorder writes within an epoch. Epoch barriers require custom hardware, and BPFS expects that the persistent memory resides directly on the memory bus. Like BPFS, Blizzard also separates ordering from durability; however, the separation is implemented in the context of a distributed system, rather than a single machine with access to persistent memory.

The Zebra file system [19] combines ideas from RAID [32] and log-based file systems [35], striping a per-client file log across a RAID array. Zebra does not provide mechanisms for asynchronous flush handling, and this constrains the level of disk parallelism that Zebra can provide to applications. Zebra uses compaction and garbage collection to manage dead block data; when such log cleaning occurs, it can introduce unpredictable performance fluctuations [37]. In contrast, when Blizzard operates in log-based asynchronous commit mode, it uses reads-before-writes to only commit full blocks of data. This smooths out the background IO traffic that is required for log maintenance. However, Section 4.7 demonstrates that if clients frequently issue misaligned writes, Blizzard's read-after-write penalty can be large, making Blizzard's simple fast acknowledgment scheme more attractive.

6 Conclusions

Blizzard exposes unmodified, cloud-oblivious POSIX applications to a fast, cloud-scale block store. This block store, which clients mount as a virtual disk, efficiently supports small, random IOs, but it coexists alongside big-data file systems, and deploys atop the same servers, disks, and switches. Using a network with full-bisection bandwidth, Blizzard provides clients with fast access to any remote disk. Using a novel striping scheme, Blizzard maximizes disk parallelism, avoids disk hotspots, and reduces IOp convoy dilation. By carefully ordering writes, Blizzard can immediately acknowledge flush requests while still providing crash consistency; with fewer write barriers, clients can issue writes faster, and better leverage the spindle parallelism of remote storage. A Blizzard prototype improves the speed of unmodified POSIX applications by up to an order of magnitude. In summary, Blizzard makes it much easier for cloud-agnostic POSIX applications to receive cloud-scale performance and availability.

References

[1] Amazon. Amazon EBS-Optimized Instances. AWS Documentation. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EBSOptimized.html. October 15, 2013.

[2] Amazon. Amazon Elastic Block Store (EBS). http://aws.amazon.com/ebs/. 2014.

[3] Amazon. Amazon Relational Database Service (Amazon RDS). http://aws.amazon.com/rds/. 2014.

[4] Amazon. Amazon Simple Email Service (Amazon SES). http://aws.amazon.com/ses/. 2013.

[5] Apache. Apache HBase. http://hbase.apache.org. 2014.

[6] D. Borthakur. The Hadoop Distributed File System: Architecture and Design. http://hadoop.apache.org/docs/r0.18.0/hdfs_design.pdf. 2007.

[7] V. Chidambaram, T. Pillai, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of SOSP*, pages 228–243, Farmington, PA, November 2013.

[8] V. Chidambaram, T. Sharma, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Consistency Without Ordering. In *Proceedings of FAST*, pages 101–116, San Jose, CA, February 2012.

[9] J. Cipar, G. Ganger, K. Keeton, C. Morrey III, C. Soules, and A. Veitch. LazyBase: Trading Freshness for Performance in a Scalable Database. In *Proceedings of EuroSys*, pages 169–182, Bern, Switzerland, April 2012.

[10] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of SOSP*, pages 133–146, Big Sky, MT, 2009.

[11] A. Edwards and B. Calder. Exploring Windows Azure Drives, Disks, and Images. Microsoft. http://blogs.msdn.com/b/windowsazurestorage/archive/2012/06/28/exploring-windows-azure-drives-disks-and-images.aspx. June 27, 2012.

[12] D. Feller. Virtual Desktop Resource Allocation. The Citrix Blog. http://blogs.citrix.com/2010/11/12/virtual-desktop-resource-allocation. November 12, 2010.

[13] R. Fellows. Storage Optimization for VDI. Tutorial: Storage Networking Industry Association. http://www.snia.org/sites/default/education/tutorials/2011/fall/StorageStorageMgmt/RussFellowsSNW_Fall_2011_VDI_best_practices_final.pdf. 2011.

[14] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. *ACM SIGOPS Operating Systems Review*, 37(5):29–43, December 2003.

[15] Google. Google Cloud SQL. https://cloud.google.com/products/cloud-sql. 2014.

[16] Google. Performance Best Practices: Minimize request overhead. https://developers.google.com/speed/docs/best-practices/request. March 28, 2012.

[17] A. Greenberg, J. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *Proceedings of SIGCOMM*, pages 51–62, Barcelona, Spain, 2009.

[18] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: a Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings. In *Proceedings of ASPLOS*, pages 229–240, Washington, DC, March 2009.

[19] J. Hartman and J. Ousterhout. The Zebra Striped Network File System. *ACM Transactions on Computer Systems*, 13(3):274–310, 1995.

[20] D. Hildebrand, A. Nisar, and R. Haskin. pNFS, POSIX, and MPI-IO: A Tale of Three Semantics. In *Proceedings of the Workshop on Petascale Data Storage*, pages 32–36, Portland, OR, November 2009.

[21] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference*, San Francisco, CA, January 1994.

[22] S. Hopkins and B. Coile. AoE (ATA over Ethernet). http://support.coraid.com/documents/AoEr11.txt. February 2009.

[23] N. Johnson. JetStress 2010: JetStress Field Guide. Microsoft. http://gallery.technet.microsoft.com/Jetstress-Field-Guide-1602d64c. March 27, 2012.

[24] E. Lee and C. Thekkath. Petal: Distributed Virtual Disks. *ACM SIGOPS Operating Systems Review*, 30(5):84–92, December 1996.

[25] A. Leung, S. Pasupathy, G. Goodson, and E. Miller. Measurement and Analysis of Large-Scale Network File System Workloads. In *Proceedings of USENIX ATC*, pages 213–226, Boston, MA, June 2008.

[26] Microsoft. Introducing Windows Azure SQL Database. http://msdn.microsoft.com/en-us/library/windowsazure/ee336230.aspx. 2014.

[27] Microsoft. Windows Azure Active Directory. http://www.windowsazure.com/en-us/services/active-directory/. 2014.

[28] Microsoft. Windows Azure Storage Scalability and Performance Targets. Windows Azure Documentation. http://msdn.microsoft.com/en-us/library/windowsazure/dn249410.aspx. June 20, 2013.

[29] Microsoft. Windows PE Technical Reference. http://technet.microsoft.com/en-us/library/dd744322(WS.10).aspx. October 22, 2009.

[30] E. Nightingale, J. Elson, O. Hofmann, Y. Suzue, J. Fan, and J. Howell. Flat Datacenter Storage. In *Proceedings of OSDI*, pages 1–15, Hollywood, CA, October 2012.

[31] E. Nightingale, K. Veeraraghavan, P. Chen, and J. Flinn. Rethink the Sync. In *Proceedings of OSDI*, pages 1–14, November 2006.

[32] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). *ACM SIGMOD Record*, 17(3):109–116, 1988.

[33] V. Prabhakaran, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *Proceedings of USENIX ATC*, pages 105–120, Anaheim, CA, April 2005.

[34] V. Prabhakaran, T. Rodeheffer, and L. Zhou. Transactional Flash. In *Proceedings of OSDI*, pages 147–160, San Diego, CA, December 2008.

[35] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[36] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. Internet small computer systems interface (iSCSI). Technical report, RFC 3720, April, 2004.

[37] M. Seltzer, K. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File System Logging Versus Clustering: A Performance Comparison. In *Proceedings of the USENIX Winter Technical Conference*, pages 249–264, New Orleans, LA, January 1995.

[38] S. Shepler, M. Eisler, and D. Noveck. Network File System (NFS) Version 4 Minor Version 1 Protocol. Technical report, RFC 5661, January, 2010.

[39] M. Steigerwald. Imposing Order. Linux Magazine, May 2007.

[40] S. Tweedie. Journaling the Linux ext2fs File System. In *Proceedings of the Fourth Annual Linux Expo*, Durham, North Carolina, May 1998.

[41] W. Vogels. File system usage in Windows NT 4.0. In *Proceedings of SOSP*, pages 93–109, Kiawah Island Resort, SC, December 1999.

[42] M. Vrable, S. Savage, and G. Voelker. BlueSky: A Cloud-Backed File System for the Enterprise. In *Proceedings of FAST*, pages 237–250, San Jose, CA, 2012.

[43] Y. Wang, M. Kapritsos, Z. Ren, P. Mahajan, J. Kirubanandam, L. Alvisi, and M. Dahlin. Robustness in the Salus Scalable Block Store. In *Proceedings of NSDI*, pages 357–370, Lombard, IL, April 2013.

[44] E. Weisstein. Linear Congruence Method. MathWorld: A Wolfram Web Resource. http://mathworld.wolfram.com/LinearCongruenceMethod.html. 2014.

[45] Y. Zhang, L. Arulraj, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. De-indirection for Flash-based SSDs with Nameless Writes. In *Proceedings of FAST*, pages 1–16, San Jose, CA, February 2012.