Scalable Virtual Machine Storage using Local Disks

Jacob Gorm Hansen
VMware
Aarhus, Denmark
jgorm@vmware.com

Eric Jul
Bell Laboratories
Alcatel-Lucent, Dublin, Ireland
eric@cs.bell-labs.com

ABSTRACT

In virtualized data centers, storage systems have traditionally been treated as black boxes administered separately from the compute nodes. Direct-attached storage is often left unused, to not have VM availabilty depend on individual hosts. Our work aims to integrate storage and compute, addressing the fundamental limitations of contemporary centralized storage solutions. We are building *Lithium*, a distributed storage system designed specifically for virtualization workloads running in large-scale data centers and clouds. Lithium aims to be scalable, highly available, and compatible with commodity hardware and existing application software. The design of Lithium borrows techniques from Byzantine Fault Tolerance, stream processing, and distributed version control software, and demonstrates their practical applicability to the performance-sensitive task of virtual machine storage.

Categories and Subject Descriptors

D.4.3 [**Operating Systems**]: File Systems Management—*Distributed file systems*; D.4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*

General Terms

Design, Experimentation, Performance, Reliability

1. INTRODUCTION

Cloud computing has motivated a new breed of distributed storage systems that trade backwards-compatibility for better scalability and the ability to run reliably on cheap commodity hardware. To reduce cost, reliability is achieved in software, for instance by using replication and integrity-preserving techniques such as strong checksums and cryptographic signatures. While interesting for new applications written specifically for deployment in the cloud, few of these new systems have been able to provide the throughput or consistency guarantees needed to act as primary storage for legacy applications running in virtual machines (VMs), for instance on VMware's ESX platform. Because they provide backwards-compatible disk semantics with high throughput and enable virtualization features like live VM migration [11, 3], shared storage (SAN or NAS) disk arrays prevail in the modern enterprise.

Virtualization in data centers was initially for consolidation of few server VMs with high throughput demands, but is now increasingly used to host many smaller VMs, for instance to provide hundreds or thousands of employees with VM-backed thin client desktops. Here, per-VM storage throughput requirements are modest (users expect performance similar to a laptop drive), but the ability to scale incrementally and tolerate bursty access patterns, such as when all VMs are powered on Monday morning, becomes important. In such scenarios, access to shared storage often becomes a bottleneck. To provide better scalability, and to bridge the current gap between VMs and low-cost cloud storage, we have designed and built Lithium. Lithium is a distributed storage system for VMs with many of the features that shared storage users have come to expect, such as clones and snapshots and support for seamless VM migration. Lithium makes VM storage location-independent and exploits the local storage capacity of compute nodes, to allow installations to grow incrementally and to reduce cost compared with non-commodity shared storage hardware. The system is aimed at applications such as virtual desktop serving, and for deployment in large-scale data centers such as those of cloud providers. It consists of a replication-optimized log-structured storage engine, and of a set of network protocols for managing replica consistency and failover in decentralized manner, without scalability bottlenecks or single points of failure.

The rest of this paper is laid out as follows: section 2 further motivates our approach, section 3 describes Lithium's overall design, data format, and replica consistency protocols, and section 4 describes our prototype implementation for the VMware ESX platform. Section 5 compares the performance of our prototype against an enterprise class Fibre Channel disk array, and section 6 concludes. This paper is a condensed and updated version of [5], and the reader is referred to that paper for additional implementation details and a full discussion of related work.

2. BACKGROUND

This work describes a distributed system that provides virtual block storage for a potentially large set of VMs running in an enterprise or cloud hosting data center. The aim is not to replace high-end storage arrays for applications that require the combined throughput of tens or hundreds of disk spindles, but rather to address the scalability and reliability needs of many smaller VMs that individually would run acceptably from one or a few local disk drives, but today require shared storage for taking advantage of features such as live VM migration. To provide location-independence and reliability, Lithium uses peer-to-peer replication at the granularity of individual VM disk writes, and its design is based on the following set of observations:

- Replication often leads to write bias, and virtualization, where each physical disk is shared by multiple concurrently running VMs, often destroys access locality. This makes exploiting disk bandwidth difficult, because most physical disk accesses end up being random.
- The need for supporting parallel IOs, for performance, may result in replica inconsistencies due to disk schedulers arbitrarily reordering writes.
- Non-dedicated networks may experience transient outages or throughput degradation that, if not masked, may cause VMs to throw timeouts or other errors back at the user. With today's high network transmission speeds and huge disk capacities, the risk of silent data corruption cannot be ignored either.

Based on the above, we decided to design Lithium to more resemble a message server or streaming database than a traditional file system. To make most of disk bandwidth, and to counter the non-determinism caused by unpredictable IO schedulers, we concluded that Lithium would need to store its data in a write-optimized, log-structured format, with explicitly ordered updates protected by strong checksums. Such as format would also support tunable degrees of replica consistency, to allow users to trade consistency for availability and performance when appropriate.

3. DESIGN

Lithium is a scalable distributed storage system optimized for the characteristics of virtual machine IO: random and mostly exclusive block accesses to large and sparse virtual disk files. For scalability reasons, Lithium does not provide a globally consistent POSIX name space, but names storage objects in a location-independent manner using globally unique incremental state hashes, similar to a key-value store but where each value is a large 64-bit address space, or *volume*. VMs see only a virtual disk abstraction, which they can format, *e.g.*, using a traditional file system or on which a database can be stored.

Lithium supports instant volume creation with lazy space allocation, instant creation of writable snapshots, and replication of volumes and snapshots with tunable consistency ranging from per-update synchronous replication to eventual consistency modes that allow VMs to remain available during periods of network congestion or disconnection.

Lithium's underlying data format is self-certifying and self-sufficient, in the sense that data can be stored on unreliable hosts and disk drives without fear of silent data corruption, and all distributed aspects of the protocol are persisted as part of the core data format. This means that Lithium can function independently, without reliance on any centralized lock management or meta-data service, and can scale from just a couple of hosts to a large cluster.

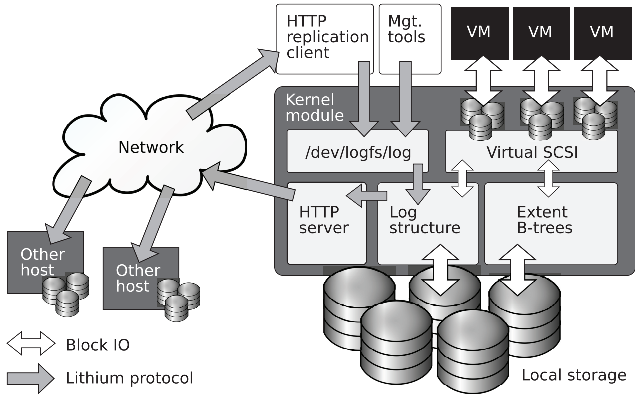

Figure 1: The components of Lithium on each host. VMs access the Lithium kernel module inside ESX through a virtual SCSI interface, and the user space replication agent and management tools append to the log through a special device node.

Lithium consists of a kernel module for the VMware ESX hypervisor, and of a number of small user space tools that handle volume creation, branching, and replication. Figure 1 shows the main components installed on a Lithium-enabled host. All data is made persistent in a single on-disk log-structure that is indexed by per-volume B-trees, to allow for random access reads. The kernel module handles performance-critical network and local storage management, while user space processes are responsible for policy-driven tasks such as volume creation, branching, replication, membership management, and fail-over. This split responsibilities design allows for more rapid development of high-level policy code, while retaining performance on the critical data path.

Like in a modern file system, space for volumes is allocated lazily when blocks are written, however, volumes differ from files in that their names are 160-bit unique identifiers chosen at random from a single flat name space, instead of paths that point to physical hosts or disk drives. There is no need for a global, hierarchical name space, and no global meta-data server that could become a bottleneck or single point of failure. Volumes are self-describing and location-independent, and can be easily replicated and migrated between hosts.

All writes to a given volume are logged to disk, and the location of the logical blocks written recorded in that volume's B-tree. Each write in the log is prefixed by a commit header with a strong checksum, update ordering information, and the logical block addresses affected by the write. Log entries can optionally be forwarded to other hosts for replication, and replica hosts maintain their own log and a B-tree for each replicated volume. The same data format is used on disk and wire.

A new volume can be created on any machine by appending an appropriate log entry to the local log. To avoid the volume being lost if the host crashes immediately, volume creation and other operations that modify the volume's replica set are implemented as distributed transactions using two-phase commit across the affected hosts. Once created, the new volume is identified by a permanent unique base id and a current version id, calculated as the incremental hash of all updates in the volume's history. Replica consistency can be verified quickly by simple comparison of version ids.

Hosts communicate over HTTP. Other hosts can create replicas of a volume merely by connecting to a tiny HTTP server inside the Lithium kernel module. They can sync up from a given version id, and when completely synced, the HTTP connection switches over to synchronous replication. As long as the connection remains open, the volume replicas stay tightly synchronized. Hosts without a volume replica can access the volume remotely using HTTP byte-range PUT and GET commands. The use of HTTP makes Lithium simple to integrate with other Cloud storage systems such as Amazon S3, and allows immutable VM images to be served from standard web servers like Apache. A simple distributed hash table (DHT) is used for keeping track of all volume replicas in the system.

A host within the replica set is selected to be the *primary* (or *owner*) for that volume. The primary serializes access, and is typically the host where the VM is running. Ownership can be transferred seamlessly to another replica, which then becomes the new primary. The integrity and consistency of replicas is protected by a partially ordered data model known as *fork-consistency*, described next.

3.1 Fork-consistent Replication

Logged updates in Lithium form a hash-chain, with individual updates uniquely identified and partially ordered using cryptographic hashes of their contexts and contents. Use of a partial rather than a total ordering removes the need for a central server acting as a coordinator for update version numbers, and allows for simple distributed branching of storage objects, which in a storage system is useful for cloning and snapshotting of volumes. The use of strong checksums of both update contents and the accumulated state of the replication state machine allows for easy detection of replica divergence and integrity errors ("forks"). Each update has a unique id and a parent id, where the unique id is computed as a secure digest of the parent id concatenated with the contents of the current update, *i.e.*, $id = h(parentid||updatecontents)$, where $h()$ is the secure digest implemented by a strong hash function (SHA-1 in our implementation). By having updates form a hash-chain with strong checksums, it becomes possible to replicate data objects onto untrusted and potentially Byzantine hardware; recent studies have found such Byzantine hardware surprisingly common [2]. Figure 2 shows how each volume is stored as a chain of updates where the chain is formed by backward references to parent updates. Fork-consistency allows a virtual disk volume to be mirrored anywhere, any number of times, and allows anyone to clone or snapshot a volume without coordinating with other hosts. Snapshots are first-class objects with their own unique base and version ids.

3.2 Replica Consistency

State-machine replication systems have been studied extensively and the theory of their operation is well understood [14]. In practice, however, several complicating factors make building a well-performing and correct VM disk replication system less than straight-forward. Examples include parallel queued IOs that in combination with disk scheduler nondeterminism can result in replica divergence, and network and disk data corruption that if not checked will result in integrity errors propagating between hosts. Finally, the often massive sizes of the objects to be replicated makes full resynchronization of replicas, for instance to ensure consistency after a crash, impractical.

In a replication system that updates data in place, a two-phase commit (2PC) is necessary to avoid replica divergence. Updates are first logged out-of-place at all replicas, and when all have acknowledged receipt, the *head* node tells them to destructively commit. Unfortunately, 2PC protocols suffer a performance overhead from having to write everything twice. If 2PC is not used, a crash can result in not all replicas applying updates that were in flight at the time of the crash. It is acceptable for the replicas to diverge, as long as the divergence is masked or repaired before data is returned to the application. In practice, this means that the cost of 2PC can be avoided as long as replicas are properly resynchronized after a crash. Previous replication systems such as Petal [9] have used a bitmap of unstable disk regions, to avoid having to resynchronize entire disks after a crash. Unfortunately bitmaps can be impractical for high-degree replication. Instead, Lithium uses update logging to also support Bayou-like eventual consistency [15].

The Lithium replication mechanism is similar to the first phase of 2PC in that updates are logged non-destructively. Once an update has been logged at a quorum of the replicas, the write IO is acknowledged back to the application VM. In eventual-consistency mode, IOs are acknowledged when they complete locally. Because updates are non-destructive, crashed or disconnected replicas can sync up by replaying of missing log updates from peers. Data lives permanently in the log, so the protocol does not require a separate commit-phase. Writes are issued and serialized at the primary replica, typically where the VM is running. We make the simplifying assumption that writes always commit to the primary's log, even when the VM tries to abort them or the drive returns a non-fatal error condition. We catch and attempt to repair drive errors (typically retry or busy errors) by quiescing IO to the drive and retrying the write, redirecting to another part of the disk, if necessary. This assumption allows us to issue IO locally at the primary and to the replicas in parallel.

Multiple IOs can be issued in parallel, a requirement for exploiting the full disk bandwidth. The strong checksum in our disk format commit headers ensures that incomplete updates can be detected and discarded. This means that we do not require a special "end-of-file" symbol: we recover the end of the log after a crash simply by rolling forward from a checkpoint until we encounter an incomplete update. To avoid problems with "holes" in the log, writes are always acknowledged back to the VM in log order, even if reordered by the drive. For mutable volumes, reads are always handled at the primary, whereas immutable (snapshot) volumes can be read-shared across many hosts. The on-disk format guarantees temporal ordering, so it is trivial to find the most recent out of a set of replicas, and to emulate disk semantics it is always safe to roll a group of replicas forward to the state of the most recent one. We use an access token, described in section 3.4, to ensure replica freshness and mutual exclusion.

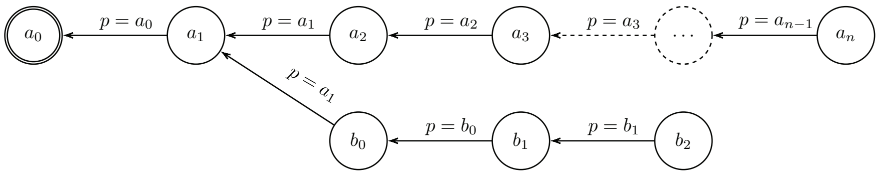

Figure 2: Partial update ordering and branches in a traditional fork-consistent system. The partial ordering is maintained through explicit "parent" back-references, and branches can be expressed by having multiple references to the same parent.

3.3 Log-structured Volumes

On each host, Lithium exposes a single on-disk log-structure as a number of volumes. Each entry in the log updates either block data or the meta-data of a volume. The state of each volume equals the sum of its updates, and a volume can be recreated on another host simply by copying its update history. Each host maintains per-volume B-tree indexes to support random access reads from each volume's 64-bit block address space. Volumes can be created from scratch, or as branches from a base volume snapshot. In that case, multiple B-trees are chained, so that reads "fall through" empty regions in the children B-trees and are resolved recursively at the parent level. Branching a volume involves the creation of two new empty B-trees, corresponding to the left and right branches, a near-instant operation. Unwritten regions consume no B-tree or log space, so there is no need to pre-allocate space for volumes or to zero disk blocks in the background to prevent information leakage between VMs.

The disk log is the only authoritative data store used for holding hard state that is subject to replication. *All* other state is host-local soft-state that in the extreme can be recreated from the log. As such, Lithium may be considered a log-structured file system (LFS) [13] or a scalable collection of logical disks [4]. Log-structuring is still regarded by many as a purely academic construct hampered by cleaning overheads and dismal sequential read performance. For replicated storage, however, log-structuring is attractive because it preserves the temporal order of writes, and because it improves write throughput compared to in-place layouts.

Non-destructive, temporally ordered writes simplify the implementation of an eventually consistent [15] replication system, for instance to support disconnected operation [7] or periodic asynchronous replication to off-site locations. Write throughput is important because replication often results in write-bias. For example, a workload with a read-skewed 2:1 read-to-write ratio without replication is transformed to a write-skewed 3:2 write-to-read ratio by triple replication. Furthermore, parallel replication of multiple virtual disk update streams exhibits poor locality because each stream wants to position the disk head in its partition of the physical platter, but with a log-structured layout all replication writes can be made sequential, fully exploiting disk write bandwidth. Other methods, such as erasure-coding or de-duplication, can reduce the bandwidth needs, but do not by themselves remedy the disk head contention problems caused by many parallel replication streams. While it is true that sequential read performance may be negatively affected by log-structuring, large sequential IOs are rare when multiple VMs compete for disk bandwidth. A recent study [10] also indicated that the general write-to-read ratio tends to be increasing. This may be due to a trend towards more archival storage, or to programmers getting better at working around the exponentially widening gap between CPU and disk performance. Finally, log-structuring can help overcome some of the inherent limitations of Flash-based storage devices, as described in [1].

The downside to log-structuring is that it creates additional fragmentation, and that a level of indirection is needed to resolve logical block addresses into physical log offsets. Interestingly, when we started building Lithium, we did not expect log-structured logical disks to perform well enough for actual hosting of VMs, but expected them to run only as live write-only backups at replicas. However, we have found that the extra level of indirection rarely hurts performance, and now use the log format both for actual hosting of VMs and for their replicas.

3.4 Mutual Exclusion

We have already described the fork-consistency model that uses cryptographic checksums to prevent silent replica divergence as a result of silent data corruption, and that allows branches of volumes to be created anywhere and treated as first-class replicated objects. We now describe our extensions to fork consistency that allow us to emulate shared-storage semantics across multiple volume replicas, and that allow us to deal with network and host failures. While such functionality could also have been provided through an external service, such as a distributed lock manager, a mechanism that is integrated with the core storage protocol is going to be more robust, because a single mechanism is responsible for ordering, replicating, and persisting both data and mutual exclusion meta-data updates. This is implemented as an extension to the fork-consistency protocol by adding the following functionality:

- A mutual exclusion *access token* constructed by augmenting the partial ordering of updates, using one-time secrets derived from a single master key referred to as the *secret view-stamp*.

- A fallback-mechanism that allows a majority quorum of the replicas to recover control from a failed primary by recovering the master key through secret sharing and a voting procedure.

3.4.1 Access Token

For emulating the semantics of a shared storage device, such as a SCSI disk connected to multiple hosts, fork-consistency alone is insufficient. VMs expect either "read-your-writes consistency", where all writes are reflected in subsequent reads, or the weaker "session-consistency", where durability of writes is not guaranteed across crashes or power losses (corresponding to a disk controller configured with write-back caching). In Lithium, the two modes correspond to either synchronous or asynchronous eventually-consistent replication. In the latter case, we ensure session consistency by restarting the VM after any loss of data.

Our SCSI emulation supports mutual exclusion through the `reserve` and `release` commands that lock and unlock a volume, and our update protocol implicitly reflects ownership information in a volume's update history, so that replicas can verify that updates were created by a single exclusive owner. To ensure that this mechanism is not the weakest link of the protocol, we require the same level of integrity protection as with regular volume data. To this end, we introduce the notion of an *access token*, a one-time secret key that must be known to update a volume. The access token is constructed so that only an up-to-date replica can use it to mutate its own copy of the volume. If used on the wrong volume, or on the wrong version of the correct volume, the token is useless.

Every time the primary replica creates an update, a new secret token is generated and stored locally in volatile memory. Only the current primary knows the current token, and the only way for volume ownership to change is by an explicit token exchange with another host. Because the token is a one-time password, the only host that is able to create valid updates for a given volume is the current primary. Replicas observe the update stream and can verify its integrity and correctness, but lack the necessary knowledge for updating the volume themselves. At any time, the only difference between the primary and the replicas is that only the primary knows the current access token.

The access token is constructed on top of fork-consistency by altering the partial ordering of the update hash chain. Instead of storing the clear text id of a version in the log entry, we include a randomly generated secret *view-stamp* s in the generation of the id for an update, so that

$$\begin{array}{rrcl} \text{instead of:} & id & = & h(parentid||update) \\ \text{we use:} & id & = & h(s||parentid||update) \end{array}$$

where h is our strong hash function. The secret view-stamp s is known only by the current primary. In the update header, instead of storing the clear text value of id, we store $h(id)$ and only keep id in local system memory (because $h()$ is a strong hash function, guessing id from $h(id)$ is assumed impossible). Only in the successor entry's *parentid* field do we store id in clear text. In other words, we change the pairwise ordering

$$\begin{array}{rlll} \text{from:} & a < b & \text{iff} & a.id = b.parentid \\ \text{to:} & a < b & \text{iff} & a.id = h(b.parentid) \end{array}$$

where $<$ is the direct predecessor relation. The difference here being that in order to name a as the predecessor to b, the host creating the update must know $a.id$, which only the primary replica does. Other hosts know $h(a.id)$, but not yet $a.id$. We refer to the most recent id as the *secret access token* for that volume. The randomly generated s makes access tokens unpredictable. While anyone can replicate a volume, only the primary, which knows the current access token id, can mutate it. Volume ownership can be passed to another host by exchange of the most recent access token. Assuming a correctly implemented token exchange protocol, replica divergence is now as improbable as inversion of $h()$. Figure 3 shows how updates are protected by one-time secret access tokens. The new model still supports decentralized branch creation, but branching is now explicit. When the parent for an update is stated as described above, the update will be applied to the existing branch. If it is stated as in the old model, *e.g.*, $b.parentid = a.id$, then a new branch will be created and the update applied there.

3.4.2 Automated Fail-over

A crash of the primary results in the loss of the access token, rendering the volume inaccessible to all. To allow access to be restored, we construct on top of the access token a view-change protocol that allows recovery of s by a remaining majority of nodes. A recovered s can be used to also recover id. To allow recovery of s, we treat it as a secret view-stamp [12], or epoch number, that stays constant as long as the same host is the primary for the volume, and we use secret sharing to disperse slices of s across the remaining hosts. Instead of exchanging the access token directly, volume ownership is changed by appending a special *view-change* record to the log. The view-change record does not contain s but its public derivate $h(s)$, along with slices of s, encrypted under each replica's host key. Thus $h(s)$ is the *public view-stamp* identifying that view.

If a host suspects that the current primary has died, it generates a new random s' and proposes the corresponding view $h(s')$ to the others. Similar to Paxos [8], conflicting proposals are resolved in favor of the largest view-stamp. Hosts cast their votes by returning their slices of s, and whoever collects enough slices to recover s wins and becomes the new primary. The new primary now knows both s and s' so it can reconstruct id using the recovered s, and append the view-change that changes the view to $h(s')$. When there is agreement on the new view, IO can commence. The former s can now be made public to revoke the view $h(s)$ everywhere. If there are still replicas running in the former view $h(s)$, the eventual receipt of s will convince them that their views are stale and must be refreshed. Should the former primary still be alive and eventually learn s, it knows that it has to immediately discard of its copy of the VM and any changes made (if running in eventual consistency mode) after the point of divergence. Discarding and restarting the VM in this case ensures session-consistency.

The use of the secret token protects against replica divergence as a result of implementation or data corruption errors. Prevention against deliberate attacks would require signing of updates, which is straightforward to add. Using signatures alone is not enough, because a signature guarantees authenticity only, not freshness or mutual exclusion as provided by the access token. We have found that access tokens are a simple and practical alternative to centralized lock servers. During development, the mechanism has

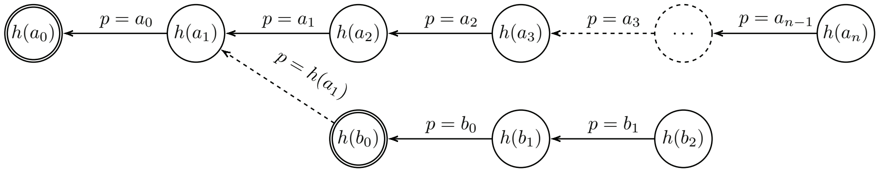

Figure 3: Update ordering and branches in Lithium. In contrast to the original fork-consistent model in Figure 2, appending an update to an existing branch requires knowledge of its clear-text *id* to state it as a parent reference. Branching is an explicit but unprivileged operation.

```
struct LogHeader {
  hash parentid, id, checksum;
  uint64_t firstBlock, numBlocks;
  bit bitVector[maxBlocksPerUpdate]; };
```

Figure 4: Format of the log entry header. The bit vector is used for zero block compression.

helped identify several implementation and protocol errors by turning replica divergence into a fail-stop condition.

4. IMPLEMENTATION DETAILS

Lithium treats each local storage device as a single log that holds a number of virtual volumes. Write operations translate directly into synchronous update records in the log, with the on-disk location of the update data being recorded in a B-tree index. The update may also propagate to other hosts that replicate the particular volume. Depending on configuration, the write is either acknowledged back to the VM immediately when the update is stable in the local log, or when all or a quorum of replicas have acknowledged the update. The first mode allows for disconnected operation with eventual consistency, while the second mode corresponds to synchronous replication.

When storing data on disk and when sending updates over the wire, the same format is used. A 512-byte log entry commit header describes the context of the update (its globally unique id and the name of its parent), the location of the update inside the virtual disk's 64-bit address space, the length of the extent, and a strong checksum. The rest of the header space is occupied by a bit vector used for compressing away zero blocks, both to save disk space and network bandwidth (in practice, this often more than offsets the cost of commit headers), and to simplify log-compaction. The header definition is shown in figure 4. For practical reasons, the on-disk log is divided into fixed size segments of 16MB each.

4.1 B-tree Indexes

Applications running on top of Lithium are unaware of the underlying log-structure of physical storage, so to support random access reads, Lithium needs to maintain a live index that maps between the logical block addresses (LBAs) used by VMs, and the actual physical locations of the data on disk. This index could be implemented as a per-volume lookup table with an entry for each logical block number in the volume, but because each Lithium volume is a 64-bit block address space, a lookup table might become prohibitively large. Instead, Lithium uses a 64-bit extent indexed B-tree to track logical block locations in the log. The B-tree is a more complex data structure than a simple lookup table or a radix tree, but is also more flexible, and designed to perform well on disk. The B-tree index does not track individual block locations, but entire extents. If a VM writes a large file into a single contiguous area on disk, this will be reflected as just a single key in the B-tree. Figure 5 shows an example B-tree index with block ranges that correspond to different log entries. In the pathological worst case, the VM may write everything as tiny, random IOs, but we have found that on average the use of extents provides good compression. For instance, a base Windows XP install has an average extent length of 29.9 512-byte disk sectors. Each B-tree extent key is 16 bytes plus 4 bytes for each child pointer, and B-tree nodes are always at least half full. In this case the space overhead for the B-tree is at most $\frac{(16+4)\times 2}{512*29.9} \approx 0.26\%$ of the disk space used by the VM. Table 1 lists the average extent sizes and resulting B-tree memory overheads we have encountered during development, along with the pathological worst cases of densely packed 4kB and 512B completely random writes. Guest OS disk schedulers attempt to merge adjacent IOs, which explains the large average extent sizes of the non-synthetic workloads. B-tree nodes are paged and cached in main memory, and as long as a similar percentage of the application's disk working set fits in the cache, performance is largely unaffected by the additional level of indirection.

Workload	Type	Avg.len	Overhead
Windows XP (NTFS)	FS	29.9	0.26%
IOZone (XFS)	FS	33.8	0.23%
PostMark (XFS)	FS	61.8	0.13%
DVDStore2 (MySQL)	DB	24.0	0.33%
4096-byte random	Synthetic	8.0	0.98%
512-byte random	Synthetic	1.0	7.81%

Table 1: Average extent sizes and worst-case B-tree memory overhead for different workloads.

We carefully optimized the B-tree performance: Most IO operations are completely asynchronous, the tree has a large fan-out (over 1,600 keys per tree node), and updates are buffered and applied in large batches to amortize IO costs. The B-tree uses copy-on-write updates and is crash-consistent. B-trees are not replicated, but if two hosts replicate the same volume, they will end up with similar B-trees. The B-trees are always kept up to date at all replicas, to allow control of a volume to migrate to another replica instantly. Periodic checkpoints ensure constant time restart recovery.

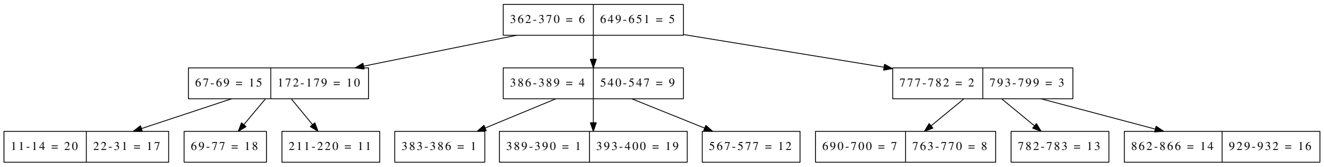

Figure 5: Simplified example of the B-tree used for mapping logical block offsets to version numbers in the log. Block number 765 would resolve to revision number 8, and block 820 would return NULL, indication that the block were empty. Multiple trees can be chained together, to provide snapshot and cloning functionality.

4.2 Log Compaction

A local log compactor runs on each host in parallel with other workloads, and takes care of reclaiming free space as it shows up. Free space results from VMs that write the same logical blocks more than once, and the frequency with which this occurs is highly workload-dependent. Free space is tracked per log-segment in an in-memory priority queue, updated every time a logical block is overwritten in one of the volume B-trees, and persisted to disk as part of periodic checkpoints. When the topmost n segments in the queue have enough free space for a compaction to be worthwhile, the log compactor reads those n segments and writes out at most $n-1$ new compacted segments, consulting the relevant B-trees to detect unreferenced blocks along the way. Unreferenced blocks are compressed away using the zero block bit vector described previously. The log compactor can operate on arbitrary log segments in any order, but preserves the temporal ordering of updates with pointer records as not to conflict with eventual consistency replication. Compaction works on entire 16MB log segments, to make the most of disk bandwidth and minimize workload interference. In a real deployment, one would likely want to exploit diurnal workload patterns to run the compaction only when disks are idle, but in our experiments, we run the compactor as soon as enough free space is available for compaction to be worthwhile.

Due to compaction, a lagging replica that syncs up from a compacted version may not see the complete history of a volume, but when the replica has been brought up to the compacted version, it will have a complete and usable copy. Though we remove overwritten data blocks, we currently keep their empty update headers to allow lagging replicas to catch up from arbitrary versions. In the future, we plan to add simple checkpoints to our protocol to avoid storing update headers perpetually.

4.3 POSIX Interface

Lithium provides block-addressable object storage, but by design does not expose a global POSIX name space. Like shared memory, faithfully emulating POSIX across a network is challenging and introduces scalability bottlenecks that we wish to avoid. The VMs only expect a block-addressable virtual disk abstraction, but some of the tools that make up our data center offering need POSIX, *e.g.*, for configuration, lock, and VM-swap files. Each VM has a small collection of files that make up the runtime and configuration for that VM, and these files are expected to be kept group-consistent. For convenience, we provide the option of formatting a Lithium volume using a tiny POSIX-compatible file system that can host the set of of files belonging to a VM, similar to a *fileset* in AFS [6]. Initial lookups for a volume are resolved through the DHT, where after the volume gets auto-mounted and can be accessed directly by VMs and local applications.

5. EVALUATION

Our evaluation focuses on the performance of the prototype when configured for synchronous 2 and 3-way replication. To measure the performance of our prototype we ran the following IO benchmarks:

PostMark PostMark is a file-system benchmark that simulates a mail-server workload. In each VM we ran PostMark with 50,000 initial files and 100,000 transactions. PostMark primarily measures IO performance.

DVDStore2 This online transaction processing benchmark simulates an Internet DVD store. We ran the benchmark with the default "small" dataset, using MySQL as database backend. Each benchmark ran for three minutes using default parameters. DVDStore2 measures a combination of CPU and IO performance.

Our test VMs ran Linux 2.6.25 with 256MB RAM and 8GB of virtual storage. Data volumes were stored either in Lithium's log-structure, or separately in discrete 8GB disk partitions. When running Lithium, log compaction ran eagerly in parallel with the main workload, to simulate a disk full condition. Lithium was configured with 64MB of cache for its B-trees. Apart from the settings mentioned above, the benchmarks were run with default parameters.

5.1 Replication Performance

Replication performance was one of the motivations for the choice of a write-optimized disk layout, because it had been our intuition that multiple parallel replication streams to different files on the same disk would result in poor throughput. When multiple VMs and replication streams share storage, most IO is going to be non-sequential. Apart from the caching that is already taking place inside the VMs, there is very little one can do to accelerate reads. Writes, however, can be accelerated through the use of log-structuring. For this benchmark, we used three HP Proliant DL385 G2 hosts, each equipped with two dual-core 2.4GHz Opteron processors, 4GB RAM, and a single physical 2.5" 10K 148GB SAS disk, attached to a RAID controller without battery backed cache. We used these hosts, rather than the newer Dells described below, because they had more local storage available for VMs and replicas, and were more representa-

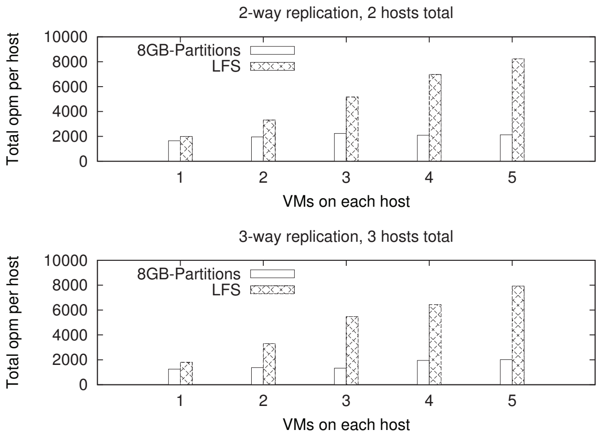

Figure 6: Performance of the DVD Store 2 OLTP benchmark in two- and three-host fully replicated scenarios.

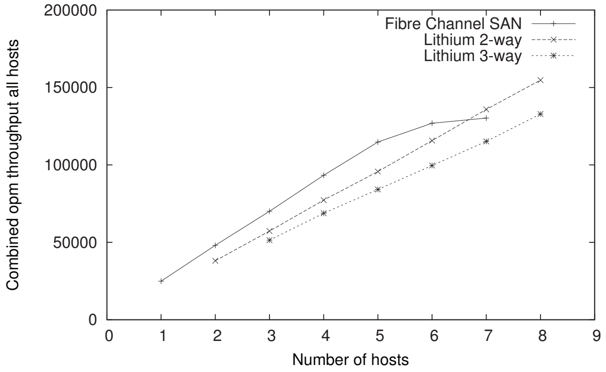

Figure 7: DVDStore2 Scalability. This graph shows the total orders per minute across all hosts, with 4 VMs per host for a varying number of hosts of Lithium in 2-way and 3-way replicated scenarios, and for a mid-range Fibre Channel storage array.

tive of commodity hardware. As a strawman, we modified the Lithium code to use a direct-access disk format with an 8GB *partition* of the disk assigned to each volume, instead of the normal case with a single shared log-structure. We ran the DVDStore2 benchmark, and used the total orders-per-minute number per host as the score. The results are shown in Figure 6, and clearly demonstrate the advantage of a write-optimized layout. Per-host throughput was 2–3 times higher when log-structuring was used instead of static partitioning.

5.2 Scalability

To test Lithium's scalability in larger clusters, we obtained the use of eight Dell 2950 III servers, each with 16GB RAM, two quad-core 3GHz Xeon processors, and with three 32GB local SAS drives in RAID-5 configuration, using a Dell perc5/i controller with 256MB of battery backed cache. Seven of the eight hosts were connected to a mid-range Fibre Channel SAN, which provided a good reference for comparison against finely tuned enterprise hardware. The SAN array, an EMC Clariion CX3-40, was configured with 15 15k Fibre Channel disks in a RAID-5 configuration, presented as single LUN mounted at each ESX host. The array was equipped with 8GB of battery backed cache, and connected to the hosts through 4GB/s Fibre Channel links. The SAN was already running in production, and we did not attempt to tune its performance for the benchmarks.

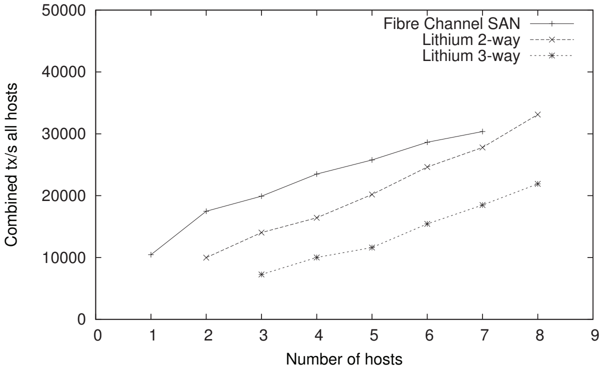

Figure 8: PostMark Scalability. Total transactions per second across all hosts, with 4 VMs per host for a varying number of hosts of Lithium in 2-way and 3-way replicated scenarios, and for a mid-range Fibre Channel storage array.

Lithium was configured for either 2- or 3-way replication over single gigabit Ethernet links, in addition to the local RAID-5 data protection. Replicas were mapped to hosts probabilistically, using a simple distributed hash table (DHT) constructed for the purpose. In this setup, Lithium was able to tolerate three or five drive failures without data loss, where the array could only tolerate a single failure. Ideally, we would have reconfigured the local drives of the servers as RAID-0 for maximum performance, as Lithium provides its own redundancy. However, the hardware was made available to us on a temporary basis, so we were unable to alter the configuration in any way.

We ran the DVDStore2 benchmark in 4 VMs per host, and varied the number of hosts. As Figure 7 shows, a three-drive RAID-5 per host is not enough to beat the high-performance array, but as more hosts are added, per-VM throughput remains constant for Lithium, whereas the array's performance is finite, resulting in degrading per-VM throughput as the cluster grows.

We also ran the PostMark benchmark (see Figure 8) in the same setting. PostMark is more IO intensive than DVDStore2, giving the SAN a larger initial advantage. However, as the SAN throughput starts to tail off, Lithium's throughput continually improves. Unfortunately, only seven hosts had working Fibre Channel links, but Lithium with double replication is likely to be as fast or faster than the array for a cluster of eight or more hosts.

Though our system introduces additional CPU overhead, *e.g.*, for update hashing, the benchmarks remained disk IO rather than CPU-bound. When we first designed the system, we measured the cost of hashing and found that a hand-optimized SHA-1 implementation, running on a single Intel Core-2 CPU core, could hash more than 270MB of data per second, almost enough to saturate the full write bandwidth of three SATA disk drives. In return for paying a modest CPU overhead, we get simple integrity checking and a scalable namespace that supports arbitrary branching of volumes without central coordination.

In summary, the performance of the Lithium prototype is on par with alternative approaches for local storage, and that its write-optimized disk layout is faster than conceptually simpler alternatives when used for for double and triple live replication. In daily use, we expect Lithium do run without local RAID protection, which is likely to be faster than the RAID-5 configuration tested here. For small setups, the Fibre Channel array is still faster, but Lithium keeps scaling, does not have a single point of failure, and is able to tolerate both host and multiple drive failures.

6. CONCLUSION

Cloud computing promises to drive down the cost of computing by replacing few highly reliable, but costly, compute hosts with many cheap, but less reliable, ones. More hosts afford more redundancy, making individual hosts disposable, and system maintenance consists mainly of lazily replacing hardware when it fails. Virtual machines allow legacy software to run unmodified in the cloud, but storage is often a limiting scalability factor.

In this paper, we have described Lithium, a fork-consistent replication system for virtual disks. Fork-consistency has previously been proposed for storing data on untrusted or Byzantine hosts, and forms the basis of popular distributed revision control systems. Our work shows that fork-consistent storage is viable even for demanding virtualized workloads such as file systems and online transaction processing. We address important practical issues, such as how to safely allow multiple outstanding IOs and how to augment the fork-consistency model with a novel cryptographic locking primitive to handle volume migration and fail-over. Furthermore, our system is able to emulate shared-storage SCSI reservation semantics and is compatible with clustered databases and file systems that use on-disk locks to coordinate access.

Lithium achieves substantial robustness both to data corruption and protocol implementation errors, and potentially unbounded scalability without bottlenecks or single points of failure. Measurements of our prototype have yielded promising results; Lithium is able to compete with an expensive Fibre Channel storage array on a small cluster of eight hosts, and is faster than traditional disk layouts when doubly or triply replicating writes between hosts.

7. ACKNOWLEDGEMENTS

The authors would like to thank Irfan Ahmad, Jørgen S. Hansen, Orran Krieger, Eno Thereska, George Coulouris, Eske Christiansen, Rene Schmidt, Steffen Grarup, Henning Schild, and Sharon Weber for input and comments that helped shape and improve this paper.

8. REFERENCES

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for ssd performance. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 57–70, Berkeley, CA, USA, 2008. USENIX Association.

[2] L. N. Bairavasundaram, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, G. R. Goodson, and B. Schroeder. An analysis of data corruption in the storage stack. *Trans. Storage*, 4(3):1–28, 2008.

[3] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd Networked Systems Design and Implementation NSDI '05*, May 2005.

[4] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh. The logical disk: a new approach to improving file systems. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 15–28, New York, NY, USA, 1993. ACM Press.

[5] J. G. Hansen and E. Jul. Lithium: virtual machine storage for the cloud. In *SoCC '10: Proceedings of the 1st ACM symposium on Cloud computing*, pages 15–26, New York, NY, USA, 2010. ACM.

[6] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, 1988.

[7] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Trans. Comput. Syst.*, 10(1):3–25, 1992.

[8] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[9] E. K. Lee and C. A. Thekkath. Petal: distributed virtual disks. *SIGOPS Oper. Syst. Rev.*, 30(5):84–92, 1996.

[10] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller. Measurement and analysis of large-scale network file system workloads. In *ATC'08: USENIX 2008 Annual Technical Conference*, pages 213–226, Berkeley, CA, USA, 2008. USENIX Association.

[11] M. Nelson, B.-H. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *Proceedings of the 2005 Annual USENIX Technical Conference*, April 2005.

[12] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *PODC '88: Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 8–17, New York, NY, USA, 1988. ACM.

[13] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[14] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[15] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 172–182, New York, NY, USA, 1995. ACM.