Towards Weakly Consistent Local Storage Systems

Ji-Yong Shin
Cornell University
jyshin@cs.cornell.edu

Mahesh Balakrishnan
Yale University
mahesh@cs.yale.edu

Tudor Marian
Google
tudorm@google.com

Jakub Szefer
Yale University
jakub.szefer@yale.edu

Hakim Weatherspoon
Cornell University
hweather@cs.cornell.edu

Abstract

Heterogeneity is a fact of life for modern storage servers. For example, a server may spread terabytes of data across many different storage media, ranging from magnetic disks, DRAM, NAND-based solid state drives (SSDs), as well as hybrid drives that package various combinations of these technologies. It follows that access latencies to data can vary hugely depending on which media the data resides on. At the same time, modern storage systems naturally retain older versions of data due to the prevalence of log-structured designs and caches in software and hardware layers. In a sense, a contemporary storage system is very similar to a small-scale distributed system, opening the door to consistency/performance trade-offs. In this paper, we propose a class of local storage systems called StaleStores that support relaxed consistency, returning stale data for better performance. We describe several examples of StaleStores, and show via emulations that serving stale data can improve access latency by between 35% and 20X. We describe a particular StaleStore called Yogurt, a weakly consistent local block storage system. Depending on the application's consistency requirements (e.g. bounded staleness, monotonic reads, read-my-writes, etc.), Yogurt queries the access costs for different versions of data within tolerable staleness bounds and returns the fastest version. We show that a distributed key-value store running on top of Yogurt obtains a 6X speed-up for access latency by trading off consistency and performance within individual storage servers.

SoCC '16 October 05-07, 2016, Santa Clara, CA, USA.
 ISBN 978-1-4503-4525-5/16/10...
DOI: http://dx.doi.org/10.1145/2987550.2987579

Devices	Throughput	Latency	Cost / GB
Registers	-	1 cycle	-
Caches	-	2-10ns	-
DRAM	10s of GB/s	100-200ns	$10.00
NVDIMM	10s of GB/s	100-200ns	$10.00
NVMM	10s of GB/s	800ns	$5.00
NVMe	2GB/s	10-100us	$1.40
SATA SSD	500MB/s	400us	$0.40
Disk	100MB/s	10ms	$0.05

Figure 1. The new storage/memory hierarchy (from a LADIS 2015 talk by Andy Warfield).

Categories and Subject Descriptors D.4.2 [*Storage Management*]: Secondary storage and storage hierarchies; C.2.4 [*Distributed Systems*]: Distributed applications; H.2.4 [*Systems*]: Concurrency; H.3.4 [*Systems and Software*]: Concurrency awareness systems

General Terms Design, Experimentation, Performance

Keywords Weak consistency, local storage

1. Introduction

The ongoing explosion in the diversity of memory and storage technology has made hardware heterogeneity a fact of life for modern cloud storage servers. Current storage system designs typically use a mix of multi-device idioms – such as caching, tiering, striping, mirroring, etc. – to spread data across a range of devices, including hard disks, DRAM, NAND-based solid state drives (SSDs), and byte-addressable NVRAM or Phase Change Memory (PCM). Each such storage medium exhibits vastly different throughput and latency characteristics; access latencies to data can vary considerably depending on which media the data resides on. Figure 1 shows the performance characteristics and cost of some of the storage options available today.

In parallel, multi-device storage systems are increasingly multi-versioned, retaining older versions of data that are typ-

ically not exposed to the application. Often, multi-versioning is a side-effect of log-structured designs that avoid writing in place; for example, SSDs expose a single-version block address space to applications, but internally log data to avoid triggering expensive erase operations on block rewrites. In other cases, tiering or caching strategies can introduce multiple versions by replicating data and synchronizing lazily; for example, SSDs typically have DRAM-based write caches that are lazily flushed to the underlying flash.

We observe that the existence of multiple versions of data within a storage system – and the non-uniform performance characteristics of the storage media that these versions reside on – creates an opportunity for trading off consistency or staleness for performance. We make a case for *weakly consistent local storage systems*: when applications access data, we want the option of providing them with stale data in exchange for better performance. This behavior is in contrast to the strong consistency or linearizability offered by existing storage systems, which guarantee that read operations will reflect all writes that complete before the read was issued [7]. Accessing older versions can provide better performance for a number of reasons: the latest version might be slow to access because it resides on a write cache that is unoptimized for reads [18], or on a hard disk stripe that is currently logging writes [16] or undergoing maintenance operations such as a RAID rebuild or scrubbing. In all these cases, accessing older versions can provide superior latency and/or throughput. Later in the paper, we describe these and other scenarios in detail.

The killer app for a weakly consistent local storage system is distributed cloud storage. Services such as S3, DynamoDB, and Windows Azure Storage routinely negotiate weak consistency guarantees with clients, primarily to mask round-trip delays to remote data centers. A client might request read-my-writes consistency, or monotonic reads, or bounded-writes consistency from the cloud service, indicating its willingness to tolerate an older version of data for better performance. For example, in the case of monotonic reads consistency, a client that last saw version 100 of a key is satisfied with any version of that key equal to or greater than 100.

Traditionally, a distributed storage service leverages weaker consistency requirements to direct the client's request to nearby servers that can provide the desired consistency (e.g., contain a version equal to or greater than 100). The server itself – typically implemented as a user-level process over a strongly consistent local storage subsystem – strenuously returns the latest value of the key that it stores (e.g., version 200), ignoring the presence of older, potentially faster versions on the underlying subsystem that would satisfy the guarantee (e.g., version 110, 125, etc.). Instead, a cloud storage service could propagate knowledge of weaker consistency requirements down to the local storage subsystem on each individual server, allowing it to return older data at faster speeds.

Accordingly, we propose a new class of local storage systems – embedded key-value stores, filesystems, and block stores – that are consistency-aware, trading off staleness for performance. We call these StaleStores. While different StaleStores can have widely differing external APIs and internal designs, they share a number of common features, such as support for multi-versioned access, and cost estimation APIs that allow applications to determine the fastest version for a particular data item. We describe several examples of StaleStores drawn from existing storage system designs in Section 2; for three such designs, we show via high-fidelity emulations that accessing older versions can improve access latency, by up to 20X, 35%, and 60% respectively.

In addition, we describe the design and implementation of a particular StaleStore: a log-structured block store called Yogurt. We implement over Yogurt a variant of a distributed cloud storage system called Pileus [19] that supports multiple consistency levels, and show that exploiting the performance/consistency trade-off within individual servers provides a 6X speed-up in access latency.

This paper makes the following contributions:

- We are the first to explore the consistency/performance trade-off within a storage server.
- We describe StaleStores, a new class of local storage systems that can weaken consistency for better performance.
- Finally, we detail the necessary APIs to trade off consistency and performance in a server, and design, implement, and evaluate a block-level StaleStore.

2. The Case for StaleStores

Our argument for StaleStores relies on two key observations. First, local storage systems are increasingly multi-versioned. Second, older versions are often faster to access. We now provide examples of such systems. In addition, we built high-fidelity emulations of three systems that are not functionally complete (e.g., they do not handle crash recovery) but faithfully mimic the I/O behavior of the originals. Using these emulations, we show that accessing older versions can significantly cut access latency.

2.1 Why are older versions sometimes faster?

S1. Single-disk log-structured stores. The simplest and most common example of a system design that internally stores faster stale versions of data is a log-structured storage system, either in the form of a filesystem [15] or block store [4]. Such systems extract sequential write bandwidth from hard disks by logging all updates. This log-structured design results in the existence of stale versions; furthermore, these stale versions can be faster to access if they are closer to the disk arm than the latest version. Previous work has

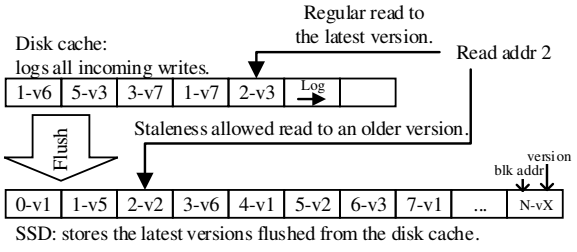
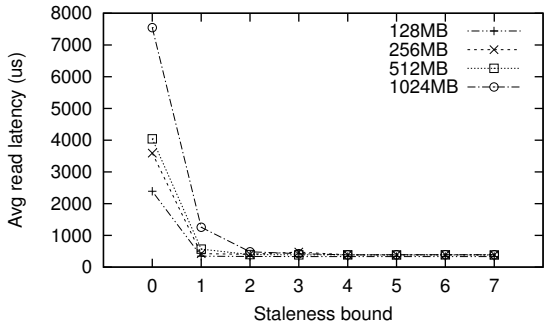
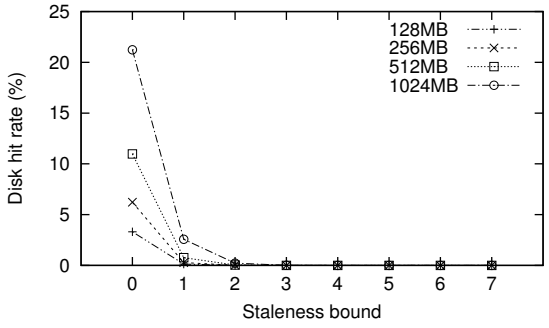


Figure 2. In the Griffin system, being able to read older versions from a SSD than the latest version from the disk cache can be faster.



(a) Read latency



(b) Disk cache hit rate

Figure 3. Read latency and disk cache hit rate of Griffin with different disk to SSD data migration trigger sizes. Accessing stale data can avoid reading from the disk.

explored storing a single version of data redundantly and accessing the closest copy [23].

S2. SSD FTLs (Flash Translation Layers). SSDs based on NAND flash act as collections of small log-structured stores: each *erase block* consists of multiple 4KB physical pages that (for many devices) must be written in sequence and are erased or reset together during garbage collection. Logging results in stale versions, which can be faster to access if the latest version happens to be in an erase block that's undergoing compaction or garbage collection.

S3. Log-structured arrays. Some designs chain a log over multiple disks. Gecko [16] is a storage array with a chained log; updates proceed to the tail drive of the log, while reads are served by all the disks in the log. In such a design, reads from disks in the body of the log are faster since they do not interfere with writes. Accordingly, reading a stale version in the body of the chained log may be faster – and less disruptive to write throughput – than reading the latest version from the tail drive.

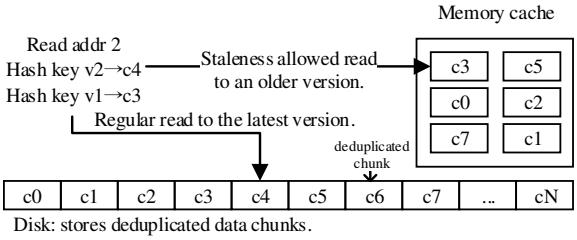
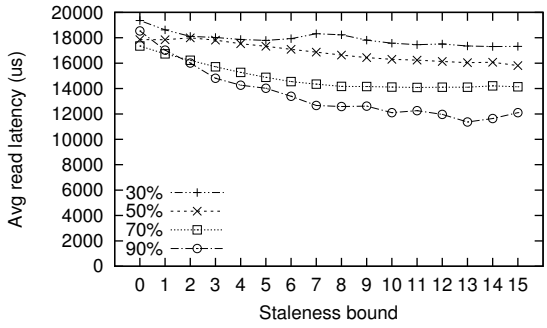


Figure 4. In a deduplicated system with cache, data items are shared with many others. If an older version is referenced by another address and is inside the cache, reading this than the latest version in the disk is faster.

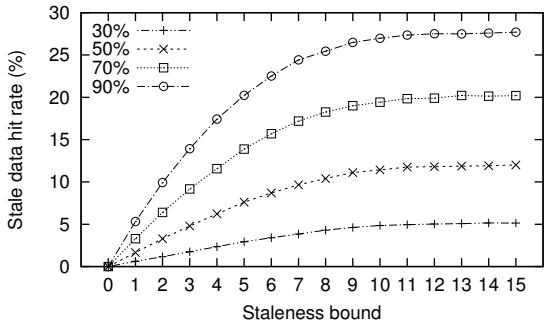
S4. Durable write caches that are fast for writes but slow for reads. Griffin [18] layers a disk-based write cache over an SSD; the goal is to coalesce overwrites before they hit the SSD, reducing the rate at which the SSD wears out. In such a system, the latest version resides on the write cache; reading it can trigger a slow, random read on the disk that disrupts write throughput. On the other hand, older versions live in the backing SSD and are much faster to access (Figure 2). A similar example is the disk-caching-disk (DCD) [9], where reads can be faster on the backing disk since the data on it has spatial locality (in contrast to the log-structured write cache, which has temporal locality).

We implemented an emulator for the Griffin system. Figure 3-(a) shows the latency benefit of serving older versions. The y-axis is the latency; the x-axis is the parameter for the bounded staleness consistency guarantee, signifying how stale the returned value can be in terms of the number of updates it omits. We run a simple block storage workload where a 4GB address space is written to and 8 threads issue random reads and writes with 9 to 1 ratio. Depending on the configuration, the Griffin system flushes data from the disk cache to SSD whenever 128MB to 1GB worth of data is written to the disk. The figure shows that allowing the returned value to be stale by even one update can reduce read latency down to 1/8 and down to 1/20 by allowing values stale by four updates. Figure 3-(b) shows the ratio of reads hitting the disk cache. Read accesses to disk can be eliminated by allowing values stale by five updates.

S5. Deduplicated systems with read caches. Systems often deduplicate data to save space. In such systems, an older version of a data item may be identical to the latest, cached version of some other data item; in this case, fetching the older version can result in a cache hit (Figure 4). Previous



(a) Read latency



(b) Cache hit rate on stale data

Figure 5. Read latency and memory cache hit rate on stale data in a deduplicated system with different deduplication rates. Accessing stale data results in higher cache utilization and lower read latency.

work has explored the use of content similarity to speed up access to data [10].

Figure 5 shows the performance and memory cache hit rate on stale data of a consistency-aware deduplication system. The system is a block store which deduplicates 4KB blocks. 8 threads randomly read and write blocks with 9 to 1 ratio within 4GB address space. The overall deduplication ratio is controlled to be 30 to 90 percent. The system uses a disk as a primary storage and 256MB DRAM as a read/write cache, which is indexed by the hash key and uses LRU policy. The performance improvement plateaus as the allowed staleness bound increases, but the performance is improved by 10 to 35% depending on the deduplication ratio (Figure 5-(a)). Such performance improvement trends follow the cache hit rate on stale data (Figure 5-(b)).

S6. Fine-grained logging over a block-grain cache. Consider a log-structured key-value store implemented over an SSD (e.g., like FAWN [1]), which in turn has an internal DRAM cache. New key-value updates are merged and written as single blocks at the tail of a log layered over the SSD's address space. As key-value pairs are overwritten, blocks in the body of the log hold progressively fewer valid key-value pairs, reducing the effectiveness of the DRAM cache within the SSD. However, if stale values can be tolerated, the effectiveness of the DRAM cache increases since it holds valid

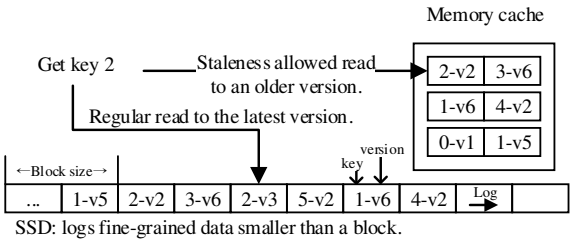
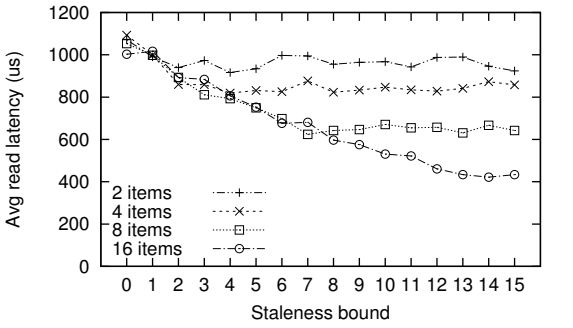
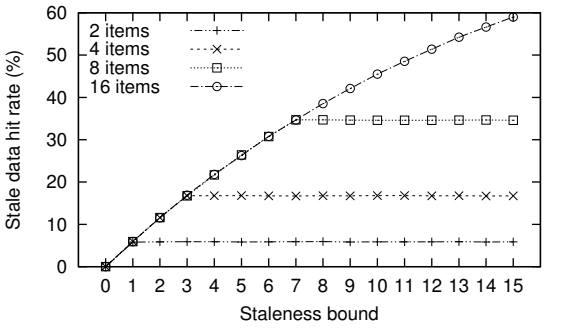


Figure 6. If data items are smaller than the cache block in a fine grained logging system, other items (e.g. 2-v2) can follow an item (e.g. 3-v6) being read into the cache. If the item that followed is an older version, accessing the older item can be faster than the latest item in the SSD.



(a) Read latency



(b) Cache hit rate on stale data

Figure 7. Read latency and memory cache hit rate on stale data in system with fine-grained logging over a block-grain cache. The read latency decreases and the cache hit rate increases when the allowed staleness of data and the number of items placed in a cache block increase.

versions – stale or otherwise – for a larger set of keys (Figure 6).

Figure 7 shows the performance of a simple emulation of a key-value store layered over an SSD with a 256MB DRAM cache. 1 million key-value pairs are present in the system and 8 threads randomly read and write them with 9 to 1 ratio. The key-value pair size is parameterized such that 2 to 16 pairs can be stored in a block. If N key-value

pairs fit in a block, at most $N-1$ key-value pairs can be wastefully loaded in the cache. Allowing access to older versions using the bounded staleness consistency re-enables utilization of potentially wasteful data items in the cache block (Figure 7-(b)) and higher utilization of DRAM cache results in increased performance up to 60% (Figure 7-(a)).

S7. Systems storing differences from previous versions. Some log-structured systems store new data as deltas against older versions. In Delta-FTL [22], when a logical address is overwritten, the system does not log an entire new 4KB block; instead, it logs a delta against the existing version. For instance, if only the first 100 bytes of the block have been changed by the overwrite, the system only logs the new 100 bytes. In such a system, accessing the latest version requires reading both some old version and one or more deltas, triggering multiple 4KB page reads. Accessing an older version can be faster since it requires fewer reads.

All the above cases keep older versions to achieve better performance, storage durability, storage utilization, ease of data maintenance, and so on, but being able to access older versions faster than the latest version is a side effect. In this paper, we explicitly utilize this side effect – whenever it is possible – to speed up storage performance. To build a system to access potentially faster older versions, the system should 1) maintain multiple versions of data, 2) be aware of access speed/cost for each version, 3) guarantee consistency semantics to safely access old versions, and 4) support new APIs for these new features.

3. Design Space for StaleStores

For any cloud storage service, the software stack on a single server typically contains a top-most layer that runs as a user-space process and exposes some high-level abstraction – such as key-value pairs and files – over the network to applications running on remote clients. This process acts in concert with other processes to implement a distributed storage service; for example, it might act as a primary or secondary replica, or as a caching node. If the distributed storage service supports weaker consistency guarantees, clients can mandate that reads satisfy some such guarantee (such as read-your-writes or monotonic reads); typically they do so by specifying some set of versions which are permissible. Many systems rely on *timestamps* that provide an ordering across versions; reads can then specify the earliest timestamp they can tolerate without violating the required guarantee.

As a concrete example of a cloud storage service that supports weaker consistency levels, the Pileus system [19] consists of a single primary server and multiple backup servers. A client writes a new key-value pair by sending it to the primary, which assigns a monotonically increasing timestamp to it before writing it to local storage. The primary then asynchronously sends the update to the backups, which apply updates in timestamp order. As a result, a global ordering exists across all updates (and consequently all versions of data). At any given point in time, each backup server contains a strict prefix of this global order corresponding to some timestamp. Clients can then obtain weaker consistency guarantees by specifying a timestamp for their reads, and contacting the closest server that is storing a prefix which extends beyond this timestamp.

For example, a client has written a key-value pair at the primary and was told by the primary that the write's timestamp is T_{44}. The primary has seen 100 writes, including the client's write, and assigned them timestamps T_1 to T_{100}. Backup A has seen writes up to timestamp T_{50}. Backup B has seen writes up to T_{35}. The client then wishes to issue a read on the same key K satisfying the read-your-writes guarantee; i.e., it requires the read to reflect its own latest write, but not necessarily the latest writes by other clients. Accordingly, it contacts the backup server closest to it with a read request annotated with T_{44}. Backup B cannot satisfy this request since it has seen writes only up to T_{35}. Backup A, on the other hand, can satisfy this request by returning any version of K with a timestamp higher than or equal to T_{44}.

In current distributed storage services, each individual server is typically single-versioned (unless the distributed service exposes reads to older versions as a feature). Specifically, existing systems do not have individual servers selectively returning older versions in order to gain better performance from their local storage stack. In the example above, we want backup A to be capable of selecting a version between T_{44} and T_{50} that can be returned the fastest from its local storage. This is the capability we seek to explore.

3.1 What is a StaleStore?

Abstractly, a StaleStore is a single-node storage system that maintains and serves multiple versions. Different StaleStores support different application-facing APIs – such as files, key-value pairs, block storage, etc. – that are augmented in similar ways to allow applications to trade off consistency for performance.

In designing the StaleStore abstraction, we observe that the information required to support consistency and performance trade-offs is typically split between the application and the store. The application (i.e., the server process implementing the distributed cloud store) understands consistency (i.e., timestamps), and the store understands performance characteristics (i.e., where data is placed and how fast it can be accessed). Required is an API that allows performance information to flow up the stack and consistency information to flow down the stack. Specifically, we push consistency information down the stack by associating versions within the multi-version store with application-level timestamps; conversely, we push performance information up the stack by allowing applications to query the estimated cost of issuing a read operation against a specific version.

Accordingly, a StaleStore API has four characteristics. In the following descriptions, we use the terms 'timestamp' and 'version number' interchangeably. In addition, we use the

Key-Value StaleStore API	Parameters	Description
Get	key, version #	Reads a key corresponding to the version #.
Put	key, version #, value	Writes a key with the specified value and version #.
GetCost	key, version #	Returns an integer cost to access the specified key with the version #.
GetVersionRange	key, version #	Returns a range of version #s within which a version of a key is valid.

Table 1. Example Key-Value StaleStore.

term 'snapshot' to define a consistent view of the data store from the viewpoint of the storage at a particular timestamp.

- **Timestamped writes:** First, writes to the StaleStore are accompanied by a monotonically increasing timestamp. This version number is global across all writes to the StaleStore; for example, for a key-value store, each put operation must have a non-decreasing timestamp, regardless of which key-value pair it touches.
- **Snapshot reads:** Second, the application should be able to read from a consistent, potentially stale snapshot corresponding to a timestamp. Read APIs are augmented with a timestamp parameter. A read operation at a timestamp T reflects all writes with a lower or equal timestamp. For example, for a key-value store, if a particular key has been updated by three puts at timestamps T_7, T_{33} and T_{56} respectively, a get operation at timestamp T_{100} will return the value inserted by the put at T_{56}, which reflects the latest update at timestamp T_{100}.
- **Cost estimation:** Third, the application should be able to query the cost of issuing a particular read operation at a snapshot. This cost is an arbitrary integer value that may not correspond to real-world metrics such as latency or throughput; all we require is that two cost estimates from the same StaleStore can be compared.
- **Version exploration:** Finally, the application should be able to determine – having read a particular version of an item – what range of timestamps that version is valid for. For example, if the application reads an item X at timestamp T_7, and that item does not change next until timestamp T_{33}, the application can optimize cost querying operations with this information, or read other items at any timestamp in between and still obtain a consistent snapshot across items.

Table 1 shows an example API for a key-value StaleStore. It provides an API for timestamped writes (*Put*), snapshot reads (*Get*), cost estimation (*GetCost*), and version exploration (*GetVersionRange*).

Why timestamps instead of consistency guarantees? Making the single-node store aware of individual guarantees (such as read-my-writes or monotonic reads consistency) is challenging; these guarantees can be application-specific and refer to application-level entities (e.g., the session consistency guarantee requires a notion of an application-level session started by a specific client). In contrast, timestamps are compact, simple and sufficient representations of consistency requirements, and are used by a wide range of systems to provide weak consistency in a distributed setting. The higher layer simply tags every read and write with a global timestamp.

What about concurrency control? One approach to implementing the above API in a real system involves guarding all data with a single, coarse lock. In this case, it's simple for application logic to ensure that writes are always in non-decreasing timestamp order, and that reads reflect writes with prior timestamps. In practice, however, the application can use fine-grained locking to issue requests in parallel, while providing the same semantics as a single lock. For example, in a key-value store, puts to different keys can proceed in parallel, while a get on a key has to be ordered after any puts to that key with a lower timestamp. We expect the application to implement concurrency control above the StaleStore API (in much the same way a filesystem implements locking above a block store API, or a key-value store implements locking above a filesystem API), while ensuring that the semantics of the system are as if a single lock guards all data.

3.2 Which layer should be a StaleStore?

The API exposed by an individual server within a cloud storage service to external clients typically mirrors the API of the cloud storage service. For example, a storage service might expose a key-value API to applications allowing them to put and get key-value pairs; each individual server exposes the same API to client-side logic used by the application to access the service. We call this the *public-facing API*.

Internally, each server runs a process (the application from the StaleStore's perspective) that implements the public-facing API over some internal, single-server storage API; we call this the *internal API*. The internal API could be provided by a filesystem like ext3, an embedded key-value store like LevelDB or RocksDB, a single block store such as Storage Spaces. These are the internal APIs that we propose augmenting to support consistency/performance trade-offs, as described above. Each of these internal subsystems could be a multi-versioned StaleStore, allowing the application to request older versions from them in exchange for better performance. Alternatively, the application could be implemented over one or more unmodified, single-versioned stor-

age subsystems, and itself act as an application-level StaleStore, managing older versions and accessing the fastest one. Below, we discuss the implications of each option:

Application-level StaleStore: In this option, the application-level storage system manages and maintains versions across unmodified single-version stores (filesystems, key-value stores, block devices), with no support from the underlying local storage stack. This approach has one significant benefit: the application is aware of the consistency guarantee required (or equivalently, of high-level constructs such as timestamps), and knows which versions will satisfy the read. It also has a significant drawback: the application is a user-space process that typically has little visibility or control over the location of data on physical devices. Multiple layers of indirection – in the form of logs, read caches and write caches – can exist between the application and raw hardware. While the application can explicitly maintain versions over a logical address space (a file or a block device), it cannot predict access latencies to individual addresses on each address space.

Filesystem / embedded key-value StaleStore: In this option, the application stores all its data in a filesystem or embedded key-value StaleStore. An important benefit of such an approach is generality and reusability: a filesystem StaleStore can be reused by multiple cloud storage systems. On the flip side, it itself operates over a logical address space – a block device – and has little visibility into where blocks are mapped, making it difficult to estimate the cost of reads to particular versions. This is particularly true with the advent of 'smart' block layers in hardware (e.g. SSDs) and software (e.g. Microsoft's Storage Spaces), which are sophisticated, multi-device systems in themselves. However, certain types of StaleStores can only be implemented at the filesystem or key-value store level; one example is scenario S6 from Section 2, in which a key-value store combines fine-grained logging with a block-grain buffer cache over a block address space with relatively uniform access latencies.

Block-level StaleStore: The third option is for a smart block layer to manage, maintain, and expose versions. The block layer has detailed information on where each block in its address space lives, and can provide accurate access latency estimates. Further, the block device shares the advantage of the filesystem: implementing tunable consistency within the block device allows new high-level storage systems – such as graph stores, new types of filesystems, table stores, databases, etc. – to easily support consistency/performance trade-offs without reimplementing the required mechanisms. We now describe the design and implementation of a block-level StaleStore called Yogurt.

4. Yogurt Design

Yogurt is a block-level StaleStore. It exposes a simple, block-level StaleStore API (shown in Table 2) that supports timestamped writes, reads and cost estimation. This API is necessary and sufficient for adding StaleStore functionality to a block store; it is analogous to the example key-value StaleStore API shown previously.

Building a block-level StaleStore poses some unique challenges. Applications might prefer to use the standard POSIX-style API for reads and writes to minimize changes to code, and also to use the existing, highly optimized I/O paths from user-space to the block storage driver. Also supporting application-level data abstractions, such as files and key-value pairs, necessitates multiple block accesses. Supporting these require some deviation from the basic StaleStore API. Specifically, Yogurt provides an alternative wrapper API where applications can specify timestamp via explicit control calls (implemented via IOCTLs) and follow those up with POSIX read/write calls.

4.1 Block-level StaleStore API

The Yogurt API is simple and matches the generic characteristics of a StaleStore API described in Section 3. *ReadVersion(block addr, version)* reads a block corresponding to the version number (specifically, the last written block with a timestamp lower than the supplied version number), and *WriteVersion(block addr, version, data)* writes a block data with the given version number. It is identical to accessing a simple block store, but with an additional version number to read and write the data.

GetCost(block addr, version) is the cost estimation API. The versioned block store computes the integer value to return which can be compared against other *GetCost* calls' results. The smaller the number, the smaller the estimated cost to access it. Depending on the underlying storage settings this number can be configured differently and more details will be presented in Section 5.

GetVersionRange(block addr, version) returns a lower and upper bounds of snapshots that contains the specified block intact. An identical block of data can be part of multiple snapshots. This API returns the version number when the block data was last written before the given version number and the version number when the block data is overwritten after the given version number.

4.2 Wrapper APIs

As mentioned previously, a standard StaleStore API – consisting of timestamp-augmented versions of the original calls – is problematic for a block store, since it precludes the use of the highly optimized POSIX calls. A second issue for applications is the granularity mismatch between the application and the block store. Application-level consistency is defined at a grain that is either smaller (e.g. small key-value pairs) or larger (e.g. large files) than a single block. In addition, a single access to an application-level construct like a key-value pair or a file often requires multiple accesses at the block level (e.g., one access to look up a key-value index or a filesystem inode; a second access to read the data). If these multiple writes are sent to the StaleStore with dif-

StaleStore APIs	Parameters	Description
ReadVersion	Block address, version #.	Reads a block corresponding to the version #.
WriteVersion	Block address, version #, data.	Writes a block with the specified version #.
GetCost	Block address, version #.	Returns an integer cost to access the specified block with the version #.
GetVersionRange	Block address, version #.	Returns the snapshot version range where the block data is intact.

Wrapper APIs	Parameters	Description
POSIX APIs		Does basic block I/Os such as read, write, seek, etc.
OpenSnapshot	Version #	Opens snapshot.
CloseSnapshot		Closes snapshot and flushes writes.

Table 2. Yogurt APIs

ferent timestamps, the application could potentially access inconsistent snapshots reflecting one write but not the other (e.g., it might see the inode write but not the subsequent data write). Required is a wrapper API that allows applications to use the POSIX calls as well as ensure that inconsistent states of the store cannot be seen.

The answer to both these questions lies in a wrapper API that exposes *OpenSnapshot* and *CloseSnapshot* calls. *OpenSnapshot(version)* opens a snapshot with the specified version number. If the version number is invalid, the operation will fail. The application that opened a snapshot can read one or more blocks within the snapshot using the POSIX read APIs until it closes the snapshot by calling *CloseSnapshot()*.

If the snapshot accessed by the *OpenSnapshot(version)* call is from the past, one cannot directly write new data onto it. To write data, the application supplies a timestamp to *OpenSnapshot()* greater than any the StaleStore has seen before; this opens a writeable snapshot. The application can then write multiple blocks within the snapshot, and then call *CloseSnapshot* to flush the writes out to the store.

Note that the *OpenSnapshot*/*CloseSnapshot* wrapper calls do not provide a full transactional API; they do not handle concurrency control or transactional isolation; the application has to implement its own locking above the wrapper API. However, these calls do provide failure atomicity over the basic StaleStore API.

Under the hood, *OpenSnapshot* simply sets the timestamp to be used for versioned reads and writes. Reads within the snapshot execute via the *ReadVersion* call in the StaleStore API; *CloseSnapshot* flushes writes to the underlying store using the *WriteVersion* call.

4.3 Versioned storage design

Yogurt implements the block-level StaleStore API over a number of raw block devices, handling I/O requests to multi-versioned data and offering cost estimates for reads. The versioned block store in Yogurt is patterned after the Gecko [16] system (see S4 in Section 2), which chains a log across multiple devices such that new writes proceed to the tail drive while reads can be served by any drive in the chain. Yogurt maintains a multi-version index over this chained log that maps each logical block address and version number pair to a physical block address on a drive in the chain (in contrast to Gecko, which does not provide multi-version access).

The wrapper layer makes sure that a block is never overwritten with the same version number and a set of writes corresponding to a new snapshot is not exposed to applications other than the one issuing the writes until all writes are persisted in the versioned block store. When *WriteVersion* is called, the versioned block store updates the multi-version index and appends new block data to the log. Similarly, *ReadVersion* simply returns the data corresponding to the address and version pair.

Because the versioned block store sits right on top of block devices, it knows the block device types, characteristics, and how busy each device is. Based on the physical location of each versioned block data, the versioned block store can estimate the cost for accessing a particular version of the block. When *GetCost* API is invoked, the multi-version index is looked up to figure out the physical location of the data, and the access cost is computed based on the storage media speed and the number of queued requests.

5. Implementation

Yogurt is implemented as a device mapper, which is a Linux kernel module similar to software RAID and LVM. The wrapper and StaleStore APIs other than the POSIX APIs are implemented as *IOCTL* calls and kernel function calls.

5.1 Snapshot Access and Read Mapping

Since modern applications are highly concurrent and serve multiple users, Yogurt should be able to service multiple snapshots to one or many applications. To do this, Yogurt identifies its users using *pid*, which is distinctively given to each thread. When a thread calls *OpenSnapshot*, all read requests from the thread are served from the opened snap-

shot until the thread calls *CloseSnapshot*. Once a thread is mapped with a snapshot, each read is tagged with the snapshot number and issued via the *ReadVersion* API.

Figure 8 shows a logical view of a multi-version index and how a snapshot is constituted. The x-axis is the logical block address and the y-axis is the snapshot version number. Each entry shows a physical block address and a version number corresponding to the logical address. The entries in the same row are the blocks that were written when the snapshot was created. Thus, a snapshot consists of the latest data blocks with version numbers less than or equal to the snapshot's version number.

When the application wishes to access an application-level data item with a certain consistency level, it translates that to a lowest acceptable version number, which we call V_{low}. It then uses the latest snapshot version as V_{up}. Once the application knows the upper and lower bounds V_{up} and V_{low}, it can issue multiple *GetCost* queries within that range. We leave the querying strategy to the application; however, one simple strategy is to assign a query budget Q, and then issue Q *GetCost* requests to a number of versions V_{query} that are uniformly selected between V_{up} and V_{low}:

$$V_{query} = V_{low} + \lfloor ((V_{up} - V_{low})/(Q-1)) \times n \rfloor, \qquad (1)$$

where $\{n \in \mathbb{Z} | 0 \leq n \leq Q\}$. For example, for upper bound 9, lower bound 5, and querying budget 3, get cost is issued to versions 5, 7, and 9. Depending on the returned costs, the application reads the cheapest version and updates V_{low}, if necessary. If the returned costs are the same for different versions, the application prefers older versions to keep the query range large.

Here, notice that if multiple blocks need to be accessed to read an object (e.g. a file that spans multiple data blocks), the blocks accessed after a block become dependent on the previously accessed block. For example, if an application reads a metadata block, the data block locations are valid only for the snapshots where the metadata block is valid. Say V_{low} and V_{up} were initially set to 0 and 10, respectively by a read semantic and the application read version 1 of logical block 2 in Figure 8. Then V_{low} and V_{up} becomes 1 and 5, respectively, which is the range the read block is valid. If version 3 of logical block 7 is read next, the version range becomes $V_{low} = 3$ and $V_{up} = 5$, which is the common range for the two blocks. Similarly, once the application opens a snapshot and reads a block, *GetVersionRange* should be called to update the common V_{low} and V_{up} range while reading the blocks.

5.2 Data Placement

To provide as many read options with different access costs as possible, it can be helpful for Yogurt to save different versions of a same block to different physical storage media. Yogurt uses two data layers to do this: the lower layer consists of the chained log with multiple disks and/or SSDs, and the higher layer is built as a memory cache over the chained

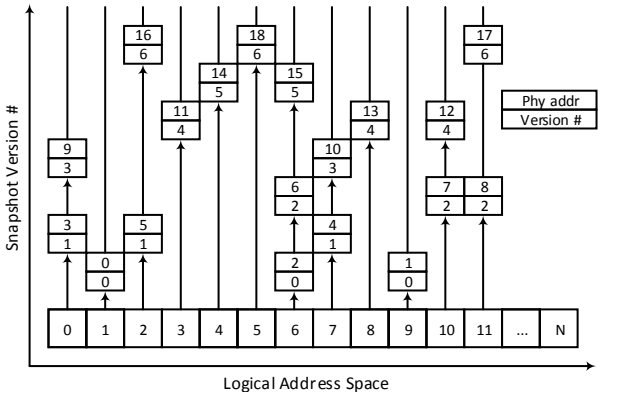


Figure 8. Logical illustration of multi-version index and snapshots

log. The memory cache is a LRU based read or read/write cache, where the data written to or read from the bottom layer is cached. When the cache acts as a read/write cache, the data writes through the cache for durability. Perfectly distributing different versions to different block devices is doable in the lower layer, but it can make the versioned block store design complicated and can cause data skew to certain block devices depending on the workload. Instead, the versioned block store uses a simpler approach: data is logged using small segments to each block device in round robin (e.g. log 16 MB segment to disk 0, log the next 16MB to disk 1, log the next 16MB to disk 2, and then back to disk 0). This results in RAID-0 like throughput behavior, by enabling independent access to each block device.

5.3 Read Cost Estimation

Based on the physical storage layer described in the previous subsection, Yogurt returns two-tiered estimated cost. For all *GetCost* calls, the versioned block store first looks up the memory cache. If the searching data block is inside the memory cache, it is always faster to read it from the memory than from either disk or SSD. To indicate this, the cost is returned as a negative value.

If the data block is not in the cache, the cost reflects the number of queued I/Os of the block device containing the block. The versioned block store can trace this using simple counters. For disks and SSDs, precise cost estimation is difficult because the internal states of the block devices are not exposed. Still, there are several known facts that can be applied for estimating the cost: 1) all writes within Yogurt are sequential log appends; 2) mixing random and sequential I/Os within disks results in overall bad performance; 3) mixing reads and writes can penalize read operations in SSDs; and 4) random read latencies of SSDs are orders of magnitude faster than those of disks. Since data blocks are read from a log, we can assume most reads will be accessing physical blocks randomly, and from 1), 2), and 3), separating reads from writes becomes important. So we add more cost

to block devices with queued writes and add small cost for queued reads. From 4) we make the cost of reading a disk an order of magnitude more expensive than reading a SSD.

To summarize, there is no cost difference among cached blocks, and cached blocks are the cheapest. SSDs are preferred over disks most of the time, unless there is an order of magnitude more I/Os queued on SSDs. Queued writes are more expensive than reads. Costs are computed as follows:

$$C_{Cache} = -1 \quad (2)$$

$$C_{SSD} = C_{rd\_ssd} \times N_r + C_{wr\_ssd} \times N_w \quad (3)$$

$$C_{Disk} = C_{rd\_disk} \times N_r + C_{wr\_disk} \times N_w \quad (4)$$

where the C variables are the costs of reading from or writing to SSD or disk, and the N variables are the number of queued reads and writes.

5.4 A Key-Value Store Example

We describe an example of a key-value store implementation to demonstrate how the Yogurt APIs can be used. The key-value store returns the fastest value of the key while satisfying the consistency constraints of each client.

The key-value store works in the following steps: 1) When a client connects to the key-value store, a session is created for the client and the latest snapshot number of the connected server is used to set up V_{low} values for the key-value pairs depending on the consistency semantics. 2) When the client issues a read to a key-value pair, V_{up} is set to the latest snapshot number of the server and *GetCost* calls are issued to different versions of the metadata block of the key. 3) Based on the returned cost of different versions of the metadata block, the key-value store calls *OpenSnapshot* to read the cheapest version of the block. 4) After the read, *GetVersionRange* is called and V_{low} and V_{up} are updated. 5) Next, the key-value store reads data blocks one by one by calling *GetCost* to versions between V_{low} and V_{up} and going through steps 3) to 4) repeatedly. 6) When the value of the key is completely read, *CloseSnapshot* is called. 7) Finally, depending on the consistency semantics, V_{low} and V_{up} are updated for future reads (e.g. under monotonic-reads, the version of the key-value pair that has been read is recorded in V_{low} and later when the client issues another read, the latest available snapshot number in the server becomes V_{up}).

As shown in the example, it is the responsibility of the application developer to wrap around the access to a single data object using *OpenSnapshot* and *CloseSnapshot*. In addition, *OpenSnapshot* should be repeatedly called within V_{low} and V_{up} range to read multiple blocks so that a data object that spans multiple blocks are read from a consistent snapshot.

6. Evaluation

To evaluate the benefit of Yogurt, we implemented a distributed storage service patterned on Pileus [19], where a client accesses a primary server and a secondary server. The primary server always has the latest data and is far away; the secondary server can be stale but is closer to the client. We tested against two variants of this system: one where the distributed service exposed a block API (matching the block store abstraction provided by Yogurt), and a second where it exposed a key-value service to clients. We call these Pileus-Block and Pileus-KV, respectively, and the latter follows the implementation of the example key-value store in the previous section.

The hardware configuration we use under Yogurt is three disks and 256MB memory cache. Data is logged to three disks in round robin in 1GB segments, using a design similar to Gecko's chain logging with smaller segment size. The memory cache can be enabled or disabled as will be described in each experiment.

Throughout the evaluation we aim to answer the following questions:

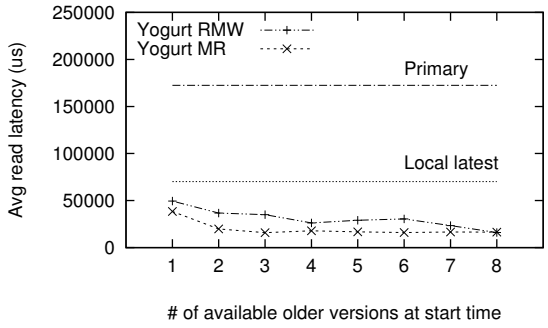
- What is the performance gain we can get by accessing stale data?
- Is there any overhead for accessing older versions?
- How well do real applications run on Yogurt?

6.1 Pileus-like Block Store

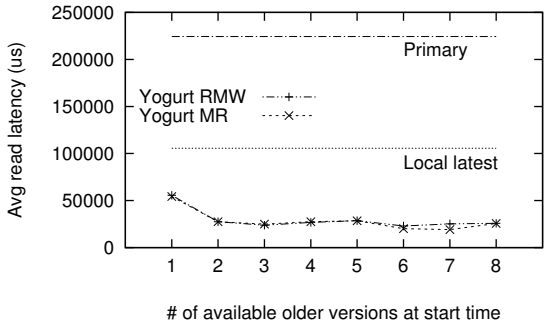
First, we measure the base performance of Yogurt when it is used with a distributed block store, comparing accessing older versions versus accessing only the latest versions. We compare Yogurt against two baseline settings, where the latest versions can be interpreted in different ways: 1) we compare against accessing the latest version within the local server, and 2) against accessing a remote primary server where the globally latest versions reside. We emulate the network latency of accessing the primary server as if it is located across the continental US from the client, delaying the response by 100ms.

For this evaluation we use two different workloads, uniform random and zipf workload that access 256K blocks. In the local server, we run a thread that aggressively writes a stream of data coming from the primary node and measured the performance from 8 threads that are reading and writing data in 9 to 1 ratio. The threads run with read-my-writes (RMW) or monotonic reads (MR) consistency guarantees and we start the threads after making N versions of data available to them. Figure 9 shows the average read latency of 3 runs.

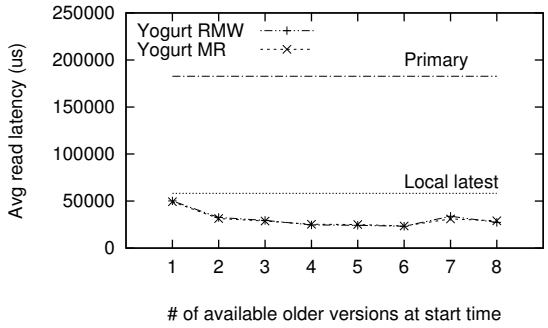
For all cases, accessing the primary server takes the longest as the added network latency is relatively huge and then comes accessing the latest version in the local storage. The latest data in the local server is mostly found in the disk that is writing data and most requests tend to concentrate on this disk. However, Yogurt can find alternative versions from different disks. The latency quickly drops to 20-25% of accessing the latest version in local storage as the *GetCost* calls enable faster data retrievals. Since there are three



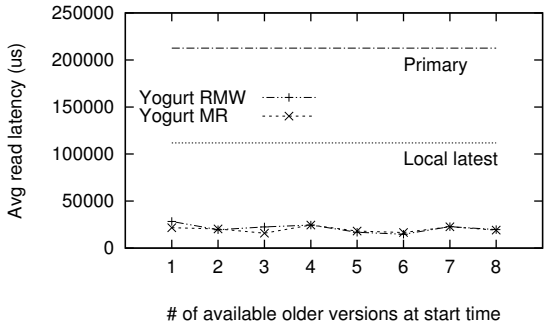
(a) Without memory cache - Uniform random



(b) Without memorycache - Zipf



(c) With memory cache - Uniform random



(d) With memory cache - Zipf

Figure 9. Performance of Yogurt under synthetic workloads.

disks the best performance is found after being able to access three or more older versions. Monotonic reads semantics show slightly better performance than read-my-writes semantics because writes from the threads that use read-my-writes limit the version range to explore before reading a data that has been written by the thread. Still, being able to explore staleness of one update can provide over 50% latency reduction.

Figures 9 (c) and (d) show the cases with memory cache. Although the overall performance of the baseline in (c) and (d) is comparable or better than that of the cases without memory cache (Figures 9 (a) and (b)), Yogurt can still return data quicker than the baselines. When the cache is missed Yogurt can bring quicker versions as shown with the case without the cache (Figure 9 (a) and (b)). Also if a certain version is in the cache (it can be an older version that has been read) Yogurt can reuse the data with better efficiency. For this reason zipf workload that has skewed data access can immediately get large performance gain (Figure 9-(d)). This result also shows that Yogurt can take advantage of heterogeneous storage media efficiently.

6.2 GetCost Overhead

To access older versions from Yogurt, applications call *GetCost* before every read to find out the lowest cost version. Comparing the cost retrieved from Yogurt is trivial as it is a simple $O(N)$ comparison of numbers. However, *GetCost* function call crosses the user space and kernel space boundary and involves copying information which can incur additional latencies.

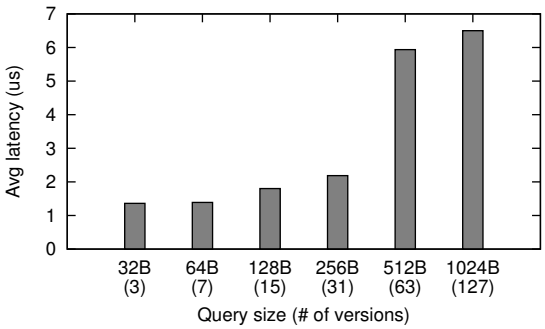


Figure 10. GetCost overhead and query size.

Figure 10 shows the *GetCost* latency of differently sized queries. Larger query size means asking for the cost of larger number of older versions. The larger the query size, the greater the *GetCost* latency. However, considering the read latency of a disk or a SSD which can be tens of microseconds to hundreds of milliseconds, the *GetCost* latency in the figure is very small. All our performance related experiments use 64B queries and results show that we can get far more latency improvements than the 1.4 microsecond overhead.

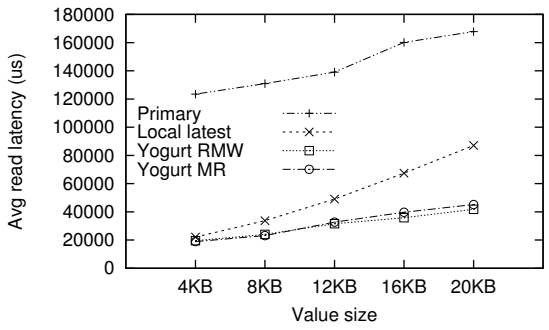


Figure 11. Key-value store's read latency and value size.

6.3 Pileus-like Key-Value Store

Pileus-KV uses a persistent hashtable over the Yogurt block address space in order to store variable-sized key-value pairs. We ran YCSB workload *A* which is composed of 50% write and 50% read on the key-value store, choosing keys according to a zipf distribution, and measured the performance. The values of the keys can be partially updated when only part of the value changes. In the experiment, the value size is varied from 4KB to 20KB, which is equivalent to 1 to 5 data blocks. To read or write a key-value pair at least one additional metadata block must be accessed to locate the block that is storing the key. We evaluate Yogurt's capability to access stale data that spans multiple blocks using *GetVersionRange* API with *GetCost* calls.

We used the same server configurations as for the Pileus-like block store and used the memory cache. There are 16 threads accessing the key-value store and a stream of incoming writes from the primary. Figure 11 shows the average read latency. As the value size grows to span multiple blocks, Yogurt can provide multiple options for selecting each block. The gap between accessing the latest block from the local storage and accessing older versions grows as the value size gets larger. The key-value store is querying costs every time before it reads, so the overall approach is a simple greedy selection. More sophisticated selection schemes can be proposed to further improve the performance, but the figure shows that for both read-my-writes and monotonic reads semantics greedy selection can already lead to better performance than the baselines.

7. Related Work

The idea of trading off consistency – defined as data freshness – for performance or availability in a distributed system has a rich history. Today, cloud services ranging from research prototypes such as Pileus [19] and production cloud services such as Amazon SimpleDB offer variable consistency levels.

A number of storage systems are multi-versioned for functionality rather than performance. These include WAFL [8] and other filesystems [3, 12], as well as block stores [5, 14]. Other systems have explored redundancy for better performance [23] and reliability [13]. To the best of our knowledge, none of these systems provide a performance vs. staleness trade-off.

Yogurt fits into a larger body of work focused on changing or augmenting the block storage and its APIs [2, 6, 11, 17, 20, 21, 24] to either simplify applications or improve performance by pushing functionality down the stack.

8. Conclusion

In this paper, we repurposed a well-known distributed systems principle within the context of a single server: storage systems should expose older versions to applications for better performance. This principle is increasingly relevant as we move towards a post-disk era of storage systems that are often internally multi-versioned and multi-device. Today, distributed storage services in the cloud can benefit from this principle by pushing relaxed consistency requirements (negotiated between the client and the service) down the stack to the storage subsystem on each server. In the future, we believe that new applications will emerge on a single machine that can work with weaker consistency guarantees in exchange for better performance.

Acknowledgments

This work is partially funded and supported by a SLOAN Research Fellowship received by Hakim Weatherspoon, DARPA MRC (FA8750-11-2-0256) and CSSG (D11AP00266), NSF (0424422, 1047540, 1053757, 1151268, 1419869, 1422544), NIST (60NANB15D327), Cisco and Intel. We would like to thank the anonymous reviewers for their constructive comments.

References

[1] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *ACM Symposium on Operating Systems Principles*, 2009.

[2] C. Chao, R. English, D. Jacobson, A. Stepanov, and J. Wilkes. Mime: a high performance parallel storage device with strong recovery guarantees. Technical report, HPL-CSP-92-9, Hewlett-Packard Laboratories, 1992.

[3] B. Cornell, P. A. Dinda, and F. E. Bustamante. Wayback: A user-level versioning file system for linux. In *USENIX Annual Technical Conference, FREENIX Track*, 2004.

[4] W. De Jonge, M. F. Kaashoek, and W. C. Hsieh. The logical disk: A new approach to improving file systems. *ACM SIGOPS Operating Systems Review*, 27(5):15–28, 1993.

[5] M. Flouris and A. Bilas. Clotho: Transparent data versioning at the block I/O level. In *International Conference on Massive Storage Systems and Technology*, 2004.

[6] G. R. Ganger. *Blurring the line between OSes and storage devices*. School of Computer Science, Carnegie Mellon University, 2001.

[7] M. P. Herlihy and J. M. Wing. Axioms for concurrent objects. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1987.

[8] D. Hitz, J. Lau, and M. A. Malcolm. File system design for an NFS file server appliance. In *USENIX Winter Conference*, 1994.

[9] Y. Hu and Q. Yang. DCD – disk caching disk: A new approach for boosting I/O performance. In *ACM/IEEE International Symposium on Computer Architectures*, 1996.

[10] R. Koller and R. Rangaswami. I/O deduplication: Utilizing content similarity to improve I/O performance. *ACM Transactions on Storage (TOS)*, 6(3):13, 2010.

[11] M. Mesnier, G. R. Ganger, and E. Riedel. Object-based storage. *IEEE Communications Magazine*, 41(8):84–90, 2003.

[12] K.-K. Muniswamy-Reddy, C. P. Wright, A. Himmer, and E. Zadok. A versatile and user-oriented versioning file system. In *USENIX Conference on File and Storage Technologies*, 2004.

[13] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Iron file systems. In *ACM Symposium on Operating Systems Principles*, 2005.

[14] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *USENIX Conference on File and Storage Technologies*, 2002.

[15] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *ACM Symposium on Operating Systems Principles*, 1991.

[16] J.-Y. Shin, M. Balakrishnan, T. Marian, and H. Weatherspoon. Gecko: Contention-oblivious disk arrays for cloud storage. In *USENIX Conference on File and Storage Technologies*, 2013.

[17] J.-Y. Shin, M. Balakrishnan, T. Marian, and H. Weatherspoon. Isotope: Transactional isolation for block storage. In *USENIX Conference on File and Storage Technologies*, 2016.

[18] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber. Extending SSD lifetimes with disk-based write caches. In *USENIX Conference on File and Storage Technologies*, 2010.

[19] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *ACM Symposium on Operating Systems Principles*, 2013.

[20] R. Y. Wang, T. E. Anderson, and D. A. Patterson. Virtual log based file systems for a programmable disk. *ACM SIGOPS Operating Systems Review*, 33:29–44, 1998.

[21] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems (TOCS)*, 14(1):108–136, 1996.

[22] G. Wu and X. He. Delta-FTL: Improving SSD lifetime via exploiting content locality. In *European Conference on Computer Systems*, 2012.

[23] X. Yu, B. Gum, Y. Chen, R. Y. Wang, K. Li, A. Krishnamurthy, and T. E. Anderson. Trading capacity for performance in a disk array. In *USENIX Symposium on Operating Systems Design and Implementation*, 2000.

[24] Y. Zhang, L. P. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. De-indirection for flash-based ssds with nameless writes. In *USENIX Conference on File and Storage Technologies*, 2012.