Lazy Verification in Fault-Tolerant Distributed Storage Systems

Michael Abd-El-Malek*, Gregory R. Ganger*, Garth R. Goodson†, Michael K. Reiter*, Jay J. Wylie*
*Carnegie Mellon University, †Network Appliance, Inc.

Abstract

Verification of write operations is a crucial component of Byzantine fault-tolerant consistency protocols for storage. Lazy verification shifts this work out of the critical path of client operations. This shift enables the system to amortize verification effort over multiple operations, to perform verification during otherwise idle time, and to have only a subset of storage-nodes perform verification. This paper introduces lazy verification and describes implementation techniques for exploiting its potential. Measurements of lazy verification in a Byzantine fault-tolerant distributed storage system show that the cost of verification can be hidden from both the client read and write operation in workloads with idle periods. Furthermore, in workloads without idle periods, lazy verification amortizes the cost of verification over many versions and so provides a factor of four higher write bandwidth when compared to performing verification during each write operation.

1. Introduction

Fault-tolerant distributed storage involves storing data redundantly across multiple storage-nodes. Some systems support only replication, while others also use more space-efficient (and network-efficient) erasure-coding approaches in which a *fragment*, which is smaller than a full copy, is stored at each storage-node. Client read and write operations interact with multiple storage-nodes, according to some consistency protocol, to implement consistency in the face of client and storage-node faults and concurrent operations.

To implement strong consistency semantics (e.g., linearizability [15]), before the system returns a value for a read, it must be verified that this same value (or a later value) will be returned to subsequent readers. In a crash failure model, it is typically necessary that sufficiently many other storage-nodes have received the value (or fragment thereof). Tolerating Byzantine failures additionally requires coping with *poisonous writes*—i.e., writes that lack integrity, meaning that not all correct storage-node fragments come from the same original value—and *stuttering* clients that repeatedly write an insufficient subset of fragments. Most consistency protocols perform such verification proactively at write-time, incurring significant expense, with extra rounds of communication or digital signatures during every write operation.

In recent work [11], we proposed a read/write consistency protocol that uses versioning storage to avoid proactive write-time verification. Rather than verification occurring during every write operation, it is performed by clients during read operations. Read-time verification eliminates the work for writes that become obsolete before being read, such as occurs when the data is deleted or overwritten. Such data obsolescence is quite common in storage systems with large client caches, as most reads are satisfied by the cache but writes must be sent to storage-nodes to survive failures. One major downside to read-time verification, however, is the potentially unbounded cost: a client may have to sift through many ill-formed or incomplete write values. A Byzantine client could degrade service by submitting large numbers of bad values. A second practical issue is that our particular approach to verification for erasure-coding, called validating timestamps, requires a computation cost equivalent to fragment generation on every read.

This paper introduces *lazy verification*, in which the storage-nodes perform verification in the background. Lazy verification avoids verification for data that has a short lifetime and is never read. In addition, it allows verification to occur during otherwise idle periods, which can eliminate all verification overhead in the common cases. Clients will only wait for verification if they read a data-item before verification of the most recent write operation to it completes.

Our lazy verification implementation limits the number of unverified write operations in the system and, thus, the number of retrievals that a reader must perform to complete a read. When the limit on unverified writes is reached, each write operation must be verified until the storage-nodes restore some slack. Limits are tracked on a per-client basis, a per-data-item basis, and on a storage-node basis (locally). Combined, the different limits can mitigate the impact of Byzantine clients (individually and collectively) while minimizing the impact on correct clients. In the worst case,

Proceedings of the 2005 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)
0-7695-2463-X/05 $20.00 © 2005 IEEE

a collection of Byzantine clients can force the system to perform verification on every write operation, which is the normal-case operation for most other protocols.

This paper describes and evaluates the design and implementation of lazy verification in the PASIS storage system [11]. Several techniques are introduced for reducing the impact of verification on client reads and writes, as well as bounding the read-time delay that faulty clients can insert into the system. For example, a subset of updated storage-nodes can verify a write operation and notify the others, reducing the communication complexity from $O(N^2)$ to $O(bN)$ in a system of N storage-nodes tolerating b Byzantine failures; this reduces the number of messages by 33% even in the minimal system configuration. Appropriate scheduling allows verification to complete in the background. Indeed, in workloads with idle periods, the cost of verification is hidden from both the client read and write operation. Appropriate selection of updates to verify can maximize the number of verifications avoided (for writes that become obsolete). The overall effect is that, even in workloads without idle periods, the lazy verification techniques implemented provide over a factor of four higher write throughput when compared to a conventional approach that proactively performs verification at write-time.

The remainder of this paper is organized as follows. Section 2 provides context on fault-tolerant storage, read-time and lazy verification, and related work. Section 3 describes how lazy verification is performed and techniques for maximizing its efficiency. Section 4 describes our prototype implementation. Section 5 evaluates lazy verification and demonstrates the value of our techniques for tuning it and for bounding faulty clients. Section 6 summarizes this paper's contributions.

2. Background and related work

This section outlines the system model on which we focus, the protocol in which we develop lazy verification, and related work.

2.1. System model and failure types

Survivable distributed storage systems tolerate client and storage-node faults by spreading data redundantly across multiple storage-nodes. We focus on distributed storage systems that can use erasure-coding schemes, as well as replication, to tolerate Byzantine failures [16] of both clients and storage-nodes. An m-of-N erasure-coding scheme (e.g., information dispersal [20]) encodes a data-item into N fragments such that any m allow reconstruction of the original. Generally speaking, primary goals for Byzantine fault-tolerant storage systems include data integrity, system availability, and efficiency (e.g., [5, 6, 11]).

Data integrity can be disrupted by faulty storage-nodes and faulty clients. First, a faulty storage-node can corrupt the fragments/replicas that it stores, which requires that clients double-check integrity during reads. Doing so is straightforward for replication, since the client can just compare the contents (or checksums) returned from multiple storage-nodes. With erasure-coding, this is insufficient, but providing all storage-nodes with the checksums of all fragments (i.e., a cross checksum [10]) allows a similar approach. Second, a faulty client can corrupt data-items with poisonous writes. A poisonous write operation [18] gives incompatible values to some of the storage-nodes; for replication, this means non-identical values, and, for erasure-coding, this means fragments not correctly generated from the original data-item (i.e., such that different subsets of m fragments will reconstruct to different values). The result of a poisonous write is that different clients may observe different values depending on which subset of storage-nodes they interact with. Verifying that a write is not poisonous is difficult with erasure-coding, because one cannot simply compare fragment contents or cross checksums—one must verify that fragments sent to storage-nodes were correctly generated from the same data-item value.

Faulty clients can also affect availability and performance by *stuttering*. A stuttering client repeatedly sends to storage-nodes a number of fragments (e.g., $m-1$ of them) that is insufficient to form a complete write operation. Such behavior can induce significant overheads because it complicates verification and may create long sequences of work that must be performed before successfully completing a read.

Our work on lazy verification occurs in the context of a protocol that operates in an asynchronous timing model. But there is no correctness connection between the timing model and verification model. Asynchrony does increase the number of storage-nodes involved in storing each data-item, which in turn increases the work involved in verification and, thus, the performance benefits of lazy verification.

2.2. Read/write protocol and delayed verification

We develop the concept of lazy verification in the context of the PASIS read/write protocol [11, 27]. This protocol uses versioning to avoid the need to verify completeness and integrity of a write operation during its execution. Instead, such verification is performed during read operations. Lazy verification shifts the work to the background, removing it from the critical path of both read and write operations in common cases.

The PASIS read/write protocol provides linearizable [15] read and write operations on data blocks in a distributed storage system [12]. It tolerates crash and Byzantine failures of clients, and operates in an asynchronous timing model. Point-to-point, reliable, authenticated channels are assumed. The protocol supports a hybrid failure model for storage-nodes: up to t storage-nodes may fail, $b \leq t$ of which may be Byzantine faults; the remainder are assumed to crash. For clarity of presentation, we focus exclusively on the fully Byzantine failure model in this paper (i.e., $b = t$), which requires $N \geq 4b+1$ storage-nodes. The minimal system configuration ($N = 4b+1$) can be supported only when the reconstruction threshold m satisfies $m \leq b+1$; in our experiments we will consider $m = b+1$ only.

At a high level, the PASIS read/write protocol proceeds as follows. Logical timestamps are used to totally order all writes and to identify erasure-coded data-fragments pertaining to the same write across the set of storage-nodes. For each write operation, the client constructs a logical timestamp that is guaranteed to be unique and greater than that of the *latest complete write* (the complete write with the highest timestamp). This is accomplished by querying storage-nodes for the greatest timestamp they host and then incrementing the greatest timestamp value received. The client then sends the data-fragments to the storage-nodes. Storage-nodes retain all versions (i.e., each written value ordered by logical timestamp) until they are garbage-collected using the techniques developed in this paper.

To perform a read, a client issues read requests to a subset of storage-nodes. Once a quorum of storage-nodes reply, the client identifies the *candidate*—the response with the greatest logical timestamp. The read operation *classifies* the candidate as *complete*, *repairable*, or *incomplete* based on the number of responses that share the candidate's timestamp value. If the candidate is classified as complete, *timestamp validation* is performed to ensure data integrity and to protect against poisonous writes. If verification is successful, the candidate's value is decoded and returned; the read operation is complete. Otherwise, the candidate is reclassified as incomplete. If the candidate is classified as repairable, it is repaired by writing data-fragments back to the original set of storage-nodes. Prior to performing repair, timestamp validation is done in the same manner as for a complete candidate. If the candidate is classified as incomplete, the candidate is discarded, previous data-fragment versions are requested (i.e., data-fragments with an earlier timestamp), and classification begins anew. All candidates fall into one of the three classifications, even those corresponding to concurrent or failed writes.

The process of *timestamp validation* ensures that erasure-coded data is not corrupted by Byzantine-faulty storage-nodes or poisoned by Byzantine-faulty clients. To perform timestamp validation, PASIS employs *validating timestamps*, in which a cross checksum [10] is included within the logical timestamp of a write operation. A cross checksum is the set of collision-resistant hashes of the N erasure-coded data-fragments for that write. Storage-nodes verify that the data-fragment sent to them in a write request corresponds to the appropriate hash in the cross checksum, before accepting the write request, in order to prevent a client from making it look like the storage-node corrupted the data-fragment. Finally, clients validate the cross checksum at read-time by regenerating all the erasure-coded data-fragments and then the cross checksum. Validating the cross checksum at read-time completes timestamp validation, checking that the write was not poisonous.

Henceforth, the term "verification", when used in the context of our protocol, will denote the above two steps: checking for write completeness and performing timestamp validation.

Note that the majority of the verification work is performed by the client during the read operation. The lazy verification approach developed in this paper addresses this issue by having the storage-nodes communicate, prior to the next read operation if possible, to complete verification for the latest complete write. Each storage-node that observes the result of this verification can inform a client performing a read operation. If at least $b+1$ confirm verification for the candidate, read-time verification becomes unnecessary. Section 3 details how lazy verification works and techniques for minimizing its performance impact.

Lazy verification also addresses the two other issues with delayed verification: garbage collection and unbounded read-time delays. To safely garbage collect unneeded versions, a storage-node must be sure that a newer value is part of a complete and correct write; that is, it needs to verify the newer value. Of perhaps greater concern are Byzantine client behaviors (attacks) that can lead to an unbounded but finite amount of work that must be completed during subsequent read operations. Specifically, sequences of poisonous or incomplete writes may have to be traversed in order to identify the latest complete write value. Lazy verification enables storage-nodes to enforce bounds on such latent work. Section 3.3 describes how such bounds are achieved and the consequences.

2.3. Related work

The notion of verifiability that we study is named after a similar property studied in the context of m-of-N *secret sharing*, i.e., that reconstruction from any m shares will yield the same value (e.g., [8, 19]). However, to address the additional requirement of secrecy, these works employ more expensive cryptographic constructions that are efficient only for small secrets and, in particular, that would be very costly if applied to blocks in a storage system. The protocols we consider here do not require secrecy, and so permit more efficient constructions.

Most previous protocols perform verification proactively during write operations. When not tolerating Byzantine faults, two- or three-phase commit protocols are sufficient. For replicated data, verification can be made Byzantine fault-tolerant in many ways. For example, in the BFT system [6], clients broadcast their writes to all servers, and then servers reach agreement on the hash of the written data value by exchanging messages. In other systems, such as Phalanx [17], an "echo" phase like that of Rampart [21] is used: clients propose a value to collect signed server echos of that value; such signatures force clients to commit the same value at all servers. The use of additional communication and digital signatures may be avoided in Byzantine fault-tolerant replica systems if replicas are *self-verifying* (i.e., if replicas are identified by a collision-resistant hash of their own contents) [18].

Verification of erasure-coded data is more difficult, as described earlier. The validating timestamps of the PASIS read/write protocol, combined with read-time or lazy verification, are one approach to addressing this issue. Cachin and Tessaro [5] recently proposed an approach based on asynchronous verifiable information dispersal (AVID) [4]. AVID's verifiability is achieved by having each storage-node send their data fragment, when it is received, to all other storage-nodes. Such network-inefficiency reduces the benefits of erasure-coding, but avoids all read-time verification.

Lazy verification goes beyond previous schemes, for both replication and erasure-coding, by separating the effort of verifying from write and read operations. Doing so allows significant reductions in the total effort (e.g., by exploiting data obsolescence and storage-node cooperation) as well as shifting the effort out of the critical path.

Unrelated to content verification are client or storage-node failures (attacks) that increase the size of logical timestamps. Such a failure could significantly degrade system performance, but would not effect correctness. Bazzi and Ding recently proposed a protocol to ensure non-skipping timestamps for Byzantine fault-tolerant storage systems [1] by using digital signatures. Cachin and Tessaro [5] incorporate a similar scheme based on a non-interactive threshold signature scheme. Validating timestamps do not ensure non-skipping timestamps, but we believe that lazy verification could be extended to provide bounded-skipping timestamps without requiring digital signatures. This will be an interesting avenue for future work.

3. Lazy verification

An ideal lazy verification mechanism has three properties: (i) it is a background task that never impacts foreground read and write operations, (ii) it verifies values of write operations before clients read them (so that clients need not perform verification), and (iii) it only verifies the value of write operations that clients actually read. This section describes a lazy verification mechanism that attempts to achieve this ideal in the PASIS storage system. It also describes the use of lazy verification to bound the impact of Byzantine-faulty clients (Section 3.3) and its application to the garbage collection of unneeded versions (Section 3.4).

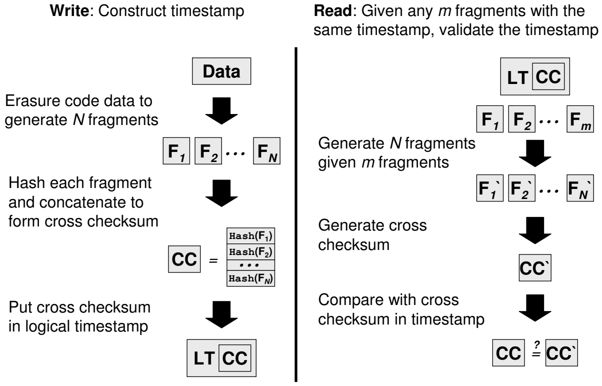

Write: Construct timestamp
Data
Erasure code data to generate *N* fragments
F_1 F_2 ··· F_N
Hash each fragment and concatenate to form cross checksum
CC = Hash(F_1) Hash(F_2) • • • Hash(F_N)
Put cross checksum in logical timestamp
LT CC
Read: Given any *m* fragments with the same timestamp, validate the timestamp
LT CC
F_1 F_2 ··· F_m
Generate *N* fragments given *m* fragments
F_1` $F_2$` ··· F_N`
Generate cross checksum
CC`
Compare with cross checksum in timestamp
CC ?= CC`

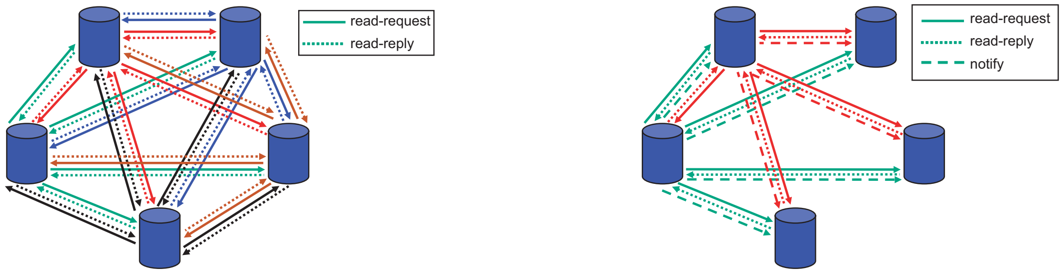
Figure 1. Illustration of the construction and validation of timestamps

3.1. Lazy verification basics

Storage-nodes perform verification using a similar process to a read operation, as described in Section 2.2. First, a storage-node finds the latest complete write version. Second, the storage-node performs timestamp validation. If timestamp validation fails, the process is repeated: the storage-node finds the previous complete write version, and performs timestamp validation on that version. Figure 1 illustrates the construction and validation of a timestamp in the PASIS read/write protocol. Timestamp validation requires the storage-node to generate a cross checksum based on the erasure-coded fragments it reads. If the generated cross checksum matches that in the timestamp, then the timestamp validates.

Once a block version is successfully verified, a storage-node sets a flag indicating that verification has been performed. (A block version that fails verification is poisonous and is discarded.) This flag is returned to a reading client within the storage-node's response. A client that observes $b+1$ storage-node responses that indicate a specific block version has been verified need not itself perform timestamp validation, since at least one of the responses must be from a correct storage-node.

When possible, lazy verification is scheduled during idle time periods. Such scheduling minimizes the impact of verification on foreground requests. Significant idle periods exist in most storage systems, due to the bursty nature of storage workloads. Golding et al. [9] evaluated various idle time detectors and found that a simple timer-based idle



(a) Without cooperation.

(b) With cooperation.

Figure 2. Communication pattern of lazy verification (a) without and (b) with cooperation

time detector accurately predicts idle time periods for storage systems. Another study showed that this type of idle time detector performs well even in a heavily loaded storage system [3].

Although pre-read verification is the ideal, there is no guarantee that sufficient idle time will exist to lazily verify all writes prior to a read operation on an updated block. Then, there is the question of whether to let the client perform verification on its own (as is done in the original PASIS read/write protocol), or to perform verification *on-demand*, prior to returning a read response, so that the client need not. The correct decision depends on the workload and current load. For example, in a mostly-read workload with a light system load, it is beneficial if the storage-nodes perform verification; this will save future clients from having to perform verification. In situations where the workload is mostly-write or the system load is high, then it is more beneficial if the clients perform verification; this increases the overall system throughput.

3.2. Cooperative lazy verification

Each storage-node can perform verification for itself. But, the overall cost of verification can be reduced if storage-nodes cooperate. As stated above, a client requires only $b+1$ storage-nodes to perform lazy verification for it to trust the result. Likewise, any storage-node can trust a verification result confirmed by $b+1$ other storage-nodes. Ideally, then, only $b+1$ storage-nodes would perform lazy verification. Cooperative lazy verification targets this ideal.

With cooperative lazy verification, once a storage-node verifies a block version, it sends a *notify* message to the other storage-nodes. The notify message is a tuple of ⟨block number, timestamp, status⟩, where "status" indicates the result of the verification. Figures 2(a) and 2(b) illustrate the messages exchanged without and with cooperative lazy verification, respectively.

We first describe the common-case message sequences for cooperative lazy verification, in concurrency- and fault-free operation. A single storage-node initiates lazy verification by performing verification and sending a notify message to the other storage-nodes. Another b storage-nodes then perform verification and send notify messages to the remaining storage-nodes. The remaining storage-nodes will thus receive $b+1$ identical notify messages, allowing them to trust the notify messages, since at least one storage-node must be correct. To reduce the number of messages required and to distribute the work of performing verification among storage-nodes, each storage-node is responsible for leading the cooperative lazy verification of a distinct range of block numbers.

If a storage-node needs to verify a block before enough notify messages are received, it will perform verification (and send notify messages), even for blocks that are outside the range it is normally responsible for. In the event of faulty storage-nodes or concurrency, the notification messages may not match. The remaining storage-nodes will then perform verification and send notify messages to the other storage-nodes, until all storage-nodes have either performed verification or received identical $b+1$ notify messages. In the worst case, all storage-nodes will perform verification.

The benefit of cooperative lazy verification is a reduction in the number of verification-related messages. Without cooperation, each storage-node must independently perform verification (i.e., effectively perform a read operation). This means each of the N storage-nodes sends read requests to, and receives read responses from, $N-b$ storage-nodes; this yields $O(N^2)$ messages without cooperation. In contrast, the common case for cooperative lazy verification is for just $b+1$ storage-nodes to perform read requests (to $N-b$ storage-nodes) and then send $N-1$ notify messages. Even in the worst case in which $2b+1$ must perform read requests, the communication complexity is still $O(bN)$. For example, even in a minimal system configuration ($N = 4b+1$) with $b = 1$, cooperation saves 33% of messages (20 messages with cooperation, versus 30 without), and this benefit improves to over 50% (128 vs. 260) at $b = 3$. Note that storage-nodes that perform cooperative lazy verification need not send notify messages back to storage-nodes from which they received a notify message.

3.3. Bounding Byzantine-faulty clients

Byzantine faulty-clients may perform poisonous writes and may stutter. Each poisonous or incomplete write introduces latent potential work into the system that may require subsequent read operations to perform additional round trips. Lazy verification, as described above, provides some protection from these degradation-of-service attacks. Storage-nodes setting the verification flag (Section 3.1) and discarding poisonous writes can decrease the cost of tolerating poisonous writes by reducing or eliminating the client verification cost. As well, if $b+1$ storage-nodes identify a client as having performed a poisonous write, then the faulty client's authorization could be revoked.

Additional steps can be taken to limit the number of write requests accepted by a storage-node but that the storage-node cannot classify as complete. This will then limit the number of latent incomplete writes a client may have to sift through before finding a correct block version during a read operation. To accomplish this, each storage-node tracks how many block versions it hosts but has not classified as complete; these versions are *possible block versions*. (Versions that are classified as complete but do not successfully verify are detected as poisonous and discarded.) Storage-nodes have per-client-per-block and per-client *possible block version thresholds*. If a storage-node receives a write request and the number of possible block versions exceeds one of these thresholds, then the storage-node blocks the write request and performs on-demand verification to see if it can remove any versions. A block version can be removed if a storage-node discovers another version with a higher timestamp that passes verification (see Section 3.4). If on-demand verification leads to some block versions being removed, the storage-node accepts the write request. Otherwise, the storage-node rejects the write request; this continues until some of these versions are classified as complete or removed.

In most environments, idle time should be sufficient to keep up with the write workload of correct clients such that the possible block version counts will not approach the thresholds unless there are significant occurrences of actively faulty clients. Section 5.4 evaluates the costs and benefits associated with different values for these thresholds. In general, higher threshold values allow higher performance during intense workloads but leave the system vulnerable to higher quantities of latent work.

3.4. Garbage collection

If lazy verification passes for a block version, i.e., the block version is complete and its timestamp validates, then block versions with earlier timestamps are no longer needed. The storage-node can delete such obsolete versions. This application of lazy verification is called garbage collection. Garbage collection allows memory and storage capacity at storage-nodes to be reclaimed by deleting unnecessary versions.

In the original presentation of the PASIS read/write protocol, capacity is assumed to be unbounded. In practice, capacity is bounded, and garbage collection is necessary. We distinguish between useful storage, which is the user-visible capacity (i.e., number of blocks multiplied by fragment size), and the history pool. The *history pool* is the capacity used for possible versions. Since the useful storage capacity must usually be determined at system configuration time (e.g., during FORMAT), the maximum size of the history pool will usually be fixed (at raw storage-node capacity minus useful storage capacity). Thus, whenever the history pool's capacity is exhausted, the storage-node must perform garbage collection (i.e., verification plus space reclamation) before accepting new write requests. (Notice that this is effectively a storage-node-wide bound on the number of possible versions, similar to the per-client-per-block and per-client limitations used to bound degradation-of-service attacks.)

Capacity bounds, garbage collection, and liveness. Capacity bounds and garbage collection can impact the liveness semantic of the PASIS read/write protocol. Without capacity bounds, reads and writes are wait-free [13]. With a capacity bound on the history pool, however, unbounded numbers of faulty clients could collude to exhaust the history pool with incomplete write operations that cannot be garbage collected—this effectively denies service. This possibility can be eliminated by bounding the number of clients in the system and setting the history pool size appropriately—the history pool size must be the product of the maximum number of clients and the per-client possible version threshold. With this approach, a faulty client can deny service to itself but not to any other client.

Garbage collection itself also affects the liveness semantic, as it can interact with reads in an interesting manner. Specifically, if a read is concurrent to a write, and if garbage collection is concurrent to both, then it is possible for the read not to identify a complete candidate. For example, consider a read concurrent to the write of version-3 with version-2 complete and all other versions at all storage-nodes garbage collected. It is possible for the read to classify version-3 as incomplete, then for the write of version-3 to complete, then for garbage collection to run at all storage-nodes and delete version-2, and finally for the read to look for versions prior to version-3 and find none. Such a read operation must be retried. Because of this interaction, the PASIS read/write protocol with garbage collection is obstruction-free [14] (assuming an appropriate history pool size) rather than wait-free.

Performing garbage collection. Garbage collection and lazy verification are tightly intertwined. An attempt to verify a possible version may be induced by history pool space issues, per-client-per-block and per-client thresholds, or the occurrence of a sufficient idle period. Previous versions of a block may be garbage-collected once a later version is verified.

Cooperative lazy verification is applicable to garbage collection. If a storage-node receives notify messages from $b+1$ storage-nodes, agreeing that verification passed for a given version of a given block, then the storage-node can garbage collect prior versions of that block. Of course, a storage-node may receive $b+1$ notify messages for a given block that have different timestamp values, if different storage-nodes classify different versions as the latest complete write. In this case, the timestamps are sorted in descending order and the $b+1^{\text{st}}$ timestamp used for garbage collection (i.e., all versions prior to that timestamp may be deleted). This is safe because at least one of the notify messages must be from a correct storage-node.

There is an interesting policy decision regarding garbage collection and repairable candidates. If a client performing a read operation encounters a repairable candidate, it must perform repair and return it as the latest complete write. However, a storage-node performing garbage collection does not have to perform repair. A storage-node can read prior versions until it finds a complete candidate and then garbage collect versions behind the complete candidate. Doing so is safe, because garbage collection "reads" do not need to fit into the linearizable order of operations. By not performing repair, storage-nodes avoid performing unnecessary work when garbage collection is concurrent to a write. On the other hand, performing repair might enable storage-nodes to discard one more block version.

Prioritizing blocks for garbage collection. Careful selection of blocks for verification can improve efficiency, both for garbage collection and lazy verification. Here, we focus on two complementary metrics on which to prioritize this selection process: number of versions and presence in cache.

Since performing garbage collection (lazy verification) involves a number of network messages, it is beneficial to amortize the cost by collecting more than one block version at a time. Each storage-node remembers how many versions it has for each block. By keeping this list sorted, a storage-node can efficiently identify its *high-yield* blocks: blocks with many versions. Many storage workloads contain blocks that receive many over-writes [23, 24], which thereby become high-yield blocks. When performing garbage collection, storage-nodes prefer to select high-yield blocks. In the common case, all but one of the block's versions will have timestamps less than the latest candidate that passes lazy verification and hence can be deleted (and never verified). Selecting high-yield blocks amortizes the cost of verification over many block versions. This is particularly important when near the per-client possible version threshold or the history pool size, since it minimizes the frequency of on-demand verification.

Block version lifetimes are often short, either because of rapid overwrites or because of create-delete sequences (e.g., temporary files). To maximize the value of each verification operation, storage-nodes delay verification of recently written blocks. Delaying verification is intended to allow rapid sequences to complete, avoiding verification of short-lived blocks and increasing the average version yield by verifying the block once after an entire burst of over-writes. As well, such a delay reduces the likelihood that verification of a block is concurrent with writes to the block. Running verification concurrent to a write, especially if only two local versions exist at a storage-node, may not yield any versions to garbage collect.

Storage-nodes prefer to verify versions while they are in the write-back cache, because accessing them is much more efficient than when going to disk. Moreover, if a block is in one storage-node's cache, it is likely to be in the caches of other storage-nodes, and so verification of that block is likely to not require disk accesses on the other storage-nodes. Garbage collecting versions that are in the write-back cache, before they are sent to the disk, is an even bigger efficiency boost. Doing so eliminates both an initial disk write and all disk-related garbage collection work. Note that this goal matches well with garbage collecting high-yield blocks, if the versions were created in a burst of over-writes.

In-cache garbage collection raises an interesting possibility for storage-node implementation. If only complete writes are sent to disk, which would restrict the history pool size to being less than the cache size, one could use a non-versioning on-disk organization. This is of practical value for implementers who do not have access to an efficient versioning disk system implementation. The results from Section 5.3 suggest that this is feasible with a reasonably large storage-node cache (e.g., 500 MB or more).

4. Implementation

To enable experimentation, we have added lazy verification to the PASIS storage system implementation [11]. PASIS consists of storage-nodes and clients. Storage-nodes store fragments and their versions. Clients execute the protocol to read and write blocks. Clients communicate with storage-nodes via a TCP-based RPC interface.

Storage-nodes. Storage-nodes provide interfaces to write a fragment at a logical time, to query the greatest logical time, to read the fragment version with the greatest logical time, and to read the fragment with the greatest logical time before some logical time. With lazy verification, storage-nodes do not return version history or fragments for writes that have been classified as poisonous. In the common case, a client requests the fragment with the greatest logical time. If a client detects a poisonous or incomplete write, it reads the previous version history and earlier fragments.

Each write request creates a new version of the fragment (indexed by its logical timestamp) at the storage-node. The storage-node implementation is based on the S4 object store [25, 26]. A log-structured organization [22] is used to reduce the disk I/O cost of data versioning. Multi-version b-trees [2, 25] are used by the storage-nodes to store fragments; all fragment versions are kept in a single b-tree indexed by a 2-tuple $\langle blocknumber, timestamp\rangle$. The storage-node uses a write-back cache for fragment versions, emulating non-volatile RAM.

We extended the base PASIS storage-node to perform lazy verification. Storage-nodes keep track of which blocks have possible versions and perform verification when they need the history pool space, when a possible version threshold is reached, or when idle time is detected. If verification is induced by a per-client-per-block possible version threshold, then that block is chosen for verification. Otherwise, the storage-node's *high-write-count table* is consulted to select the block for which verification is expected to eliminate the highest number of versions. The high-write-count table lists blocks for which the storage-node is *responsible* in descending order of the number of unverified versions associated with each.

Each storage-node is assigned verification responsibility for a subset of blocks in order to realize cooperative lazy verification. Responsibility is assigned by labelling storage-nodes from $0...(N-1)$ and by having only storage-nodes k **mod** N, $(k+1)$ **mod** N, $\cdots$, $(k+b)$ **mod** N be responsible for block k. In the event of failures, crash or Byzantine, some storage-nodes that are responsible for a particular block may not complete verification. Therefore, any storage-node will perform verification if it does not receive sufficient notify messages before they reach a possible version threshold or exceed history pool capacity. So, while $b+1$ matching notify messages about a block will allow a storage-node to mark it as verified, failure to receive them will affect only performance.

Client module. The client module provides a block-level interface to higher-level software. The protocol implementation includes a number of performance enhancements that exploit its threshold nature. For example, to improve the responsiveness of write operations, clients return as soon as the minimum number of required success responses are received; the remainder of the requests complete in the background. To improve the read operation's performance, only m read requests fetch the fragment data and version history; the remaining requests only fetch version histories. This makes the read operation more network-efficient. If necessary, after classification, extra fragments are fetched according to the candidate's timestamp.

PASIS supports both replication and an m-of-N erasure coding scheme. If $m=1$, then replication is employed. Otherwise, our base erasure code implementation stripes the block across the first m fragments; each *stripe-fragment* is $\frac{1}{m}$ the length of the original block. Thus, concatenation of the first m fragments produce the original block. (Because "decoding" with the m stripe-fragments is computationally less expensive, the implementation always tries to read from the first m storage-node for any block.) The stripe-fragments are used to generate the $N-m$ *code-fragments*, via polynomial interpolation within a Galois Field. The implementation of polynomial interpolation is based on publicly available code [7] for information dispersal [20]. This code was modified to make use of stripe-fragments and to add an implementation of Galois Fields of size 2^8 that use lookup tables for multiplication. MD5 is used for all hashes; thus, each cross checksum is $N \times 16$ bytes long.

We extended the PASIS client module to use the flag described in Section 3.1 to avoid the verification step normally involved in every read operation, when possible. It checks the "verified" flag returned from each contacted storage-node and does its own verification only if fewer than $b+1$ of these flags are set. In the normal case, these flags will be set and read-time verification can be skipped.

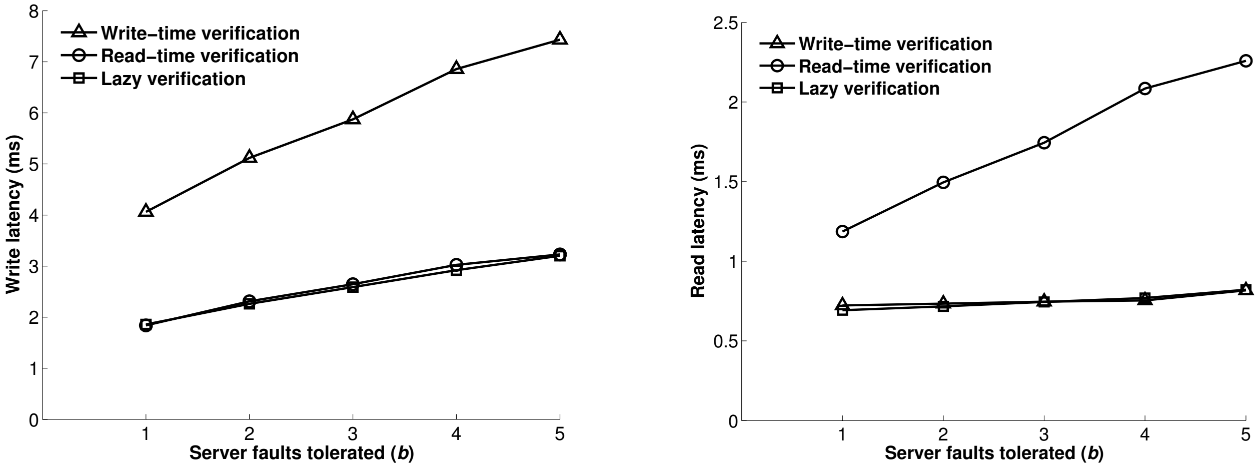
5. Evaluation

This section quantifies benefits of lazy verification. It shows that significant increases in write throughput can be realized with reasonably small history pool sizes. In fact, lazy verification approaches the ideal of zero performance cost for verification. It also confirms that the degradation-of-service vulnerability inherent to delayed verification can be bounded with minimal performance impact on correct clients.

5.1. Experimental setup

All experiments are performed on a collection of Intel Pentium 4 2.80 GHz computers, each with 1 GB of memory and an Intel PRO/1000 NIC. The computers are connected via an HP ProCurve Switch 4140gl specified to perform 18.3 Gbps/35.7 mpps. The computers run Linux kernel 2.6.11.5 (Debian 1:3.3.4-3).

Micro-benchmark experiments are used to focus on performance characteristics of lazy verification. Each experi-



(a) Write latency

(b) Read latency

Figure 3. Operation latencies for different verification policies

ment consists of some number of clients performing operations on a set of blocks. Experiments are run for 40 seconds and measurements are taken after 20 seconds. (The 20 second warm-up period is sufficient to ensure steady-state system operation.) For the benchmarks in this paper, the working set and history pool are sized to fit in the storage-nodes' RAM caches. Given that the storage-node cache is assumed to be non-volatile, this eliminates disk activity and allows focus to stay on the computation and networking costs of the protocol.

5.2. Verification policies and operation latencies

A storage system's verification policy affects client read and write operation latencies. We ran an experiment to measure the operation latencies for three different verification policies: proactive write-time verification, read-time verification, and lazy verification. The write-time verification policy is emulated by having each storage-node perform verification for a block immediately after it accepts a write request—this is similar to the system described by Cachin and Tessaro [5]. With the read-time verification policy, the client incurs the cost of verification on every read operation.

We ran experiments for different numbers of tolerated server faults (from $b = 1$ to $b = 5$). For each experiment, we used the minimal system configuration: $N = 4b + 1$. The erasure coding reconstruction threshold m equals $b + 1$ in each experiment. This provides the maximal space- and network-efficiency for the value of N employed. A single client performs one operation at a time. The client workload is an equal mix of read and write operations. After each operation, the client sleeps for 10 ms. This introduces idle time into the workload.

Figure 3(a) shows the client write latency for the different verification policies. The write-time verification policy has a higher write latency then the two other verification policies. This is because the storage-nodes perform verification in the critical path of the write operation. For both the read-time and lazy verification policies, the client write latency increases slightly as more faults are tolerated. This is due to the increased computation and network cost of generating and sending fragments to more servers.

Figure 3(b) shows the client read latency for the different verification policies. For the read-time verification policy, the client read latency increases as more faults are tolerated. This is because the computation cost of generating the fragments to check the cross checksum and validate the timestamp increases as b increases. For the write-time verification policy, verification is done at write-time, and so does not affect read operation latency. For the lazy verification policy, sufficient idle time exists in the workload for servers to perform verification in the background. As such, the read latency for the lazy verification policy follows that for the write-time verification policy.

These experiments show that, in workloads with sufficient idle time, lazy verification removes the cost of verification from the critical path of client read and write operations. Lazy verification achieves the fast write operation latencies associated with read-time verification, yet avoids the overhead of client verification for read operations.

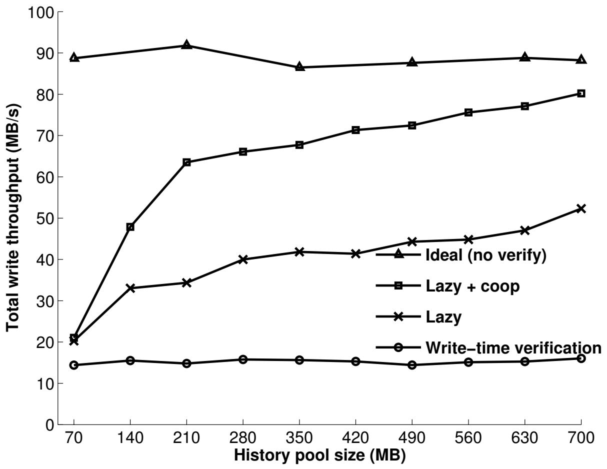


Figure 4. Write throughput, as a function of history pool size, for four verification policies

5.3. Impact on foreground requests

In many storage environments, bursty workloads will provide plenty of idle time to allow lazy verification and garbage collection to occur with no impact on client operation performance. To explore bounds on the benefits of lazy verification, this section evaluates lazy verification during non-stop high-load with no idle time. When there is no idle time, and clients do not exceed the per-client or per-block-per-client thresholds, verification is induced by history pool exhaustion. The more efficient verification and garbage collection are, the less impact there will be on client operations.

In this experiment, four clients perform write operations on 4096 32 KB blocks, each keeping eight operations in progress at a time and randomly selecting a block for each operation. Also, the system is configured to use N=5 storage-nodes, while tolerating one Byzantine storage-node fault ($b = 1$) and employing 2-of-5 erasure-coding (so, each fragment is 16 KB in size, and the storage-node working set is 64 MB).

Figure 4 shows the total client write throughput, as a function of history pool size, with different verification policies. The top and bottom lines correspond to the performance ideal (zero-cost verification) and the conventional approach of performing verification during each write operation, respectively. The ideal of zero-cost verification is emulated by having each storage-node replace the old version with the new without performing any verification at all. As expected, neither of these lines is affected by the history pool size, because versions are very short-lived for these schemes. Clearly, there is a significant performance gap ($5\times$) between the conventional write-time verification approach and the ideal.

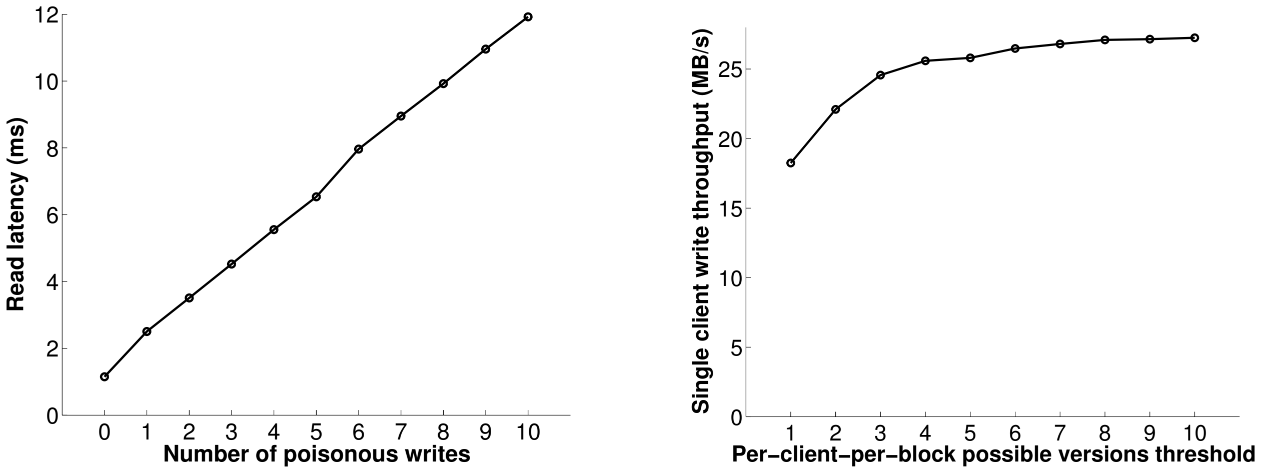
The middle two lines correspond to use of lazy verification with ("Lazy + coop") and without ("Lazy") cooperative lazy verification. Three points are worth noting. First, with lazy verification, client write throughput grows as the history pool size grows. This occurs because a larger history pool allows more versions of each block to accumulate before history pool space is exhausted. As a result, each verification can be amortized over a larger number of client writes. (Recall that all earlier versions can be garbage collected once a later version is verified.) Second, with a reasonably-sized 700 MB history pool size, cooperative lazy verification provides client write throughput within 9% of the ideal. Thus, even without idle time, lazy verification eliminates most of the performance costs of verification and garbage collection, providing a factor of four performance increase over conventional write-time verification schemes. Third, cooperative lazy verification significantly reduces the impact of verification, increasing client write throughput by 53–86% over non-cooperative lazy verification for history pool sizes over 200 MB.

5.4. Performance with faulty clients

In the absence of lazy verification, unverified versions build up, and there is a possibility that Byzantine clients have inserted poisonous writes. A client performing a read operation will discover, during its verification phase, that the writes are poisonous and read an earlier version. To measure the cost of read operations in the presence of poisonous writes, we ran an experiment in which a single client performed a single outstanding read operation as we varied the number of poisonous writes. Figure 5(a) shows the client's read latency.

On the second and seventh data values, there is a slightly higher than normal increase in the client's read latency. This is due to our implementation of client history reading. Being optimistic, on an initial read, clients request both the latest timestamp and its associated data value. If the client's verification results in the client needing to read more versions, it will first have to read more history from the storage-nodes. Currently, storage-nodes return history in 5-version units. This explains the slight increase in read latency that occurs every time a client reads past another five versions.

In order to bound the impact of a Byzantine client, one can set a per-client-per-block limit on the number of possible versions. Given such a limit, Figure 5(a) can be used to determine the worst case client read operation latency. For example, if the per-client-per-block bound is 3, and there is a single Byzantine faulty client performing poisonous writes, then a read operation is expected to take no more than 4.5 ms. However, such limits can adversely affect write throughput, as it limits the verification amortization as well as decreases the chances of waiting long enough for an



(a) Read latency versus number of poisonous writes in absence of lazy verification.

(b) Write throughput versus the per-client-per-block possible versions threshold.

Figure 5. Dealing with faulty clients

idle time period. Figure 5(b) shows the impact on a single client's write throughput, as we vary the limit on the number of possible versions per block. The client has 8 outstanding requests. As can be seen, a limit of five or six possible versions leads to good client write throughput, while bounding the harm a Byzantine client can inflict via poisonous writes.

6. Summary

Lazy verification can significantly improve the performance of Byzantine fault-tolerant distributed storage systems that employ erasure-coding. It shifts the work of verification out of the critical path of client operations and allows significant amortization of work.

Measurements show that, for workloads with idle periods, the cost of verification can be hidden from both the client read and write operation. In workloads without idle periods, lazy verification and its concomitant techniques—storage-node cooperation and prioritizing the verification of high-yield blocks—provides a factor of four greater write bandwidth than a conventional write-time verification strategy.

Acknowledgements

We thank Gregg Economou and Eno Thereska for technical assistance and Dushyanth Narayanan for flexible internship work hours. We thank the CyLab Corporate Partners for their support and participation. This work is supported in part by Army Research Office grant number DAAD19-02-1-0389, by NSF grant number CNS-0326453, and by Air Force Research Laboratory grant number FA8750-04-01-0238. We thank the members and companies of the PDL Consortium (including APC, EMC, Equallogic, Hewlett-Packard, Hitachi, IBM, Intel, Microsoft, Network Appliance, Oracle, Panasas, Seagate, and Sun) for their interest, insights, feedback, and support.

References

[1] R. A. Bazzi and Y. Ding. Non-skipping timestamps for Byzantine data storage systems. DISC, 2004.

[2] B. Becker, S. Gschwind, T. Ohler, P. Widmayer, and B. Seeger. An asymptotically optimal multiversion b-tree. *VLDB Journal*, **5**(4):264–275, 1996.

[3] T. Blackwell, J. Harris, and M. Seltzer. Heuristic cleaning algorithms in log-structured file systems. USENIX Annual Technical Conference, pages 277–288. USENIX Association, 1995.

[4] C. Cachin and S. Tessaro. Asynchronous verifiable infromation dispersal. Symposium on Reliable Distributed Systems. IEEE, 2005.

[5] C. Cachin and S. Tessaro. Brief announcement: Optimal resilience for erasure-coded Byzantine distributed storage. International Symposium on Distributed Computing. Springer, 2005.

[6] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, **20**(4):398–461, November 2002.

[7] W. Dai. *Crypto++*. http://www.cryptopp.com/.

[8] P. Feldman. A practical scheme for non-interactive verifiable secret sharing. IEEE Symposium on Foundations of Computer Science, pages 427–437. IEEE, 1987.

[9] R. Golding, P. Bosch, C. Staelin, T. Sullivan, and J. Wilkes. Idleness is not sloth. Winter USENIX Technical Conference, pages 201–212. USENIX Association, 1995.

[10] L. Gong. Securely replicating authentication services. International Conference on Distributed Computing Systems, pages 85–91. IEEE Computer Society Press, 1989.

[11] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient Byzantine-tolerant erasure-coded storage. International Conference on Dependable Systems and Networks, 2004.

[12] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. *The safety and liveness properties of a protocol family for versatile survivable storage infrastructures*. Technical report CMU–PDL–03–105. Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh, PA, March 2004.

[13] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages*, **13**(1):124–149. ACM Press, 1991.

[14] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: double-ended queues as an example. International Conference on Distributed Computing Systems, pages 522–529. IEEE, 2003.

[15] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, **12**(3):463–492. ACM, July 1990.

[16] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, **4**(3):382–401. ACM, July 1982.

[17] D. Malkhi and M. K. Reiter. An architecture for survivable coordination in large distributed systems. *IEEE Transactions on Knowledge and Data Engineering*, **12**(2):187–202. IEEE, April 2000.

[18] J.-P. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine storage. International Symposium on Distributed Computing, 2002.

[19] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. Advances in Cryptology - CRYPTO, pages 129–140. Springer-Verlag, 1991.

[20] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, **36**(2):335–348. ACM, April 1989.

[21] M. K. Reiter. The Rampart toolkit for building high-integrity services. *Theory and Practice in Distributed Systems* (Lecture Notes in Computer Science 938), pages 99–110, 1995.

[22] 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. ACM Press, February 1992.

[23] C. Ruemmler and J. Wilkes. UNIX disk access patterns. Winter USENIX Technical Conference, pages 405–420, 1993.

[24] C. Ruemmler and J. Wilkes. *A trace-driven analysis of disk working set sizes*. HPL–OSR–93–23. Hewlett-Packard Company, April 1993.

[25] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger. Metadata efficiency in versioning file systems. Conference on File and Storage Technologies, pages 43–58. USENIX Association, 2003.

[26] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-securing storage: protecting data in compromised systems. Symposium on Operating Systems Design and Implementation, pages 165–180. USENIX Association, 2000.

[27] J. J. Wylie, G. R. Goodson, G. R. Ganger, and M. K. Reiter. A protocol family approach to survivable storage infrastructures. FuDiCo II: S.O.S. (Survivability: Obstacles and Solutions), 2nd Bertinoro Workshop on Future Directions in Distributed Computing, 2004.