

Olive: distributed point-in-time branching storage for real systems

Marcos K. Aguilera Susan Spence Alistair Veitch
Hewlett-Packard Laboratories, Palo Alto, California, USA

Abstract. This paper describes Olive, the first distributed block storage system to provide consistent point-in-time branching. Point-in-time branching allows users to recursively and quickly snapshot or clone the storage state. It has a wide range of applications including testing new deployments or upgrades without disrupting a running system, quickly provisioning large homogeneous systems, and preserving old versions of data. Olive provides block-level access and strong consistency for broad applicability, allowing it to branch file systems, database systems, and every other storage application that ultimately stores data on block storage. Olive is distributed and replicated to provide fault tolerance and availability. Providing strong consistency for branching in a replicated distributed system is a technical challenge that we address in this work. We evaluate Olive and show that branching typically takes a few tens of milliseconds, and so it has little impact on I/O's.

1 Introduction

Distributed replicated storage systems provide many benefits over single-server solutions, like better scalability and cheaper reliability. Scalability comes from dividing work among many servers, and cheaper reliability comes from replicating over unreliable commodity hardware instead of using specialized fault-tolerant hardware. Distribution, however, poses challenges: variable asymmetric network delays, and the inability to distinguish a node that has crashed from one that is slow to respond, make it impossible for nodes to always be consistent with each other. These inconsistencies must be dealt with, ideally in a way transparent to users.

This paper proposes a new scheme to provide *point-in-time branching* of storage, or the ability to recursively fork off storage branches that can evolve independently, in a replicated distributed system. Branching storage includes two basic functions: snapshots and clones. A *snapshot* is a read-only virtual copy that preserves the state of storage at a given point in time, while a *clone* is a virtual copy that can change independently of its source.¹ These storage branches are recursive, meaning that branches can be created off other branches.

¹Clones are sometimes called “writable snapshots”, but we avoid this term since it is an oxymoron.

Branching storage has many uses that become more important as the size of storage increases without a corresponding increase in data transfer rates—a trend that has made it increasingly difficult to manipulate ever larger data volumes. As an example application, suppose that a user wishes to install and test a software upgrade without disturbing the current working version. Without branching storage, this could involve copying large amounts of application data and environment state, which can take hours or days. With branching storage, the user can simply create a storage clone very quickly, and install the upgrade on the clone, without disturbing the original version. As another application, suppose that an administrator wishes to provision storage to many homogeneous computers from a “golden” copy, as is often needed in a computer lab, store, or data center. Without branching storage, this involves copying entire storage volumes many times. With branching distributed storage, the administrator can simply clone the golden copy once for each computer. Besides being fast to create, clones are space efficient because they share common unmodified blocks.

This paper describes Olive, a novel point-in-time branching storage system that is efficient, distributed, broadly applicable, and fault tolerant. Providing point-in-time branching for distributed replicated systems raises new consistency issues because of the need to simultaneously coordinate replicas and capture distributed state when there are many outstanding operations. We believe our techniques to handle these issues are applicable not just to Olive, but also to other distributed replicated storage systems.

While designing Olive, our goal was to maximize its applicability to real systems. To do so, we made two broad design choices: (1) *Provide branching at the lowest level: block storage.* Branching functionality can be designed at various levels, including database systems, file systems, object stores, or block storage. By choosing block storage, Olive can be used to branch file systems, database systems, or any application that ultimately stores data on block storage. (2) *Preclude changes to storage clients.* Changes to storage clients are a huge inhibitor for adoption of new storage solutions, due to the large existing base of clients and applications. Thus, Olive does not

require storage clients to be modified in any way: they do not need to run special protocols, install special drivers, or use special storage paradigms. In fact, we took this goal to an extreme, by showing that Olive can support an industry-standard storage protocol, iSCSI [22]. Olive presents clones and snapshots as regular block volumes on a network.

Olive provides replicated storage distributed over a network for fault tolerance. We use quorum-based data replication for high-availability, which provides three benefits. First, the system requires only a quorum (e.g., a majority) of replicas to be accessible to a client at any time. Second, the accessible quorum can vary with different clients, due to their placement in the network for example, and over time, due to transient network behavior or brick failures. Third, if a client cannot access a quorum, part of the storage becomes temporarily unavailable to that client, but neither storage integrity nor other clients are affected.

To broaden applicability, Olive also provides a very strong form of consistency, linearizability [9], which allows Olive to be used by applications that are very demanding on storage consistency, in addition to less demanding applications. Roughly speaking, linearizability requires that operations appear to occur at a single point in time between the start and end of the operation. For branching storage, this means that if a clone or snapshot is requested at time t_1 and completes at time $t_2 > t_1$ then it will capture the global state of storage at a single point in time between t_1 and t_2 . Note that linearizability implies other forms of consistency used in storage systems, like sequential consistency, causal consistency, and crash consistency. Crash consistency means that a branch initially captures some state that could have resulted from a crash, which is important because applications typically know how to recover from such states.

We implemented Olive within the Federated Array of Bricks (FAB), a low-cost distributed storage system that provides block-level storage and uses distributed data replication for fault tolerance [3, 20]. We show that branching a volume typically takes a few tens of milliseconds so it has little impact on I/O's. Experiments also validate the consistency of our scheme.

In summary, Olive is the first distributed block storage system to provide point-in-time branching. This is achieved without any client changes, which is an important consideration for applicability to real systems. A key contribution of Olive is its scheme to provide strong consistency for distributed replicated storage, by ensuring that replication and branching are coordinated carefully. As far as we know, Olive is the first system to tackle this issue.

This paper is organized as follows. In Section 2 we explain the assumed environment and the requirements for Olive. Section 3 explains the version tree, a simple struc-

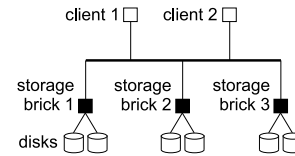


Figure 1: Distributed storage setting.

ture that is used both internally and by users of Olive. We explain the exact consistency that Olive provides in Section 4. Olive's efficiency comes from sharing data between branches; we explain how this is done in Section 5. Section 6 covers the main algorithms in Olive, which are responsible for providing consistency. Section 7 has a discussion on data layout on physical storage. We describe the evaluation of Olive in Section 8 and related work in Section 9.

2 Environment and requirements

Environment. We consider a distributed system where nodes communicate with each other by sending point-to-point messages over network links. Every pair of nodes can send messages to each other. Nodes may fail by crashing; a crashed node stops executing and becomes unresponsive. We do not consider malicious failures. Network links may fail by dropping messages, but we assume that if a message is repeatedly sent and the destination does not crash then the destination eventually receives the message (no permanent partitions). We do not assume that the network is synchronous or that network delays are bounded.

Some nodes in the network are storage nodes or *bricks*; together, they implement the storage system. Other nodes have clients running storage applications, like file systems or database systems (Figure 1). In an ideal world, storage clients can execute custom protocols to read and write data, and these protocols can implement replication for fault tolerance. But in practice, deploying custom protocols at clients is very difficult; clients instead use standard storage protocols to send a read or write request to a *single* brick. This brick can use custom protocols to execute the client request, and then returns the result to the client, if any, using the standard storage protocol.

Requirements. Our goal in producing Olive is to obtain a distributed block-based storage system that provides point-in-time branching. Distributed means that Olive is implemented by multiple storage nodes, and it is usable by multiple application nodes. Block-based storage means that storage is accessed through fixed-length units called blocks, typically with 512 bytes each. Olive is expected to have the usual attributes of a general-purpose storage service: good reliability, availability, and performance. Providing point-in-time branching means to implement two functions: *snapshots* and *clones* of a storage volume. A snapshot of a volume is a data collection that *retains* the past contents of a volume despite updates.

Snapshots are useful for archiving data or in other situations where old versions of data are needed. For broad usability, we require that a snapshot be accessible as a storage volume, so that applications that run on regular storage can also run on snapshots. The volume from which a snapshot originates is called its *source*.

A clone of a volume is another volume that starts out as a virtual copy but may later diverge, allowing data to evolve in multiple concurrent ways. For flexibility, we require all volumes to be clonable, including clones and snapshot volumes themselves. Cloning a volume results in a new writable volume whose initial contents are identical to the source volume. While the data in the clone can change, the snapshot retains its original data. It is possible to clone a snapshot multiple times, for example to experiment with different evolutions of data from the same snapshot.

We also make the following further requirements of Olive:

- *Do not require client changes.* For broadest applicability, clients should use a standard network storage protocol for reading and writing. There are currently no standard interfaces for requesting snapshots and clones, so we allow some flexibility here, but the interface should be minimal and intuitive, so that it can be replaced with a standard interface once it emerges.
- *Provide a strong form of consistency.* A storage volume may be cloned or snapshotted while there are outstanding I/O operations, because storage clients, such as file systems, can have concurrent activities, and most cannot be expected to pause their activities when a snapshot is made (and requiring this would violate the previous requirement). For broadest applicability, in these cases snapshots and clones should provide a strong form of consistency so that the service can be used with all applications. Roughly speaking, a consistent snapshot ensures that the data that it preserves reflects the state of storage at some point in the past. And a consistent clone ensures that its initial state is a consistent snapshot. We later describe the exact consistency guarantees provided by Olive.
- *Avoid performance disruptions.* When creating new branches or when using data volumes that are branches, the performance of the storage should not suffer significant impact.
- *Be space efficient.* Storage branches may share lots of common data, and in those cases, the system should avoid having multiple physical copies in storage.

3 The version tree

The version tree is a simple data structure that Olive uses to describe the relationship between various storage

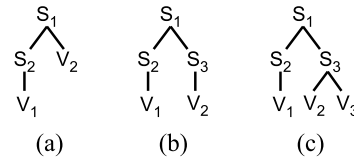


Figure 2: Examples of version trees.

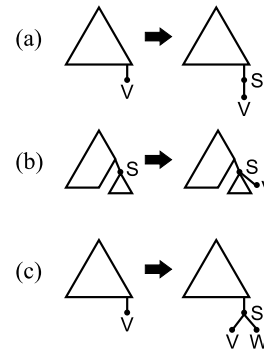


Figure 3: How branching operations affect the version tree: (a) Taking a snapshot S of a writable volume V . (b) Making a clone V of a snapshot S . (c) Making a clone W of a writable volume V .

branches, where each branch is a volume. In the context of Olive, a volume is a set of blocks, but in general it could be any set of data objects that are branched together. Nodes in the version tree correspond to volumes, where a leaf node corresponds to a writable volume, and the ancestors of the leaf node correspond to all its snapshots, with the most recent snapshots closer to the leaf. Inner nodes in the tree are always (read-only) snapshot volumes. Figure 2 (a) is an example of such a tree. There are two leaves, V_1 and V_2 , which are writable volumes, and two snapshots of V_1 , S_1 and S_2 . S_1 is also a snapshot of V_2 . This case might occur if V_2 is created as a clone of S_1 . Figure 2 (b) shows how the tree changes if a user takes a snapshot S_3 of V_2 : S_3 is the parent of V_2 because S_3 is V_2 's most recent snapshot. And Figure 2 (c) shows what happens if the user subsequently creates a clone of S_3 .

In general, taking a snapshot of a writable volume V results in replacing V with a new node S and adding V as a child of S . This is depicted in Figure 3 (a), where the triangles represent the version trees before and after taking a snapshot. Cloning a snapshot S results in creating a new child V of S (Figure 3 (b)). Cloning a writable volume V corresponds to creating a snapshot of V and then cloning the snapshot. The result is that V is replaced with a snapshot S and two children, V and a new node W (Figure 3 (c)). Note that snapshot S is a by-product of cloning V ; the reason for this will become clear later, but roughly speaking it is because we want both V and W to initially share allocation of their data, allowing for space efficiency.

The version tree allows a user to visualize all previous

states of a given writable volume V , as well as to understand where in the past two volumes have diverged. As we will see later, the version tree is also needed by Olive to maintain data consistency.

4 Consistency provided

Intuitively, a replicated storage system that supports branching needs to maintain two forms of consistency:

- *Replica consistency*: ensuring that replicas have compatible states; and
- *Branching consistency*: ensuring that snapshots and clones have meaningful contents.

Replica consistency is a form of consistency that is internal to the storage system, while branching consistency is visible to storage clients.

The branching consistency provided by Olive is *linearizability* [9], a strong and well-understood condition used often in concurrent systems. Roughly speaking, linearizability considers operations that have non-zero durations, and requires each operation to appear to take place at a single point in time, and this point must be between the start and the end of the operation. For branching storage, by definition the operations are read, write, clone, and take snapshot; the start of an operation is when a client requests the operation, and the end is when the client receives a response or acknowledgement. For example, Figure 4 (a) shows a timeline where time flows to the right. There are four operations: three writes to volume V for blocks B_1 , B_2 , and B_3 with data x , y , and z , respectively, and one clone operation on V . These operations have start and end times represented by the endpoints of the lines below each operation. Linearizability requires operations to appear to take effect at a single point on these lines. Figures 4 (b), (c), and (d) show some points in time where each operation could appear to occur in accordance with linearizability. In (b), the clone operation “happens” at a point after the write to B_1 but before the other writes; as a result, the clone incorporates the first write but not the others. In (c) and (d), the clone operation happens at a different place and so the clone incorporates different sets of writes. All these behaviors are allowed by linearizability. One behavior *not* allowed by linearizability is for the clone to incorporate the writes to B_1 and B_3 , but not the one to B_2 , because there is no way to assign points to each operation to produce this behavior.

Linearizability captures the intuition that if two operations are concurrent then they may be ordered arbitrarily, while if they are sequential then they must follow real time ordering.

Olive achieves branching consistency and replica consistency by building upon an existing protocol for replicated storage [3, 20]. This protocol solves the problem of how new writes are propagated to the replicas, and how reads reconcile data from possibly divergent

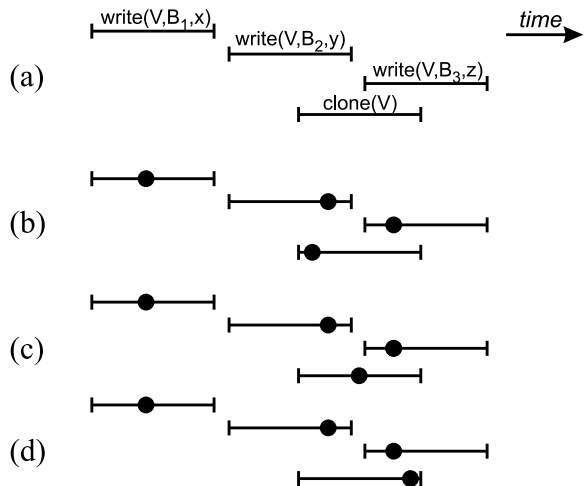


Figure 4: Depiction of linearizability. (a) shows an execution history with three writes and a clone operation. (b)-(d) show three allowable ways to linearize this execution.

replicas, without any branching. The problems that we solve are how to propagate information about new storage branches, how this information interacts with reads, writes and the replicated storage protocol to achieve linearizable branching, and how and when different branches can share data for efficiency. This is explained later in Section 6.

Relation to other forms of consistency. Linearizability is similar to sequential consistency [13], but different because sequential consistency allows an operation to appear to take place after the operation has completed or before the operation has begun. For example, with sequential consistency, the clone could exclude all writes in Figure 4 (a), i.e., the clone appears to occur at a point before all writes. This could occur with an implementation that did not see frequent writes because they are still in some buffer; this implementation, however, would not satisfy linearizability.

Linearizability implies *crash consistency*. Roughly speaking, crash consistency is a consistency condition for snapshots (or clones) that requires that the state captured by a snapshot be one that could have resulted from halting the system unexpectedly (crash). For this definition to be precise, one needs to define what are the allowable states produced by a crash, but typically it means that completed writes are incorporated while outstanding writes may be incorporated partly. It is not difficult to show that linearizability implies this property, in the sense that a branching storage system satisfying linearizability ensures crash consistency of its snapshots or clones.

Crash consistency means that a branch initially captures some state that could have resulted from a crash, which is important because applications typically know how to recover from such states. The recovery procedure typically involves writing data to the volume, and might

require user choices, such as whether to delete unattached inodes, and so it is performed at a clone derived from a snapshot.

5 Data sharing

Storage volumes that are related by branching may have lots of common data, which can share the storage medium. This provides not just space efficiency, but also time efficiency since sharing allows branches to be created quickly by simply manipulating data structures. We next explain these data structures. Throughout this paper, we use the standard terminology that *logical* offsets or blocks are relative to the high-level storage volume, whereas *physical* offsets or blocks are relative to the storage medium (disks).

Each storage node or *brick* needs logical-to-physical maps that indicate where each logical address of a volume is mapped. This is a map from $\langle \text{volume}, \text{logical-offset} \rangle$ to $\langle \text{disk}, \text{physical-offset} \rangle$. Because it takes too much space to maintain this map on a byte-per-byte basis, the map is kept at a coarser granularity in terms of *disk allocation units*, which are chunks of L consecutive bytes where L is some multiple of 512 bytes. L is called the *disk allocation size*, and it provides a trade-off between flexibility of allocation and the size of the map. It is also useful to think in terms of the reverse physical-to-logical map, which indicates the volume and logical offset that correspond to each disk and physical offset. This map is one-to-many, because storage volumes may share physical blocks. The *sharing list* of a physical block B is the result of applying this map to block B : it indicates the set of all storage volumes that share B (strictly speaking, the map also indicates the logical offset where the sharing occurs, but this offset is the same for all volumes sharing B). When a write occurs to a block that is being shared, the sharing is broken and the sharing list shrinks. Sharing can be broken in two ways: either the volume being written gets allocated a new block B' (move-on-write), or the volume being written retains B while the old contents of B are copied to B' (copy-on-write). In the common case, there will be exactly one volume V in the sharing list of B or B' (the volume where the write occurs) and the other list will be equal to the original list minus V (the volumes that should preserve the contents before the write). However, there are situations in which the split will result in more than one volume in both B and B' . Those situations are due to *recovery reads*, which we will cover in the next section.

6 Algorithm

A key contribution of Olive is its algorithm for reading and writing data while providing strong consistency and supporting branching. To explain the algorithm, we first provide some background on quorum-based data replication in Section 6.1; this is not a novelty of Olive. The

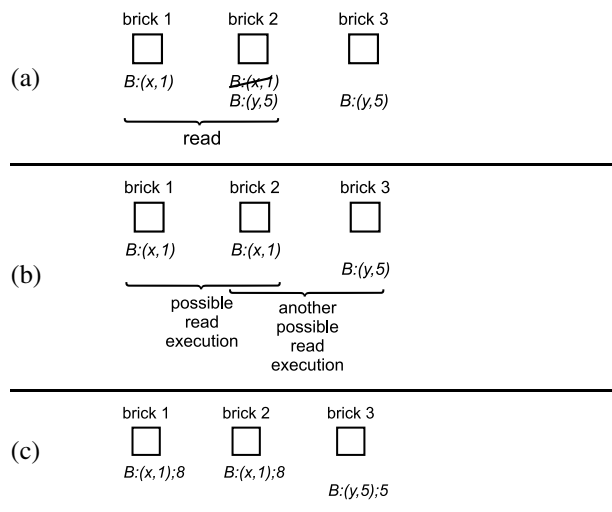


Figure 5: (a) Using timestamps to resolve replica divergence. (b) Non-determinism that arises from partial writes. (c) Largest seen timestamp, shown after semi-colons.

novelty is how to provide branching storage on top of the replication scheme, as we explain in Sections 6.2–6.8. Due to lack of space, this paper does not have proofs of correctness; they will be provided in the full version.

6.1 Quorum-based data replication

To tolerate failures, Olive uses quorum-based replication [5, 2, 15, 16, 6] modified to work with real distributed storage systems [3, 20], which we now explain. Storage is replicated at many storage nodes or *bricks*, such that data is accessible if a quorum of bricks are operational and accessible through the network. In this paper, a quorum means a majority, but other types of quorums are possible [4]. Majorities can vary with time because of variance in network delays and brick load, causing one or another brick to be temporarily slow to respond, or because of brick crashes. To write new data, a *coordinator* propagates the data with a timestamp to a majority of bricks; the timestamp comes from the coordinator's local clock, which is nearly synchronized with other coordinators' clock most of the time. To read data, a coordinator queries the data at a majority of bricks and decides which data is the most recent using the timestamp. Because any two majorities intersect, at least one brick returns to the coordinator the most recently written data. Figure 5(a) shows an example. Three bricks store data for block B ; other blocks are not shown. Initially, data x with timestamp 1 is stored at bricks 1 and 2, a majority; later, data y with timestamp 5 is stored at bricks 2 and 3, another majority; later, a read gets data from bricks 1 and 2, and y is chosen since it has a higher timestamp.

Partial writes. A partial write occurs when the coordinator crashes while writing to some block B , causing the new data to be propagated to only a minority of replicas.

The system is left in a non-deterministic state, because a subsequent read may return either new or old value, depending on whether the majority that reads intersect the minority that wrote. For example, Figure 5(b) shows y at a minority of bricks. A subsequent read will return different values depending on which majority responds first, as determined by network and other delays: if the majority is bricks 1 and 2, the read returns x , but if the majority is bricks 2 and 3, the read returns y due to its higher timestamp. This could lead to the problem of *oscillating reads*: as majorities change over time, consecutive reads return different values even though there are no writes. To prevent this, the coordinator executes a *repair phase*, in which it *writes back* or propagates the value read to a majority of replicas with a new timestamp. In the example, the repair phase writes back x or y with a higher timestamp, say 8, to a majority of bricks.²

If some coordinator writes while another coordinator reads, the repair phase of the read may obliterate an ongoing write. In Figure 5(b), y may be at a minority of bricks not because the coordinator crashed, but because it has not yet finished propagating y ; as both write and read coordinators continue to execute, the write back of x may obliterate the write of y . This problem is addressed through an initial *announce phase*, in which a coordinator announces to a majority of bricks the timestamp that it wants to use; each brick remembers the largest announced timestamp. Thus, writes execute two phases: announce and propagate (Figure 6). For reads, the announce phase can be combined with querying the data at bricks, so reads also execute two phases: announce+query and propagate. Each phase may involve a different majority of bricks. In the second phase, if a coordinator propagates a value with a smaller timestamp than the largest announced timestamp at a brick, the brick rejects the value and returns an error, which causes the coordinator to return an error to the client. Typical clients (e.g., an operating system) then retry the operation, for a few times. Figure 5(c) shows the largest announced timestamp for block B at each brick after the semicolons; this timestamp is different from the timestamp of the data, shown in parenthesis. Bricks 1 and 2 have been announced timestamp 8 of an ongoing read that will later write back x with this timestamp. Meanwhile, an ongoing write with timestamp 5 has propagated y to brick 3; when it tries to propagate y to bricks 1 or 2, an error will occur because these bricks saw timestamp 8. This will cause a client to retry writing y . More generally, the announce phase helps to deal with a *stale timestamp*: if some value at a brick has timestamp T then T has been announced at a majority of bricks, and so a coordinator

²Some quorum-based replication schemes write back x or y with its original timestamp, which prevents oscillation from y to x , but still allows one oscillation from x to y , thus allowing a failed write to take effect at an unpredictable arbitrary time in the future. This violates the limited effect property [1], and so Olive does not employ such schemes.

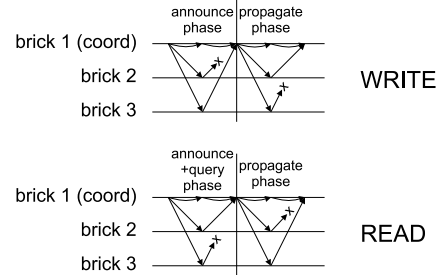


Figure 6: Two-phase write and read operation.

that tries to execute with a smaller timestamp gets an error. The announce phase also allows an important simple optimization for reading: in the first phase, if all bricks return the same data with same timestamp, and indicate that no higher timestamp has been announced, then the repair phase is not needed since the data already has the highest timestamp among all bricks including those that did not respond.

In the above description, the coordinator reads or writes a single block, but the scheme allows operation on a range of blocks, by packaging together information for multiple blocks in the messages of the protocol.

We now explain how branching works with quorum-based storage replication, by using the version tree (Section 3) to determine what content should continue to be shared after a write or a read.

6.2 Using the version tree to determine sharing

Recall that the sharing list of a physical block B is the set of all storage volumes that share B . The sharing list changes over time as new data gets written to volumes. For example, consider the version tree in Figure 2 (c), and suppose that logical block b of volumes S_1 , S_3 , V_2 and V_3 are sharing the same physical block B . Then, the sharing list for B is $\{S_1, S_3, V_2, V_3\}$. If a user writes new data to block b of volume V_3 then a new physical block B' is allocated for volume V_3 (assuming move-on-write instead of copy-on-write), and the sharing list for B is reduced to $\{S_1, S_3, V_2\}$; the sharing list created for B' is $\{V_3\}$.

Read-only snapshots may get their blocks updated too, because of the repair phase of reads. For example, if the sharing list for B is $\{S_1, S_3, V_2, V_3\}$ and there is a read on snapshot S_3 that requires writing back to S_3 then a new physical block B' is allocated for the data being written back (assuming move-on-write) and the sharing list for B' is set to $\{S_3, V_2, V_3\}$, while the sharing list for B gets reduced to $\{S_1\}$.

The general rule for splitting a sharing list L is that the volume V being written (or written back) and all its children in L should share the newly written contents, while the other volumes in L should share the old contents. This is consistent with the fact that descendants of node V represent later versions of that node, and so if there is a

change on V to fix nondeterminism then descendants of V that are sharing data with V also need to fix the nondeterminism in the same way.

6.3 Achieving consistency

Recall that Olive must provide two forms of consistency: replica consistency and branching consistency. We now explain the details of the new mechanisms used to achieve them; keep in mind that each form of consistency cannot be provided in isolation, but rather require a single scheme that provides both. Thus, the separation of techniques below is merely for didactic purposes.

Replica consistency. To create a new storage branch, a user sends a request to one of the bricks, say c . Brick c decides how the version tree needs to be updated, based on the type of the new branch as explained earlier, and then propagates this update to other bricks. This propagation is done with uniform reliable broadcast [7, 8], which ensures that if one brick receives the update then all live bricks also receive it, despite failures, thus ensuring eventual consistency. Olive uses a simple algorithm to implement uniform reliable broadcast, described in Section 6.8.

While the propagation is happening, however, bricks will have divergent version trees. For example, if we take a new snapshot of V_2 from the situation in Figure 2 (a), bricks will eventually arrive at the tree in Figure 2 (b), where S_3 is the new snapshot. If a new write occurs, and all bricks have the tree in (b), then the write results in a copy-on-write on all replicas to preserve the contents for snapshot S_3 . But what happens if the new snapshot is still propagating, and some bricks have (a), while others have (b)?

In our scheme, the coordinator for the write decides what to do, and the replicas just follow that decision. We implement this by having a version number associated with writable volumes; this number is incremented every time the volume gets a new snapshot. The number is the depth of the volume's node in the tree if snapshots are not deleted, but could be higher if snapshots are deleted. For example, Figure 7 shows a version tree with two writable volumes V_1 and V_2 (leaves), assuming no snapshots have been deleted. The current version of V_1 and V_2 is 3; version 2 of V_1 is snapshot S_2 , while version 1 is snapshot S_1 . When executing a write on a volume, the coordinator reads the volume's version according to its local view; bricks receiving a write from the coordinator use that number to decide where the write gets stored. For example, if the coordinator decides to write version 2 of volume V_2 (because the coordinator's version tree is slightly out of date and does not have snapshot S_3 yet), then a brick that has snapshot S_3 will store the new data by overwriting data for S_3 rather than doing a copy-on-write. This ensures that replicas treat all writes consistently.

Branching consistency. To achieve branching consistency, the coordinator of a write checks that the version

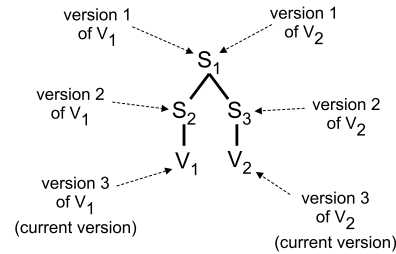


Figure 7: Version numbers associated with *writable* volumes V_1 and V_2 for a given version tree.

number that it wants to pick is the same at a majority of bricks. If not, the coordinator picks the highest version number seen and retries, until successful—this is called the *version-retry technique*, which is key to obtaining consistency. This process is piggybacked on the first phase of the two-phase write protocol (from Section 6.1), and so it has little additional cost if the coordinator does not have to retry, which is the common case. In the second phase of the two-phase write, which is when the data actually gets written to each brick, the coordinator tells bricks which version k it picked along with the data and timestamp to be written. When a brick receives this information, it stores the data in the appropriate physical block according to the logical-to-physical map. If that block is being shared with many volumes, the sharing may have to be broken according to the general rule in Section 6.2.

The above technique, whereby the coordinator retries the first phase until a majority of replica bricks have identical version numbers, effectively delays writes while a snapshot is taken. This is different from the well-known but simplistic technique of pausing I/O's during a snapshot, in which the coordinator acts in three phases: it first tells bricks to pause their I/O's, then it tells bricks that branching has occurred, and finally it tells bricks to resume I/O's. This simplistic technique is slow because there are three sequential phases, where each phase requires all bricks (not just a majority) to acknowledge before moving to the next phase. Requiring all bricks to respond eliminates the benefits of quorums. In contrast, with our scheme only a quorum of bricks needs to respond, and we embed the necessary delays within the write protocol without the need for explicit pause and resume actions. The result is less time to take snapshots (and hence smaller write delays during snapshots), and less complex handling of failures: with the simplistic scheme, there has to be a way to resume paused bricks if the coordinator fails, whereas with our scheme uniform reliable broadcast ensures that the snapshot information eventually propagates to the live replicas, regardless of failures, and so a write does not get stuck.

For branch consistency, reads also need to be handled carefully. First, recall that reads use a repair phase to fix nondeterministic state that arise from partial writes. For

example, consider Figure 5(b), where there is a partial write of y to block B in volume V , and assume that a subsequent read coordinator obtains responses from bricks 1 and 2, and so the coordinator picks x as the value to be read; the coordinator will then write back x to a majority of bricks with a new higher timestamp. How do we perform this write back with branching storage? Each of the three bricks have a sharing list for block B , and for consistency, it is important that the writing back of x occurs not just at volume V , but also at all ancestor volumes of V in the version tree, that appear in the sharing list for block B at the brick that returned x to the read coordinator.

For example, suppose that there are three bricks, *brick 1*, *brick 2* and *brick 3*, with version tree as in Figure 2 (a), and some logical block B has value x , which is shared between volumes S_1 and V_2 at all bricks. Now suppose there is a write for B in volume V_2 with data y , but the write is partial and only reaches *brick 3*, due to a failure of the write coordinator. Thus, at *brick 3*, the sharing of B has been broken, but this is not so at the other bricks. Now suppose that a new snapshot of V_2 is taken resulting in the version tree as in Figure 2 (b) at all bricks.

The resulting situation for logical block B is that *brick 3* has data x for S_1 and data y for $\{S_3, V_2\}$, while *brick 1* and *brick 2* have data x for $\{S_1, S_3, V_2\}$. Now suppose there is a read for B in volume V_2 . While executing the read, suppose *brick 1* and *brick 2* respond to the coordinator, but *brick 3* is slow. Then x is picked as the value being read, and there is a write back of x for volumes S_1 , S_3 , and V_2 with a new timestamp. This causes *brick 3* to restore back the sharing between S_1 , S_3 and V_2 . It also causes all bricks to adopt the new timestamp for B in volumes S_1 , S_3 and V_2 , not just for V_2 . The reason is to ensure that y cannot be read for any of these volumes; in fact, when the system resolves the nondeterminism for V_2 by deciding that the failed write of y never occurred then it must make a consistent decision for the previous snapshots S_1 and S_3 .

6.4 Creating new snapshots and clones

We now explain how Olive creates snapshots of writable volumes, clones of snapshots, and clones of writable volumes.

Creating a snapshot of a volume V is a very simple operation: it simply requires updating the version tree and incrementing the version number of V at a majority of bricks. This is done using uniform reliable broadcast, to ensure that the updates are propagated regardless of failures. The brick creating a snapshot waits until a majority of bricks have acknowledged the updates before telling the user that the operation is completed. This is necessary because reads to the snapshot should be prohibited until a majority of bricks have received the update: otherwise, two reads to the same snapshot could return different data (this could happen if a write occurs to the volume being

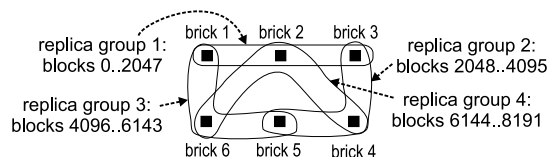


Figure 8: Replica groups storing different blocks.

snapshotted).

To create a clone of a snapshot S , a brick updates the version tree, propagates the update using uniform reliable broadcast, and waits for a majority of acknowledgements.

To create a clone of a writable volume V , a brick simply creates a snapshot S of V and then creates a clone of S , using the above procedures.

Note that if two clients take snapshots simultaneously of the same volume, there is a chance that both get the same snapshot. This is not problematic since snapshots are read-only. As for clones, it is desirable to actually create multiple clones, and so the coordinator adds a unique identifier to clones, namely, an id for the coordinator plus an increasing number.

6.5 Multiple replica groups

So far, we have been assuming that all storage blocks are replicated across all bricks. Instead, if there are many bricks, it may be desirable to replicate blocks at some of the bricks in a way that spreads load, as shown in Figure 8. The set of bricks that replicate a block is called a *replica group*. A real system with many bricks will have many replica groups, and in general they need not intersect.

To snapshot a volume, bricks at all replica groups must coordinate to ensure that branching takes effect atomically. Our algorithm is designed so that, while the relatively infrequent snapshot operation must contact a majority of bricks in every replica group, the more common read and write operations only has to contact the replica group of the block involved. To do this, we introduce the notion of a *stable* version: version v of a volume is stable iff the current version is at least v at a majority of bricks in all replica groups. The algorithm ensures that if a brick reads or writes using version v then v is stable. Note that if v is stable then no operations in any replica group will use a version smaller than v , because of the version-retry technique (Section 6.3). This provides consistency across replica groups.

A variable *stableVersion* keeps the largest version that a brick knows to be stable, and it is updated as follows. When a coordinator takes a snapshot of a volume, it uses a *uniform confirmed broadcast* to ensure that a volume's *stableVersion* is only incremented after a majority of bricks in every replica group of the volume has incremented their *currentVersion*. Roughly speaking, uniform confirmed broadcast ensures that (1) either all correct bricks (in all replica groups) deliver a message, or none

do, and (2) if the broadcaster does not fail then all correct bricks get the message. It also provides a *confirmation* of the broadcast through the primitive *confirm-deliver*. Roughly speaking, *confirm-deliver(m)* informs a brick that message *m* has been delivered at a quorum of bricks, where quorum in our system means a majority of bricks in each replica group. Uniform confirmed broadcast ensures that (1) either all bricks get *confirm-deliver(m)*, or none do, and (2) if some brick delivers *m*, then all correct bricks get *confirm-deliver(m)*. We show how to implement uniform confirmed broadcast in Section 6.8, using a simple 4-phase protocol.

When a brick delivers a message with a new version, it updates its *currentVersion* variable. When a brick gets a confirmation for this message, it updates its *stableVersion* variable. This ensures that a version *currentVersion* is stable when $stableVersion \geq currentVersion$.

When a coordinator starts a read or write on a volume, it initially waits until its version *currentVersion* for the volume is stable. It does not need to contact other bricks to determine this, because it keeps track of versions already known to be stable using the *stableVersion*, as described above. In the common case, the coordinator should not even have to wait, since *currentVersion* is likely to be stable already. Once *currentVersion* is stable, the read or write operation is executed on this version of the volume.

It is worth noting that, while a 4-phase protocol for uniform confirmed broadcast may slow down the snapshot operation, the reads and writes are only delayed during one of those 4 phases: after a message with a new version is received but before its confirmation (this is the time when *currentVersion* is not stable). Given that taking snapshots are not as frequent as performing I/O, the 4-phase protocol does not severely impact the system as a whole. This is confirmed by our experiments.

6.6 Multi-volume branching

Sometimes it is useful to clone or snapshot many volumes simultaneously and atomically—an operation that we call *multi-volume branching*. For example, a database system may store the log and tables in separate volumes, which is a typical scenario. In that case, if the table and log volumes are cloned separately then the cloned log may be out of sync with the cloned tables. With multi-volume branching, the cloning of two or more volumes can occur atomically, thus ensuring consistency between them. In terms of linearizability, this means that the cloning of all volumes appear to take place at a single point in time, rather than having a different point for each volume.

Olive provides multi-volume branching using exactly the same mechanisms as replica groups: stable versions and uniform confirmed broadcast. A snapshot or clone operation uses uniform confirmed broadcast to contact all bricks that serve the volumes being branched. The broadcast carries new version numbers for each volume being

branched. When the message is delivered, it causes a brick to increment the *currentVersions* of the volumes, causing a brief delay on new writes to those volumes. Soon after, the confirmation of delivery makes those versions stable, allowing new writes to continue.

6.7 Deleting storage branches

A user deletes a volume by sending a request to one of the bricks, who acts as the coordinator, as for other storage operations. The coordinator reliably broadcasts the request to all bricks.

Upon receipt of the request, a brick *p* does the following. It first removes the volume from the sharing list of all physical blocks. If the sharing list has become empty for a physical block, the block is marked as free. Brick *p* then updates the version tree by marking the volume's node as deleted, but the node is not yet removed from the tree, for two reasons: First, the node may have children that are not deleted, and so it should remain in the tree while the children are there. Second, even if the node has no children, another coordinator may be trying to branch the volume while it is being deleted. The actual removal of nodes from the tree happens through a periodic pruning where entire branches are removed: a node is only removed if all its children are marked deleted. This periodic pruning is done with a two-phase protocol that quits after the first phase if any node to be pruned is being branched.

6.8 Implementing uniform confirmed broadcast

Figure 9 gives an algorithm for uniform confirmed broadcast, by using point-to-point messages. (It is also easy to modify the algorithm to implement uniform reliable broadcast instead, by replacing the number 4 with 2 in lines 1 and 12 and removing lines 8 and 9.) It works as follows. To broadcast a message, a brick proceeds in 4 phases. In each phase, the brick sends the message and phase number to all bricks, and waits to receive acknowledgements from a quorum of bricks (where in our system, a quorum means a majority of bricks in each replica group). When a brick receives a message from a phase, it sends back an acknowledgement to the sender. In addition, if the phase is 2, the brick delivers the message, and if the phase is 3, the brick confirms the message.

If a brick receives a message for phases 1, 2, or 3, but does not receive a message for the following phase after a while, then the brick suspects that the sender has failed, and takes over the job of the sender (lines 10–14). In practice, one should add some random delay to the checking for this condition; Otherwise, if all bricks check at the same time, then lots of bricks will take over the job of the sender. This will not affect correctness, but may result in lots of messages and extra delay.

7 Data layout on physical storage

When mapping logical addresses to physical storage, it is desirable to preserve locality by placing adjacent logical

```

To broadcast( $m$ ):
1  for  $i \leftarrow 1$  to 4 do
2    send  $\langle \text{NEW}, i, m \rangle$  to all
3    wait to receive  $\langle \text{NEW-REP}, i, m \rangle$  from a majority from each replica group
4    return
   upon receive  $\langle \text{NEW}, i, m \rangle$  from  $q$ 
5     send  $\langle \text{NEW-REP}, i, m \rangle$  to  $q$ 
6     if  $i = 2$  and has not yet delivered  $m$ 
7       then deliver  $m$ 
8     if  $i = 3$  and has not yet confirmed  $m$ 
9       then confirm  $m$ 
Task check:
10  repeat periodically forever
11    if for some  $m$  and  $j \leq 3$ , received  $\langle \text{NEW}, j, m \rangle$  long ago, but
      never received  $\langle \text{NEW}, j + 1, m \rangle$  then
12      for  $i \leftarrow j$  to 4 do
13        send  $\langle \text{NEW}, i, m \rangle$  to all
14        wait to receive  $\langle \text{NEW-REP}, i, m \rangle$  from a majority

```

Figure 9: Implementation of uniform confirmed broadcast, used for handling multiple replica groups or for multi-volume branching.

addresses in adjacent physical addresses. It can be impossible to simultaneously preserve locality, share data between volumes, and ensure that writes execute efficiently. Indeed, efficient writing to branching storage involves using move-on-write or copy-on-write, but both schemes destroy locality of either the branched volume or source volume.

This issue, however, has little to do with the distributed or replicated nature of storage: it also applies to centralized systems, like disk arrays. In Olive, we did not come up with new solutions to this problem, but just ensured enough flexibility to support existing solutions. Indeed, Olive allows general maps of logical-to-physical storage (Section 5), and so it can support the following:

- Prioritize writable volumes over snapshot volumes, by using copy-on-write instead of move-on write. This preserves locality of the former to the detriment of the latter.
- Use volume priorities to choose between copy-on-write or move-on-write to preserve locality of the higher priority volume.
- Allocate branched blocks near the original, to preserve locality of all volumes; for example, leave an empty disk track between tracks of data, if space allows, and then use the empty track for move-on-write or copy-on-write.
- Defragment volumes based on volume priorities.

Another placement issue arises when a brick has many disks with different performance, such as fast, small, expensive disks and slow, large, cheap disks. Then, more commonly-used volumes could employ the faster disks, while seldom-used volumes employ the slower disks. Olive can support that, because the logical-to-physical mappings allow different physical disks to be used for dif-

ferent parts of a logical volume. For example, a volume may be placed a fast disk, while its snapshots are placed partly on the fast disk (for blocks shared with the source volume) and partly on the slow disk (for blocks that have diverged from the source volume). These techniques are not specific to distributed storage; existing solutions for centralized storage can be applied at each brick of Olive. Currently, Olive does not support splitting the sharing of storage across bricks—in other words, copy-on-write or move-on-write to a remote brick.

8 Evaluation

We implemented Olive within the Federated Array of Bricks (FAB), a distributed storage system [3, 20] that provides block-level storage with reliability and availability comparable to high-end disk arrays but without their high price tags. Bricks are built from commodity off-the-shelf hardware, and they provide an iSCSI interface to storage. For fault tolerance, FAB uses quorum-based data replication, as explained in Section 6.1. We modified FAB to keep track of a version tree (Section 3) and implement the scheme described in Sections 6.2–6.8. In FAB, the set of contiguous blocks stored at a replica group is called a *segment*, typically of size 1 GB. The *segment map* indicates for each segment which replica group stores it.

Olive inherits good availability and I/O performance from FAB, described in [3, 20], and so our evaluation of Olive focuses on branching and how it affects the system. The metrics we consider are the following:

- *Branch latency*: the time taken to create a clone or snapshot. This metric is important to users of the system, who (presumably) want these to be created “as soon as possible”.
- *I/O delay while branching*: the length of time I/O’s are delayed *while* a snapshot or clone is being created. This metric affects overall throughput and helps define acceptable frequencies with which storage branches can be made. This and the branch latency are key metrics for evaluating the general performance of branching storage.
- *Metadata size*: the size of the sharing lists, logical-to-physical maps, and segment maps required for each storage branch. This information is needed for each storage branch, and should not be too large.
- *I/O latency for a branched volume*: the latency for I/O *after* a branch has been created. One expects somewhat higher latency in this case for the first write to each block, which requires splitting the sharing by doing copy-on-write or move-on-write. This and the metadata size results are more specific to FAB than Olive, although they help illustrate some key overheads in the overall performance of the system.

Acceptable values for the above metrics vary per application, and since storage is intended to be a general-purpose service, the better the numbers, the more useful the service is.

8.1 System configuration

We use rack-mounted servers as bricks, each with two 1GHz Pentium 3 CPUs,³ 2GB of memory, three Seagate Cheetah 32GB SCSI disks (15K rpm, 3.6ms average seek time), and two Intel Gigabit Ethernet interfaces. They run Debian 3.0 with the Linux 2.6.9 kernel. The first 6GB of one disk hosts Linux and FAB/Olive software, while the remaining 90GB is used for FAB/Olive storage. Up to 20 machines are bricks, and other machines generate workload as needed.

In all experiments, we use 3-way replication and ensure that segments and data are distributed evenly to provide similar load to bricks.

For some experiments, we used an alternative branching algorithm from the original version of FAB. This algorithm uses Paxos to distribute a “create clone” message to all bricks, which then update their global volume metadata. This algorithm does not delay I/O’s or synchronize them with the clone operation, which may cause bricks to process I/O and the clone creation in different orders. We refer to this algorithm as the *straw man algorithm*.

8.2 Metadata size

Metadata for branching storage includes the logical-to-physical maps and the sharing lists, kept in memory. There are 8 bytes for each entry in the segment map and 12 bytes for each entry in the sharing list. Figure 10 shows metadata size per brick per branch, for small, medium, and large volumes stored on 8 or 16 bricks, using either 512 KB or 1 MB disk allocation sizes (see Section 5), and a segment size of 1 GB. Metadata size doubles when the number of bricks is halved because each brick stores a larger part of the storage volume.

From these numbers, with a 1 MB disk allocation size, 16 bricks and 200 MB of memory in each brick for metadata, a large volume can have over 50 branches, while a medium volume can have over 400 branches. While we keep metadata in main memory, it is possible to page this information, allowing for a virtually unlimited number of branches.

8.3 Branch latency

There are two major components to the user-visible time for creating branches (clones or snapshots). The first component is the time for uniform confirmed broadcast, which we expect to depend mostly on the number of bricks. The second is the time for each brick to copy/create/modify volume metadata; this time is bro-

³Only one CPU per brick is actively used during the evaluation, because FAB is single-threaded.

Volume size	# bricks	Metadata size (das=512 KB)	Metadata size (das=1024 KB)
24 GB	8	216 KB	108 KB
24 GB	16	108 KB	54 KB
192 GB	8	1730 KB	866 KB
192 GB	16	866 KB	434 KB
1536 GB	8	13836 KB	6924 KB
1536 GB	16	6924 KB	3468 KB

Figure 10: Amount of metadata per brick per branch for a small, medium, and large volume in a system with 1 GB segment size, 8 or 16 bricks, using 512 KB or 1024 KB disk allocation sizes (das).

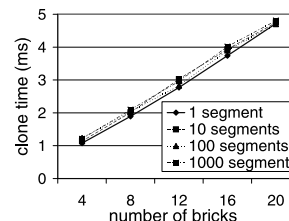


Figure 11: Clone creation latency as a function of number of bricks and segments. The segment size was fixed, and volume size chosen to result in the desired number of segments. Error bars show 95% confidence intervals.

ken up as two subcomponents: handling volume metadata (segment maps) and handling disk metadata (sharing lists). The first subcomponent should vary only with the number of segments in the volume, while the second subcomponent should depend mostly on whether the storage location has ever been written (if not, physical disk space will not have been allocated, and there will be no data to copy), and the amount of physical storage allocated on each brick. Neither subcomponent should depend on the number of existing branches of a volume.

Figure 11 shows the latency for clone creation versus the numbers of bricks and segments in a volume. The volume had no back-end physical storage allocated, which eliminates disk metadata copying. The number of segments varies from 1 to 1024, which with 1 GB segments represents a 1 TB volume. From the figure, we see that latency varies negligibly with number of segments, which indicates that snapshot latency depends primarily on the time for broadcast, not on segment handling. Indeed, separate measurements show that segment handling time is less than 30 μ s in all experiments. The latency increases linearly with the number of bricks, indicating that the bottleneck is the overhead of sending and receiving uniform confirmed broadcast messages (the number of messages for this operation is also linear in the number of bricks). We conclude that optimizing the broadcast mechanism could significantly improve clone latency, if necessary.

We repeated these experiments using the straw man algorithm and got far worse results (approximately 50 ms higher latency in all cases), which indicates our custom

Disk allocation size	Disk allocation units per brick	Metadata size per brick
128 KB	73 728	864 KB
512 KB	18 432	216 KB
1024 KB	9 216	108 KB
4096 KB	2 304	27 KB

Figure 12: Size of metadata for a small 24 GB volume with 3-way replication as we vary the disk allocation size, for a system with 8 bricks and segment size of 1 GB. These numbers are identical for a 48 GB volume and 16 bricks.

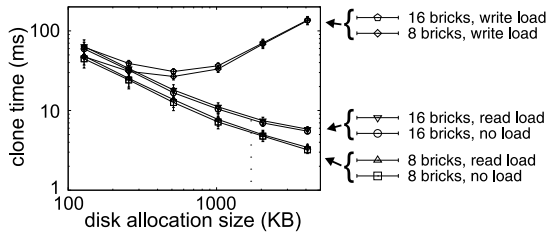


Figure 13: Clone creation latency while varying the size of disk allocation, and type of I/O workload (either none, or 100% read or write) on the system. Error bars show 95% confidence intervals

protocol provides not only better consistency but also better performance.

We also conducted experiments to measure how disk metadata size affects branching latencies. Since the disk metadata size is dominated by the number of disk allocation units, we varied the disk allocation size from 128 KB to 4096 KB while holding the volume and segment size constant, to produce varying amounts of metadata.

For small volumes with 24 GB or 48 GB, the parameters used are shown in Figure 12, and results are shown in Figure 13 (the curves labeled “no load”). It can be seen that disk metadata handling contributes significantly to latency. The copying of metadata starts during phase 2 of the broadcast, and theoretically proceeds in a separate thread independently of future broadcast phases. In practice, since FAB uses only one CPU and non-preemptive threads, the metadata copying thread typically must finish before processing of later broadcast phases, which adds to clone latency. Moreover, when copying the metadata, all of the volume’s metadata is locked, preventing I/O’s from completing. We conclude that one could reduce clone time and I/O latencies during clone by using multiple processors and allowing for incremental metadata copying.

For a medium volume of size 192 GB (8 bricks) or 384 GB (16 bricks), the latencies are ≈ 8 times larger than for the small volume size. This suggests using correspondingly larger disk allocation sizes to obtain the same performance.

Figure 13 also shows clone latency when the system is under load; we used a random workload with 1 KB

reads or writes. For the read workload, the clone latency rises slightly because processing of clone requests compete with ongoing I/O. For the write workload, the clone latency rises much more with increasing disk allocation size, because of copy-on-write: each 1KB write results in 3 times the disk allocation size of data movement. This I/O and memory activity delays the handling of the broadcast messages of clone operations, increasing the clone times, although they are still well within acceptable margins. This information can be used to fine-tune the desirable range for disk allocation unit size. 512KB to 1MB yields efficient clone operations, while providing a good trade-off between efficiency, locality and metadata size.

8.4 I/O delay while branching

This metric is the length of time that I/O’s are delayed while branching executes. In Olive, branching may cause a coordinator to retry the first phase of the I/O protocol while a volume version is not stable (Section 6.5).

Thus, the I/O delay while branching is $T_0 + T_1$, where T_0 is the time it takes between phases 2 and 3 of uniform confirmed broadcast (which is when a version number is not stable), and T_1 is the time it takes to retry the first phase of the protocol. T_0 is about 1/4 of the total time to create a new branch (see Section 8.3) and should not be more than a few tens of milliseconds. T_1 is a fraction of a millisecond, and it is dominated by T_0 . We conclude that the total delay is about 1/4 of the time to create a new branch. Figure 14 shows the read latencies for a random workload while a snapshot is taken. The system had 16 bricks, and disk allocation size was 512KB. Data was in cache to avoid large variations from disk I/O and further accentuate the snapshot overheads. From Figure 13, the total snapshot time is ≈ 18 ms. The actual I/O delay is ≈ 4.8 ms, as seen in the enlarged part of Figure 14. This confirms that I/O delays are equal to about 1/4 of the time to create a branch. Some I/O operations are delayed by up to 4ms, visible in the brief period immediately after the snapshot delay.

8.5 I/O latency and throughput for a branched volume

We now compare the latency of I/O between a non-branched volume and a branched volume. A branched volume requires breaking data sharing the first time that a block is written, by using copy-on-write or move-on-write.

For read operations, we expect the I/O latency and throughput to be unchanged, and this is shown in Figure 14. Figure 15 shows the result for write, where there is a visible latency penalty, resulting from copy-on-write, rather than snapshot overheads. Over time, as the underlying storage is increasingly separated, the throughput and latency return to normal, as seen by the time-average points in the figure.

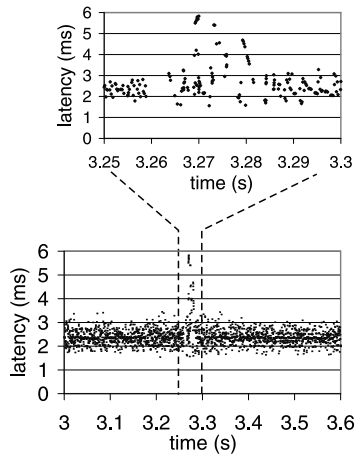


Figure 14: Read latencies while a branch is created. Each point shows the latency of one I/O. The period surrounding the snapshot is magnified, showing a gap while the branch is created at around time 3.26s.

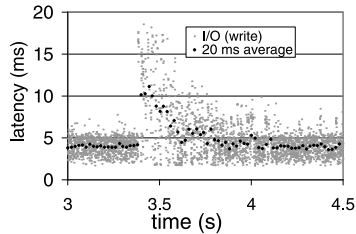


Figure 15: Write latencies while a branch is created. Grey points show the latency of one I/O; black points show the average latency in the previous 20ms.

8.6 Correctness

We did some regression tests to test branch consistency. The first test issues many pairs of synchronous writes while snapshots are taken, and then tests consistency by verifying that the snapshots do not incorporate the second write of a pair without the first. While Olive did not show any consistency errors, approximately 25% of the snapshots taken using the straw man solution did.

In the second test, we introduce partial writes to a volume to simulate coordinator failures, and then we take a snapshot and read the source and snapshot volumes while bricks fail and recover. These reads should generate a write back for either the old data (data before the partial write) or new data, depending on how it executes. Inconsistency occurs if a snapshot has the new data while its source has the old data, since the snapshot would represent a state that never happened in its source volume. While Olive carefully considers which versions are updated on a write back, the straw man solution simply writes back to the volume read. Our implementation did not show any inconsistencies, whereas the straw man did.

8.7 Evaluation summary

We have shown that the time to create a branch in Olive is primarily affected by the time taken for the broadcast, disk metadata manipulation and copy-on-writes. The most important parameter for our system was the disk allocation size, with larger sizes resulting in less metadata (and consequently faster branch operations) but more copy-on-write overhead (and slower branches). A size of 512KB seems ideal, resulting in snapshot times of a few tens of milliseconds in the worst case. In that case, the I/O delay while branching is also a few milliseconds. These latencies are on the same order as the I/O latencies themselves, and smaller than for those I/O's that must go to disk.

Better performance is possible by improving the time to do a copy-on-write or move-on-write locally on a brick, and increasing the concurrency within each brick, so that branch operations do not get queued behind I/O operations. However, this has nothing to do with the distributed nature of Olive, and involve existing techniques that are already in use in commercial products like disk arrays.

9 Related work

As we mentioned, Olive is built on top of FAB [3, 20], which provides distributed block storage, but without branching capabilities. For fault tolerance, FAB uses quorum-based data replication, as described in Section 6.1. There are many *centralized* or *single-server systems* that can capture consistent versions of data for backups or future perusal, including file systems (e.g., [19, 10, 21]), and database systems (e.g., [24, 18]). In all these systems, data is in a single place, so there are no issues of distribution and replication of data for consistency. State-of-the-art disk arrays support point-in-time branching (both snapshots and clones), and other forms of point-in-time copy. These systems are also centralized.

Petal [14] is a distributed replicated block storage system that supports (read-only) snapshots without consistency; intention to provide consistency has been announced [14], but no schemes have been proposed in the literature. Frangipani [25] is a distributed file system built on top of Petal; it provides (read-only) snapshots of file systems, by using the underlying snapshot facility of Petal. Consistency is obtained by pausing I/O at all nodes before taking a Petal snapshot, causing a potentially disruptive system hiccup. Moreover, if there are failures during the snapshot, nodes will be left in a paused state for even longer. Snapshots in Frangipani need not worry about distributed replicated data, since that is provided by Petal. Neither Petal nor Frangipani support clones.

Gifford was the first to propose replication of data at a majority of nodes; the work assumes some transactional support in the form of a stable file system [5]. Algorithms for quorum-based data replication over message-passing were proposed in the context of emulating shared mem-

ory using message-passing where some nodes may fail but the memory should retain its contents [2, 15, 16, 6]. Emulating shared memory means implementing primitives to read and write data, and so these algorithms lead to data replication schemes. Algorithms for quorum-based data replication in the context of real storage systems have been proposed for FAB [3, 20]. None of these algorithms support branching. A snapshot scheme has been proposed for FAB [11], but it does not provide consistency.

The work that is closest to ours is the Timeline [17] system, which provides consistent (read-only) snapshots for Thor, a system where a set of servers provide persistent storage for objects, and clients access these objects using a transactional interface. In essence, Timeline gets snapshot consistency by using logical clocks [12] implemented in a way different than the usual, but similar to what is suggested by Welch [26]. There are four main differences between our work and Timeline. First, Timeline does not support quorum-based replication of data. Second, Timeline requires modifications to storage clients: they piggyback and propagate timestamps internal to Thor. Moreover, Timeline clients must use the Thor abstraction for object-based storage, which precludes the use of existing storage applications. Third, Timeline does not support clones. And lastly, Timeline provides a weaker consistency guarantee than linearizability, which leads to inconsistency if the application nodes communicate outside of Thor (e.g., by sending messages over the network). Another scheme for taking (read-only) snapshots in Thor is Snap [23], but there are no details of how to ensure consistency with multiple servers.

10 Conclusion

In this paper, we described Olive, a distributed storage system that provides point-in-time branching. With Olive, storage branches can be created efficiently while providing a strong form of consistency. Olive provides a block-level interface to storage and requires no changes to storage clients. Today, we expect branching to be triggered by operators for tasks like “what-if” testing or quick storage provisioning. In the future, branching could be triggered by management applications to control the behavior of other applications by forking or rolling back their state.

Acknowledgments We thank Minwen Ji, Arif Merchant, and Yasushi Saito for insightful discussions. We thank Jay Wylie, our shepherd Steven Hand, and the anonymous referees for suggestions that improved the paper.

References

- [1] M. K. Aguilera and S. Frolund. Strict linearizability and the power of aborting. Technical Report HPL-2003-241, HP Laboratories, 2003.
- [2] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, January 1995.
- [3] S. Frolund, A. Merchant, Y. Saito, S. Spence, and A. Veitch. A decentralized algorithm for erasure-coded virtual disks. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2004)*, pages 125–134, June 2004.
- [4] H. Garcia-Molina and D. Barbara. How to assign votes in a distributed system. *Journal of the ACM*, 32(4):841–860, October 1985.
- [5] D. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th Symposium on Operating Systems Principles (SOSP 1979)*, pages 150–162, December 1979.
- [6] S. Gilbert, N. Lynch, and A. Shvartsman. Rambo II: Rapidly reconfigurable atomic memory for dynamic networks. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2003)*, pages 259–268, June 2003.
- [7] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. J. Mullender, editor, *Distributed Systems*, chapter 5, pages 97–145. Addison-Wesley, 1993.
- [8] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR 94-1425, Cornell University, Dept. of Computer Science, Cornell University, Ithaca, NY 14853, May 1994.
- [9] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.
- [10] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 235–246, January 1994.
- [11] M. Ji. Instant snapshots in a federated array of bricks. Technical Report HPL-2005-15, HP Laboratories Palo Alto, January 2005.
- [12] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
- [13] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9), September 1979.
- [14] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 1996)*, pages 84–92, October 1996.
- [15] N. A. Lynch and A. A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS 1997)*, pages 272–281, June 1997.
- [16] N. A. Lynch and A. A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proceedings of the 16th International Conference on Distributed Computing (DISC 2002)*, pages 173–190, October 2002.
- [17] C.-H. Moh and B. Liskov. TimeLine: A high performance archive for a distributed object store. In *Proceedings of the First Symposium on Networked Systems Design an Implementation (NSDI 2004)*, pages 351–364, March 2004.
- [18] G. Ozsoyoglu and R. T. Snodgrass. Temporal and real-time databases: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):513–532, August 1995.
- [19] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.
- [20] Y. Saito, S. Frolund, A. Veitch, A. Merchant, and S. Spence. Fab: building distributed enterprise disk arrays from commodity components. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems (ASPLOS 2004)*, pages 48–58, October 2004.
- [21] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the elephant file system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP 1999)*, pages 110–123, December 1999.
- [22] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zedner. RFC3720: Internet small computer systems interface (iSCSI). <http://www.faqs.org/rfcs/rfc3720.html>, 2004.
- [23] L. Shrira and H. Xu. Snap: Efficient snapshots for back-in-time execution. In *Proceedings of the 21st International Conference on Data Engineering (ICDE 2005)*, pages 351–364, March 2004.
- [24] M. Stonebraker. The design of the POSTGRES storage system. In *Proceedings of the 13th International Conference on Very Large Data Bases (VLDB 1987)*, pages 289–300, September 1987.
- [25] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP 1997)*, pages 224–237, October 1997.
- [26] J. L. Welch. Simulating synchronous processors. *Information and Computation*, 74(2):159–171, August 1987.