Speculative Recovery: Cheap, Highly Available Fault Tolerance with Disaggregated Storage

Nanqinqin Li, Anja Kalaba, Michael J. Freedman, Wyatt Lloyd, and Amit Levy, *Princeton University*

https://www.usenix.org/conference/atc22/presentation/li-nanqinqin

This paper is included in the Proceedings of the
2022 USENIX Annual Technical Conference.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-29-8

Speculative Recovery: Cheap, Highly Available Fault Tolerance with Disaggregated Storage

Nanqinqin Li, Anja Kalaba, Michael J. Freedman, Wyatt Lloyd, and Amit Levy
Princeton University

Abstract

The ubiquity of disaggregated storage in cloud computing has led to a nascent technique for fault tolerance: instead of utilizing application-level replication, newly-launched backup instances recover application state from disaggregated storage (REDS) after a primary's failure. Attractively, REDS provides fault tolerance at a much lower cost than traditional replication schemes, wherein at least two instances are running. Failover in REDS is slow, however, because it sequentially first detects primary failure and only then starts recovery on a backup.

We propose *speculative recovery* to accelerate failover and thus increase the availability of applications using REDS. Instead of proceeding with failover sequentially, speculative recovery safely and efficiently parallelizes detecting primary failure and running recovery on a backup, by employing our new `super` and `collapse` primitives for disaggregated storage. Our implementation and evaluation of speculative recovery demonstrate that it considerably reduces failover time.

1 Introduction

Replicated, network-attached storage devices have all but replaced local disks in cloud settings. Such *disaggregated disks* provide a host of useful features, including scalable storage capacity and performance, convenient data backup, and disk fault-resilience [16, 33, 49].

Their ubiquity has led developers to begin leveraging *disk* fault-tolerance to achieve *application* fault-tolerance [12, 15, 50]. When an application running on one instance, the primary, uses a disaggregated disk, its state survives its failure. This enables an emerging fault-tolerance technique we term recovery from disaggregated storage (REDS) where a backup instance recovers application state from the disaggregated disk and continues serving the application. In general, single-node applications can use REDS unmodified as long as they are crash-consistent—i.e. they persist state updates to disk before *externalizing* them to clients and are able to recover state from disk after a crash [27, 39, 58]. This includes most relational databases, local key-value stores, and file systems.

REDS is an alternative to the traditional *application-level replication*, where the application running on the primary continuously replicates its state to at least one backup [20, 22, 24, 42, 44, 56, 62, 66]. Application-level replication provides *high availability* since it ensures that a backup can service requests immediately should the primary fail. However, application-level replication is also expensive as each backup runs an entire instance of the application, requiring as much CPU, memory, storage, network resources, etc., as the primary.

In contrast, REDS only requires running a single instance of the application at a time but sacrifices availability since failover can be slow. In particular, REDS requires that the disaggregated disk be detached from a potentially faulty instance *before* initiating recovery on a new one. As a result, REDS risks long recovery periods on the new instance when the original may have come back online faster, e.g., when a transient networking issue resolves itself, or waiting too long to determine the original instance has indeed failed.

In this paper, we introduce *speculative recovery*, an application fault-tolerance technique that leverages disaggregated disks to achieve resource efficiency similar to REDS with significantly higher availability. Speculative recovery begins as soon as the primary *appears* unavailable, e.g., when it stops responding to health checks. It immediately begins recovery on a new backup instance by creating an independent clone of the disk and attaching it to the backup, while the primary instance is not interfered with to allow it an opportunity to come back in parallel. Whichever instance, the primary or the backup, becomes available first serves the application while the other is deallocated. This reduces unavailability to the minimum of either the primary becoming available again or the backup's speculative recovery completing.

There are two major challenges in realizing speculative recovery on existing disaggregated storage systems. The first is ensuring application correctness, i.e., linearizability [37], when both the primary and the backup are using a clone of the same application disk. This requires that updates to the disk from one instance do not interfere with the other, and that the external world only ever sees the effects of updates from one instance. The second challenge is ensuring good disk performance for the backup instance to recover the application. Many existing disaggregated storage systems have designs for disk clones that provide poor performance.

To address these challenges, speculative recovery introduces new primitives, `super` and `collapse`, for disaggregated disks. `super` allows a disk to be in a *superposition* temporarily where two independent versions of the disk are allowed to diverge until a `collapse` when one is observed, and it appears as though the other never existed. In essence, `super` provides disk clones with isolation and good performance, and `collapse` guarantees correctness by ensuring only one, primary or backup, of the clones can be observed.

`super` uses copy-on-write to achieve effective isolation, and the ephemeral nature of superposition enables a new design we term *collocated-clone* that minimizes the negative performance impact of copy-on-write. With collocated-clone, a disk clone directly refers to its parent's allocation table to locate data blocks, eliminating the overhead of re-populating the clone's own allocation table, which is a major bottleneck in some existing disaggregated storage systems. Collocated-clone also adopts a minimal data path by keeping all data blocks of a clone on the same storage shards as the corresponding blocks of its parent. We believe that such collocation does not skew the data distribution of a storage cluster given that only one clone continues after `collapse`.

`collapse` uses a *dirty bit* to ensure only one clone of the disk is ever externally observable. The dirty bit reflects whether there have been any updates to the disk from the primary after `super` is invoked. If so, `collapse` determines that the primary may have been observed and then aborts speculative recovery by deallocating the disk clone and the backup. Otherwise, `collapse` ensures no future writes from the primary will be accepted and then informs the backup that it can start externalizing state updates.

We implement `super` and `collapse` based on Ceph [68], an open-source distributed storage system, and use them to implement speculative recovery from disaggregated storage (SpecREDS). Our evaluation compares SpecREDS to REDS for three stateful applications: MySQL, PostgreSQL, and MariaDB. We find that our collocated-clone design achieves near-normal disk performance that supports application recovery up to an order of magnitude faster compared to Ceph's native clone design. Such improvement enables SpecREDS to achieve significantly faster failover in some scenarios.

In summary, the main contributions of this paper include:

- Speculative recovery, which increases the availability of applications that achieve cheap fault tolerance using REDS.
- The `super` and `collapse` primitives and their designs including collocated-clone for disk cloning with near-normal performance and the dirty bit for guaranteeing correctness.

2 Highly Available Applications

Stateful data center applications strive to provide high availability in the face of individual machine failures. This is often exacerbated on the cloud because developers may have no way to recover data from a virtual machine's or a container's disk after a failure. Practitioners today adopt both traditional fault-tolerance techniques at the application-level as well as cloud-native techniques that rely on disaggregated storage.

2.1 Application-level Replication

A standard approach to highly available fault tolerance for stateful applications is to replicate the application across multiple compute instances (physical machines, VMs, containers, etc.). Commonly, applications use primary-backup replication where a primary instance handles all client requests and forwards the execution logs to backup instances. If the primary fails, backups are ready to be promoted with minimal overhead since their local state is already up-to-date.

However, application-level replication has two major drawbacks. First, it can be costly. Because backups require redundant compute resources—CPU, memory, etc.—adding a backup costs as much as hosting the original application. Second, support for application-level replication is often implemented separately for each application [53, 56]. While many stateful applications support replication, including MySQL, PostgreSQL, and MongoDB, many do not, including SQLite, LevelDB, and RocksDB.

2.2 Recovery From Disaggregated Storage

Two recent trends have enabled alternative fault-tolerance strategies. First, cloud platforms have adopted disaggregated storage [28, 41, 45, 52] to provide virtual block devices to enable more efficient resource management and provide more reliable services [16, 33, 49]. Data stored on these disaggregated disks is striped and replicated across a storage area network to provide highly available and highly durable block devices that can outlast failures of the compute instances they are attached to. Second, provisioning compute instances (VMs or containers) has become fast—new compute instances can be spawned in seconds rather than in minutes [2, 5, 13, 47].

As a result, practitioners have adopted an alternative fault-tolerance mechanism, REDS, leveraging disaggregated disks and fast provisioning [12, 15, 50]. In REDS (Figure 1), instead of maintaining live backup replicas of the application, a backup instance is only spawned after the primary instance is presumed down. The disaggregated disk is then moved from the failing primary to the new backup and the application is restarted on the backup. Since the application data stored on disk persist through machine failures, the backup can recover the application to a consistent pre-failure point.

REDS provides fault tolerance to stateful applications at virtually no additional cost, since only a single instance is provisioned most of the time, with at most a short overlap of a primary and backup instance during failures. Moreover,

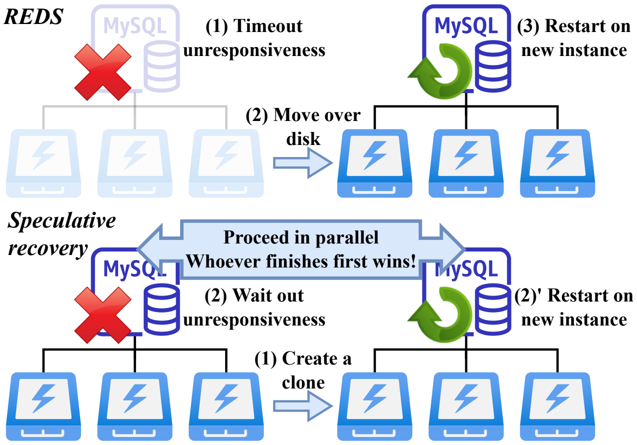

REDS
MySQL
(1) Timeout unresponsiveness
MySQL
(3) Restart on new instance
(2) Move over disk
Speculative recovery
Proceed in parallel
Whoever finishes first wins!
MySQL
(2) Wait out unresponsiveness
MySQL
(2)' Restart on new instance
(1) Create a clone

Figure 1: **REDS vs speculative recovery.** REDS sequentially times out the unresponsiveness and restarts the application on a new instance by moving over the application disk, whereas speculative recovery parallelizes the two instances.

unlike application-level replication, it does not require explicit support from the application and can thus support any crash-consistent [27,39,58] application—i.e., any application that persists state changes before externalizing them and can recover to a consistent state from disk following a crash failure. This includes most relational databases, local key-value stores, and file systems.

2.2.1 Lower Application Availability

Compared to application-level replication, REDS suffers from lower availability due to the relatively long process of restarting the application after failure has occurred.

As shown in Figure 1, the failover process in REDS includes two steps: (1) determining whether the primary instance has failed and (2) recovering the application on a backup instance. Step 1, the timeout phase, is typically achieved using a timeout of unresponsiveness for the primary to avoid spurious downtime during Step 2. Step 2 recovers the application by spawning a new instance as the backup, moving over the application disk, and restarting the application.

These steps *must* happen sequentially since both require exclusive access to the disk. For the timeout phase, the primary needs the disk attached in case it becomes responsive again. For the recovery phase, the backup needs the disk to restart the application. As a result, downtime following a failure is dictated by the sum of the timeout and the recovery phase.

Clearly, short recovery and short timeouts would improve availability. Much of recovery—spawning a new virtual machine or container and attaching a disaggregated disk to these instances—is relatively fast and becoming faster in modern data center infrastructure. For example, an AWS EC2 virtual machine can be allocated and spawned in a few seconds with optimized operating system distributions [57], while containers as well as other cloud virtualization techniques can allocate runtime environments an order of magnitude faster [13,47]. Similarly, disaggregated disks, such as a Ceph block device, can be attached to a new virtual machine or container in a few hundred milliseconds.

Application recovery time, on the other hand, is less predictable—the same application might require a few seconds or several minutes to recover depending on, e.g., the state of the application's write-ahead-log. As a result, very short timeouts risk triggering such long recoveries unnecessarily when the primary's unresponsiveness is ephemeral (e.g. the monitor is faulty, a packet is lost, etc.). For instance, if a temporary network problem leads to 6 seconds of unavailability for the primary, using a short 5 second timeout to trigger a 1 minute recovery leads to 65 seconds of unavailability. In contrast, a longer timeout would have only 6 seconds of unavailability in this scenario. In practice, "primary-is-failed" timeouts are often quite long—for example, Kubernetes uses a default 5 minute timeout to detect node failure [43].

3 Introducing Speculative Recovery

REDS can result in poor availability because the primary must be marked irreversibly failed before recovery can be attempted on a backup. Timeout lengths are chosen conservatively to avoid long recoveries if the failure is temporary, and recovery cannot begin until *after* this timeout is expired. In the worst case, this results in downtime during a long timeout followed by more downtime until a long recovery completes.

This is fundamental to REDS because there is no way to predict the future. When downtime is detected, we do not yet know if the primary has actually failed, if the failure can self-heal quickly, or if a failed health-check was actually due to temporary network issues or a faulty monitor, etc. We also cannot know how long it would take for a backup to be ready to process requests from clients—depending on the disk state when the primary failed, it could be seconds or minutes.

But suppose an *oracle* did know, at the moment of apparent failure, how long recovery would take on a backup as well as how long the primary's apparent failure would last. Such an oracle could achieve considerably better availability by avoiding timeouts completely while avoiding a slow recovery when the primary's failure is temporary. In particular, the oracle's optimal decision would be to choose the shorter of waiting for the primary to self-heal or immediately beginning recovery on a backup without waiting.

We propose a new failover design, *speculative recovery*, that makes similarly optimal choices *in practice*, without knowing the future. Speculative recovery pursues both paths (Figure 1) in parallel and either aborts recovery if the primary becomes available first, or irreversibly marks the primary failed if recovery completes first.

To accomplish this, speculative recovery creates and attaches an independent clone of the primary's disk to a new backup instance immediately when primary downtime is detected. The backup begins recovery from the cloned disk, potentially in parallel with the primary's continued operation if it is not actually failed. This results in a "superposition" where the parent and child disks are permitted to temporarily diverge, as long as neither is observed externally. Once one of them is observed (i.e., if the primary becomes available or when clients are redirected to a fully recovered backup), the superposition collapses and the unobserved disk is destroyed. Specifically, this superposition state collapses in two cases:

- **Observing the primary.** If any *writes* to the parent disk are observed, the primary is assumed to be available and the superposition is collapsed by aborting recovery on the backup and deallocating the child disk.
- **Observing the backup.** If recovery on the backup completes successfully and no writes have been issued to the parent disk, the superposition is collapsed by deallocating the parent disk, destroying the primary, and promoting the backup to be the new primary by pointing all clients to it.

Thus, speculative recovery on the backup can complete though the primary *may* still be operational, while guaranteeing external correctness. As long as the application is crash-consistent, observing clients cannot distinguish between speculative recovery and REDS, except that failover may appear much faster. If a client receives an acknowledgement from the primary for a request that modifies application state, crash consistency mandates that the primary must have written to the disk, which would halt failover to the backup, in turn ensuring the backup is never observable.

Conversely, if a client is directed to communicate with the backup, the primary cannot have acknowledged any state-modifying operations or is no longer servicing client requests, and thus the backup's state is consistent with all previous reads from the primary.

To realize these important properties, speculative recovery introduces two new disaggregated storage primitives: `super` and `collapse`. `super` produces a temporal, performant disk clone using copy-on-write (COW) semantics, resulting in a superposition in which the parent disk (attached to the primary) and the child disk (attached to the backup) diverge from the same state. `collapse` destroys the parent disk if and only if it has not changed since `super`, otherwise it destroys the, yet unobserved, child disk.

It is critical that disk clones spawned by `super` are fast to create and performant, so as not to slow down recovery on the backup significantly. `super` uses a new form of COW disk, *collocated-clone*, that improves COW writes over existing designs by up to an order of magnitude and performs almost as well as a regular, non-COW disk. Similarly, `collapse` must operate atomically—it must determine whether any writes have been made to the parent disk *and* block future writes if not, atomically—but should also not unduly delay failover. `collapse` uses a single, global *dirty bit* for the entire parent disk to track whether writes have occurred in the superposition, allowing `collapse` to use a simple protocol with only a single round trip to one storage shard. In both cases, these designs are enabled by the temporal nature of the superposition.

4 Design

This section details our design for speculative recovery. It describes the system components, the design of `super`, the design of `collapse`, why and when speculative recovery is correct, and finally discusses some performance concerns.

4.1 Components and Overview

A speculative recovery system consists of three components: (1) an instance pool to host applications; (2) disaggregated storage that provides highly durable and highly available virtual disks to applications with the `super` and `collapse` primitives; (3) a failure monitor that monitors the health of the running application instances and coordinates speculative recovery for failed instances.

When the failure monitor presumes the primary instance is unhealthy, e.g., if the monitor fails to connect to the application, it initiates speculative recovery. It invokes `super` on the primary's disaggregated disk which creates a lightweight clone using COW semantics (§4.2). In addition, `super` causes the parent disk to begin tracking writes to support the `collapse` protocol (§4.3). Next, the monitor spins up a new backup instance from the same application boot image as the primary, except with the cloned child disk attached in place of the parent. When the backup finishes restarting the application, the monitor calls `collapse`, which either atomically promotes the backup if there have been no writes to the parent disk or deallocates it if there have been writes.

4.2 `super`: Creating a Disk Superposition

As the backup instance boots and starts up the application, it may write to the child disk. For example, *fsck* might fix corruption in the file system and the application may replay and commit or rollback uncommitted transactions from its write-ahead log (WAL). Meanwhile, the primary is still allowed to function should it become available before recovery on the backup is complete. As a result, the parent and child disks are likely to diverge. However, this divergence retains application correctness because the backup is not observable to clients until after it is determined that the primary has not acknowledged any state-modifying requests.

This design is relatively simple to realize using existing primitives in disaggregated disks. In particular, many disaggregated disks provide copy-on-write clones that are quick to create. In principle, this should allow speculative recovery

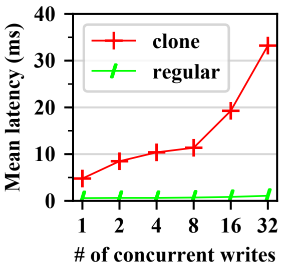


Figure 2: **Concurrent writes on EBS.** Writes are issued simultaneously in batches of 1 write to 32 concurrent writes.

to explore both paths simultaneously—waiting out the unresponsiveness on the primary and recovering the application on the backup—and achieve the same outcome as an oracle. In addition, COW clones provide the child disk with the same level of durability guarantee as the parent since dirtied data blocks are copied as new blocks and thus can be applied the same replication schema (e.g., three-way replication).

Unfortunately, existing designs for COW disk clones perform very poorly for recovery workloads. We conducted black-box experiments on EBS to measure the I/O performance of EBS clones. EBS supports clones by first creating a snapshot from a volume and then creating a new volume from that snapshot. Figure 2 shows the write performance of EBS clones with varying levels of parallelism. Normally, concurrent writes on a regular EBS volume can exploit disk parallelism well (the green line): the average latency when 32 writes are in-flight is only 2.6x the latency of a single write. However, for a cloned EBS volume (the red line), this relation becomes 7x, indicating significant performance bottlenecks for a cloned volume under highly parallel writes.

To understand the underlying reasons, we instrumented the open-source Ceph codebase where a similar behavior exists: on a cloned disk, the average latency with 32 writes in-flight is 7.1x the latency of a single write (more results and details are described in §6.2). We discovered two fundamental problems with the Ceph clone implementation, and we speculate that these may be general to many other clone designs.

First, because disaggregated disks typically treat a clone's dirtied blocks like any other new disk block, most COW designs copy dirtied blocks to different storage shards than the ones hosting the original blocks. This results in considerable overhead compared to modifying blocks in place. Second, each dirtied block requires allocating a new location in the storage area network, which is typically a blocking operation. As a result, concurrent writes that touch mostly newly dirtied blocks are performed in *sequence* rather than in parallel.

In short, copying dirty blocks to new locations over the network increases single write latency significantly, while serialized allocation eliminates most of the parallelism benefit for concurrent writes. These overheads are reasonable for typical uses of COW-cloned disks, where COW writes, and particularly concurrent writes, are infrequent [25]. However, a recovery workload is often write-intensive. As a result, these overheads can dramatically increase the time to recover applications—in some cases from seconds on a regular disaggregated disk to several minutes on a COW clone.

4.2.1 Collocated-Clone

`super` addresses both performance issues, copying overhead and serialization of COW writes, using a mechanism we term *collocated-clone*. Rather than treating copied dirty blocks the same as newly allocated blocks, collocated-clone reuses the parent's allocation table to collocate child blocks with their corresponding parent blocks. This accomplishes two things. First, copying a dirtied block *never* traverses the network, as child blocks are always on the same shard as the parent blocks. Second, COW writes never require a blocking allocation operation as the parent's allocation table already contains enough information to derive the child block's location—specifically, it is always on the same shard as the parent's and its name can be derived from the parent block's name.

As a result, COW writes in collocated-clone require only marginally more work than normal writes. The dirtied block must be copied, but only locally—incurring local disk overhead, but not network overhead. Moreover, these writes *never* require a new block allocation, so concurrent writes are always just as parallelizable as on a regular, non-COW disk.

Collocated-clone is not suitable for many uses of COW-clones because it risks amplifying any skew in the original disk's allocation. However, in speculative recovery, clones are temporary: after a short period of coexistence, it is either the child being deallocated or the child succeeding the parent and carrying on. Shards only need to have sufficient extra storage to store dirtied blocks temporarily.

In addition, collocated-clone only provides limited isolation between the parent and child. Because collocated-clone does not require the parent to do COW, the parent can directly update its data blocks in case it self-recovers. Thus, if the parent updates a data block the child has not copied, the child can see those updates, breaking the isolation. Again, this is permissible in the special semantics of superposition since if the parent is ever updated, the child will never be externalized.

4.3 `collapse`: Collapsing a Superposition

By allowing the parent and child disks to diverge in their superposition, speculative recovery introduces potential application inconsistency that must be hidden from clients. To prevent such inconsistencies, `collapse` uses a single disk-global dirty bit to indicate whether there have been writes applied to the parent disk since the creation of the child. It must also have a means of atomically promoting the backup instance to be the new primary, even with in-flight operations from the old primary.

Tracking primary writes. When super is invoked on a disk, its disk-global *dirty* and *allow-write* bits are initially set to false and true, respectively, on a fault-tolerant tracking shard in the storage cluster (this may simply be one of the data shards). When a shard of the parent disk receives a write request, before servicing the write, it requests permission to perform the write from the tracking shard. If the allow-write bit is true, the tracking shard sets the dirty bit and allows the shard to proceed with the write. Otherwise it responds that the shard should reject the write.

Atomic promotion. collapse operations are performed on the tracking shard. This shard atomically checks the dirty bit and, if it is still false, sets the allow-write bit to false, preventing any future write attempts to the parent disk. It then returns an acknowledgment that the parent disk is disabled and being deallocated. Otherwise, if the dirty bit is true, it responds that the parent disk has been observed, that the backup should be taken down, and begins asynchronously deallocating the child disk, aborting failover.

Tracking disk modification using a disk-global dirty bit allows collapse to complete quickly, as the only atomic operation is limited to a single node in the disaggregated storage cluster, avoiding expensive multi-node protocols such as two-phase commit. Such a tracking mechanism may be unnecessary and inappropriate for long-lived COW-clones that may have subsequent children and grandchildren. However, due to the ephemeral nature of the superposition, and because it is at most one clone of a disk at any given time, this design allows collapse to be supported efficiently.

After promotion of the backup is complete, the primary might still be able to service client reads from its in-memory cache, even though its disk has been deallocated by collapse. This would externalize potentially stale values. To prevent this, a stronger method is needed to sever the old primary from the clients. The specific mechanisms to achieve this may be cloud-platform dependent, but one option is to use an “elastic IP” [18] to remap the old primary’s IP address to the newly promoted primary, automatically rerouting clients. Other mechanisms such as using a firewall to block the primary’s access to the network would also work.

4.4 Correctness and the Failure Model

Speculative recovery ensures correctness, i.e., linearizability [37], by ensuring two properties. First, only one instance of an application is accessible to clients at any point in time. Second, if the backup is promoted and becomes accessible, its state begins from the previous primary’s last acknowledged changes and thus it looks like a continuation of the old primary. The first property is achieved trivially using atomic promotion. The second property is achieved using super and collapse in sequence for a crash-consistent application: a backup is only promoted by collapse if there have been no writes since super, and thus the disk it recovers from must include the previous primary’s last acknowledged changes as required by crash consistency.

Our design of collapse uses writes to the primary’s disk as a signal of liveness to abort failover. This will correctly detect crash failures where the primary stops completely. However, it will not detect more nuanced kinds of failures such as partial failures or fail-slow failures. For example, even if the primary is disconnected from the clients, it may still write to disk for internal operations such as log rotation and garbage collection. This means that writes to the primary’s disk may not always reflect client-visible application state changes, and collapse would abort failover in these cases. In addition, fail-slow failures, where applications are slow but not inaccessible, can occur [26, 34, 40, 60]. In these failure situations, speculative recovery can be falsely and repeatedly aborted, causing an increased failover latency. In all these cases, speculative recovery should fall back to REDS by using a timeout to force failover when recovery is aborted repeatedly.

5 Implementation

We implemented a prototype speculative recovery system, SpecREDS, and deployed it on AWS EC2. The instance pool is implemented as a docker container pool on top of EC2 compute instances, where application images can be directly pulled from the docker registry. The failure monitor is implemented as a simple daemon process that pings the application instances with read-only queries to determine connectivity and health. As an independent component, the monitor also needs to be fault-tolerant. Many orchestration architectures provide fault-tolerant monitors such as those in EC2 Auto Scaling groups, Kubernetes, etc.

Our implementation of the disaggregated storage layer is based on Ceph [68], an open-source distributed storage system. On top of its backend object store called RADOS, Ceph provides highly durable and highly available block storage called rbd (Rados Block Device) that can remotely attach a rbd disk as a Linux block device through its kernel driver. SpecREDS focuses on the block interface due to its prevalent adoption on cloud and its simpler interface. We believe that the concept of speculative recovery can be applied to other cloud storage interfaces like network file systems [17] and object stores [19]. The implementation is based on Ceph release v16.2.4. The artifact of SpecREDS is publicly available. Please refer to the appendix for the artifact description.

Background on Ceph rbd. We give a short background on rbd necessary to understand our implementation. rbd also provides disk clone functionality: a disk snapshot is first taken, then a disk clone can be created from that snapshot. While clone creation is fast, rbd ’s native clone implementation has the performance problems of copying over the network and serialized concurrent COW writes, as discussed in §4.2.

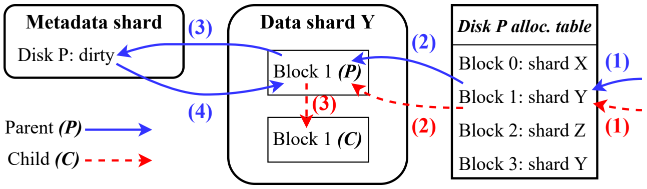


Figure 3: **Parent and child write path after `super`.** The parent and child disks are assigned ID ***P*** and ***C***, respectively. Only the first parent write to shard Y performs steps 3 and 4.

`rbd` implements the functionality of an "allocation table" with two separate utilities. First, each `rbd` disk has an object map, a bit map indicating the *existence* of the disk's data objects. A COW write needs to update the child's object map by marking the corresponding bit "dirty", which is a blocking operation due to locking, causing the effect of serialized concurrent COW writes. Second, the location of a data object is *calculated* deterministically by an algorithm based on the object name and the cluster layout [69]. Since objects have unique names, a child object will likely be placed on a different shard than its parent. For clarity, this section assumes that a `rbd` disk has an "allocation table" that combines the two utilities, as shown in Figure 3.

In addition, a `rbd` clone disk depends on its parent snapshot, and such dependency prevents the parent from deletion unless the cloned child is deleted first. As a result, for repeated failovers, the latest child will carry a chain of parent dependencies. These parents keep taking up space even though they are not needed anymore, as well as the child keeps suffering from COW penalties even after the failover is complete.

Collocation by reusing parent's "allocation table." To accomplish this, `super` directly assigns the parent disk to the child, including all data objects and the object map. This achieves two things. First, a COW write *never* needs to update the object map since by reusing parent's object map, the corresponding bit is already updated by the parent. Second, the child uses the same object names as the parent to locate objects, allowing for collocation of parent and child objects.

To differentiate, the parent and the child are assigned a unique ID. When accessing the disk, they identify themselves to the storage cluster using that ID. This means that creating the child disk is fast because it only involves the assignment of a unique ID. The names of the objects are tagged with the unique ID to identify the object ownership (parent or child).

To determine how to serve a child I/O, the data shard first checks the existence of the child object and the corresponding parent object by directly querying the backend object store. COW is performed for a child write if the child object does not exist but the parent object does. Figure 3 demonstrates this process (dashed red arrows). By reusing the allocation table, child access will be directed to the same shard holding the corresponding parent objects (steps 1 and 2) and thus allowing for collocation (step 3).

Object size. Another factor affecting COW performance is the object size since objects are the minimal unit of copying. But if a COW write contains some whole objects, copying is unnecessary for these objects. With large objects, data copying imposes huge overhead; with small objects, writes are more likely to contain whole objects to reduce copying overhead, but the overhead of allocating more smaller objects could overwhelm and thus degrade the overall performance. Our benchmark shows that `rbd` 's default 4 MB object size is not ideal for many database applications whose default page size is only 4 KB to 64 KB. The the sweet spot for these applications is around 64 KB.

Dirty bit tracking and atomic promotion. `collapse` elects the data shard that stores the parent disk's metadata (which is a single object) as the tracking shard. When `super` is invoked on the parent disk, its unique ID is registered to the tracking shard and then broadcasted to all data shards. The data shards then add the ID to a tracking list. When receiving a write with ID in the tracking list, the data shard must ask the tracking shard for permission to proceed (blue solid arrow, step 3 in figure 3). If permission is granted, the data shard can then submit this write and remove the ID from the tracking list; otherwise, this write must be rejected.

The tracking shard, by default, grants permission to any data shard requesting (step 4 in figure 3), sets the dirty bit associated with the ID, and notifies the other data shards to remove the ID from their tracking lists. The tracking shard also persists the dirty bit by writing it to the disk's metadata, allowing it to be replicated along with the metadata. In case the current tracking shard fails, another shard holding a replica of the metadata is elected the new tracking shard.

When initiating an atomic promotion, based on the dirty bit status of the parent disk, the tracking shard performs either of the two actions *atomically*: (1) if the dirty bit is set, the tracking shard returns an error to indicate that promotion is rejected and the child disk should be deallocated; (2) otherwise, the tracking shard starts rejecting all requests-for-permission to the parent disk and acknowledges that promotion can proceed.

Dirty bit tracking adds one additional RTT to at most the number of writes equivalent to the number of data shards in the cluster. Our evaluation shows that this has negligible performance impact (§6.2).

Deallocation with garbage collection. `collapse` deallocates the child disk by asynchronously garbage-collecting all objects associated with the child's unique ID. Similarly, the parent disk is deallocated by asynchronously garbage-collecting parent objects that have a corresponding child object and reassigning those who do not to the child's ownership. After this process is complete, the child no longer depends on the parent and no longer needs to do COW. Asynchronous garbage collection minimizes the performance impact to the storage cluster's normal operation.

6 Evaluation

This evaluation answers the following questions:

- How does the performance of collocated-clone disks compare to that of normal disks and general-clone disks? (§6.2)
- What is the recovery latency for various applications and failure scenarios when using a collocated-clone disk compared to using a normal disk and a general-clone disk? (§6.3)
- How does the failover latency of SpecREDS compare to REDS? (§6.4)
- What are the overheads of SpecREDS over REDS in terms of application performance after recovery, resource overhead, and overhead due to false positives? (§6.5)

We find that our implementation of a collocated-clone disk provides disk-level performance *close* to that of a normal disk and is much faster than a general-clone disk (§6.2). This performance translates to recovery latency when using a collocated-clone disk being close to using a normal disk (§6.3). This similar recovery latency leads to speculative recovery always providing failover latency comparable to REDS and often providing much lower failover latency across a wide variety of failover scenarios (§6.4).

6.1 Experimental Setup

We conducted our evaluation on EC2. The SpecREDS storage layer has four storage shards by EC2 instance type i3en with access to 7500 GB NVMe local SSDs and 25 Gbps network bandwidth. As shown in Table 1, our storage layer delivers performance comparable to popular cloud storage services.

For the primary and backup instances, we use the m5n instance type with 16 vCPUs, 64 GB RAM, and 25 Gbps network bandwidth, and for the application clients, we use an instance with 32 vCPUs, 128 GB RAM, and 25 Gbps network bandwidth. All instances are in the same availability zone as each other and the storage layer. We also set up a simple docker orchestrator environment on the primary and backup instances where applications are running in docker containers. The client instance runs oltpbench [31] with 100 virtual clients sending requests to the active instance. The primary is initially the active instance, while the failover process with our orchestrator makes the backup the active instance as it completes. The failure monitor, which pings the instances every second, runs as a separate daemon process on the client machine. We believe that this setup mimics existing systems like EC2 Auto Scaling groups and GCP Kubernetes Engine.

We pick three representative database applications: MySQL with InnoDB, PostgreSQL, and MariaDB with RocksDB. These applications meet the requirements of REDS and are widely used. The oltpbench client loads these application by running the TPC-C workload [65].

	KIOPS	**Tput (MB/s)**	**Latency (ms)**
EBS gp3	16/16	1000/1000	0.5/0.7
GCP SSD PD	15/15	245/245	0.6/0.7
Our storage layer	75/26	1000/630	0.38/2.0

Table 1: **Raw disk performance.** Comparing the raw disk performance of EBS General Purpose SSD (gp3), GCP SSD Persistent Disk, and our storage layer. Numbers in each cell are for read/write.

6.2 Disk-level Performance

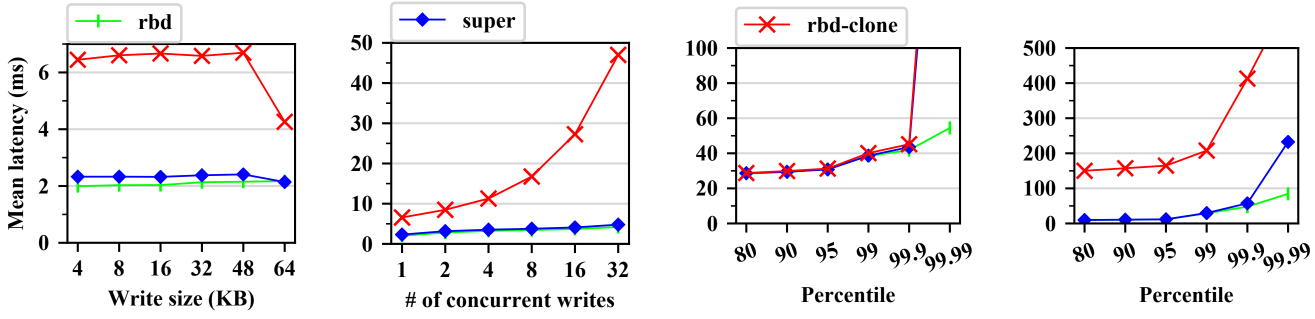
To build up to an end-to-end availability comparison, we start by showing a disk-level performance comparison between a regular, non-COW `rbd` disk, a collocated-clone disk implemented with `super`, and a general-clone disk implemented with native `rbd` cloning (`rbd-clone`). We use an object size of 64 KB for all disks. The experiments examine single write performance, concurrent write performance, performance for real recovery workloads, and the impact of the dirty bit on the parent disk performance. Our results indicate that the main source of improvement comes from the elimination of object map update operations, which could increase the latency of a single write by 6.1x under highly parallel I/Os.

Single COW write latency. As the first set of experiments, we isolate the latency impact of the COW designs when there is no concurrency with an experiment that issues *single writes*, where only a single write is in flight at a time. Figure 4a compares the mean latency (averaged over 20,000 writes) of COW writes with `super` and `rbd-clone` to normal writes with `rbd` for varying write sizes. A closed-loop client issues writes to random offsets.

For writes smaller than the object size, write latency on `super` is 14% higher than `rbd` while `rbd-clone` is 220% higher. `super` provides this similar performance because it avoids an object map update operation and does the copy locally instead of having to transmit data over the network. When the write size equals the object size (64 KB), no COW is necessary. This isolates the latency effect of object map update operations. For `rbd-clone`, this results in 2 ms of added latency (the update operation is basically another write), while `super` has identical performance to normal writes because it does not need to update the object map.

Concurrent COW writes. Next, we evaluate the latency of COW write under varying levels of concurrency. Figure 4b shows the mean latency of 4 KB COW writes as we vary concurrency from 1 write in flight at a time up to 32 writes in flight. A closed-loop client simultaneously issues *n* writes to random offsets, waits for all of their responses, and then repeats this process.

The latency of `rbd-clone` increases sharply with concurrency: the mean latency with 32 writes in flight is 6.1x higher than the mean latency with 1 write in flight. We found that this high latency is due to parallel object map update opera-



(a) Single writes

(b) Concurrent writes

(c) Read CDF

(d) Write CDF

Figure 4: **COW performance comparison. (a)** Latency for varying size COW writes with no concurrency; **(b)** Latency for 4 KB COW writes under increasing levels of concurrency; **(c)/(d)** Read/write CDFs from replaying a trace of recovery operations.

tions being serialized by the client's disk driver. In contrast, super provides similar performance to rbd under concurrency because it avoids updates to the object map: both have comparable mean latency that ranges from 2 ms with a concurrency of 1 to 4 ms with a concurrency of 32.

Performance on real recovery workloads. To quantify how these improvements of super translate to performance in real application recovery workloads, Figures 4c and 4d show read and write CDFs collected from replaying a recovery workload trace with fio [32]. The trace captures the recovery work for a Postgres database with 20 GB TPCC data and 1 GB of WAL at the time of an injected kernel panic failure.

For read, all three disks have similar read latency up to p99.9. super and rbd-clone have higher read latency beyond p99.9 due to the overhead of COW that may occupy a majority of disk throughput under high load and cause heavy I/O contention. For write, the write latency of rbd-clone is much higher than rbd. In contrast, the write latency of super is comparable to rbd up to p99.9.

Dirty bit tracking overhead. As discussed in §5, the dirty bit tracking mechanism of SpecREDS may impact the performance of the parent disk because some parent writes require an additional round trip to set the dirty bit. This could become a problem if the primary instance self-heals and continues serving the application. We performed an experiment that invokes super every second while measuring raw IOPS on the parent disk. Even under such an extreme condition, the parent disk achieves the same IOPS numbers. Thus, the only overhead of dirty bit tracking is increased latency for the few writes that set the dirty bit when the primary is still alive.

Summary. We believe that these disk-level improvements of super, as shown in Figure 4, can achieve recovery latency very close to a regular rbd disk in real failure scenarios, enabling end-to-end application availability improvement for SpecREDS, as presented in the next two subsections.

6.3 Application Recovery Latency

We ran a series of experiments to understand how disk-level performance of the three disk types (rbd, super, and rbd-clone) affects recovery latency as we vary failure type and failure timing. SpecREDS operates on a disk clone (super by default or rbd-clone) with COW penalties, which increases application recovery latency compared to REDS using a regular rbd disk without COW. It is critical for such latency increase to be relatively minor to show practical improvement in end-to-end application availability (§6.4)

For these experiments, the application initially runs in a container on the primary instance, handling requests from the clients. Then, a failure is injected to the primary. To isolate recovery latency, the failure monitor detects loss of connectivity with no timeout and immediately initiates failover, and the application restarts on the backup instance. The recovery latency is measured at the client side as the length of time between when the TPC-C throughput drops to zero and when it resumes. Failures are injected either by synchronously stopping the docker container and unmounting the disk (clean failures) or by causing a kernel panic (unclean failures).

We found the type of failure has a major effect on recovery latency. Stopping the primary container tries to gracefully shut down the application (this is the case for MariaDB but not for MySQL and Postgres), and unmounting the disk flushes file system cache such that the file system is not corrupted. Therefore, the disk is in a cleaner state and can recover faster. Kernel panic, on the other hand, immediately crashes the instance without giving a chance to clean up, leaving the disk in an unclean state that takes longer for the backup to recover. In addition, we found the size of WAL at time of failure also significantly impacts recovery latency.

Our full range of experiments have recovery latencies that vary from 1–70 s when run on rbd. We capture block-level traces of those recovery workloads with blktrace and then replay them with fio on rbd, super, and rbd-clone. Replaying traces ensures the workload is identical for all three disks.

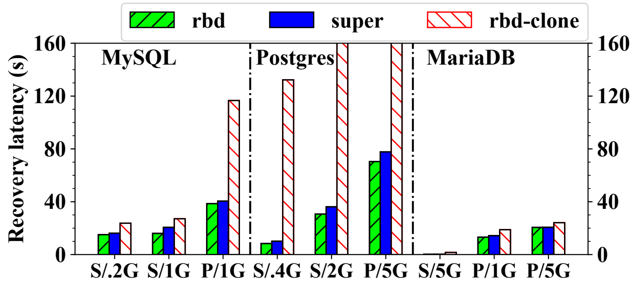


Figure 5: **Application recovery latency from various disk states.** Recovery latency is shown for our three applications running on `rbd`, `super`, and `rbd-clone`. Failures are injected using docker stop (**S**) or a kernel panic (**P**). Labels are failure types followed by WAL size in GB.

To make results legible while demonstrating the effect of varying WAL sizes and failure types, we select three scenarios to show for each application. The recovery latency for each disk with these scenarios is shown in Figure 5. In all cases, we see that `super` improves performance over `rbd-clone`. This is especially pronounced for Postgres whose recovery workload is generally more write-intensive, exacerbating the write performance bottlenecks in `rbd-clone` shown in §6.2. Further, recovery on `super` is only slightly slower than recovery on `rbd` by 13% on average.

6.4 End-to-end Failover Latency

To quantify the effect of speculative recovery for complete end-to-end failover scenarios, we simulated various failover scenarios and compare the latency across REDS (using `rbd`), SpecREDS (using `super` by default), SpecREDS (using `rbd-clone`), and the oracle model (using `rbd`). The oracle model shows the lower bound on failover latency: it runs recovery on a `rbd` disk immediately after a primary issues its last write (or simply waits for primary to come back online, whichever is shorter). Thus, the oracle shows failover latency without either REDS's timeout or SpecREDS's slower disk performance. On the other hand, REDS initiates recovery after a full timeout, while SpecREDS initiates much sooner after only *one second* of an unresponsive ping.

The simulations explored three variables: the primary-is-failed timeout, the recovery latency for the backup, and if/when the primary self-heals. Results are divided into broad categories depending on the timeout length (short, medium, or long), recovery length (short or long), and whether the primary self-heals after the timeout but before backup recovery completes (true or false positive recovery for REDS). Results with a long timeout (e.g., the Kubernetes default timeout of five minutes) are similar to a medium timeout but have even higher failover latency for REDS, so we only show results for a medium timeout. Results with false positive recovery for REDS all similarly inflate only the latency of REDS, so we only show one of these results. This leads to five categories.

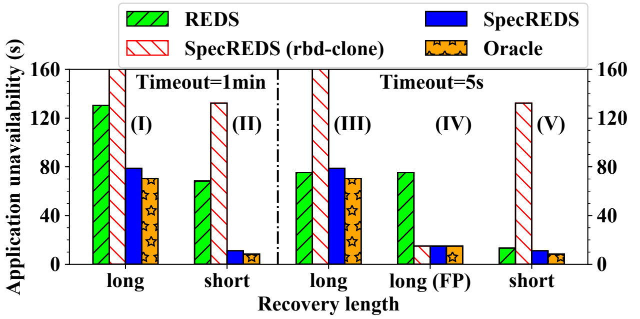


Figure 6: **End-to-end failover latency.** Representative failover scenarios, picked by varying the lengths of timeout and recovery. Bar group IV shows a false positive (FP) failover for REDS

Figure 6 shows a representative result from each of these five categories. The medium timeout is one minute and the short timeout is five seconds. The recovery latencies are picked from the results in §6.3. The long recovery is from Postgres with an unclean failure and a 5 GB WAL (around 70 seconds of recovery latency on `rbd`). The short recovery is from Postgres with a clean failure and a 0.4 GB WAL (around 8 seconds on `rbd`).

The two leftmost bar groups show scenarios with medium timeouts and demonstrate one major part of SpecREDS's availability improvement over REDS. Because SpecREDS starts recovery early without waiting for a full timeout, it completes failover much sooner and thus significantly reduces application unavailability.

The three rightmost bars in Figure 6 demonstrate short timeout failure scenarios. Bar groups III and V shows similar performance for REDS and SpecREDS with a long (III) or short (V) recovery. In these cases, SpecREDS start recovery slightly sooner than REDS. But, its recovery takes slightly longer because its `super` disk is slightly slower than the `rbd` disk used by REDS. With a long recovery (III), this makes REDS's unavailability marginally shorter than SpecREDS. With a short recovery (V), this makes SpecREDS's unavailability marginally shorter than REDS. Finally, bar group IV shows a false positive failover where the primary is available again (we used 15 seconds for illustration) shortly after the timeout. SpecREDS decreases unavailability considerably in this scenario by allowing the primary to continue instead of committing to recovery on the backup with no turning back.

Overall, there are three takeaways. First, the failover latency of SpecREDS (`rbd-clone`) is consistently the highest, indicating that the improved performance of the `super` disk is the key to achieving the availability improvement of SpecREDS. Second, SpecREDS achieves significantly lower failover latency when REDS uses a medium timeout (bar groups I and

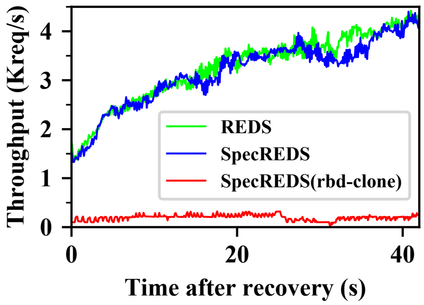


Figure 7: **Throughput after recovery.** Time 0 is right after recovery completes and clients resume.

II) because this timeout dominates REDS's unavailability; SpecREDS also achieves lower failover latency for false positives when REDS uses a short timeout (IV), while achieving similar failover latency in other cases (III and V). Third, SpecREDS is always close to the oracle lower bound, suggesting it achieves most of the possible availability improvement for a REDS-based fault tolerance scheme.

6.5 Other SpecREDS Overheads

To understand the other overheads of SpecREDS, we evaluated application performance immediately after recovery is complete, analyzed production health monitor logs to estimate the resource overhead of SpecREDS, and discussed the performance overhead on the storage layer's normal operation due to false positives.

Application performance after recovery. After the backup instance completes recovery and gets promoted, collapse asynchronously transfers parent objects to the child. During this time, COW is still used for writes that go to objects whose ownership has not yet been transferred. Figure 7 compares application performance following recovery (Postgres, unclean failure with 1 GB WAL) on REDS, SpecREDS, and SpecREDS (rbd-clone). SpecREDS using rbd-clone has low throughput due to the continued impact of COW because the parent-child dependency still exists, as discussed in §5. In contrast, we see that SpecREDS has a throughput curve very similar to REDS. Thus, we conclude that SpecREDS adds negligible overhead to application performance after recovery.

Resource overhead of SpecREDS. Due to running two instances concurrently during speculative recovery and the possibility of aborted recovery, SpecREDS incurs additional resource overhead compared to REDS. The key to understanding SpecREDS's resource overhead is to see how often it would be incurred. We analyzed a complete collection of health monitor logs from more than 80 production caching servers for the past five years. On average, a server is reported inaccessible once every 2.8 days. Of these reported events, 90% are transient: the server becomes accessible again within 10 seconds. Even with such a high false positive rate, the frequency of possible server inaccessible event is quite small. The resource overhead of SpecREDS would be, on average, an unnecessary backup instance allocation for up to 10 seconds once every 3.1 days: a 0.004% overhead.

Performance overhead due to false positives. False positives that trigger speculative failover that is aborted impose performance overhead on the storage layer due to garbage collection (GC). This may be troublesome since short-lived failures are common in today's data center networks [21,48,54,60], which could introduce frequent GC that could harm the storage layer's normal operation. Our log analysis described above, however, found that a server is reported inaccessible once every 2.8 days on average, meaning that SpecREDS incurs GC overhead only once every few days. Moreover, GC incurs minor performance overhead, since GC is asynchronous and thus does not block regular disk I/O operations, as shown in Figure 7.

7 Related Work

This section reviews related work on application-level replication, state machine replication, virtual machine replication, slow recovery in databases, shared storage clustering, disk snapshotting, and other related uses of speculation within systems. The most closely related work is the industry's adoption of REDS, which is introduced in §2.2 and discussed extensively throughout the paper.

Application-level replication. This is a widely implemented technique for providing high-availability fault tolerance. SQL databases, including MySQL, PostgreSQL, Microsoft SQL Server, as well as NoSQL databases such as MongoDB, replicate client transactions synchronously and persistently to backups before responding to clients [7–10]. This can provide excellent performance with failover latency shorter than SpecREDS. However, it requires an extensive implementation for each individual application since the replication logic and implementation are application-specific. Many useful persistence applications do not provide high-availability at the application layer, including SQLite, LevelDB, and RocksDB [1, 3, 4, 6, 11]. In contrast, Both REDS and SpecREDS support these applications without any modification or explicit support, since they work at the block-device layer.

Application-level replication requires multiple application instances at all times to provide fault tolerance: at least the primary instance and one backup. In contrast, REDS and SpecREDS only run a single instance almost at all times, which makes it far cheaper.

Application-level replication may also have lower performance in normal operation since it runs expensive replication protocols for client requests. We believe that this argument needs meticulous measurements to validate because REDS and SpecREDS do not eliminate the need for a replication protocol but instead runs it at the storage level. In addition, the replicas in application-level replication can provide read-only throughput. Though disaggregated storage can also offer better disk-level read throughput from data replicas, single application instance often cannot fully utilize it due to bottlenecks at CPU and network bandwidth [41]

State machine replication (SMR). This technique provides high availability for applications that use its interface, which is typically a log of requests executed in order [62]. SMR is typically implemented either using a consensus algorithm like Paxos [44] or a primary-backup approach [24]. SMR can often provide shorter recovery times than SpecREDS. But, like application-level replication, it requires multiple instances and thus is more costly than SpecREDS.

Virtual machine (VM) replication. This technique provides application-agnostic high-availability fault tolerance [23, 29, 53, 63]. This technique replicates an entire VM and thus can make any application or a collection of applications fault tolerant. However, VM replication is heavyweight because it replicates the entire virtual machine (e.g., all changes to memory must be replicated before they are externalized to provide linearizability). Also, it requires at least two instances at all times to provide fault tolerance. Speculative recovery supports the smaller set of applications that are crash consistent, but is much lighter weight and provides high availability at a much lower cost

Database slow recovery. This is a technique that precedes the cloud by decades where logs are periodically shipped to a backup that stores, but does not apply, them until a failover is needed. This similarly requires fewer backup resources in the normal case but results in slower recovery. REDS and SpecREDS build on this technique to provide a similar tradeoff more generally for any crash-consistent application and in a cloud-native way by using disaggregated storage to provide the backup its own copy of the disk instead of requiring any computation from a backup.

Shared storage clustering. This technique allows a storage volume to be attached to and accessible from multiple application instances at the same time, enabling faster failover in a clustered application setup without dismounting and remounting the volume to another instance [51]. The cloud-native version of this technique is "multi-attach" [14]. These techniques require a standby backup instance, which is not the case for REDS and SpecREDS.

Snapshots and checkpoints. Other forms of storage copy such as snapshots and checkpoints are widely used for data backup and rollback-based disaster recovery [38, 46, 67]. Many cloud platforms also support automatically taking snapshots of application disks on a user-specified schedule. However, this method does not provide linearizability amid failures because updates following the latest snapshot will be lost.

Speculation. This is a widely used technique to accelerate the performance of systems. Here we discuss a few of these systems that inspired us. Zyzzyva [42] is a Byzantine fault tolerance SMR protocol where the replicas speculatively execute client requests without agreeing on a single total ordering, and it is then the client's responsibility to observe and help resolve any inconsistencies. Speculative recovery adopts a similar idea that inconsistency can be allowed temporarily and resolved later.

Speculative Paxos [59] is a SMR protocol where replicas speculatively execute client requests based on the message delivery order provided by the underlying network layer. In cases where this order is violated, a reconciliation protocol is in place to rollback inconsistent operations. Such inconsistencies are detected before externalizing. Speculative recovery is similar in that inconsistencies cannot be externalized. This is also inspired by a similar idea in "rethink the sync" [55] where external clients are the real observer of the system.

Speculation is also widely adopted for tolerating tail latency in data-parallel computing such as Hadoop and Spark [64,70]. When a computing job is taking an unexpectedly long time, the same job will be sent to another worker, and the system uses the results from whichever finishes first. Hedged requests are a similar technique that is used for applications that access many backend systems [30] as well as other domains such as RAID storage arrays [35,36]. Speculative recovery is similar to these techniques in that there are two racing paths and latency is determined by the first path to finish.

8 Conclusions

We presented speculative recovery, a cheap, highly available fault-tolerance scheme based on disaggregated storage for crash-consistent applications. At the core of speculative recovery are the two new primitives, `super` and `collapse`, for disaggregated storage. `super` provides performant disk clones with the novel collocated-clone design, and `collapse` ensures application correctness, i.e., linearizability, in a failover process with a disk-global dirty bit. Speculative recovery achieves the same level of resource efficiency as REDS with significantly higher availability in most failover scenarios.

Acknowledgments

We thank our anonymous shepherd and reviewers for their many constructive comments. We thank Khiem Ngo and Jeffrey Helt for their helpful discussions. We thank CloudLab [61] for providing compute resources used in the development of this project. This material is based upon work supported by the National Science Foundation under Grants No. 1763546, 2028869, and 2106530.

References

[1] About SQLite. https://www.sqlite.org/about.html.

[2] Docker. https://www.docker.com/.

[3] How we use RocksDB at Rockset. https://rockset.com/blog/how-we-use-rocksdb-at-rockset/.

[4] LevelDB Store. https://activemq.apache.org/leveldb-store.

[5] Linux Containers. https://linuxcontainers.org/.

[6] Litereplica: Replication Support for SQLite. http://litereplica.io/sqlite-replication.html.

[7] Microsoft SQL Server Replication. https://docs.microsoft.com/en-us/sql/relational-databases/replication/sql-server-replication?view=sql-server-ver15.

[8] MongoDB Replication. https://docs.mongodb.com/manual/replication/.

[9] MySQL Replication. https://dev.mysql.com/doc/refman/8.0/en/replication.html.

[10] PostgreSQL Replication. https://www.postgresql.org/docs/9.2/runtime-config-replication.html.

[11] rocksplicator, RocksDB Replication. https://github.com/pinterest/rocksplicator.

[12] StatefulSets – Kubernetes. https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/.

[13] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '20, pages 419–434, Santa Clara, CA, February 2020. USENIX Association.

[14] Amazon. Attach a volume to multiple instances with Amazon EBS Multi-Attach. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-volumes-multi.html.

[15] Amazon. EC2 Auto Scaling groups. https://docs.aws.amazon.com/autoscaling/ec2/userguide/AutoScalingGroup.html.

[16] Amazon. Elastic Block Storage. https://aws.amazon.com/ebs.

[17] Amazon. Elastic File System. https://aws.amazon.com/efs/.

[18] Amazon. Elastic IP addresses. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/elastic-ip-addresses-eip.html.

[19] Amazon. Simple Storage Service (S3). https://aws.amazon.com/s3/.

[20] Michael Baentsch, Georg Molter, and Peter Sturm. Introducing Application-Level Replication and Naming into Today's Web. *Computer Networks and ISDN Systems*, 28(7–11):921–930, May 1996.

[21] Peter Bailis and Kyle Kingsbury. The network is reliable: An informal survey of real-world communications failures. *Queue*, 12(7):20–32, 2014.

[22] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. Fault-Tolerance in the Borealis Distributed Stream Processing System. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, page 13–24, New York, NY, USA, 2005. Association for Computing Machinery.

[23] T. C. Bressoud and F. B. Schneider. Hypervisor-Based Fault Tolerance. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, page 1–11, New York, NY, USA, 1995. Association for Computing Machinery.

[24] Navin Budhiraja, Keith Marzullo, Fred B Schneider, and Sam Toueg. Distributed systems. *ch. The Primary-Backup Approach*, pages 199–216, 1993.

[25] Ceph. rbd Persistent Read-only Cache. https://docs.ceph.com/en/latest/rbd/rbd-persistent-read-only-cache/.

[26] Mike Y. Chen, Anthony Accardi, and Dave Patterson. Path-Based Failure and Evolution Management. In *First Symposium on Networked Systems Design and Implementation*, NSDI '04, San Francisco, CA, March 2004. USENIX Association.

[27] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 228–243, New York, NY, USA, 2013. Association for Computing Machinery.

[28] Brian Cho and Ergin Seyfe. Taking Advantage of a Disaggregated Storage and Compute Architecture. In *Spark+AI Summit 2019*, SAIS '19, April 2019.

[29] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. In *5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '08, San Francisco, CA, April 2008. USENIX Association.

[30] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Communications of the ACM*, 56(2):74–80, February 2013.

[31] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proceedings of the VLDB Endowment*, 7(4):277–288, 2013.

[32] fio. Flexible I/O tester. https://fio.readthedocs.io/en/latest/fio_doc.html.

[33] Google. GCP Persistent Disks. https://cloud.google.com/persistent-disk.

[34] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems. In *16th USENIX Conference on File and Storage Technologies*, FAST '18, pages 1–14, Oakland, CA, February 2018. USENIX Association.

[35] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O. Suminto, Cesar A. Stuardo, Andrew A. Chien, and Haryadi S. Gunawi. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 168–183, New York, NY, USA, 2017. Association for Computing Machinery.

[36] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S. Gunawi. LinnOS: Predictability on Unpredictable Flash Storage with a Light Neural Network. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20, pages 173–190. USENIX Association, November 2020.

[37] Maurice Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[38] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, feb 1988.

[39] Yige Hu, Zhiting Zhu, Ian Neal, Youngjin Kwon, Tianyu Cheng, Vijay Chidambaram, and Emmett Witchel. TxFS: Leveraging File-System Crash Consistency to Provide ACID Transactions. In *2018 USENIX Annual Technical Conference*, USENIX ATC '18, pages 879–891, Boston, MA, July 2018. USENIX Association.

[40] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray Failure: The Achilles' Heel of Cloud-Scale Systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, page 150–155, New York, NY, USA, 2017. Association for Computing Machinery.

[41] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash Storage Disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[42] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 45–58, New York, NY, USA, 2007. Association for Computing Machinery.

[43] Kubernetes. kube-controller-manager. https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager/.

[44] Leslie Lamport. Paxos Made Simple, Fast, and Byzantine. In *Procedings of the 6th International Conference on Principles of Distributed Systems*, OPODIS '02, pages 7–9, 2002.

[45] Sergey Legtchenko, Hugh Williams, Kaveh Razavi, Austin Donnelly, Richard Black, Andrew Douglas, Nathanael Cheriere, Daniel Fryer, Kai Mast, Angela Demke Brown, Ana Klimovic, Andy Slowey, and Antony Rowstron. Understanding Rack-Scale Disaggregated Storage. In *9th USENIX Workshop on Hot Topics in Storage and File Systems*, HotStorage '17, Santa Clara, CA, July 2017. USENIX Association.

[46] LVM-HOWTO. Taking a Backup Using Snapshots. https://tldp.org/HOWTO/LVM-HOWTO/snapshots_backup.html.

[47] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is Lighter (and Safer) than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 218–233, New York, NY, USA, 2017. Association for Computing Machinery.

[48] Shicong Meng, Arun K. Iyengar, Isabelle M. Rouvellou, Ling Liu, Kisung Lee, Balaji Palanisamy, and Yuzhe Tang. Reliable State Monitoring in Cloud Datacenters. In *Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing*, CLOUD '12, pages 951–958, 2012.

[49] Microsoft. Azure blob storage. https://azure.microsoft.com/en-us/services/storage/blobs.

[50] Microsoft. High availability in Azure Database for PostgreSQL – Single Server. https://docs.microsoft.com/en-us/azure/postgresql/concepts-high-availability.

[51] Microsoft. Use Cluster Shared Volumes in a failover cluster. https://docs.microsoft.com/en-us/windows-server/failover-clustering/failover-cluster-csvs.

[52] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. Gimbal: Enabling Multi-Tenant Storage Disaggregation on SmartNIC JBOFs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 106–122, New York, NY, USA, 2021. Association for Computing Machinery.

[53] Umar Farooq Minhas, Shriram Rajagopalan, Brendan Cully, Ashraf Aboulnaga, Kenneth Salem, and Andrew Warfield. RemusDB: Transparent High Availability for Database Systems. *Proceedings of the VLDB Endowment*, 4(11):738–748, August 2011.

[54] Srihari Nelakuditi, Sanghwan Lee, Yinzhe Yu, Zhi-Li Zhang, and Chen-Nee Chuah. Fast Local Rerouting for Handling Transient Link Failures. *IEEE/ACM Transactions on Networking*, 15(2):359–372, April 2007.

[55] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the Sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 1–14, USA, 2006. USENIX Association.

[56] Haochen Pan, Jesse Tuglu, Neo Zhou, Tianshu Wang, Yicheng Shen, Xiong Zheng, Joseph Tassarotti, Lewis Tseng, and Roberto Palmieri. Rabia: Simplifying State-Machine Replication Through Randomization. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 472–487, New York, NY, USA, 2021. Association for Computing Machinery.

[57] Colin Percival. EC2 boot time benchmarking. https://www.daemonology.net/blog/2021-08-12-EC2-boot-time-benchmarking.html.

[58] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14, pages 433–448, Broomfield, CO, October 2014. USENIX Association.

[59] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *12th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '15, pages 43–57, Oakland, CA, May 2015. USENIX Association.

[60] Rahul Potharaju and Navendu Jain. When the Network Crumbles: An Empirical Study of Cloud Network Failures and Their Impact on Services. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SoCC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[61] Robert Ricci, Eric Eide, and CloudLab Team. Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications. *login USENIX Magazine*, 39(6), 2014.

[62] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[63] Rahul Singh, David Irwin, Prashant Shenoy, and K.K. Ramakrishnan. Yank: Enabling Green Data Centers to Pull the Plug. In *10th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '13, pages 143–155, Lombard, IL, April 2013. USENIX Association.

[64] Riza O. Suminto, Cesar A. Stuardo, Alexandra Clark, Huan Ke, Tanakorn Leesatapornwongsa, Bo Fu, Daniar H. Kurniawan, Vincentius Martin, Maheswara

Rao G. Uma, and Haryadi S. Gunawi. PBSE: A Robust Path-Based Speculative Execution for Degraded-Network Tail Tolerance in Data-Parallel Frameworks. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 295–308, New York, NY, USA, 2017. Association for Computing Machinery.

[65] TPC-C. An On-Line Transaction Processing Benchmark. http://www.tpc.org/tpcc/.

[66] Peng Wang, Kaiyuan Zhang, Rong Chen, Haibo Chen, and Haibing Guan. Replication-Based Fault-Tolerance for Large-Scale Graph Processing. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, pages 562–573, 2014.

[67] Andrew Warfield, Russ Ross, Keir Fraser, Christian Limpach, and Steven Hand. Parallax: Managing storage for a million machines. In *Proceedings of the 10th Conference on Hot Topics in Operating Systems*, HotOS '05, page 4, USA, 2005. USENIX Association.

[68] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 307–320, USA, 2006. USENIX Association.

[69] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, pages 31–31, 2006.

[70] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '08, page 29–42, USA, 2008. USENIX Association.

A Artifact Appendix

Abstract

The artifact provides a framework for evaluating SpecREDS as shown in the evaluation section of the paper. The artifact includes the source code of SpecREDS's disaggregated storage layer (based on Ceph), configuration files and pre-captured application recovery traces, and handy scripts for instrumenting the experiments. Readers can easily use this artifact to reproduce figures shown in the paper.

Scope

There are two main claims from the paper that the artifact seeks to validate: **(1)** the disk-level I/O performance of `super`, our novel design and implementation of light-weight, fast disk clones, is *close* to that of a regular, non-COW disk, while significantly outperforming Ceph's existing clone implementation `rbd-clone`; **(2)** SpecREDS using `super` can bring practical end-to-end application availability improvement over REDS in various failover scenarios.

Specifically, the paper uses Figures 4 and 5 to prove the point of the first claim and Figures 6 and 7 to prove the second claim. The artifact contains experiments to reproduce these four figures, and readers should be able to compare them with the original figures in the paper to validate the claims.

Contents

The artifact contains the following items

- The source code of SpecREDS's disaggregated storage
- A simple tool for measuring disk-level performance
- Pre-configured configs, disk images, and traces
- Scripts for instrumenting all experiments
- Detailed readmes

Hosting

The artifact is hosted on our public GitHub repository at https://github.com/princeton-sns/specreds. The tag for the OSDI/ATC artifact evaluation is `atc22ae`. To get started, please follow the detailed instructions in the repo.

Requirements

The artifact does not require special hardware or software, but we highly recommend running the artifact on CloudLab with a `c220g2` or `c220g5` machine where the artifact is tested to be reproducible. If not available, we recommend using a machine with at least 16 CPU cores, 64 GB memory, 400 GB of free space on an SSD, and Ubuntu 20.04. We also provide a qcow2 image for booting up a QEMU VM.