Asynchronous Prefix Recoverability for Fast Distributed Stores

Tianyu Li*
litianyu@csail.mit.edu
MIT CSAIL

Badrish Chandramouli
badrishc@microsoft.com
Microsoft Research

Jose M. Faleiro
jmf@microsoft.com
Microsoft Research

Samuel Madden
madden@csail.mit.edu
MIT CSAIL

Donald Kossmann
donaldk@microsoft.com
Microsoft Research

ABSTRACT

Accessing and updating data sharded across distributed machines safely and speedily in the face of failures remains a challenging problem. Most prominently, applications that share state across different nodes want their writes to quickly become visible to others, without giving up recoverability guarantees in case a failure occurs. Current solutions of a fast cache backed by storage cannot support this use case easily. In this work, we design a distributed protocol, called Distributed Prefix Recovery (DPR) that builds on top of a sharded cache-store architecture with single-key operations, to provide cross-shard recoverability guarantees. With DPR, many clients can read and update shared state at sub-millisecond latency, while receiving periodic prefix durability guarantees. On failure, DPR quickly restores the system to a prefix-consistent state with a novel non-blocking rollback scheme. We added DPR to a key-value store (FASTER) and cache (Redis) and show that we can get high throughput and low latency similar to in-memory systems, while lazily providing durability guarantees similar to persistent stores.

ACM Reference Format:
Tianyu Li, Badrish Chandramouli, Jose M. Faleiro, Samuel Madden, and Donald Kossmann. 2021. Asynchronous Prefix Recoverability for Fast Distributed Stores. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3458454

1 INTRODUCTION

The rise of cloud computing has resulted in applications that increasingly work with data *distributed* across machines. In a modern cloud-based relational database or key-value store, the storage tier is typically provisioned and scaled separately from the compute tier running application logic. Examples of this include raw cloud storage such as Amazon S3 [2] and Azure Storage [6], as well as database tiers that are backed by raw storage, such as Amazon DynamoDB [1]. This model incurs high per-operation latency, which cannot be hidden when applications perform dependent cross-shard operations, without introducing complex distributed coordination.

To avoid this latency penalty, many applications resort to caches (such as Redis [19]) in front of the storage tier. Caches can ameliorate the performance limitations of storage; they can immediately service read and write requests on cache-resident data, limiting synchronous interactions with the storage tier. This, however, comes at the expense of consistency, recoverability, and increased complexity of application logic. Specifically, today's caching-based solutions have no notion of a "commit" that is resilient to failure. Upon failure, the system is left in an inconsistent state, where some writes are recovered and others are not, and reads are either stale or lost. Clients must ensure idempotence of operations, or reason about application-level correctness, which is challenging [8, 45].

The problem is made worse on modern *serverless* offerings such as Azure Functions [18] or AWS Lambda [17]. Applications on such platforms are often structured as *workflows* of operators. For workflow durability [14, 15], operators interact synchronously with storage for resilient logging and state persistence, during inter-operator communication. Caching would risk creating inconsistency in the face of failures, particularly as the serverless compute layer scales at a finer grain and works with ephemeral instances [36].

In summary, no existing distributed data processing system provides the benefits of caching without renouncing strong failure guarantees and consistent recovery. Such a system can provide good performance while simplifying application logic and easing the adoption of distributed cloud storage solutions. We aim to address this with our design, called *distributed prefix recovery* (*DPR*).

Distributed Prefix Recovery (DPR)

In DPR, storage is split into disjoint partitions or shards, where each shard spans volatile memory and durable storage. We refer to such storage as a *cache-store*; examples include key-value stores such as FASTER [21], Redis, and logging systems such as Kafka [16]. Applications interact with a distributed cache-store using *sessions* of read and write operations, each operation being on a single shard.

DPR offers *prefix recoverability* to client sessions. Upon failure, the recovered state corresponds to a recent known valid prefix for each session [44]. This state respects all dependencies between operations within a session, where an operation is causally dependent on all preceding completed operations in the same session. Applications can easily limit unnecessary dependencies and boost performance by interacting with the store via multiple sessions.

DPR separates the notion of operation *completion* from operation *commit*. The application is allowed to work on uncommitted state in caches directly, and DPR guarantees to make application state durable asynchronously, in a way that respects dependencies.

*Work started during internship at Microsoft Research.

SIGMOD '21, June 20–25, 2021, Virtual Event, China.

ACM ISBN 978-1-4503-8343-1/21/06.
https://doi.org/10.1145/3448016.3458454

Applications working with DPR can forge ahead using uncommitted (distributed) state if deemed safe, or choose to defer critical (e.g., user-facing) transitions until commit.

By decoupling recoverability from operation completion, applications working with DPR are not bottlenecked on the *latency* of operation commit and are only limited by the memory and network *throughput*. Writes to durable storage and distributed coordination happen off the critical path of request processing. Therefore, systems implementing DPR can serve compute tasks at close to local-memory latency and throughput in the common case.

DPR can work with any cache-store. We build DPR on top of a cache-store abstraction that combines fast but volatile operations and asynchronous checkpoints. This abstraction covers diverse storage systems including key-value stores, caches, and persistent logs. As a result, one can add DPR to any existing unmodified system with little effort, as we demonstrate with D-Redis, a distributed version of Redis with DPR guarantees. Deeper integration into a highly-tuned cache-store provides further benefits: we present D-FASTER, a distributed cache-store that builds on FASTER [21], our high-performance open-source[1] single-node key-value store that supports *concurrent prefix recoverability* (CPR) [44] – a similar guarantee to DPR, but only for a single node. D-FASTER uses FASTER's non-blocking checkpoints to orchestrate a DPR commit as a lightweight metadata-only operation. Upon failure, if some uncommitted state in surviving nodes is lost, D-FASTER coordinates an asynchronous non-blocking rollback to a prefix-consistent state.

DPR operates without any additional coordination beyond the normal cross-shard traffic due to client operations. This property makes DPR lightweight to add to an existing system. It does so by piggybacking on client messages to orchestrate the global commit, along with a lightweight metadata store that all nodes independently access off the critical path. The key insight is that we can limit coordination to the machines accessed by sessions that create data dependencies.

Example Use Cases for DPR

Example 1 (Cloud Telemetry). *Consider a cloud application that inserts telemetry data into a distributed cache-store. A telemetry service can continuously read uncommitted data and write back aggregate per-key metrics, with a guarantee that the aggregates will not commit without the contributing data committing as well. Further, a feed service can read such metrics and offer tentative results (e.g., as a dashboard or feed) at very low latency for immediate consumption, while also depicting committed views as they become available lazily. Finally, a fault-detection service can perform data analysis and write a fault report, with a guarantee that the report will not commit unless the data it depends on commits as well.*

Example 2 (Serverless Workflows). *A serverless application consists of a graph of operators executing user logic. A typical approach for durability, used in systems such as Azure Durable Functions [14] and Temporal Workflows [15], is to write each operator's input to some shard of a persistent log such as Kafka (the* cache-store*). Naively executed, every enqueue has to wait for a commit, resulting in high end-to-end latencies. Using DPR, a dequeue from the cache-store by a downstream operator can view enqueues by preceding operators before they commit, providing lower operation completion latencies, while at the same time providing lazy commits that the application may use to expose results to the outside world.*

[1]FASTER is available at https://github.com/microsoft/FASTER.

Summary of Contributions

- We propose the DPR model, which enables distributed cache-stores to offer prefix recovery guarantees. DPR decouples operation completion from commit to avoid bottlenecking on commit latency while avoiding cross-shard coordination overhead.
- We present a DPR-based distributed key-value store, D-FASTER, which features a novel asynchronous non-blocking rollback technique for active nodes.
- We also describe D-Redis, an unmodified Redis that offers DPR guarantees by wrapping it in a new library we have developed (libDPR) that can add DPR to any unmodified cache-store.
- We evaluate D-FASTER and D-Redis in a cloud deployment to demonstrate their scalability, performance, and recovery guarantees. We show that D-FASTER achieves throughput and latency at the level of pure in-memory caches while providing recoverability.

Before describing these features in detail, we begin with a high-level overview of the DPR architecture and client model.

2 DPR ARCHITECTURE

DPR makes it possible for systems to expose to clients a unified API over a sharded set of *cache-stores*, representing paired caches and corresponding durable backing stores. Each cache-store completes client requests and immediately makes their effects visible to other clients before durability. At the same time, DPR reports commits asynchronously to the client as prefixes. Clients may choose to withhold operation result until it is committed for a familiar durable-store experience or proceed ahead with uncommitted state for maximum performance. In the latter case, when a failure occurs, the next call to DPR will return an error with the exact prefix that survived the failure so clients can react accordingly.

The system restores global state to a consistent previous state upon failure. It provides *prefix consistency* [44], where a recovered state contains all the effects of client operations in a linearizable execution schedule up to some point, and none after. The schedule respects the natural session expectation that operations logically depend on all prior completed operations in the session.

We illustrate this model in Figure 1. The system consists of a number of shards, each with a cache component and a durable store. Multiple caches may be backed by the same durable store. These cache-store shards form a global logical database. Client applications interact only with the (sharded) cache-store abstraction, where the cache responds to both read and write requests, with lazy write-back to the store. The exact mapping of application keys to shards is orthogonal to DPR and may be dynamic, using techniques standard in distributed databases [52], such as shared metadata tables. We uniquely assign exactly one shard as the owner for any key, and route all read/write requests to the owner. Because cache operations do not block on the backend durable store, this scheme is sufficient to support high throughput on a single key without compromising read freshness or consistency of operations.

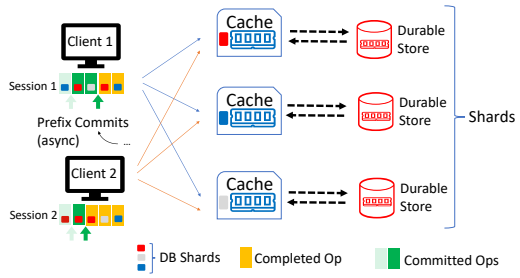


Figure 1: Overall Architecture and Client Model

Clients use the concept of a *session* to interact with the cache-store shards. A session consists of a sequence of *operations* that access and update data. For example, operations for a key-value API include `read`, `upsert`, and `delete`, whereas a log supports `enqueue` and `dequeue`. Where our API diverges from traditional architectures is that we explicitly decouple operation *completion* from operation *commit*. Operation completion does not need to wait on expensive cross-shard synchronization or flushes to durable storage. Instead, the system asynchronously informs clients of commits of their prefixes, and clients can proceed with uncommitted operations.

DPR supports both synchronous and asynchronous operation completion. Operations in a client session are linearizable [34]; an operation is logically dependent on all previously completed operations on that session, and these dependencies are honored by our lazy commit guarantees. DPR builds on the recent notion of prefix consistency in shared memory within one failure domain in CPR [44] and extends it to the multi-node setting across failure domains and sharded memory. Sessions form the logical unit for determining dependencies in client operations with DPR; clients may establish multiple sessions in order to explicitly decouple dependencies and improve performance. Seen from a traditional transaction processing viewpoint, state is made visible to applications at operation completion and before commit, but with implicit commit dependencies that are respected from a recovery perspective. Importantly, sessions may *wait for commit* at any time and ensure that all preceding operations have been made durable, making it easy to behave like traditional durable stores with group commit.

3 DISTRIBUTED PREFIX RECOVERY

We now present the DPR protocol in detail. DPR operates over a set of clients interacting with a set of abstract shards, which we call `StateObjects`. `StateObjects` provide the following API:

- **`Op()`**: executes a read/write operation and returns uncommitted
- **`Commit() -> (token, committed)`**: commits (or makes durable) a prefix of previously uncommitted **`Op()`s**, and returns a unique token and description of committed operations for each client.
- **`Restore(token)`**: restores the `StateObject` to some committed state identified by the token.

We note that this API is straightforward to implement in a single-node cache-store. `Op()` can be implemented by returning after operation completion on the cache. `Commit()` can reuse existing group commit mechanisms implemented by most modern stores [44]. Group commit boundaries are typically triggered by an internal timer [49]; instead, one can modify these systems to allow group commit boundaries to be explicitly triggered via an API call. Finally, `Restore()` can leverage failure recovery mechanisms already implemented by existing systems. As a different example, a simple write-ahead or operation log with periodic group commit [16] may also be viewed as a `StateObject` implementation.

A client session issues `Op()`s in sequence to multiple shards of `StateObjects` and maintains a `SessionOrder` – a linearizable order of `Op()`s that respects the natural partial order induced by operation completion: Op_1 precedes Op_2 in a `SessionOrder` if Op_1 completes before Op_2 begins. In Figure 2, our running example, we show two `SessionOrders`, S_1 and S_2 each issuing 4 operations to 3 `StateObjects`, containing objects A, B, and C respectively. `SessionOrders` determine the recovery behavior of DPR – if Op_1 precedes Op_2, Op_2 is dependent on Op_1, and should not be recovered without Op_1. We write the recovery tokens of individual `StateObjects` in the form of *Object-version*, e.g., A-2 is the second committed token of A. Each token captures some prefix of client operations performed on the `StateObject`. In Figure 2, we write beside each operation the token it is captured in. In order to provide actionable information to clients on failure, the system produces *DPR-cut*s and reports *DPR-guarantee*s to clients:

Definition 3.1. DPR-cut – A DPR-cut is a set of tokens, such that after calling `Restore()` with all included tokens, the system is recovered to a prefix consistent state for every client.

Definition 3.2. DPR-guarantee – A DPR-guarantee is a mapping from each client to a point on its `SessionOrder`, such that under failure, every operation before is recovered, but none after.

As an example, we depict in Figure 2 a DPR-guarantee of up to operation 2 of S_1 (S_1-2), and operation 1 of S_2 (S_2-1), with a dotted line. The system backs this guarantee with a DPR-cut that consists of A-1 and B-1, shown with a dashed circle on the right. Recovering to this DPR-cut restores the corresponding DPR-guarantee for all sessions. Our goal is to design a lightweight protocol that: 1) asynchronously gives DPR-guarantees, which we cover in this section, and 2) correctly restores system state to the guaranteed DPR-cut in the event of a failure, which we will describe in Section 4.

3.1 Modeling

A consistent DPR-cut must satisfy *dependencies* introduced by client sessions. Importantly, we track dependencies at the granularity of *versions*, which is the aggregate state of each `Commit()`, instead of individual operations. Within versions, there exist operation dependencies across `SessionOrder` induced by shared access (e.g., read-write dependencies on a key). These are resolved through `StateObject` implementations, as most modern linearizable `StateObject` implementations support consistent checkpointing that respects these dependencies. DPR focuses instead on cross-shard dependencies, which are always introduced by a client session issuing consecutive operations to different `StateObjects`. This is true because we assume a single-owner, non-transactional model where all read and write accesses to a key are sent to the same `StateObject`, and thus all such dependencies appear as intra-version dependencies. We more formally define dependency as follows. A token B-n depends on A-m if either

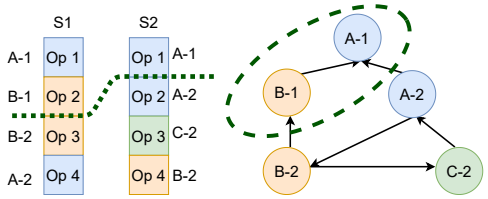


Figure 2: Example of a Precedence Graph – precedence graphs model token dependencies, and `SessionOrders` determine how tokens connect to each other.

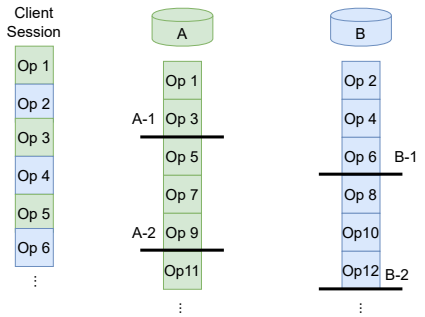


Figure 3: No Cuts without Coordination – calling `Commit()` in an uncoordinated manner may never yield a DPR-cut.

- there exists a `SessionOrder`, where an operation included in *A-m* is immediately followed by an operation included in *B-n* (dependency by *precedence*).
- there exists some token *C-p*, such that *B-n* depends on *C-p*, and *C-p* depends on *A-m* (dependency by *transitivity*).

We can visualize this relationship with the *precedence graph*. Every version is a vertex in the graph, and a directed edge goes from token *B-n* to *A-m* if *B-n* depends on *A-m* by precedence. We show an example precedence graph in Figure 2. Intuitively, a set of tokens form a DPR-cut iff the set of tokens they capture is closed under the transitive property of dependency.

3.2 Ensuring Progress in DPR

It is important for the DPR protocol to ensure *progress* by continuously giving non-trivial guarantees, such that any operation will be eventually recoverable. One might think that progress is guaranteed as long as the system periodically calls `Commit()` on each `StateObject`. Unfortunately, this is **not** the case, and Figure 3 shows a counter-example. Assume there are two single-threaded `StateObjects`, *A*(green) and *B*(blue). A client issues operations in sequence that alternates between manipulating *A* and *B*. The black lines denote each call to `Commit()`, which captures every operation above the line and emits the token next to it. *A*-1 and *B*-1 are staggered such that they cannot form a DPR-cut. From that point forward, the `Commit()` calls are synchronized to happen every 3 operations. By induction, we can see that given any pair of tokens, they do not form a DPR-cut, and therefore, the system is never able to give any non-trivial DPR-guarantee. This problem manifests on the precedence graph as each version having an infinite dependency set — any token in this trace depends on all future tokens.

To prevent this, `StateObjects` must coordinate calls to `Commit()`. Consider the following algorithm: each client maintains local version counter V_s, the largest version number a session has seen, and appends V_s to every request. A `StateObject` checks V_s against its next token version v, and executes the request only if $v \geq V_s$; otherwise it calls `Commit()` until $v \geq V_s$ before executing. The `StateObject` returns v to client afterwards, and client sets $V_s = max(V_s, v)$. We sketch a proof for why this algorithm ensures DPR progress. Under this scheme, we have *monotonicity* in that a version never depends on versions with larger version numbers. Observe that monotonicity is preserved under transitivity. By induction, if *B-n* depends on *C-p*, and *C-p* depends on *A-m*, $n \geq p$ and $p \geq m$ by induction hypothesis, and thus $n \geq m$. We therefore only need to prove the algorithm preserves monotonicity under precedence. This is ensured because V_s can only increase by construction. Our mechanism is similar to the classic Lamport clock algorithm [39], where V_s encodes dependency information as part of the client requests with no additional cross-shard coordination.

3.3 Finding DPR-guarantees

Recall from Section 3.1 that DPR-cuts are transitive closures on the precedence graph, and we can find the latest DPR-cut building the maximal such closures. The challenge is that the precedence graph is distributed. Take Figure 2 again for an example, the dependency between *B*-2 and *C*-2 results from $S_1 - 4$, which is chronologically after the last client interaction with *C*, and *C* is not aware of its existence. Because DPR coordinates `StateObjects` solely through client requests, a `StateObject` is only aware of precedence that exists at the time of the requests, or outgoing edges on the graph. To find DPR-cuts, the algorithm must join this information. Additionally, the cluster must achieve fault-tolerant consensus on the current DPR-cut. Otherwise, failure can cause a `StateObject` to renege on its guarantee and recover to an earlier cut.

We now sketch an exact algorithm to solve this problem with a simple implementation, outlined in the upper half of Figure 4. We persist all information on durable shared storage (e.g., an ACID database) as the fault-tolerant consensus mechanism, and additionally provision a compute node (the coordinator). Each `StateObject` adds a version and its dependencies to the (durable) precedence graph after each local checkpoint. Periodically, the coordinator traverses the precedence graph to find the maximal transitive closure and persists it as the current DPR-cut. Because the coordinator only sees committed versions, the computed cut is always safe, albeit slightly outdated. We illustrate this on the right of Figure 4. Here, `BuildDependencySet(v)` traverses the precedence graph with breadth-first search starting from `v` and returns the visited set, whereas `UpdateCutAtomically` atomically updates the durable store such that the cut is never partially read. The compute node can simply be restarted in case of failure, as it is stateless.

3.4 Approximations and Optimizations

The exact algorithm presented above can have scalability issues in very large clusters because of the write bottleneck on the underlying store. In particular, the size of the precedence graph may scale quadratically to the size of the cluster. To improve performance, we need to avoid persisting the entire graph. Consider an approach

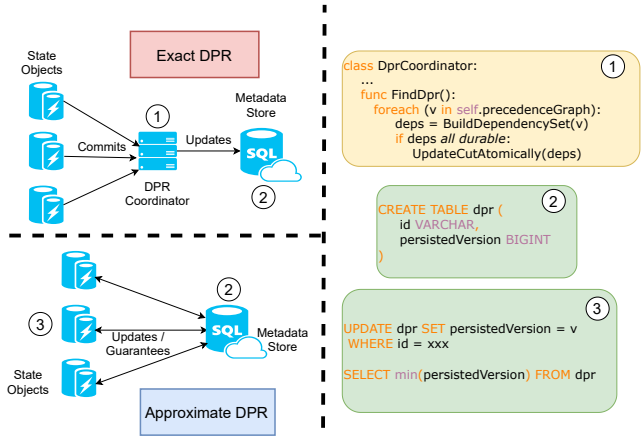


Figure 4: Summary of DPR Tracking Algorithms – we offer exact and approximate algorithms with accuracy-scalability tradeoffs.

where the coordinator keeps the graph purely in-memory. Upon failure, the precedence graph is lost; when a new version arrives, the coordinator cannot be certain of its dependency set in the lost subgraph and therefore cannot safely advance the DPR cut.

It is possible to overcome this obstacle through approximations. We show one such algorithm on the bottom left of Figure 4. `StateObjects` now write only committed version numbers to the durable store and discard dependency information. At any time, let V_{min} be the smallest version number in the durable store. Then, a DPR-cut in the system consists of all tokens of version V_{min}. This is correct because no versions can depend on larger versions. One problem remains: two `StateObjects` will not synchronize their versions if no clients operated across them, and they will advance versions at their own pace. Approximate DPR introduces a false dependency between them such that if a `StateObject` A calls `Commit()` less frequently than B, B can only advance its position in the DPR-cut at A's pace. To address this, we can additionally track V_{max}, the largest version number in the durable store. We can then program each `StateObject` to periodically read its value and fast-forward its next checkpoint to be at least V_{max}, which allows for a lagging `StateObject` to catch up in bounded time.

The approximate algorithm does not store the precedence graph, and is sufficiently lightweight for the DPR cut computation to be directly pushed down to storage (e.g., through aggregates in SQL) without the need for a coordinator node. Due to its imprecision, however, it may be most useful when used as a fault-tolerant fallback for the exact algorithm. Consider again the implementation where the coordinator keeps the precedence graph only in memory, except now the approximate algorithm is run in parallel. On recovery, the exact algorithm is temporarily unable to commit with an incomplete graph, but eventually the approximate algorithm will advance the DPR cut past the missing subgraph, at which point the coordinator can proceed as normal. Such a hybrid algorithm has improved scalability over the exact algorithm, but retains high precision in the failure-free common case.

4 NON-BLOCKING FAILURE RECOVERY

We now present an algorithm that restores the system to a consistent DPR-cut upon `StateObject` failures. A failure in the system potentially affects all other `StateObjects`, as any `StateObject` may have outstanding dependencies on lost operations. Although unavoidable, we would like to limit its impact by allowing `StateObjects` to recover and resume operation and not do so in lock-step with others. We call this *non-blocking recovery*.

4.1 Assumptions

Even though recovery is non-blocking from DPR's perspective, clients may still observe unavailability if the `StateObject` implementation has a blocking `Restore()` implementation. We describe in 5 how D-FASTER provides a non-blocking implementation. We assume throughout this section that an external entity, a cluster manager such as Kubernetes [9] or Azure Service Fabric [4] helps with failure detection and restarts. We focus on the handling steps in response to an identified failure in DPR. The cluster manager restarts failed servers in bounded time and restores them to their latest guaranteed checkpoint. The rest of the cluster now needs to rollback to the last known DPR cut to maintain prefix consistency. The cluster manager orchestrates this by temporarily halting DPR progress and sending a message to each worker to rollback, resuming progress only after all workers have reported back with completion. Clients are notified of failure the next time they perform an operation or check on their commit status.

4.2 Tracking World-lines

Although DPR is temporarily blocked during recovery, ideally operations continue normally without guarantees. However, the system is temporarily inconsistent during recovery, and client operations to the system can result in anomalies. Consider the execution trace given in Figure 5, where some other failure results in both A and B calling `Restore()`. A client issues operations to both `StateObjects` while system recovery is underway. After A recovers, the next time the client issues an operation to A, A propagates information about the failure to the client. The client computes the surviving prefix, performs any necessary application-level failure handling, and proceed to issue operation 11 to B. At this time, the client assumes that it is operating in a post-recovery world, but B has not recovered. A subsequent call to `Restore()` from B wipes out operation 11, which violates the prefix guarantee.

Intuitively, the state of the system is partitioned into a pre-recovery world and a post-recovery world during recovery. Any operation in the pre-recovery world is no longer protected by DPR, and a post-recovery operation must not depend on it. We formalize this notion with the concept of *world-lines*. A world-line is an uninterrupted trajectory of system state evolution through time, where the effect of every operation is eventually recoverable as prefix of later operations. We show a model of this on the right-hand side of Figure 5, where time flows downwards, and the horizontal position of a point represents the state of the system. When a failure causes the system to resume at some prefix, it may evolve differently onwards (i.e., a new world-line "branches off" from a prefix). The two world-lines co-exist until recovery is complete.

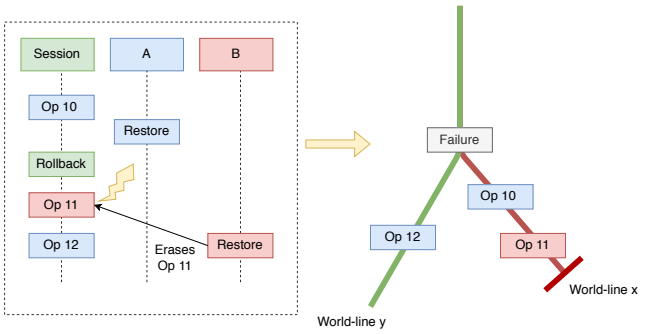


Figure 5: Prefix Anomaly during Recovery – `StateObjects` recovering at their own pace naïvely can cause anomalies.

The key to a correct non-blocking recovery implementation is to ensure that `SessionOrders` only operate on `StateObjects` that are on the same world-line as they are. We use a mechanism similar to viewstamps [42] in traditional distributed systems, and augment operation versions with a world-line id. More concretely, the cluster manager assigns a serial id to each failure; workers use this to uniquely identify world-lines, as world-lines only spawn due to failures. Each `StateObject` and `SessionOrder` maintains a world-line counter. Clients append their world-line to every request, and `StateObjects` execute a request only if their world-lines match. Otherwise, the `StateObject` either returns an error if its world-line number is larger, or delays execution until after recovery if smaller. A `StateObject` advances its world-line by calling `Restore()`, and a `SessionOrder` advances its world-line by computing a surviving prefix and reporting it to the client. This is similar to the DPR progress protocol, except that incrementing the counter is now associated with error-handling actions instead of `Commit()`.

4.3 Proof Sketch

We now briefly sketch how one may show the unified DPR scheme of DPR guarantees and non-blocking recovery to be correct. We first define the key properties that constitute correctness:

(1) *Prefix Recoverability* – All committed operations are persistent, and form a valid prefix of system state
(2) *Progress* – Every issued operation is eventually either completed and committed, or rolled back
(3) *Rollback Convergence* – Given a trace with finite failures, DPR eventually restores the system to a valid prefix

Prefix recoverability follows by structural induction on our definition of dependencies. We require that a valid trace of DPR have infinitely many `Commit()` calls for each `StateObject`. Then, we can show that our scheme leaves each version with a finite dependency set, which will eventually be persisted, thus committing all of its operations. Coupled with failure handling that rolls back any uncommitted results, we can show progress. Finally, intuitively, given finite failures, there exists an infinite suffix of a failure trace that is failure-free. We can then show rollback convergence by inductively proving that our failure handling preserves prefix-consistency and always terminates on said failure-free trace. Additionally, we note that existing work in the theory community [29, 33, 47, 53] has shown that algorithms similar to individual pieces of DPR are correct under a different system model. A formal proof is left for future work.

5 FAST DPR WITH D-FASTER

We now present D-FASTER, a distributed key-value cache-store that supports low-latency read/write operations and asynchronous recoverability through DPR. Clients can interact with D-FASTER in a traditional client-server setting or locally through co-location with cache-store servers.

5.1 `StateObject` Implementation

We use FASTER [21] as our `StateObject` implementation. FASTER is a popular open-source single-node multi-threaded key-value store library that provides fast and scalable linearizable reads, upserts, and read-modify-write operations. Clients use sessions, sequential logical threads of execution, to interact with FASTER. At the core of FASTER is a latch-free hash table that maps keys to logical addresses on a persistent record log. FASTER uses `HybridLog`, which spans main memory and persistent storage and supports in-place updates. Only `HybridLog` records on persistent storage, or currently being persisted are immutable and updated via read-copy-updates. Most in-memory records are instead modified in-place, which effectively compresses the log between flushes, and eliminates contention on the log tail. FASTER provides *concurrent prefix recoverability* (CPR) guarantees to its clients, which is similar to DPR but only across threads on a single-node. Threads in CPR coordinate loosely to write to the unified log, and fail as a unit, in contrast to the DPR model. FASTER is the ideal implementation for a `StateObject` in our DPR model — the in-memory, mutable portion of the log operates as the fast, caching layer, and the persistent part of the log serves as the durable store.

5.2 D-FASTER Architecture

We wish to provide an API for D-FASTER that resembles a single-node experience. Unlike FASTER, D-FASTER cannot be completely embedded into application code, as it needs to manage cross-worker communications and DPR maintenance. On a single node, however, we would like to support *co-located execution*, where application threads operate on D-FASTER directly via shared memory. For example, in a serverless framework, D-FASTER can run on the servers that execute user functions, and allow co-located functions to access and share local state at memory speeds.

We illustrate D-FASTER's architecture in Figure 6. D-FASTER workers share nothing and coordinate through two services: DPR-tracking and ownership mapping. DPR-tracking runs the DPR algorithm in Section 3, and ownership tracking maps keys to assigned workers. A cluster manager monitors the health of these workers and triggers failure recovery. Within each worker, D-FASTER runs FASTER, which spans its `HybridLog` to include cloud storage as well. Each D-FASTER worker owns a unique slice of the global keyspace, managed by a FASTER instance, and uses FASTER's CPR state machine to orchestrate asynchronous `Restore()` and `Commit()` calls. Clients lookup key owners through the ownership mapping service and connect to workers directly.

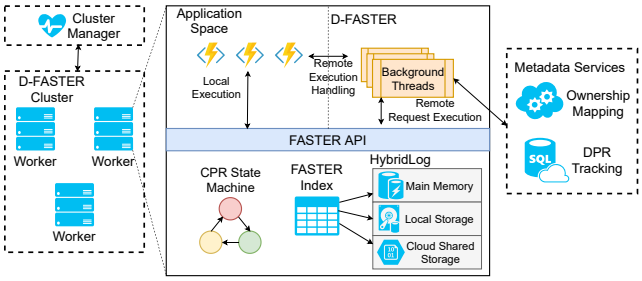


Figure 6: Overview of D-FASTER Architecture – D-FASTER builds on top of FASTER to provide a unified key-value store interface across shards, and allows for co-located execution.

D-FASTER client sessions are identified by a globally unique id. When a session operates on a worker, the worker creates a corresponding FASTER session using the unique id. Application code issues operations to D-FASTER, which first checks if the local worker owns the key. If so, the wrapper forwards the operation to local FASTER and completes the operation on the calling thread. Otherwise, the wrapper sends the operation to the owner for remote execution. Clients can choose to either wait for the operation to complete or entrust a background thread with handling the remote worker response through a `PENDING` mechanism, discussed in Section 5.4. Each D-FASTER worker additionally maintains a pool of background threads that execute requests on behalf of remote clients, trigger local FASTER checkpoints, and DPR operations.

5.3 Metadata Management

D-FASTER requires three pieces of metadata information to function: DPR tracking, cluster membership, and key ownership. We have already covered DPR tracking in Section 3. The other two are required for any state-of-the-art distributed key-value store, but their implementation can interplay with DPR, making it necessary to co-design them. We track all three pieces of information with an Azure SQL database for fault-tolerance. Clients cache information of key ownership and cluster membership locally, and only consult the SQL DB when changes occur. Broadly, our approach for metadata management follows techniques from our prior work – an elastic client-server version of FASTER called Shadowfax [38]; we focus only on key DPR-related design aspects below.

Cluster Membership Changes. D-FASTER stores a mapping from workers to their version number as the current DPR cut. We use this table as the source of truth for cluster membership in D-FASTER. Adding a worker or removing an empty worker is equivalent to adding ore dropping a row in the DPR table. Non-empty workers first migrate all keys before leaving.

Key Ownership Tracking: We use an additional ownership table to map keys to the workers that currently own them. It is unrealistic to keep track of ownership information on a per-key basis, so we introduce the concept of a *virtual partition*. Each virtual partition consists of related keys that co-locate, and users provide a mapping from keys to virtual partitions. Hash- and range-based partitioning schemes are supported by default.

Ownership Validation and Transfer: Before each operation, a worker needs to validate its ownership of the key. In D-FASTER workers do so against its local view of the ownership mapping, rather than the remote metadata store, and reject requests that fail to validate. Workers can use leases to guard against outdated ownership information, as is standard practice. When transferring ownership, the old owner renounces ownership locally before updating the metadata store; the key is temporarily without an owner during transfer, and the client retries until the transfer is complete. To ensure DPR correctness in the face of ownership transfer, we defer ownership transfer to checkpoint boundaries, such that ownership is static within versions. Shadowfax explores how to perform ownership transfers; we omit the details in this paper.

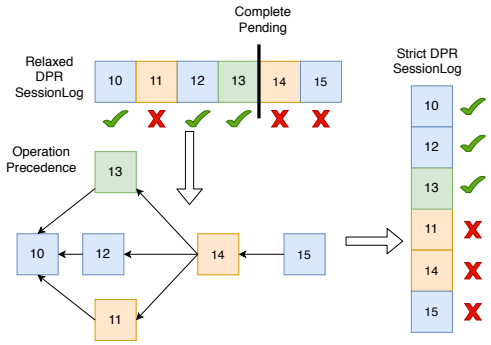


Figure 7: Equivalence of Relaxed DPR and Strict DPR – relaxed DPR merely renames operations by their start time.

5.4 Relaxed CPR/DPR with Pending Operations

FASTER uses operation begin time to number each operation in a session, and provides a strict prefix guarantee based on these serial numbers. Operations that started before a prefix boundary must finish execution before later operations. With high-latency I/O devices, sessions that may go dormant after issuing an operation, or in a distributed setting such as with D-FASTER, such a strict guarantee would block commits. It is important for client sessions to issue multiple unrelated requests in parallel to mask operation latency, and such parallelism should not block commits. Fortunately, prefix recoverability only requires linearizability for each `SessionOrder`, i.e., the linearizable order does not have to correspond exactly to FASTER serial numbers. Therefore, we need to simply relax FASTER's strict prefix scheme while retaining prefix recoverability. We call our approach *relaxed CPR* for FASTER and correspondingly, *relaxed DPR* for D-FASTER.

Both FASTER and D-FASTER execute a client request immediately if the request is in local memory, and return a `PENDING` status otherwise. Clients either wait on the operation or issue more operations in parallel before waiting on them as a group. The client session maintains a buffer of (completed and in-progress) asynchronous operations. Later operations do not depend on such `PENDING` operations until explicit resolution, by calls to `CompletePending()` on the session. Relaxed CPR/DPR defines its `SessionOrder` using operation start time, but weakens the prefix consistency guarantee. Specifically, recovered prefixes may now have missing `PENDING` operations that are clearly identified in an *exception list* as part of

the commit. Relaxed CPR is now the default option in FASTER, but users may revert to strict CPR if so desired.

We show an example of failure recovery with relaxed DPR in Figure 7. Operations shown in blue (10, 12, 15) are local and execute immediately, whereas others execute remotely and go `PENDING`. The recovery status of each operation is indicated on the side. This seemingly leads to a violation of DPR properties: the recovered prefix (10, 12, 13) misses operation 11. Observe, however, that this is not an anomaly — 12 and 13 are concurrent with 11 and cannot depend on it. It is therefore possible to find another equivalent linear execution order that satisfies strict DPR. We give a concrete example in Figure 7. We show the operation precedence that constrains our execution order on the bottom left, representing read-write or write-write dependencies. We then reorder the previous example into a strict-DPR-consistent `SessionOrder` on the right while satisfying these constraints. One can similary show that relaxed DPR is merely a restating of the strict DPR guarantee that uses an alternative ordering within `SessionOrders`.

5.5 Non-blocking `Restore` Implementation

We now present how D-FASTER supports non-blocking `Restore()` by extending FASTER's checkpointing mechanism. As discussed in [44], FASTER's checkpoints are already non-blocking. This is achieved through a state machine abstraction, where threads loosely coordinate to step through a series of global transitions. The system maintains a global "current state" variable and the current version number; each thread periodically refreshes its local view of that global state, executing logic to catch up as needed. We advance the system state after all threads have reached that state, or when some external criteria (e.g., disk write persistence) are met. Normally, FASTER threads operate in the **REST** state at some version v, and threads update mutable records in-place without additional coordination. When checkpointing, the system first waits for all threads to observe the start of a checkpoint and move to the global state to $v + 1$. Threads catch up to $v + 1$ at their own pace and disable in-place updates for records in v, so the system can capture a precise image of the system state as of v when all threads enter $v + 1$. When the checkpoint is complete, the system goes back to **REST** and resumes normal operation in $v + 1$. D-FASTER uses a similar mechanism for `Restore()`, as shown in Figure 8. When `Restore()` is invoked to recover from v to some earlier v_{safe}, the system disables in-place updates and move to **THROW** in $v + 1$. Similar to checkpointing, this creates a fuzzy cut-off of v entries on the log — after all threads enter **THROW**, no more entries from v can appear in the log. The system then proceeds to mark all entries in $(v_{safe}, v]$ invalid in **PURGE** before moving back to **REST**.

To support operations on the database during rollbacks, we leverage FASTER's hash index implementation that handles hash collision with chaining. As different versions of an entry share a key, one can accesses all versions that are not garbage-collected by traversing the hash chain. We modify FASTER to ignore all entries in $(v_{safe}, v]$ when in **THROW** or **PURGE**, so threads only see keys in or before v_{safe}. D-FASTER only garbage-collects FASTER log entries that are in the DPR guarantee so this mechanism functions.

D-FASTER conveys a failure to the client by throwing an exception upon completing the transition to **Throw**. We assume that the client has properly handled the failure when they issue the next request, and allow them to operate in $v + 1$ post-recovery. Implementing rollbacks in the checkpoint state machine also prevents concurrent checkpoints from occurring, as the system allows at most one state machine to execute at a time.

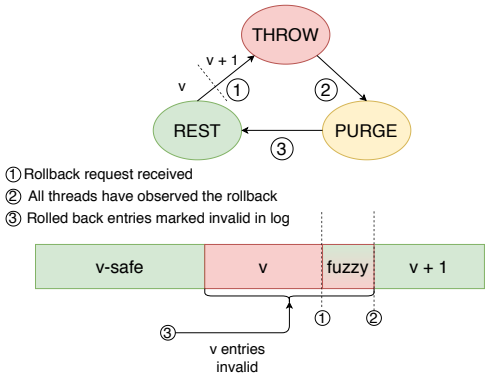


Figure 8: Overview of Rollback State Machine – threads first coordinate to draw fuzzy end of the lost versions. Readers ignore these entries while we mark them invalid in the background.

6 GENERAL DPR WITH D-REDIS

We now describe a library called libDPR that can help add DPR semantics to an unmodified cache-store. We use Redis with libDPR to build D-Redis, Redis with DPR guarantees.

Figure 9 shows the architecture of D-Redis. We color code each component by the software they belong to – red for the D-Redis wrapper, blue for unmodified Redis, and green for libDPR.

The D-Redis wrapper consists of two components. The client-side wrapper provides a session-based interface for applications to issue operations. Messages are serialized into batches, enhanced with a DPR-specific header, and sent to the server. The server-side wrapper reads messages, does DPR-specific work, forwards operations to Redis, and returns DPR-enhanced responses to the client, which replies to the application.

libDPR implements all necessary logic to track, find, and report DPR guarantees. It operates on a per-batch basis. On the client-side, libDPR assigns sequence numbers to operations in the session, tracks committed prefixes and computes dependencies for message headers. It also tracks committed session prefixes and determines if the client needs to roll back due to server failures. On the server-side, libDPR is invoked before and after each request batch is processed. Before a batch is processed, libDPR uses the header and DPR information to determine if it is safe to execute. It may need to delay execution, e.g., until a local commit is issued, or reject a batch in case of failure. Once libDPR clears a batch for processing, it eventually becomes recoverable. The library supports custom version tracking by the server wrapper while executing a batch on the cache-store in parallel to ongoing commits. Finally, the server-side libDPR computes the response header, including per-operation version information, to be sent back to the client. libDPR invokes `Commit()` and `Restore()` calls on `StateObject`, in order to commit periodically. The library also tracks world-lines and rolls back the `StateObject` in events of failure.

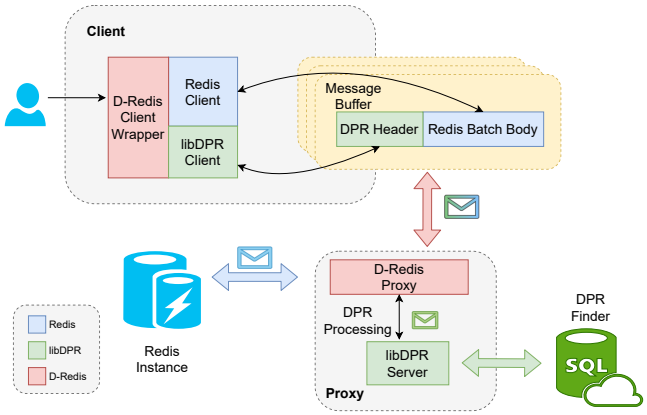


Figure 9: Architecture of D-Redis Using libDPR

The D-Redis server wrapper is simpler than D-FASTER, because Redis is single-threaded. There is one latch associated with the wrapper. We take an exclusive latch before invoking Redis checkpoints using the BGSAVE command for `Commit()`. For each incoming batch, the wrapper takes a shared latch before calling libDPR and processing it. This ensures that all operations in a batch belong to the same version. We use periodic LASTSAVE calls in the background to determine when a checkpoint is finished. Finally, `Restore()` is implemented by restarting the Redis instance in question.

7 EVALUATION

We review our implementation and setup in Section 7.1, and then address the following questions in subsequent sections:

- Whether DPR preserves FASTER's scalability and performance when adding distributed recoverability guarantee (Section 7.2).
- Whether DPR supports low operation and commit latencies at high throughput, even over cloud storage (Section 7.2).
- Whether a co-location model can benefit applications (Section 7.3).
- Whether the DPR recovery model allows for fast failure recovery in D-FASTER (Section 7.4).
- Whether libDPR can efficiently add DPR to Redis. (Section 7.5).
- How DPR compares to other recoverability guarantees in terms of performance impact. (Section 7.6).

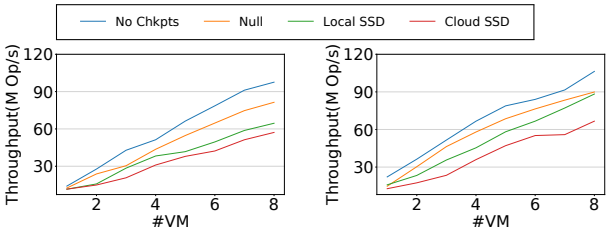
7.1 Experimental Setup

Setup – We performed our experimental evaluation on the Azure public cloud [24]. We ran all experiments on a cluster of Standard DS14 v2 machines [10] running Windows Server 2019-Datacenter with 128 vCPUs across 8 VMs (16 vCPUs per VM). Each VM uses accelerated networking that offloads networking logic onto physical hardware [7]. DPR's backend metadata store is an Azure SQL database [5] with a maximum of 8 vCores and 64 GB data size.

Workload – We used the YCSB-A workload [23] for most experiments shown, with 250 M distinct 8-byte keys and 8-byte values with uniform or Zipfian access patterns. We describe workloads as R:BU for the fraction of reads and blind updates. The keyspace is sharded by hash value into equal chunks, and each VM owns one shard. For co-located benchmarks we first used a random number generator to classify an operation as local or global. For local operations, we picked keys from the co-located server's local keyspace, otherwise, we picked keys from the global keyspace.

Implementation – We implemented D-FASTER in C# on top of FASTER. The network layer is similar to Shadowfax [38] but runs natively on C# with co-location support. All network communication is over TCP. Each client thread maintains a window of outstanding remote operations with size w. Clients stop issuing remote operations and wait when their window is full. Clients accumulate up to b messages before sending as a batch to reduce networking overhead. We used $w = 16b$ unless otherwise noted, which, on average, allows two batches to be in-flight simultaneously to each worker in an 8-machine cluster. The FASTER index is sized at #keys/2 hash bucket entries. Servers periodically perform fold-over `HybridLog` checkpoints by marking the log read-only and flushing. For all experiments shown, we use approximate DPR on Azure SQL; we have also experimented with other DPR algorithms but found minimal differences in performance at a cluster size of 8.

Other Systems – We use Redis as the integration target for libDPR; Redis clients use pre-computed command batches (batching and windowing is as before). We also experiment with Apache Cassandra (with the default YCSB driver [13]) and Redis as baselines of our study on the performance impact of recoverability levels.



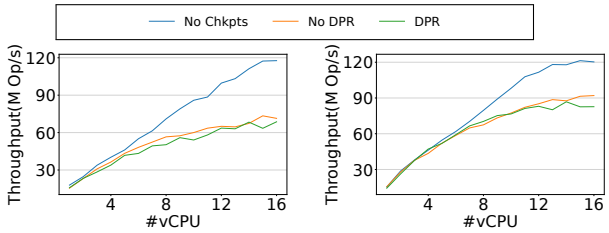
(a) uniform 50:50 **(b)** Zipfian($\theta = 0.99$) 50:50

Figure 10: Scaling out D-FASTER

7.2 Client-Server Performance

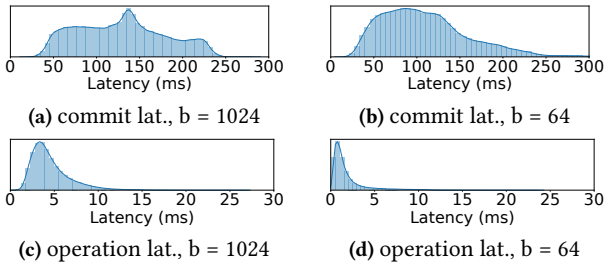
We first show D-FASTER's throughput (Figure 10 and Figure 11) and latency (Figure 12) with clients issuing requests from dedicated client VMs. We issued requests from 8 client machines. Unless otherwise specified, we took a checkpoint every 100 ms on every D-FASTER worker. We ran these experiments across 3 different storage backends – null, local SSD, and Azure Premium SSD (cloud SSD). The null storage device completes every I/O request instantaneously but incurs all the overhead associated with checkpointing and DPR. It represents a theoretical upper bound for the performance of D-FASTER's recoverability model. Local SSD is the temporary storage disk attached to each VM, and Azure Premium SSD is a replicated cloud storage service that is highly available and fault-tolerant [3]. We observed that checkpoints over Premium SSD took 2 to 3 times longer to complete than local SSD. We took an initial checkpoint before the start of measurement, except when the benchmark is marked as "no checkpoint". Under "no checkpoint", the FASTER log is entirely mutable and we do not invoke the checkpointing code path, imitating a pure in-memory cache. All experiments in this subsection run for 30 seconds and report average throughput.

Server Throughput: Figure 10 shows that D-FASTER can effectively scale-out. D-FASTER achieved close to 60M operations



(a) uniform 50:50 **(b)** Zipfian(θ = 0.99) 50:50

Figure 11: Scaling up D-FASTER



(a) commit lat., b = 1024 **(b)** commit lat., b = 64

(c) operation lat., b = 1024 **(d)** operation lat., b = 64

Figure 12: Latency Distribution of D-FASTER

per second with persistence guarantees, which was only about 40% slower compared to running without guarantees. As expected, using slower cloud storage resulted in slightly lowered performance, but as we will show later, this reduction of performance mostly stems from the underlying FASTER instances spending more time in the slow code-path completing checkpoints, instead of directly waiting on storage. We also plot how well D-FASTER scales up on larger machines in Figure 11. We manually pinned the D-FASTER process to a subset of cores on each of the 8 VMs. D-FASTER still scaled up to the number of vCPUs on the VM compared to no checkpoints, but at a reduced rate, due to the overhead of checkpointing. We additionally show D-FASTER's performance when taking uncoordinated checkpoints without DPR, and see that DPR added minimal overhead. When running with Zipfian, we see an improvement in performance, because the frequently accessed keys were quickly copied and subsequent accesses were in-place in FASTER. Performance is 20% higher across the board in Figure 10, and thread scalability is better in Figure 11. We have also experimented with other configurations, including with read-modify-write workloads and read-mostly workloads, and observe similarly that DPR does not slow down D-FASTER, and that D-FASTER operates at close to in-memory performance despite frequent checkpoints. We omit these experiments due to space limitations and show only results from the 50:50 Zipfian configuration in subsequent experiments.

Latency: Figure 12 plots the distribution of operation commit and operation completion latency for D-FASTER. We sampled 0.1% of the requests when running 8 servers with all cores enabled, and recorded the time between operation start and completion, and later, commit. An average request committed in ≈150ms, or one checkpoint interval plus the time to perform a checkpoint and DPR-induced delays. On the other hand, client operation completed in a few milliseconds; most of this latency was due to client batching. When we ran the benchmark with a smaller batch size of 64, we

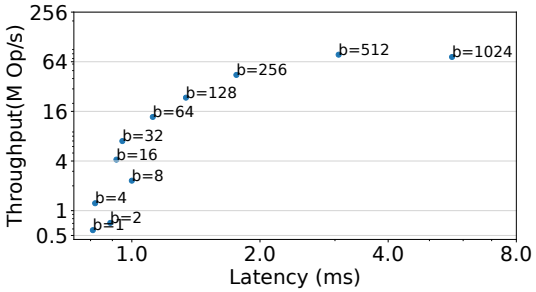


Figure 13: Throughput-Latency Trade-off

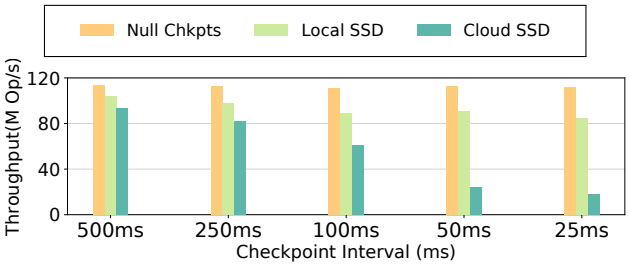


Figure 14: Impact of Storage Backend on Throughput

achieved sub-millisecond latency, albeit at reduced throughput. This reduced load also resulted in faster and more stable commits. Finally, we observed these latencies to be on par with Redis as well.

Throughput-Latency Trade-off: Figure 13 plots D-FASTER's performance with different batch sizes at 100ms checkpoints. Both the x-axis and y-axis are log-scale. The sweet spot for this trade-off appeared at b=64, where we achieved the highest throughput with around 1 millisecond of latency. After this point, we increased throughput at the expense of latency. The system was saturated at 90M ops/s, achieved with b=512. Increasing batch size beyond that added latency without throughput benefit.

Sensitivity to Storage Latency: Figure 14 shows the performance impact of different storage backends on D-FASTER. We ran three configurations — null, local storage, and cloud storage while varying checkpoint intervals. Even though the 3 have orders of magnitude difference in latency, the resulting throughput difference was only about 15% on less frequent checkpoints. This is because D-FASTER spent more time in reduced performance mode when a checkpoint was underway. The performance impact of cloud storage was more pronounced as we increased checkpoint frequency. This is because persisting a DPR checkpoint on cloud storage took 50ms (on average), and D-FASTER thrashed at a checkpoint frequency of 50ms or lower. Overall, D-FASTER can tolerate high-latency storage and retain performance.

7.3 Co-location Performance

We now evaluate the performance of D-FASTER running in co-location mode. Here co-location means we used the same cluster configuration from before, but instead issued client requests from the servers themselves so that there was no network communication between clients and local servers. Note that some client requests still needed to be sent to remote servers for remote keys.

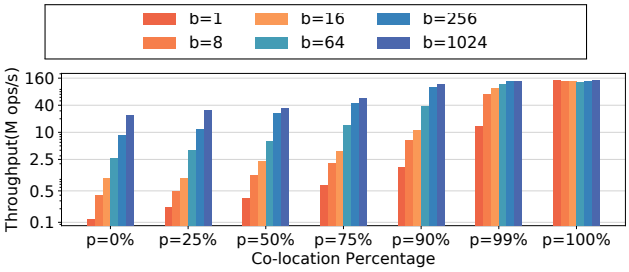


Figure 15: Co-location Throughput

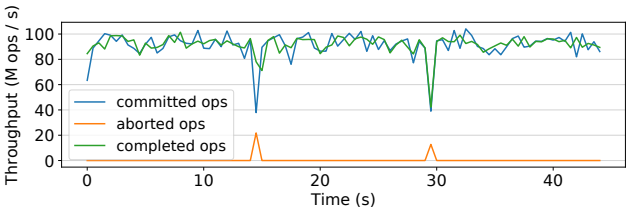
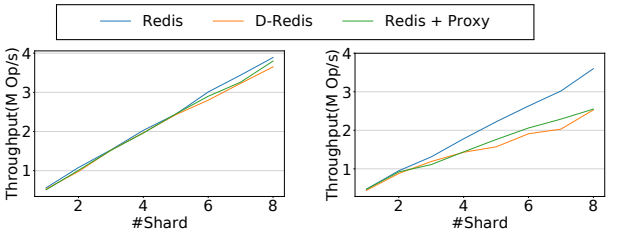


Figure 16: Impact of Recovery on Throughput

Figure 15 shows the throughput of operations with co-located D-FASTER running the same benchmark as the previous section, varying the percentage of requests that are remote. We pinned one client thread to every vCPU enabled for the benchmark, and only handled remote requests using spare cycles. Figure 15 shows that co-located D-FASTER achieved the same level, or even higher performance as dedicated servers when most requests were co-located. Performance declined to less than half of what a dedicated server achieved for a higher remote percentage. This is for two reasons: 1) the remote code-path is much more expensive than local execution, and co-located clients took away CPU resources that could have otherwise been used by servers on high remote percentage runs, and 2) co-located clients took up network resources to send requests, which created contention with request serving. The advantage of co-located execution is more pronounced for applications with limited batching within a session. In Figure 15, we also show the performance of D-FASTER while varying batch sizes. We plot throughput in log scale due to the large shifts in performance across configurations. Figure 15 shows that D-FASTER achieved significantly higher throughput when executing a higher percentage of operations locally, as local operations were not impacted by the reduction in parallelism and batching. This makes D-FASTER ideal for such applications if there are opportunities for co-location.

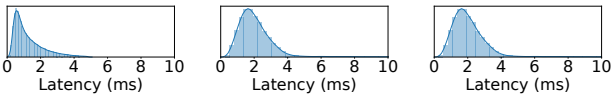
7.4 Failure and Recovery

We now show the performance of D-FASTER with failures. We used the same workload and setup as earlier remote client experiments (Section 7.2). Instead of introducing a crash, we simulated a worker failure by notifying workers of a new world-line, forcing all workers to rollback to the latest DPR cut. We plot the throughput of operation commit and completion in Figure 16 against time. We ran the system for 45 seconds, collecting throughput every 250 ms, and introduced 3 total failures – one at the 15-second mark, and the other two in short succession at the 30-second mark, such that the third failure happened while the system is recovering from the second failure. We truncate the first and last 250 ms of data and



(a) Saturated (w=8192, b=1024) **(b)** Unsaturated (w=1024, b=16)

Figure 17: Throughput Comparison of D-Redis vs. Redis



(a) Redis **(b)** D-Redis **(c)** Redis + Proxy

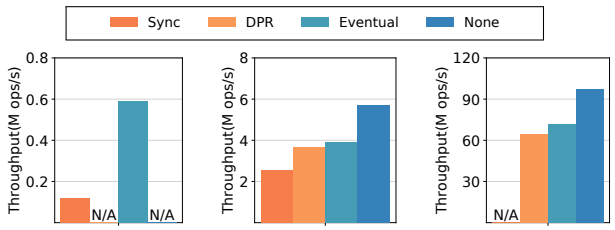
Figure 18: Latency Distribution of D-Redis vs. Redis

only show the system running in a stable state. The results show that D-FASTER recovered quickly from failures in less than 200 ms. During failure, commit progress was temporarily halted as D-FASTER detects that recovery is in progress, and some operations were lost in the rollback at 15s. Operation throughput saw only a minor drop, as clients paused operations to compute their recovery state and workers delayed requests from post-recovery clients. For the failure scenario at 30s, D-FASTER treated the nested failures simply as two failure-and-recovery sequences. The second failure and recovery did not add noticeable overhead in overall recovery time, as very few operations succeeded in between the two failures. Due to the additional coordination overhead between workers, however, operation throughput had a larger dip. Thus, fewer operations aborted as fewer operations were executed during the recovery window. Overall, the experiment shows that our recovery scheme is efficient, capable of resuming normal operation in 100s of ms even in the face of repeated failures.

7.5 D-Redis: Unmodified Redis with DPR

We now evaluate the performance of D-Redis in comparison to Redis. We deployed D-Redis on the same cluster as D-FASTER, with 8 servers each running a Redis instance and a D-Redis proxy, and executed the same YCSB-A (50:50) workload as before. We ran the benchmark for 5 minutes with one checkpoint (commit for DPR). We ran three configurations for each experiment: 1) Redis, 2) Redis through a proxy that forwards every packet, to control for the change in networking pattern, and 3) D-Redis.

We show the results in Figure 17. We first saturated Redis with large w and b and observe that D-Redis did not reduce the throughput or scalability of Redis while adding prefix recoverability. To test the impact of D-Redis on latency, we ran the same benchmark with much smaller w and b, such that the baseline Redis performance was slightly below maximum throughput. In this scenario, D-Redis was still scalable but incurred around 30% higher latency than the Redis baseline; see Figure 18. Recall that unlike vanilla Redis, D-Redis clients first talk to a proxy, which forwards requests to a local Redis instance. To understand the source of increased latency, we repeated the experiment with a pass-through proxy



(a) Cassandra **(b)** D-Redis **(c)** D-FASTER

Figure 19: Throughput Impact of Recoverability Guarantees

without DPR. As shown, D-Redis does no worse than this alternative. This suggests that the new network pattern (via proxy) is the dominant factor for the slowdown, rather than the DPR algorithm itself. Overall, while DPR via proxy can be a quick and effective addition to existing cache-stores to add prefix recoverability, an integrated solution such as D-FASTER can avoid the extra proxy latency with additional developer effort.

7.6 Performance vs. Recoverability

Lastly, to better understand the tradeoffs between performance and recoverability properties, we evaluate on different systems 4 types of common recoverability guarantees:

(1) None – not recoverable in failure
(2) Eventual – operation returns immediately with background persistence (e.g., returning before `fsync()` completes)
(3) DPR – operation returns with asynchronous guarantees
(4) Synchronous – operation returns only after persistence

We ran 3 systems (D-FASTER, D-Redis, and Apache Cassandra [11]) each supporting some of the listed guarantees, to compare their throughput. We used the same uniform YCSB-A (50:50) benchmark on an 8-node cluster. For D-FASTER, we emulate eventual recoverability by turning off DPR. For D-Redis, we turn on write-ahead-logging in Redis to guarantee synchronous recoverability [12], and turn off DPR for eventual recoverability. For Cassandra, we disable replication and set commit log option to `periodic` for eventual recoverability, and `group` for synchronous recoverability. Note that not every system supports every recoverability level.

We show the results in Figure 19. We mark a configuration with `N/A` if the system in question does not support it. Note the different Y axes; performance across different systems is not directly comparable due to differences in our benchmarking setup and performance goals of respective systems. We observe that in both D-Redis and D-FASTER, DPR provides performance similar to that of eventual recoverability, despite providing prefix guarantees. In comparison, synchronous recoverability schemes have a much larger impact on system throughput. This trend can be seen in all three systems, despite the different order of magnitude of throughput they operate at. This shows that DPR is a general and viable option to provide prefix recoverability while maintaining high performance.

8 RELATED WORK

Single node storage systems. Main-memory resident database systems have long employed asynchronous recoverability mechanisms to avoid bottlenecking on storage latency. Early lock release allows transactions to release locks before persisting their commit records [27, 30]. Aether [35] and ERMIA [37] extended early lock release for better scalability on modern multi-core hardware. Recently proposed epoch-based recoverability avoids the synchronization bottleneck of a centralized log by loosely coordinating threads using epochs [21, 44, 49, 54]. Amazon Aurora [50] and Orleans [28], while distributed, funnel transactions through a centralized node to guarantee recoverability. Unlike this prior work, D-FASTER's asynchronous recoverability is fully distributed and thus needs to additionally handle cross-shard consistency and partial system failures (in the form of individual node failures).

Distributed storage systems. Strongly consistent distributed storage systems have typically used synchronous recoverability mechanisms; via synchronous replication [20, 22], two-phase commit [41], or both [25, 32]. Consequently, single-key write throughput is limited by the protocol latency. To avoid this bottleneck, prior work on deterministic databases, such as Calvin [48], removes recoverability logic from the requests' critical path by replicating transaction commands in a log before execution and deterministically executing the log. DPR instead returns client operations once they complete (rather than commit) and guarantees recoverability asynchronously.

Caching and fault-tolerance. Much prior work has proposed using session-based consistency guarantees to increase the scalability of storage systems [26, 40, 44], which DPR builds on. TxCache [43] serves transactionally consistent but stale state to applications and synchronously writes transactions through to storage. ChronoCache [31] similarly uses session-based guarantees while predictively executing queries to speed up reads further. D-FASTER instead provides linearizable consistency for client operations and speeds up writes with asynchronous recoverability. LineageStash [51] provides asynchronous recoverability for task-parallel systems by tracking non-durable lineage metadata of each task In D-FASTER, this approach would effectively need to forward keys and values, which is impractically more state to forward than task lineage metadata. Cloudburst [46] circumvents the IO latency bottleneck in serverless storage offerings via weakly consistent caching: reads can observe stale data and writes to the same key are asynchronously merged . Effectively, Cloudburst's weakening of consistency removes conflicts between requests that would otherwise need to block to guarantee recoverability. D-FASTER instead maintains linearizable consistency and focuses on recoverability.

9 CONCLUSION

Accessing and updating shared distributed data in a consistent yet speedy manner, in the presence of failures, is a challenging and important problem in today's cloud and serverless setting. Current solutions using fast caches over durable stores are insufficient. Our solution is DPR, a distributed protocol for such "cache-store" architectures that decouples operation completion from operation commit and provides strong consistency and asynchronous recovery guarantees to applications. We build D-FASTER and D-Redis to show its ability to deliver performance with durability guarantees.

Acknowledgments. We are grateful for the support of the MIT DSAIL@CSAIL member companies. We also thank Phil Bernstein and the anonymous reviewers for their comments and suggestions.

REFERENCES

[1] Amazon DynamoDB. https://aws.amazon.com/dynamodb/, 2020.
[2] Amazon S3. https://aws.amazon.com/s3/, 2020.
[3] Azure Disk Storage oerview. https://docs.microsoft.com/en-us/azure/virtual-machines/managed-disks-overview, 2020.
[4] Azure Service Fabric. https://azure.microsoft.com/en-us/services/service-fabric/, 2020.
[5] Azure SQL Database serverless. https://docs.microsoft.com/en-us/azure/azure-sql/database/serverless-tier-overview, 2020.
[6] Azure Storage - Secure cloud storage. https://azure.microsoft.com/en-us/services/storage/, 2020.
[7] Create a Windows VM with accelerated networking using Azure PowerShell. https://docs.microsoft.com/en-us/azure/virtual-network/create-vm-accelerated-networking-powershell, 2020.
[8] How do I make my Lambda function idempotent to prevent inconsistencies and data loss in my application? https://aws.amazon.com/premiumsupport/knowledge-center/lambda-function-idempotent/, 2020.
[9] Kubernetes. https://kubernetes.io/, 2020.
[10] Memory optimized Dv2 and Dsv2-series. https://docs.microsoft.com/en-us/azure/virtual-machines/dv2-dsv2-series-memory, 2020.
[11] Apache Cassandra. https://cassandra.apache.org/, 2021.
[12] Redis Persistence. https://redis.io/topics/persistence, 2021.
[13] YCSB. https://github.com/brianfrankcooper/YCSB, 2021.
[14] Azure Durable Functions. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview, retrieved 12-Feb-2020.
[15] Temporal Workflows. https://temporal.io/, retrieved 12-Feb-2020.
[16] Kafka. https://kafka.apache.org/, retrieved 12-Feb-2021.
[17] Amazon Lambda. https://aws.amazon.com/lambda/, retrieved 19-Sep-2020.
[18] Azure Functions. https://azure.microsoft.com/en-us/services/functions/, retrieved 19-Sep-2020.
[19] Redis. https://redis.io/, retrieved 19-Sep-2020.
[20] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, 2011.
[21] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18. ACM, 2018.
[22] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2), 2008.
[23] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *SoCC*, 2010.
[24] M. Copeland, J. Soh, A. Puca, M. Manning, and D. Gollob. *Microsoft Azure: Planning, Deploying, and Managing Your Data Center in the Cloud*. Apress, USA, 1st edition, 2015.
[25] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *OSDI*, 2012.
[26] K. Daudjee and K. Salem. Lazy database replication with snapshot isolation. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB '06, page 715–726. VLDB Endowment, 2006.
[27] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. In *SIGMOD*, 1984.
[28] T. Eldeeb and P. A. Bernstein. Transactions for distributed actors in the cloud. 2016.
[29] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, Sept. 2002.
[30] D. Gawlick and D. Kinkade. Varieties of concurrency control in IMS/VS fast path. *DE Bull*, 8(2), 1985.
[31] B. Glasbergen, K. Langendoen, M. Abebe, and K. Daudjee. Chronocache: Predictive and adaptive mid-tier query result caching. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2391–2406, New York, NY, USA, 2020. Association for Computing Machinery.
[32] J. Gray and L. Lamport. Consensus on transaction commit. *TODS*, 31(1), 2006.
[33] J. . Helary, A. Mostefaoui, R. H. B. Netzer, and M. Raynal. Preventing useless checkpoints in distributed computations. In *Proceedings of SRDS'97: 16th IEEE Symposium on Reliable Distributed Systems*, pages 183–190, 1997.
[34] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *TOPLAS*, 12(3), 1990.
[35] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Aether: a scalable approach to logging. *PVLDB*, 3(1-2), 2010.
[36] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson. Cloud programming simplified: A berkeley view on serverless computing, 2019.
[37] K. Kim, T. Wang, R. Johnson, and I. Pandis. Ermia: Fast memory-optimized database system for heterogeneous workloads. In *SIGMOD*, 2016.
[38] C. Kulkarni, B. Chandramouli, and R. Stutsman. Achieving High Throughput and Elasticity in a Larger-than-Memory Store. *Proc. VLDB Endow.*, 14(8), 2021. https://arxiv.org/abs/2006.03206.
[39] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.
[40] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 401–416, New York, NY, USA, 2011. Association for Computing Machinery.
[41] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the r* distributed database management system. *TODS*, 11(4), 1986.
[42] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *PODC*, 1988.
[43] D. R. Ports, A. T. Clements, I. Zhang, S. Madden, and B. Liskov. Transactional consistency and automatic management in an application data cache. In *OSDI*, 2010.
[44] G. Prasaad, B. Chandramouli, and D. Kossmann. Concurrent Prefix Recovery: Performing CPR on a Database. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 687–704, New York, NY, USA, 2019. ACM.
[45] V. Sreekanti, C. Wu, S. Chhatrapati, J. E. Gonzalez, J. M. Hellerstein, and J. M. Faleiro. A fault-tolerance shim for serverless computing. In *EuroSys*, 2020.
[46] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. M. Faleiro, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov. Cloudburst: Stateful functions-as-a-service, 2020.
[47] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, Aug. 1985.
[48] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD*, 2012.
[49] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, 2013.
[50] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. Amazon Aurora: Design considerations for high throughput cloud-native relational databases. In *SIGMOD*, 2017.
[51] S. Wang, J. Liagouris, R. Nishihara, P. Moritz, U. Misra, A. Tumanov, and I. Stoica. Lineage stash: fault tolerance off the critical path. In *SOSP*, 2019.
[52] C. Wu, J. Faleiro, Y. Lin, and J. Hellerstein. Anna: A KVS for Any Scale. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 401–412, 2018.
[53] Yi-Min Wang. Consistent global checkpoints that contain a given set of local checkpoints. *IEEE Transactions on Computers*, 46(4):456–468, 1997.
[54] W. Zheng, S. Tu, E. Kohler, and B. Liskov. Fast databases with fast durability and recovery through multicore parallelism. In *OSDI*, 2014.