Transactional Causal Consistency for Serverless Computing

Chenggang Wu
UC Berkeley
cgwu@berkeley.edu

Vikram Sreekanti
UC Berkeley
vikrams@berkeley.edu

Joseph M. Hellerstein
UC Berkeley
hellerstein@berkeley.edu

ABSTRACT

We consider the setting of serverless Function-as-a-Service (FaaS) platforms, where storage services are disaggregated from the machines that support function execution. FaaS applications consist of compositions of functions, each of which may run on a separate machine and access remote storage.

The challenge we address is improving I/O latency in this setting while also providing application-wide consistency. Previous work has explored providing causal consistency for individual I/Os by carefully managing the versions stored in a client-side data cache. In our setting, a single application may execute multiple functions across different nodes, and therefore issue interrelated I/Os to multiple distinct caches. This raises the challenge of *Multisite Transactional Causal Consistency* (MTCC): the ability to provide causal consistency for all I/Os within a given transaction even if it runs across multiple physical sites. We present protocols for MTCC implemented in a system called HydroCache. Our evaluation demonstrates orders-of-magnitude performance improvements due to caching, while also protecting against consistency anomalies that otherwise arise frequently.

ACM Reference Format:
Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. 2020. Transactional Causal Consistency for Serverless Computing. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3318464.3389710

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3389710

1 INTRODUCTION

Serverless computing has gained significant attention recently, with a focus on Function-as-a-Service (FaaS) systems [2, 13–15, 23, 28]. These systems—e.g., AWS Lambda, Google Cloud Functions—allow programmers to upload arbitrary functions and execute them in the cloud without having to provision or maintain servers.

These platforms enable developers to construct applications as compositions of multiple functions [17]. For example, a social network generating a news feed might have three functions: authenticate a user, load posts in that user's timeline, and generate an HTML page. Functions in the workflow are executed independently, and different functions may not run on the same physical machine due to load balancing, fault tolerance, and varying resource requirements.

For developers, the key benefit of FaaS is that it transparently autoscales in response to workload shifts; more resources are provisioned when there is a burst in the request rate, and resources are de-allocated as the request rate drops. As a result, cloud providers can offer developers attractive consumption-based pricing. Providers also benefit from improved resource utilization, which comes from dynamically packing the current workload into servers.

FaaS platforms achieve flexible autoscaling by *disaggregating* the compute and storage layers, so they can scale independently. For example, FaaS applications built on AWS Lambda typically use AWS S3 or DynamoDB as the autoscaling storage layer [4]. This design, however, comes at the cost of high-latency I/O—often orders of magnitude higher than attached storage [13]. This makes FaaS ill-suited for low-latency services that would naturally benefit from autoscaling—e.g. webservers managing user sessions, discussion forums managing threads, or ad servers managing ML models. These services all dynamically manipulate data based on request parameters and are therefore sensitive to I/O latency.

A natural solution is to attach caches to FaaS compute nodes to eliminate the I/O latency for data that is frequently accessed from remote storage. However, this raises challenges around maintaining consistency of the cached data—particularly in the context of multi-I/O applications that may run across different physical machines with different caches.

Returning to the social network setting, consider a scenario where Alice updates her photo access permissions to

ban Bob from viewing her pictures and then posts a picture that makes fun of him. When Bob views his timeline, the cache his request visits for Alice's permissions may not yet reflect her recent update, but the cache visited to fetch posts may include the picture that she posted. The authentication process mistakenly will allow Bob to view the picture, creating a consistency anomaly [8, 19, 24].

The source of this inconsistency is that reads and writes fail to respect *causality*: Bob first observes a stale set of permissions, then observes a photo whose write was influenced by a newer permission set. Such anomalies can be prevented by transactional causal consistency (TCC), the strongest consistency model that can be achieved without expensive consensus protocols [5, 19, 21] required in stricter consistency models such as serializable transactions.

However, providing TCC for low-latency applications creates unprecedented challenges in a serverless environment, where cluster membership rapidly changes over time due to autoscaling infrastructure and user requests span multiple compute nodes. Recent systems such as Cure [1] and Occult [22] enforce TCC at the storage layer, so FaaS-layer caches would need to access storage for each causally consistent I/O, which reintroduces network roundtrips and violates our low-latency goal. Moreover, the consistency mechanisms in prior work rely on fixed node membership, which we cannot assume of an autoscaling system.

An alternative approach is Bolt-on Causal Consistency [7] (BCC). BCC enforces consistency in a cache layer similar to the one proposed here and does not rely on fixed-size clusters. However, BCC only guarantees Causal+ Consistency [19], which is weaker than TCC and inadequate to prevent the anomaly described above. BCC also does not guarantee consistency across multiple caches.

To solve these challenges, we present HydroCache, a distributed caching layer attached to each node in a FaaS system. HydroCache simultaneously provides low-latency data access and introduces *multisite transactional causal consistency* (MTCC) protocols to guarantee TCC for requests that execute on multiple nodes. Our MTCC protocols do not rely on the membership of the system, and HydroCache does not interfere with a FaaS layer's crucial autoscaling capabilities. In summary, this paper's contributions are:

(1) The design of HydroCache, which provides low latencies while also guaranteeing TCC for individual functions executed at a single node (Section 3).
(2) Efficient MTCC protocols to guarantee TCC for compositions of functions, whose execution spans multiple nodes (Section 4).
(3) An evaluation that shows TCC offers an attractive trade-off between performance and consistency in a serverless setting and HydroCache achieves a 10× performance improvement over FaaS architectures without a caching layer while simultaneously offering stronger consistency (Section 5).

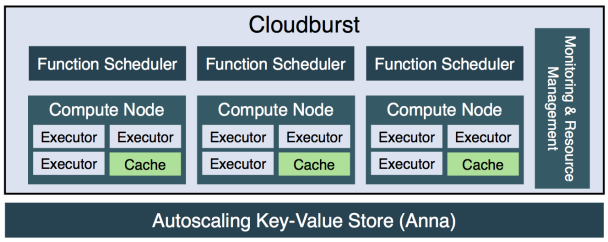

Figure 1: Cloudburst architecture.

2 BACKGROUND

In this section, we briefly introduce the system architecture within which we implement HydroCache. We also define causal consistency and its extensions, which are essential for understanding the material in the rest of the paper.

2.1 System Architecture

Figure 1 shows an overview of our system architecture, which consists of a high-performance key-value store (KVS) Anna [31, 32] and a function execution layer Cloudburst [30]. We chose Anna as the storage engine as it offers low latencies and flexible autoscaling, a good fit for serverless. Anna also supports custom conflict resolution policies to resolve concurrent updates. As we discuss in Section 2.2, this provides a necessary foundation to support causal consistency.

Cloudburst deploys KVS-aware caches on the same nodes as the compute workers, allowing for low-latency data access. We build our own function execution layer as there is no way to integrate our cache into existing FaaS systems.

In Cloudburst, all requests are received by a scheduler and routed to worker threads based on compute utilization and data locality heuristics. Each compute node has three function executor threads, each of which has a unique ID. As we discuss in Section 2.2, these IDs are used to capture causal relationships. The compute threads on a single machine interact with one HydroCache instance, which retrieves data for the function executors as necessary. The cache also transparently writes updates back to the KVS.

Cloudburst users write functions in vanilla Python, and register them with the system for execution. The system enables low-latency function chaining by allowing users to register function compositions forming a *DAG* of functions. DAG execution is optimized by automatically passing results from one function executor to the next. Each DAG has a single *sink* function with no downstream functions, the results of which are either returned to the user or written to Anna.

2.2 Causal Consistency

Causal Consistency (CC). Under CC, reads and writes respect Lamport's "happens-before" relation [18]. If a read of key a_i (i denotes a version of key a) influences a write of key b_j, then a_i *happens before* b_j, or b_j depends on a_i; we denote this as $a_i \rightarrow b_j$. *happens before* is transitive: if $a_i \rightarrow b_j \wedge b_j \rightarrow c_k$, then $a_i \rightarrow c_k$. In our system, dependencies are explicitly generated during function execution (known as explicit causality [6, 16]). A write causally depends on keys that the function previously read from storage.

A key k_i has four components $[k, VC_{k_i}, deps, payload]$; k is the key's identifier, VC_{k_i} is a *vector clock* [26, 29] that identifies its version, *deps* is its dependency set, and *payload* is the value. VC_{k_i} consists of a set of $\langle id, clock\rangle$ pairs where the *id* is the ID of a function executor thread that updated k_i, and the *clock* represents that thread's monotonically growing logical clock. *deps* is a set of $\langle dep\_key, VC\rangle$ pairs representing key versions that k_i depends on.

During writes, the VC and dependency set are modified as follows. Let thread e_1 write a_i. Thread e_2 then writes b_j with $a_i \rightarrow b_j$. a_i will have $VC_{a_i} = \langle\langle e_1, 1\rangle\rangle$ and an empty dependency set, and b_j will have $VC_{b_j} = \langle\langle e_2, 2\rangle\rangle$ and a dependency set $\langle\langle a_i, VC_{a_i}\rangle\rangle$. If another thread e_3 writes b_k such that $b_j \rightarrow b_k$, then b_k will have $VC_{b_k} = \langle\langle e_2, 2\rangle, \langle e_3, 3\rangle\rangle$ and a dependency set $\langle\langle a_i, VC_{a_i}\rangle\rangle$. In this example, 1, 2 and 3 are the values of logical clocks of e_1, e_2 and e_3 during the writes. Dependencies between versions of the same key are captured in the key's VC, and dependencies across keys are captured in the dependency set.

Given a_i and a_j, $a_i \rightarrow a_j \iff VC_{a_i} \rightarrow VC_{a_j}$. Let E be a set that contains all executor threads in our system. We define $VC_i \rightarrow VC_j$ as $\forall e \in E \mid e \notin VC_i \vee VC_i(e) \leq VC_j(e)$ and $\exists e' \in E \mid (e' \notin VC_i \wedge e' \in VC_j) \vee (VC_i(e') < VC_j(e'))$. In other words, VC_j "dominates" VC_i if and only if all $\langle id, clock\rangle$ pairs of VC_j are no less than the matching pairs in VC_i and at least one of them dominates. If $a_i \nrightarrow a_j \wedge a_j \nrightarrow a_i$, then a_i is *concurrent* with a_j, denoted as $a_i \sim a_j$.

CC requires that if a function reads b_j, which depends on a_i ($a_i \rightarrow b_j$), then the function can subsequently only read $a_k \mid a_k \nrightarrow a_i$—i.e. $a_k == a_i$, $a_i \rightarrow a_k$, or $a_k \sim a_i$.

Causal+ Consistency (CC+) [19] is an extension to CC that—in addition to guaranteeing causality—ensures that replicas of the same key eventually converge to the same value. To ensure converence, we register a conflict resolution policy in Anna by implementing the following definitions:

Definition 1 (Concurrent Version Merge). *Given two concurrent versions a_i and a_j, let a_k be a merge of a_i and a_j (denoted as $a_k = a_i \cup a_j$). Then $VC_{a_k} = VC_{a_i} \cup VC_{a_j} = \langle\langle e, c\rangle | \langle e, c_i\rangle \in VC_{a_i} \wedge \langle e, c_j\rangle \in VC_{a_j} \wedge c = \max(c_i, c_j)\rangle\rangle$.*

The merged VC is the key-wise maximum of the input VCs. Note that the above definition has a slight abuse of notation: e might not exist in one of the two VCs. If $\langle e, c_i\rangle \notin VC_{a_i}$, we simply set $\langle e, c\rangle = \langle e, c_j\rangle \in VC_{a_j}$ and vice versa.

In addition to merging the VCs, we also merge the dependency sets using the same mechanism above. Finally, we merge the payloads by taking a set union ($a_k.payload = \langle a_i.payload, a_j.payload\rangle$). When a function requests a key that has a set of payloads, applications can specify which payload to return; by default we return the first element in the set to avoid type error. If a_i and a_j are not concurrent (say $a_i \rightarrow a_j$), then a_j overwrites a_i during the merge.

Transactional Causal Consistency (TCC) [1, 22] is a further extension of CC+ that guarantees the consistency of reads and writes for a *set* of keys. Specifically, given a read set R, TCC requires that R forms a *causal snapshot.*

Definition 2. *R is a causal snapshot $\iff \forall(a_i, b_j) \in R, \nexists a_k \mid a_i \rightarrow a_k \wedge a_k \rightarrow b_j$.*

That is, for any pair of keys a_i, b_j in R, if a_k is a dependency of b_j, then a_i is not allowed to happen before a_k; it can be equal to a_k, happen after a_k, or be concurrent with a_k. Note that TCC is stronger than CC+ because issuing a sequence of reads to a data store that guarantees CC+ does not ensure that the keys read are from the same causal snapshot.

In the social network example, the application explicitly specifies that Alice's photo update depends on her permission update. TCC then ensures that the system only reveals the *new* permission and the funny picture to the application, which rejects Bob's request to view the picture.

TCC also ensures *atomic visibility* of written keys; either all writes from a transaction are seen or none are. In our context, if a DAG writes a_i and b_j and another DAG reads a_i and b_k, TCC requires $b_k == b_j \vee b_j \rightarrow b_k$.

3 HYDROCACHE

In this section, we introduce the design of HydroCache and discuss how it achieves TCC for individual functions executed at a single node. To achieve both causal snapshot reads and atomic visibility, each cache maintains a single *strict causal cut C* (abbreviated as *cut*) which we define below.

Definition 3. *C is a cut $\iff \forall k_i \in C$,*
$\forall d_j \in get\_tuple(k_i.deps), \exists d_k \in C \mid d_k == d_j \vee d_j \rightarrow d_k$.

A cut requires that for *any* dependency, d_j, of any key in C, there is a $d_k \in C$ such that either the two versions are equal or d_k happens after d_j. We formally define this notion:

Definition 4. *Given two versions of the same key k_i and k_j, we say that k_i supersedes k_j ($supersede(k_i, k_j)$) when $k_i == k_j \vee k_j \rightarrow k_i$. Similarly, given two sets of key versions T and S, let K be a set of keys that appear in both T and S. We say*

that T supersedes S if $\forall k \in K$, *let* $k_i \in T$ *and* $k_j \in S$, *we have* $supersede(k_i, k_j)$.

Note that a cut differs from a causal snapshot in two ways. First, cuts are closed under dependency: If a key is in the cut, so are its dependencies. Second, the *happens-before* constraint in a cut is more stringent than in a causal snapshot: The key d_k must be equal-to or happen after every dependency d_j associated with other keys in the cut—concurrency is disallowed. We will see why this is important in Section 3.2.

Updating the Local Cut. HydroCache initially contains no data, so it trivially forms a cut, C. When a function requests a key b that is missing from C, the cache fetches a version b_j from Anna. Before exposing it to the functions, the cache checks to see if all of b_j's dependencies are superseded by keys already in C. If a dependency a_i is not superseded, the cache fetches versions of a, a_m, from Anna until a_i is superseded by a_m. HydroCache then recursively ensures that all dependencies of a_m are superseded. This process repeats until the dependencies of all new keys are superseded. At this point, the cache updates C by merging the new keys with keys in C and exposes them to the functions. If a cache runs out of memory during the merge of requested keys, the requesting function is rescheduled on another node.

The cache "subscribes" to its cached keys with Anna, and Anna periodically pushes new versions to "refresh" the cache. New data is merged into C following the same process as above. When evicting key k, all keys depending on k are also evicted to preserve the cut invariant.

In the rest of this section, we first discuss how HydroCache guarantees TCC at a single node—providing a causal snapshot for the read set (Section 3.1) and atomic visibility for the write set (Section 3.2). We then discuss garbage collection and fault tolerance in Section 3.3.

3.1 Causal Snapshot Reads

From Definition 3, we know that for any pair of keys a_i, b_j in a cut C, if $a_k \rightarrow b_j$, then a_i supersedes a_k—either $a_k == a_i \vee a_k \rightarrow a_i$. This is stronger than the definition of a causal snapshot (Definition 2), as that definition permits a_k to be concurrent with a_i. Since each function reads from the cut in its local cache, the read set trivially forms a causal snapshot.

As we show below, this stricter form of causal snapshot (disallowing a_k and a_i to be concurrent) also ensures atomic visibility. Therefore, from now on, we consider this type of causal snapshot and abbreviate it as *snapshot*.

3.2 Atomic Visibility

Say a function writes two keys, a_i and b_j; in order to make them atomically visible, HydroCache makes them mutually dependent—$a_i \rightarrow b_j$ and $b_j \rightarrow a_i$. If another function reads a snapshot that contains a_i and b_k, since $b_j \rightarrow a_i$, the snapshot ensures that $b_k == b_j \vee b_j \rightarrow b_k$, satisfying atomic visibility. When executing a DAG, in order to ensure that writes across functions are mutually dependent, all writes are performed at the end of the DAG at the sink function.

There is, however, a subtle issue. Recall that VCs consist of the IDs of threads that modify a key along with those threads' logical clocks. All functions performing writes through an executor thread *share* the same ID. This introduces a new challenge: Consider two functions F and G both using executor e to write key versions a_i and a_j. When e writes these two versions—say a_i first, then a_j—a_i may be overwritten since e attaches a larger logical clock to $VC(e)$ of a_j. However, a_i and a_j may in fact be logically concurrent ($a_i \sim a_j$) since F may not have observed a_j before writing a_i and vice versa. This violates atomic visibility: If a function writes a_i and b_k, a_i can be overwritten by a concurrent version from the same thread. For a later read of a and b, b_k is visible but a_i is lost.

To prevent this from happening, each executor thread keeps the latest version of keys it has written. When a function writes a_i at executor e, e inspects its dependency set to see whether this write depends on the most recent write of the same key, a_{latest}, performed by e. If so, e advances its logical clock, updates $VC_{a_i}(e)$, writes to Anna, and updates a_{latest} to a_i. If not, then $a_i \sim a_{latest}$. This is because since a_{latest} is not in a_i's dependency set, $a_{latest} \nrightarrow a_i$, and since e wrote a_{latest} before a_i, $a_i \nrightarrow a_{latest}$. Since the versions are not equal, we have $a_i \sim a_{latest}$. In this case, e first merges a_i and a_{latest} following Definition 1 to produce a_k, advances its logical clock and updates $VC_{a_k}(e)$, writes to Anna, and sets a_{latest} to a_k. Doing so prevents each executor from overwriting keys with a potentially concurrent version.

3.3 Discussion

Dependency Metadata Garbage Collection. Causal dependencies accumulate over time. For a key $b_j \mid a_i \rightarrow b_j$, we can safely garbage collect $\langle a_i, VC_{a_i} \rangle \in b_j.deps$ if all replicas of a supersede a_i. We run a background consensus protocol to periodically clear this metadata.

Fault Tolerance. When writes to Anna fail due to storage node failures or network delay, they are retried with the same key version, guaranteeing idempotence. Function and DAG executions are *at least once*. Heartbeats are used to detect node failures in the compute layer, and unfinished functions and DAGs at a failed node are re-scheduled at other nodes.

4 MTCC PROTOCOLS

Although the design introduced in Section 3 guarantees TCC for individual functions executed at a single node, this is insufficient for serverless applications. Recall that a DAG in Cloudburst consists of multiple functions, each of which can be executed at a different node. To achieve TCC for the DAG,

we must ensure that a read set spanning *multiple* physical sites forms a distributed snapshot. A naïve approach is to have all caches coordinate and maintain a large distributed cut across all cached keys at all times. This is infeasible in a serverless environment due to the enormous traffic that protocol would generate amongst thousands of nodes.

In this section, we discuss a set of MTCC protocols we developed to address this challenge while minimizing coordination and data shipping overheads across caches. The key insight is that rather than eagerly constructing a distributed cut, caches collaborate to create a snapshot of each DAG's read set during execution. This leads to significant savings for two reasons. First, snapshots are constructed per-DAG; the communication to form these snapshots is combined with that of regular DAG execution, without any global coordination. Second, snapshots are restricted to holding the keys read by the DAG, whereas a cut must include all keys' transitive dependencies.

We start with the *centralized* (CT) protocol in which all functions in a DAG are executed at a single node. We then introduce three protocols—*optimistic* (OPT), *conservative* (CON), and *hybrid* (HB)—that allow functions to be executed across different nodes. Throughout the rest of this section, we assume the read set of each function is known, but we return to cases in which the read set is unknown in Section 4.6.

4.1 Centralized (CT)

Under CT, all functions in a DAG are executed at a single node, accessing a single cache. This significantly simplifies the challenge of providing TCC, as we do not need to worry about whether reads from cuts on different nodes form a snapshot. Before execution, the cache creates a snapshot from the local cut that contains the read set of all functions in the DAG. All reads are serviced from the created snapshot to ensure that they observe the same cut, regardless of whether the cut is updated while the DAG is executing.

The main advantage of CT is its simplicity: Since all functions are executed on the same node, there is no network cost for passing results across nodes. However, CT suffers from some key limitations. First, it constrains scheduling: Scheduling happens at the DAG level instead of at the level of individual functions. The distributed nature of our scheduler sometimes leads to load imbalances, as schedulers do not have a global view of resource availability. When the scheduling granularity becomes coarse (from function to DAG), the performance penalty due to load imbalance will be amplified. Second, CT requires the data requested by all functions to be co-located at a single cache. The overheads of fetching data from remote storage and constructing the cut can be significant if there are many cache misses or if the read set is large. Finally, CT limits the amount of parallelism in a DAG to the number of executor threads on a single node. In Section 5.2, we evaluate these limitations and quantify the trade-off between CT and our distributed protocols.

4.2 Towards Distributed Snapshots

The goal of our distributed protocols below is to ensure that each DAG observes a snapshot as computation moves across nodes. This requires care, as reading arbitrary versions from the various local cuts may not correctly create a snapshot.

4.2.1 Theorems. Before describing our protocols, we present simple theorems and proofs that allow us to combine data from each node to ensure the distributed snapshot property.

Definition 5 (Keysets and Versionsets). *A keyset $\widetilde{R}$ is a set of keys without specified versions. A versionset R is a* binding *that maps each $k \in \widetilde{R}$ to a specific version k_i.*

In subsequent discussion, we notate keysets with a tilde above. As a mild abuse of notation, we will refer to the intersection of a keyset $\widetilde{K}$ with a versionset V; this is the maximal subset of V whose keys are found in the keyset $\widetilde{K}$. Each element in the versionset only contains the key identifier and its VC; dependency and payload information are not included.

Definition 6 (Keyset-Overlapping Cut). *Given a versionset V and a keyset $\widetilde{K}$, we say that V is a keyset-overlapping cut for $\widetilde{K}$ when $\forall k_i \in V$, we have $k \in \widetilde{K} \wedge \forall d_j \rightarrow k_i$, if $d \in \widetilde{K}$, then $\exists d_{js} \in V \mid supersede(d_{js}, d_j)$.*

In essence, a keyset-overlapping cut for $\widetilde{K}$ is similar to a cut with the relaxation that it only consists of keys and dependencies that overlap with $\widetilde{K}$.

Lemma 1. *Given a keyset $\widetilde{K}$ and a cut C, $S = C \cap \widetilde{K}$ is a keyset-overlapping cut for $\widetilde{K}$.*

Proof. This follows directly from Definition 6. S is the intersection of the versionset C and $\widetilde{K}$; the fact that C is a cut ensures that the conditions of Definition 6 hold. □

Definition 7 (Versionset Union). *Given two versionsets V_1 and V_2, their union $V_3 = V_1 \cup V_2$ includes all keys k_m such that:*

$$k_m = \begin{cases} k_i \in V_1 & \nexists k_j \in V_2 \\ k_i \in V_2 & \nexists k_j \in V_1 \\ k_i \cup k_j & \exists k_i \in V_1 \wedge \exists k_j \in V_2 \end{cases}$$

We show keyset-overlapping cuts are closed under union:

Theorem 1 (Closure Under Union). *Given keyset $\widetilde{K}$, let $S_1 = C_1 \cap \widetilde{K}, S_2 = C_2 \cap \widetilde{K}$ be keyset-overlapping cuts for $\widetilde{K}$. Then $S_3 = S_1 \cup S_2$ is a keyset-overlapping cut for $\widetilde{K}$.*

Proof. Let $k_3 \in S_3$ and $d_3 \rightarrow k_3$. Since $S_3 = S_1 \cup S_2$, we know $k_3 = k_1 \cup k_2$ where $k_1 \in S_1$, $k_2 \in S_2$, and $d_3 = d_1 \cup d_2$ where $d_1 \rightarrow k_1$, $d_2 \rightarrow k_2$.

If $d \in \widetilde{K}$, according to Definition 6, $\exists d_{1s} \in S_1 \mid supersede(d_{1s}, d_1)$ and $\exists d_{2s} \in S_2 \mid supersede(d_{2s}, d_2)$. It follows that $supersede(d_{1s} \cup d_{2s}, d_1 \cup d_2)$, where $d_1 \cup d_2 = d_3$ and $d_{1s} \cup d_{2s} = d_{3s} \in S_3$; we have $supersede(d_{3s}, d_3)$. This holds for all dependencies in $\widetilde{K}$. We omit cases where $\nexists k_1 \in S_1$ or $\nexists k_2 \in S_2$ as they follow trivially from set union and Definition 6. Therefore, S_3 is a keyset-overlapping cut for $\widetilde{K}$. □

We conclude with a simple lemma that ensures the snapshot property that is the goal of our protocols in this section.

Lemma 2. *Every keyset-overlapping cut is a snapshot.*

Proof. Let S be a keyset-overlapping cut for keyset $\widetilde{K}$. For $(a_i, b_j) \in S$, if $a_k \rightarrow b_j$, since $a \in \widetilde{K}$, from Definition 6 we know $supersede(a_i, a_k)$. The same holds for other pairs of keys in S. Therefore, S is a snapshot. □

4.3 Optimistic (OPT)

OPT is our first MTCC protocol. The idea is to eagerly start running the functions in a DAG and check for violations of the snapshot property at the time of each function execution. If no violations are found—e.g. when updates are infrequent so the cuts at different nodes are roughly in sync—then no communication costs need be incurred by constructing a DAG-specific snapshot in advance. Even when violations are found at some node, we can potentially adjust the versionset being read at that node to re-establish the snapshot property for the DAG. However, we will see that in some cases the violation cannot be fixed, and we must restart the DAG.

OPT validates the snapshot property in two cases: when an upstream function triggers a downstream function (linear flow), and when multiple parallel functions accumulate their results (parallel flow). We present an algorithm for each case.

4.3.1 Linear Flow Validation. In the linear flow case, we have an "upstream" function in the DAG that has completed with its readset bound to specific versions, and an about-to-be-executed "downstream" function whose readset is still unbound. If we are lucky, the current cut at the downstream node forms a snapshot with the upstream function's readset; if not, we will try to modify the downstream readset to suit.

Specifically, given a versionset R_u read until now in the DAG, a downstream function F_d must bind a keyset $\widetilde{R}_d$ to a versionset R_d, where $R_d \cup R_u$ is a snapshot. This require two properties: (Case I) R_d supersedes R_u's dependencies, $R_u.deps$ (see Definition 4), and (Case II) R_u supersedes $R_d.deps$.

Algorithm 1 shows the validation process. When an upstream function F_u triggers F_d, it sends the results (R_u, S_u) from running Algorithm 1 on F_u. R_u is the versionset read by

Algorithm 1 Linear Flow Validation

```
Input: a, S_u, R_u, R̃_d, R̃_DAG, C
 1: if a == True then
 2:     return "Abort"
// Ensure all keys in R̃_d are available for execution
 3: R_remote := ∅
 4: for k ∈ R̃_d do
 5:     if k_i ∉ C ∧ k_j ∈ S_u then
 6:         R_remote.add(k_j)
 7:     else if k_i ∉ C ∧ k_j ∉ S_u then
 8:         C.update(k)
// Case I
 9: R_local := C ∩ R̃_d
10: for k_i ∈ S_u do
11:     if k_j ∈ R_local ∧ !supersede(k_j, k_i) then
12:         R_remote.add(k_i)
// Case II
13: abort := False
14: S_map := {}  // an empty map
15: for k_i ∈ R_local do
16:     S_i = RetrieveCut(k, R̃_DAG, C)  // Algorithm 2
17:     S_map[k_i] = S_i
18:     for m_i ∈ S_i do
19:         if m_j ∈ R_u ∧ !supersede(m_j, m_i) then
20:         // key k violates Case II. Try to move it to R_remote
21:             if k_j ∉ S_u then // cannot read k from upstream to
    fix
22:                 abort = True
23:                 break
24:             else  // can read k from upstream to fix
25:                 R_remote.add(k_j)
26:                 R_local.remove(k_i)
27: if abort then
28:     return "Abort"
29: else
30:     S_d := ∅
31:     for k_i ∈ R_local do
32:         S_d = S_d ∪ S_map[k_i]
33:     S_d.version() // create temporary versions for keys in S_d
34:     R_d := R_local
35:     for k_i ∈ R_remote do
36:         R_d.merge(fetch(k_i))
37:     return S_new_u = S_u ∪ S_d, R_new_u = R_u ∪ R_d
```

F_u and any of its upstream functions. S_u is a versionset with two properties: It is a keyset-overlapping cut for the DAG's read set ($\widetilde{R}_{DAG}$), and $R_u \subseteq S_u$—all keys in R_u are present in S_u. Since S_u is a snapshot (Lemma 2), we know S_u supersedes the dependencies of R_u. We show later how S_u is constructed and prove its properties in Theorem 2. Recall that S_u and R_u only contain the id and vector clock for each key. Dependency metadata and payloads are *not* shipped across functions.

In lines 1-2 of Algorithm 1, we check if the upstream validation process decided to abort (a is an abort flag from

Algorithm 2 RetrieveCut

Input: $k, \widetilde{R}_{DAG}, C$
1: $C' := \varnothing$
2: $to\_check := \{k_i \in C\}$ // k_i is the version of k in C
3: **while** $to\_check \neq \varnothing$ **do** // transitively add dependencies
4: **for** $v_j \in to\_check$ **do** . // of k to construct the cut C'
5: **for** $d_k \in v_j.deps$ **do**
6: **if** $d_l \notin C'$ **then** // C' does not contain d
7: C'.add($d_m \in C$)
8: to_check.add($d_m \in C$)
9: to_check.remove(v_j)
10: return $C' \cap \widetilde{R}_{DAG}$ // a keyset-overlapping cut for $\widetilde{R}_{DAG}$

the upstream). If so, we also abort. Otherwise, beginning on line 3, we ensure that the keys $\widetilde{R}_d$ to be read in F_d are available. For each $k \in \widetilde{R}_d$ that is not present in the local cut C, if k exists in S_u as k_i, we add it to the versionset R_{remote} (line 6); it will be fetched from upstream at the end of the algorithm. Otherwise, we update C to include k (line 8) following the Local Cut Update process described in Section 3.

Next, we begin handling the two cases mentioned above. In Case I (lines 9-12), we start by forming a candidate mapping for $\widetilde{R}_d$, R_{local}, by simply binding $\widetilde{R}_d$ to the overlap of the local cut C (line 9). We then check each element of R_{local} to see if it supersedes the corresponding element of $R_u.deps$. It is sufficient to check if each element of R_{local} supersedes the corresponding element of S_u, which in turn supersedes the element of $R_u.deps$. When we discover a violation, we add the corresponding key from S_u to R_{remote}.

In Case II (line 13), we ensure that R_u supersedes $R_d.deps$. To do so, we identify elements in R_{local} whose dependencies are not superseded by the corresponding keys in R_u (line 19).

For $k_i \in R_{local}$, it is sufficient to construct S_i, a keyset-overlapping cut for $\widetilde{R}_{DAG}$ that contains k_i and check if R_u supersedes S_i. S_i is created as follows: we first construct a cut C' from C that includes k_i and intersect C' with $\widetilde{R}_{DAG}$ to get S_i (Algorithm 2). According to Lemma 1, S_i is a keyset-overlapping cut for $\widetilde{R}_{DAG}$. We remove elements not in $\widetilde{R}_{DAG}$ as they need not be checked for supersession. This optimization significantly reduces the amount of causal metadata we ship across nodes (line 10 of Algorithm 2).

If R_u does not supersede S_i, then k_i cannot be included in R_d, and we try to use the upstream versions instead. If k does not exist upstream, we fail to form a snapshot and abort (lines 21-22). Otherwise, we add an older version $k_j \in S_u$ to R_{remote} (line 25) and remove k_i from R_{local} (line 26).

At this point we can construct S_d, a union of all keyset-overlapping cuts for $\widetilde{R}_{DAG}$ that supersedes dependencies of R_{local} (lines 31-32). By Theorem 1, S_d is a keyset-overlapping cut for $\widetilde{R}_{DAG}$. The cache creates temporary versions that are stored locally for keys in S_d in case they need to be fetched by other caches during the distributed snapshot construction (line 33). Finally, we initialize R_d to R_{local} and fetch the keys in R_{remote} to merge into R_d. R_d is now provided to F_d for execution, and $S_{new\_u} = S_u \cup S_d$, $R_{new\_u} = R_u \cup R_d$ are used for validation in subsequent functions.

Correctness. S_{new_u} passed to subsequent functions must have the same properties as S_u. Recall that S_u has two properties. First, it is a keyset-overlapping cut for $\widetilde{R}_{DAG}$. Second, $R_u \subseteq S_u$. For the first property, recall that S_d is also a keyset-overlapping cut for $\widetilde{R}_{DAG}$. Hence by Theorem 1, we know $S_{new\_u} = S_u \cup S_d$ is also a keyset-overlapping cut for $\widetilde{R}_{DAG}$. We now show $R_{new\_u} \subseteq S_{new\_u}$.

Theorem 2. *If the validation process in Algorithm 1 succeeds, let $R_{new\_u} = R_u \cup R_d$ and $S_{new\_u} = S_u \cup S_d$. Then $R_{new\_u} \subseteq S_{new\_u}$.*

Proof. We prove by induction. We first prove the base case where there is no upstream. In this case, $R_u = \varnothing, S_u = \varnothing$, so it is sufficient to show $R_d \subseteq S_d$. Recall by construction S_d contains all keys in R_{local} (line 34-35). Since there is no upstream function, we do not fetch any data to update R_d (line 39), so we have $R_d = R_{local}$. Hence, $R_{new\_u} \subseteq S_{new\_u}$.

Inductive hypothesis: Given R_u, S_u such that $R_u \subseteq S_u$, we want to show $R_{new\_u} \subseteq S_{new\_u}$. It is sufficient to show that $R_u \subseteq S_{new\_u}$ and $R_d \subseteq S_{new\_u}$. For the first part, let $k_i \in R_u$, $k_m \in S_d$ and $k_j \in (S_u \cup S_d)$. Since $R_u \subseteq S_u$, we know $k_i \in S_u$, and therefore $k_j = k_i \cup k_m$. Since Case II of the validation process ensures that $supersede(k_i, k_m)$, we know $k_i \cup k_m = k_i$, and therefore $k_i == k_j$. If $k_m \notin S_d$, it is trivially true that $k_i == k_j$. Hence, $R_u \subseteq S_{new\_u}$.

We now prove the second half of the inductive hypothesis. If $k \in \widetilde{R}_d$, there are two cases: either k is read from C (and potentially from upstream) or k is not read from C. In the first case, let $k_m \in C$. By construction, we know $k_m \in S_d$. Suppose $k_j \in S_u$ and $k_i \in R_d$. If k is also read from the upstream, then $k_i = k_m \cup k_j$. Otherwise, $k_i = k_m$. Note that since Case I ensures that $supersede(k_i, k_j)$, we can still express k_i as $k_m \cup k_j$. Therefore, regardless of whether k is read from the upstream, we always have $k_i = k_m \cup k_j$. Also, since $k_m \in S_d$ and $k_j \in S_u$, we know $k_i = (k_m \cup k_j) \in (S_u \cup S_d)$. We now have $R_d \subseteq S_{new\_u}$. If k is not read from C, $k_i \in R_d$ is read from S_u, so $k_i \in S_u$, and Case II ensures that $k_j \notin S_d$. Hence $R_d \subseteq S_{new\_u}$, and S_{new_u} has the same properties as S_u. □

4.3.2 Parallel Flow Validation. When multiple parallel upstream functions $(U_1, U_2, ..., U_n)$ accumulate their results to trigger a downstream function, we need to validate if the versionsets read across these parallel upstreams form a snapshot. To this end, we check if their read sets $(R_1, R_2, ..., R_n)$ supersede $(S_1, S_2, ..., S_n)$, each S_i being a keyset-overlapping cut for $\widetilde{R}_{DAG}$ that contains each upstream U_i's read set R_i.

Algorithm 3 Parallel Flow Validation

Input: $(S_1, R_1, a_1), (S_2, R_2, a_2), ..., (S_n, R_n, a_n)$

1: $a := \bigvee_{i=1}^{n} a_i$
2: **if** $a == True$ **then**
3: return "Abort"
4: $S := \bigcup_{i=1}^{n} S_i$
5: **for** $i \in [1, n]$ **do**
6: **for** $k_m \in R_i$ **do**
7: **if** $!supersede(k_m, k_n \in S)$ **then**
8: return "Abort"
9: $R := \bigcup_{i=1}^{n} R_i$
10: return S, R

Algorithm 4 CreateSnapshot

Input: $\widetilde{R_i}, \widetilde{R}_{DAG}, C$

1: **for** $k \in \widetilde{R_i} \mid k \notin C$ **do**
2: C.update(k)
3: $S_i := \varnothing$
4: **for** $k \in \widetilde{R_i}$ **do**
5: $S_i = S_i \cup$ RetrieveCut$(k, \widetilde{R}_{DAG}, C)$ // Algorithm 2
6: S_i.version() // create temporary versions for keys in S_i
7: return S_i

In Algorithm 3, we first check if any upstream function aborts due to linear flow validation. If so, we also abort. Otherwise, we create S, a union of all upstream S_is that contains the read sets of all parallel upstreams; it follows that S supersedes the dependencies of all parallel upstream read sets. For each k_m in each read set R_i, we check if k_m supersedes the corresponding $k_n \in S$. Since we are validating between parallel upstreams whose functions have already been executed, OPT cannot perform any "repair" as in Algorithm 1. Therefore, if validation fails, we abort. Note that S has exactly the same properties as S_i if validation succeeds. We omit the proof as it is almost the same as Theorem 2.

4.4 Conservative (CON)

CON is the opposite of OPT: Instead of lazily validating read sets as the DAG progresses, the scheduler coordinates with all caches involved in the DAG request to construct a distributed snapshot of $\widetilde{R}_{DAG}$ before execution. Each function's corresponding cache first creates S_i, a keyset-overlapping cut for $\widetilde{R}_{DAG}$, such that S_i contains the function's read set. According to Theorem 1 and Lemma 2, the distributed snapshot S can then be formed by taking the union of all S_is.

In Algorithm 5, the scheduler instructs each function F_i's cache to create S_i (line 4) via Algorithm 4. If a key in $\widetilde{R_i}$ is missing from C, the cache updates C to include the key (lines 1-2 of Algorithm 4). Then, for each key $k \in \widetilde{R_i}$, the cache

Algorithm 5 Distributed Snapshot Construction

Input: $(F_1, \widetilde{R_1}), (F_2, \widetilde{R_2}), ..., (F_n, \widetilde{R_n}), \widetilde{R}_{DAG}$

1: $S := \varnothing$
2: $R := []$ // empty list
3: **for** $i \in [1, n]$ **do**
4: $S_i = F_i$.GetCache.CreateSnapshot$(\widetilde{R_i}, \widetilde{R}_{DAG})$ // Algorithm 4
5: $R_i = S_i \cap \widetilde{R_i}$
6: R.append(R_i)
7: $S = S \cup S_i$
8: $R_{remote} := \{\}$ // an empty map
9: **for** $R_i \in R$ **do**
10: **for** $k_m \in R_i$ **do**
11: **if** $!supersede(k_m, k_n \in S)$ **then**
12: R_{remote}[i].add(k_n) // cache i needs to fetch k_n
13: **for** $i \in R_{remote}$ **do**
14: F_i.GetCache.Fetch(R_{remote}[i])

creates a keyset-overlapping cut for $\widetilde{R}_{DAG}$ that includes k and unions all the keyset-overlapping cuts to create S_i (line 5). It then creates temporary versions for keys in S_i in case they need to be fetched by other caches during the distributed snapshot construction (line 6). After that, the scheduler forms R_i, the versionset that F_i reads from the local cut, by binding $\widetilde{R_i}$ to the overlap of S_i (line 5 of Algorithm 5). After all S_is are created, the scheduler unions them to create S (line 7). It then inspects the R_i of each function. If a key $k_m \in R_i$ cannot supersede $k_n \in S$ (line 11), then the corresponding cache fetches from remote caches to match k_n (line 14). The scheduler only begins execution after all remote reads finish; each function reads from a partition of S.

4.5 Hybrid (HB)

The OPT protocol starts a DAG immediately without coordination but is susceptible to aborts, and there is no guarantee as to how many times it retries before succeeding. On the other hand, the CON protocol never aborts but has to pay the cost of coordinating with all caches involved in a request to construct a distributed snapshot.

The hybrid protocol (HB) combines the benefits of the OPT and CON protocols. HB (run by the scheduler) starts the OPT subroutine and simultaneously performs a simulation of OPT. This simulation is possible because OPT only needs the read set of each function and the local cut at each cache to perform validation, and the simulation process can get the same information by querying the caches. The simulation is much faster than executing the request because no functions are executed and no causal metadata is passed across nodes. The CON subroutine is activated only when the simulation aborts. HB includes some key optimizations that enable the two subroutines to cooperate to improve performance.

Pre-fetching. After the simulation, we know what data must be fetched from remote storage during each function's validation process (Algorithm 1). In this case, the HB protocol notifies caches involved in the request to pre-fetch relevant data before the OPT subroutine reaches these caches.

Early Abort. When our simulation aborts, HB notifies all caches to stop the OPT process to save unnecessary computation. This is especially useful for DAGs with parallel functions: A function is not aware that a sibling has aborted in Algorithm 1 until they "meet" and abort in Algorithm 3.

Function Result Caching. After each function is executed under OPT, its result and key versions read are stored in the cache. If OPT aborts, the function is re-executed under CON, and if CON's key versions match the original execution's, we skip execution and retrieves the result from the cache. This data is cleared immediately after the DAG finishes.

4.6 Discussion

We now discuss a few important properties of our MTCC protocols and how to handle cases when $\widetilde{R}_{DAG}$ is unknown.

Interaction with Autoscaling. Recall the naïve approach at the beginning of Section 4 in which all caches coordinate to maintain a large distributed cut at all times. Such an approach requires knowing the membership of the system, which dynamically changes in a serverless setting. This protocol will thus require expensive coordination mechanisms to establish cluster memberhsip before each cut update; the autoscaling policy cannot act while the cut is being updated.

On the other hand, none of protocols described in this section rely on node membership to achieve TCC. Each cache independently maintains its own cut, and only a small number of caches involved in a DAG request needs to communicate to ensure the snapshot property. Therefore, HydroCache guarantees TCC in a way that is orthogonal to autoscaling.

Repeatable Read. If two functions in a single request both read key k, it is natural to expect that they will read the same version of k [5]. The CT and CON protocols trivially achieve repeatable read because their snapshots are constructed prior to DAG execution. For OPT, repeatable read is a simple corollary of Theorem 2. Given R_u, R_d, S_{new_u}, $k_i \in R_u \Rightarrow k_i \in S_{new\_u}$, and $k_j \in R_d \Rightarrow k_j \in S_{new\_u}$. Hence $k_i == k_j$, and OPT and HB achieve repeatable read.

Versioning. Our protocols do not rely on aggressive multi-versioning; they create *temporary* versions (line 33 of Algorithm 1 and line 6 of Algorithm 4) so that remote caches can retrieve the correct versions of keys to construct snapshots. These versions are garbage collected after each request finishes, significantly reducing storage overhead.

Unknown Read Set. When the readset is unknown, CON can no longer pre-construct the distributed snapshot. Instead, we rely on OPT to "explore" the DAG's read set as the request progresses and validate if the keys read so far form a snapshot. When the validation fails, we invoke CON to construct a distributed snapshot for all keys read thus far *before* retrying the request. This way, the next trial of OPT will not abort due to causal inconsistencies between keys that we have already explored. We switch between OPT and CON until the read sets of all functions in the DAG are fully explored.

With an unknown $\widetilde{R}_{DAG}$, we can no longer perform the optimization in line 10 of Algorithm 2 to reduce causal metadata shipped across nodes. Finally, as OPT explores new keys, the protocol may abort multiple times. However, in practice, a DAG will likely only read a small number of keys that are updated very frequently. These keys are the primary culprits for aborts, and the number of aborts will roughly be bounded by the number of such write-heavy keys.

5 EVALUATION

This section presents a detailed evaluation of HydroCache. We first study aspects of HydroCache in isolation: MTCC's performance (§5.2), a comparison to other consistency models (§5.3), and scalability (§5.4). We then evaluate HydroCache's broader benefits by comparing its performance and consistency against cache-less architectures (§5.5).

5.1 Experiment Setup and Workload

Our experiments were run in the `us-east-1a` AWS availability zone (AZ). Function schedulers were run on AWS `c5.large` EC2 VMs (2 vCPUs and 4GB RAM), and compute nodes used `c5.4xlarge` EC2 VMs (16 vCPUs and 32GB RAM); hyperthreading was enabled. Each compute node had 3 function executors that shared a HydroCache. Cloudburst was deployed with 3 scheduler nodes; the system's autoscaling policy enforced a minimum of 3 compute nodes (9 Python execution threads and 3 caches total). Clients were run on separate machines in the same AZ.

Unless otherwise specified, for each experiment, we used 6 concurrent benchmark threads, each sequentially issuing 500 DAG execution requests. The cache "refresh" period for cut updates was set to 100ms. Our dataset was 1 million keys, each with an 8-byte payload. Caches were pre-warmed to remove the data retrieval overheads from the KVS.

Our benchmarks evaluate two DAG topologies: a linear DAG and a V-shaped DAG. Linear DAGs are chains of three sequentially executed functions, where the result of each upstream function is passed in as an argument to the downstream function. V-shaped DAGs also contain three functions, but the first two functions are executed in parallel, and their results are passed as arguments to the third function.

Every function takes two arguments, except for the sink function of the V-shaped DAG, which takes three arguments.

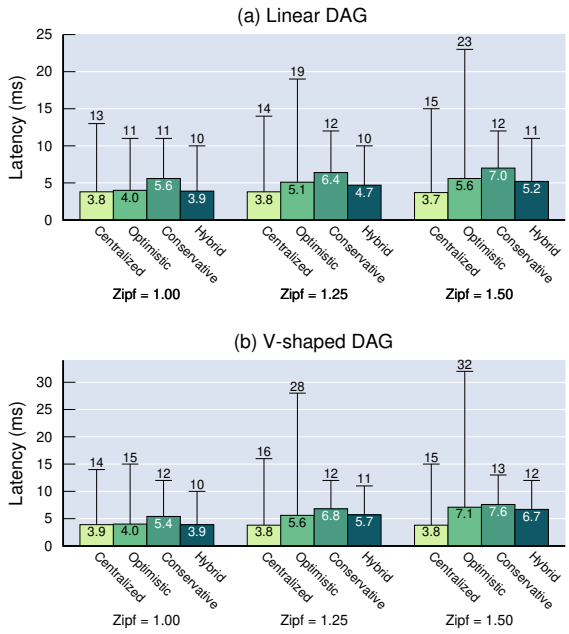

Figure 2: Median (bar) and P99 (whisker) latencies across different protocols for executing linear and V-shaped DAGs.

The arguments are either a reference to key in Anna (drawn from a Zipfian distribution) or the result of an upstream function. At the end of a DAG, the sink function writes its result into a key randomly chosen from the DAG's read set; the write is causally dependent on the keys in the read set. Each function returns immediately to eliminate function execution overheads, and we assume the read set is known.

5.2 Comparison Across Protocols

In this section, we evaluate the protocols proposed in Section 4. Figure 2 shows the end-to-end DAG execution latency for our protocols, with varying topologies and Zipfian distributions (1.0, 1.25, and 1.5). Our experiments show that HB has the best performance of the three distributed protocols and highlight a trade-off between CT and HB, which we discuss in more detail. We omit discussion of the experiments with a Zipfian coefficient of 1.25, as the performance is in between that of the other two contention levels.

Centralized (CT) achieves the best median latency in all settings because each request has only one round of communication with a single cache to create its snapshot before execution. This avoids the additional overhead of passing causal metadata across caches; neither DAG topology nor workload contention affect performance. Furthermore, function results within a DAG are passed between threads rather than between nodes, avoiding expensive network latencies. Nonetheless, its 99th percentile latency is consistently worse than the conservative protocol's (CON) and the hybrid protocol's (HB), because requiring all functions to execute on a single node leads to more load imbalance (Section 4.1).

Optimistic (OPT) achieves excellent performance for linear DAGs (Figure 2 (a)) when the Zipfian coefficient is set to 1.0, a moderately contended distribution. Key accesses are spread across the entire key space, so each read likely accesses the most recent update, which has already been propagated and merged into the cut of all caches. As a result, 88% of DAG executions see local cuts that form a distributed snapshot without any intervention, significantly improving performance.

For the most contended workload (Zipf=1.5), the median latency increases by 40% due to increased data shipping costs (line 39 of Algorithm 1) to construct a snapshot; data shipping occurred in 82% of DAG executions. Correspondingly, 99th percentile latency was 2.1× worse. Under high contention, the probability of the OPT protocol's validation phase failing increases, which leads to more aborts and retries. In this experiment, 8% of the DAGs aborted at least once, and in the worst case, a DAG was retried 7× before succeeding.

OPT performs similarly for V-shaped DAGs (Figure 2 (b)). The key difference is that increasing contention significantly degrades 99th percentile latencies. By design, OPT is unaware of the causal metadata required across the two parallel functions until parallel flow validation (Algorithm 3). The probability of validation failure is much higher since repair cannot be performed during Algorithm 3. For the most contended workload, 75% of DAGs were aborted at least once; in the worst case, a request required 14 retries.

Conservative (CON)'s median latency is 40% higher than OPT's due to the coordination prior to DAG execution. However, 99th percentile latency is more stable for high-contention workloads, with an increase of 1ms from the least to most contention. Each cache already has a snapshot for the DAG's read set before executing, so requests never abort.

Hybrid (HB) offers the best median and 99th percentile latency in all settings. To explain the performance improvements, Tables 1 and 2 show how often each protocol subroutine was activated for each topology and contention level.

Under moderate contention (Zipf=1.0), HB has OPT's advantages of immediately executing the DAG without coordination. We see that in a large majority of cases—90% for linear DAGs and 83% for V-shaped DAGs—no data fetching is required; the OPT subroutine of HB simply passes causal metadata along the DAG. Much of the DAG has already been executed under HB by the time the CON protocol finishes constructing its snapshot. This explains the 44% and 38% improvements in median latency for linear DAGs and V-shaped DAGs, respectively.

		HB Subroutines (Linear)		
		OPT Metadata Only	OPT Data Fetch	CON
Zipf	1.0	90%	7%	3%
	1.25	37%	55%	8%
	1.5	10%	81%	9%

Table 1: Percentage of different HB subroutines activated across contention levels for linear DAGs. The first column shows when OPT subroutine succeeds and only causal metadata is passed across nodes. The second column means the OPT subroutine succeeds but data is shipped across nodes to construct the snapshot. In the third column, OPT subroutine aborts and the DAG is finished by the CON subroutine with data shipping.

		HB Subroutines (V-shaped)		
		OPT Metadata Only	OPT Data Fetch	CON
Zipf	1.0	83%	5%	12%
	1.25	29%	17%	54%
	1.5	10%	12%	78%

Table 2: Percentage of different HB subroutines activated across different contention levels for V-shaped DAGs.

Under high contention (Zipf=1.5), HB offers 35% better median latency than CON for linear DAGs. However, for V-shaped DAGs, the performance improvement is less than 15%. The reason, seen in Table 2, is that the OPT subroutine aborts frequently under high contention: In 78% of the cases, the CON subroutine is activated. Nevertheless, for the remaining 22% of requests, the OPT subroutine succeeds, leading to a moderate improvement in median latency.

Interestingly, the median latency of HB is even lower than OPT. There are two reasons for this: First, the CON subroutine prevents aborts to which OPT is susceptible. Second, while the OPT subroutine executes, the CON subroutine pre-fetches data to help OPT construct the snapshot (see Section 4.5). At the 99th percentile, HB matches the performance of CON and significantly outperforms OPT due to the avoided aborts.

***Takeaway**: HB consistently achieves the best performance among the distributed protocols, by taking advantage of optimistic execution while avoiding repeated aborts.*

5.2.1 Centralized vs. Hybrid. From the previous section, it is clear that HB is the best distributed protocol, but CT achieves better median latencies while compromising on 99th percentile latencies. We now turn to the question of whether our system should choose to use CT, as it is a simpler protocol that achieves reasonable performance.

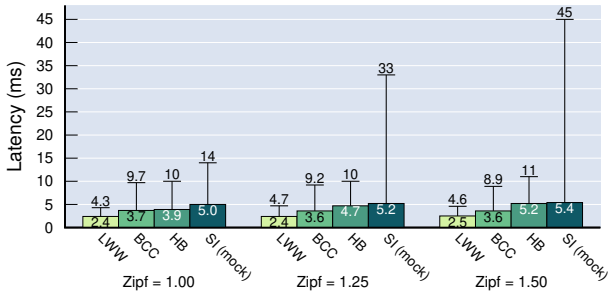

Figure 3: Median and P99 latencies between LWW, BCC, HB, and simulated SI protocols.

As discussed in Section 4.1, CT has three key limitations: coarse-grained scheduling, forcing all data to a single node, and limited parallelism. In this section, we have not seen the data retrieval overhead as all caches were warmed up in advance. As we show in Section 5.5, the overhead of remote data fetches can be significant, especially for large data.

To better understand the limitation due to parallelism, we use a single benchmark thread to issue V-shaped DAGs with varying fanout (number of parallel functions), ranging from 1 to 9. To emphasize the performance gains from parallelism, each parallel function executes for 50ms, and the sink function returns immediately (for brevity, no figure is shown for this experiment). We observe that under a workload with moderate contention, HB's performance (median latency) is relatively stable, as it parallelizes sibling functions across nodes. However, CT executes all functions on the same node, so parallelism is limited to the three executors on that node. Therefore, we observe latency jumps as fanout grows from 3 to 4 and from 6 to 7. For DAGs with fanout greater than 7, HB outperforms CT by 3×.

***Takeaway**: Many factors affect the optimal protocol for a given workload, including load balancing, cache hits rates, and the degree of parallelism within a DAG. In general, the HB protocol offers the most flexibility.*

5.3 Consistency Overheads

In this section, we compare the performance of the HB protocol against two weaker consistency protocols (last-writer-wins (LWW) and Bolt-on Causal Consistency (BCC) [7]) and one strong consistency protocol (Snapshot Isolation (SI)). We begin by discussing LWW and BCC and return to SI in Section 5.3.1. We evaluate linear DAGs in this experiment and vary the workload's contention level.

The LWW protocol attaches a timestamp to each write, and concurrent updates are resolved by picking the write with the largest timestamp. LWW only guarantees eventual replica convergence for individual keys but offers the best

performance as there are no constraints on which key versions can be read. BCC only guarantees CC+ (see Section 2.2) for keys read within individual functions.

As shown in Figure 3 LWW and BCC are insensitive to workload skew. No causal metadata need be passed across nodes, and no data is shipped to construct a distributed snapshot. HB, as discussed previously (Section 5.2), incurs higher overheads under high contention due to the data fetching overhead incurred by the OPT subroutine and the coordination overhead incurred by the CON subroutine.

Under moderate contention (Zipf=1.0), HB matches the performance of BCC and is 62% slower than LWW. Under high contention, HB is 44% slower than BCC and 2× slower than LWW. However, Table 1 shows that 90% of DAGs require data shipping across caches under high contention. Since BCC does not account for multiple caches, over 90% of the BCC requests violated the TCC guarantee.

In addition to latency, we measure the maximum causal metadata storage overhead for each key in the working set for the HB protocol. Under moderate contention (Zipf=1.0), the median metadata overhead is 120 bytes and the 99th percentile overhead is 432 bytes. Under high contention (Zipf=1.5), the median and the 99th percentile overheads increase to 300 bytes and 852 bytes, respectively. Under high contention, both the keys' vector clock lengths and dependency counts increase: The 99th percentile vector clock length is 9 and the dependency count is 7.

5.3.1 Snapshot Isolation. HydroCache relies on Anna for its storage consistency; both components are coordination-free by design. However, there are databases that provide "strong" isolation with serverless scaling. Notably, AWS Serverless Aurora [27] provides Snapshot Isolation (SI) via its PostgreSQL configuration. SI is stronger than TCC in two ways: (1) it guarantees that reads observe all committed transactions, and (2) at commit time, SI validates writes and aborts transactions that produce write-write conflicts. TCC allows transactions to observe stale data and also allows concurrent updates, asynchronously resolving conflicts via the convergent conflict resolution policy (set union in our case).

To determine whether a strongly consistent serverless framework could compete with HydroCache and Anna, we conduct two experiments. As a baseline, we replace Anna with Aurora-PostgreSQL (with SI) and measure performance. We warm the Aurora database by querying every key once in advance of the experiment, so that all future requests hit Aurora's buffer cache and there are no disk accesses on the critical path. Second, we perform a more apples-to-apples comparison using HydroCache and Anna, replaying the cache misses and abort/retry patterns observed in Aurora.

In the first experiment, we observe that SI is over an order of magnitude slower than HB at both the median and the 99th percentile across all contention levels. (Due to space constraints, no figure is shown for this experiment.) There are four likely reasons for this gap. The first three echo the guarantees offered by SI. (a) Guarantee (1) requires that when a transaction first reads any key k, it must bypass our cache and fetch k from an up-to-date database replica. (b) Both guarantees (1) and (2) require coordination inside Aurora to ensure that replicas agree on the set of committed transactions. (c) Guarantee (2) causes transactions to abort/retry. The fourth reason is a matter of system architecture: (d) Aurora is built on PostgreSQL, almost certainly resulting in more query overhead than HydroCache and Anna.

The absolute numbers from this experiment do not provide much insight into the design space due to reason (d). However, workload traces can be used to simulate the performance of SI in HydroCache and Anna.

Therefore, in our second experiment, we take the Aurora trace (accounting for cache misses and abort/retry count) and run it under LWW (our fastest option) using HydroCache and Anna. This is a *lower bound* on the latency of a full SI implementation, as it doesn't account for coordination overheads (reason (b)). The overhead of coordination protocols such as two-phase commit is at least 17ms as reported by Google Spanner [9] and worsens as the system scales.

The SI (mock) bars in Figure 3 show the simulated results. Median latency is worse than HB's due to the added network round-trip overhead during cache misses. While HB's 99th percentile latency is insensitive to workload skew, SI's tail latency is over 3x worse as we increase the contention from Zipf=1.0 to 1.5. Under high contention, a large number of transactions concurrently update a single key and only one is allowed to commit. All other transactions are aborted and retried, contributing to the high tail latency. In fact, 23% of the transactions are retried at least once, and in the worst case, a transaction is retried 18 times before committing. Thus, SI does not meet our goal of low-latency function serving.

***Takeaway**: MTCC protocols incur a reasonable overheads—at most 2× performance penalty and sub-KB metadata overheads—compared to weaker protocols while preventing anomalies on up to 90% of requests. MTCC also avoids the cache misses and expensive aborts caused by snapshot isolation.*

5.4 Scalability

Thus far, we have only studied small DAGs of length at most 3. In this section, we explore the scalability of our MTCC protocol as DAG length increases. Figure 4 reports the performance of the HB protocol as a function of the length of a linear DAG for different contention levels. We normalize (divide) the latency by the length of the DAG. Ideally, we would expect a constant normalized latency for each skew.

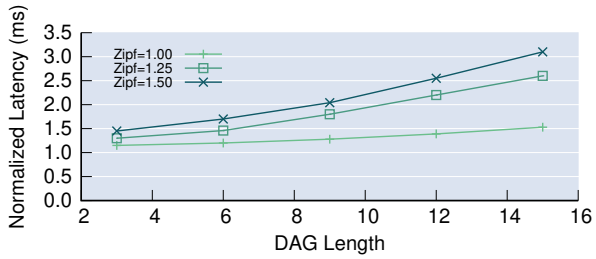

Figure 4: Normalized median latency as we increase the length of a linear DAG.

In practice, as we increase DAG length from 3 to 15, the normalized latency grows under all contention levels. Longer DAGs with larger read sets require creating larger snapshots, so the volume of causal metadata being passed along the DAG increases linearly with respect to DAG length. Furthermore, the protocol must communicate across a larger number of caches to construct the snapshot as the DAG grows. Under moderate contention (Zipf=1.0), the normalized latency grows from 1.15ms to 1.53ms (a 33% increase), while under high contention (Zipf=1.5), the normalized latency doubles.

The reason for the difference across contention levels is that the OPT subroutine avoids data fetches for 91% of requests under moderate contention. Under high contention, 75% of requests require OPT to ship data across caches, and 12% of requests require the CON subroutine to communicate with all caches, significantly increasing latencies.

***Takeaway**: Our MTCC protocols scale well from short to long DAGs under moderate contention, with normalized latencies only increasing by 33%; however, high contention workloads require large amounts of expensive data shipping, leading to a 2× increase in normalized latency.*

5.5 Life Without HydroCache

Finally, we investigate the benefits of HydroCache (using the HB protocol) relative to a cache-less architecture. We study two different architectures: One fetches data directly from Anna, and another fetches data from AWS ElastiCache (using Redis in cluster mode). Although ElastiCache is not an autsocaling system, we include it in our evaluation because it is an industry-standard, memory-speed KVS [25].

5.5.1 Caching Benefits. We first study the performance benefits of HydroCache. We begin with caches empty, and execute linear DAGs with 3 functions. In previous experiments, the reads and writes were drawn from the same distribution. Here, we draw the read set from a Zipfian distribution of 100,000 keys with a coefficient of 1.5. We study two, less-skewed distributions for the write set—a uniform distribution and a Zipfian distribution with coefficient 0.5. This variation in distributions is commonly observed in many social network applications [12].

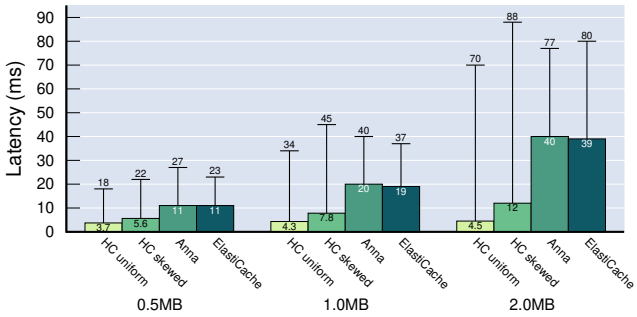

Figure 5: Median and P99 latency comparison against architectures without HydroCache.

Writing new key versions forces HydroCache to exercise the MTCC protocol, which highlights its performance overhead under different levels of write skew. To focus on the benefit of caching large payloads, we configure each DAG executed with HydroCache to write a small (1 byte) payload to avoid the overhead of expensive writes to the KVS. To prevent this write from clobbering the original large payload—thereby reducing the cost of future KVS reads—we attach the 1 byte payload with a vector clock concurrent with what is stored in Anna. This payload will be *merged* with the original payload via set union so that further reads to the same key still fetch both payloads. Since ElastiCache does not support payload merge, we simplify the workload of cache-less architectures by making the DAG read-only.

Figure 5 compares HydroCache, Anna, and ElastiCache as we increase payload size from 0.5MB to 2MB. Anna and ElastiCache exhibit similar performance for all payload sizes. HydroCache with uniform random writes outperforms both systems by 3× at median for small payloads and 9× for large payloads. 99th percentile latencies are uniform across systems because cache misses incur remote data fetches.

With a slightly skewed write distribution (Zipf=0.5), our protocol is forced to more frequently ship data between executor nodes to construct a distributed snapshot. For large payloads, the median latency is only 3.5× better than other systems, and 99th percentile latency is in fact 14% higher.

***Takeaway**: HydroCache improves performance over cache-less architectures by up to an order-of-magnitude by avoiding expensive network hops for large data accesses.*

5.5.2 Consistency Benefits. We measure how many inconsistencies HydroCache prevents relative to Anna and ElastiCache, neither of which guarantee TCC. To track violations in the other systems, we embed causal metadata directly into each key's payload when writing it to storage. At the end of a request, we extract the metadata from the read set to check whether the versions formed a snapshot.

	Write Skew					
	Uniform	0.5	0.75	1.0	1.25	1.5
HydroCache	0%	0%	0%	0%	0%	0%
Anna	0.02%	0.4%	1.9%	6.6%	15%	21%
ElastiCache	0.03%	0.4%	2.0%	6.6%	14%	20%

Table 3: Rates of inconsistencies observed with varying write skew for HydroCache/Anna, Anna only, and ElastiCache.

Table 3 shows the rates of TCC violations for different write skews. The number of inconsistencies increases significantly as skew increases. For the two most skewed distributions, over 14% of requests fail to form a causal snapshot.

***Takeaway**: In addition to improving performance, HydroCache prevents up to 21% of requests from experiencing causal consistency anomalies that occur in state-of-the-art systems like Redis and Anna.*

6 RELATED WORK

Many recent storage systems provide causal consistency in various settings, including COPS [19], Eiger [20], Orbe [10], ChainReaction [3], GentleRain [11], Cure [1] and Occult [22]. However, these are fixed-deployment systems that do not meet the autoscaling requirements of a serverless setting. [1, 3, 10, 11, 22] rely on linear clocks attached to each data partition to version data, and they use a fixed-size vector clock comprised of the linear clocks to track causal dependencies across keys. The size of these vector clocks is tightly coupled with system deployment—specifically, the shard and replica counts. Correctly modifying this metadata when the system autoscales requires an expensive coordination protocol, which we rejected in HydroCache's design. [19] and [20] reveal a new version only when all of its dependencies have been retrieved. This design is susceptible to "slowdown cascades" [22], in which a single straggler node limits write visibility and increases the cost of write buffering.

By extending the Bolt-On Causal Consistency approach [7], HydroCache guarantees TCC at the caching layer, separate from the complexity of the autoscaling storage layer. Each cache creates its own causal cut without coordination, eliminating the possibility of a slowdown cascade, which removes concerns about autoscaling at the compute tier. Our cache tracks dependencies via individual keys' metadata rather than tracking the linear clocks of fixed, coarse-grained data partitions. This comes at the cost of increased dependency metadata overhead; we return to this in Section 7.

Another causal system that employs a client-side caching approach is SwiftCloud [33]. That work assumes that clients are resource-poor entities like browsers or edge devices, and the core logic of enforcing causal consistency is implemented in the data center to which client caches connect. We did not consider this design because constructing a causal cut for an entire datacenter is expensive, especially in a serverless setting where the system autoscales. Moreover, SwiftCloud is not designed to guarantee causal consistency across *multiple* caches, one of the main contributions of our paper.

7 CONCLUSION AND FUTURE WORK

Disaggregating compute and storage services allows for an attractive separation of concerns around autoscaling resources in a serverless environment; however, it introduces performance and consistency challenges for applications written on FaaS platforms. In this paper, we presented HydroCache, a distributed cache co-located with a FaaS compute layer that mitigates these limitations. HydroCache guarantees transactional causal consistency for individual functions at a single node, and we developed new multisite TCC protocols that offer the same guarantee for compositions of functions spanning multiple physical nodes. This architecture enables up to an order-of-magnitude performance improvements while also preventing a plethora of anomalies to which existing cloud infrastructure is susceptible.

While HydroCache significantly outperforms existing systems, there is a variety of future work we plan to explore:

Metadata Garbage Collection. As discussed in Section 3.3, HydroCache periodically garbage collects (GCs) dependency metadata through a background consensus protocol. However, we do not GC vector clock entries (node IDs) when function executors leave the system. Simultaneously GCing node IDs and dependencies requires extra care, as it introduces race conditions. To GC a vector clock entry from a particular key k's VC, we must ensure that keys which depend on k have had their dependency metadata correctly GCed to match k's newly garbage collected VC. Furthermore, for frequently-written keys, the length of the vector clock can grow linearly with the number of nodes in the system. Therefore, we also plan to explore mechanisms to garbage collect VC entries corresponding to nodes that are still active.

Dynamic Scheduling. We are interested exploring in two, more advanced scheduling techniques. First, we want to explore policies that choose between the CT and HB protocols based on workload characteristics and resource availability. Second, our policies have assumed that functions are executed at predetermined locations, and we ship the required data to the function's node. However, it might be cheaper to ship functions to where the data lives—e.g., given a large key k on node A and a small function f on node B, we would rather ship f to A than k to B. Exploring network- and data-aware scheduling techniques is an exciting future direction.

REFERENCES

[1] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro. Cure: Strong semantics meets high availability and low latency. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 405–414. IEEE, 2016.

[2] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt. SAND: Towards high-performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, 2018.

[3] S. Almeida, J. a. Leitão, and L. Rodrigues. Chainreaction: A causal+ consistent datastore based on chain replication. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 85–98, New York, NY, USA, 2013. ACM.

[4] Aws Lambda - case studies. https://aws.amazon.com/lambda/resources/customer-case-studies/.

[5] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly available transactions: Virtues and limitations. *Proceedings of the VLDB Endowment*, 7(3):181–192, 2013.

[6] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. The potential dangers of causal consistency and an explicit solution. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 22. ACM, 2012.

[7] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772, New York, NY, USA, 2013. ACM.

[8] 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. *Proc. VLDB Endow.*, 1(2):1277–1288, Aug. 2008.

[9] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, and et al. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3), Aug. 2013.

[10] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 11:1–11:14, New York, NY, USA, 2013. ACM.

[11] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel. GentleRain: Cheap and scalable causal consistency with physical clocks. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–13. ACM, 2014.

[12] F. Fu, L. Liu, and L. Wang. Empirical analysis of online social networks in the age of web 2.0. *Physica A: Statistical Mechanics and its Applications*, 387:675–684, 01 2008.

[13] J. M. Hellerstein, J. M. Faleiro, J. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu. Serverless computing: One step forward, two steps back. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*, 2019.

[14] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Serverless computation with OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver, CO, 2016. USENIX Association.

[15] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Menezes Carreira, K. Krauth, N. Yadwadkar, J. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson. Cloud programming simplified: A Berkeley view on serverless computing. Technical Report UCB/EECS-2019-3, EECS Department, University of California, Berkeley, Feb 2019.

[16] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4):360–391, Nov. 1992.

[17] Common lambda application types and use cases. https://docs.aws.amazon.com/lambda/latest/dg/applications-usecases.html.

[18] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[19] W. Lloyd, M. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *SOSP'11 - Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 401–416, 10 2011.

[20] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pages 313–328, 2013.

[21] P. Mahajan, L. Alvisi, and M. Dahlin. Consistency, availability, convergence. Technical Report TR-11-22, Computer Science Department, University of Texas at Austin, May 2011.

[22] S. A. Mehdi, C. Littley, N. Crooks, L. Alvisi, N. Bronson, and W. Lloyd. I can't believe it's not causal! scalable causal consistency with no slowdown cascades. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 453–468, Boston, MA, Mar. 2017. USENIX Association.

[23] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, Boston, MA, 2018. USENIX Association.

[24] R. Pang, R. Caceres, M. Burrows, Z. Chen, P. Dave, N. Germer, A. Golynski, K. Graney, N. Kang, L. Kissner, J. L. Korn, A. Parmar, C. D. Richards, and M. Wang. Zanzibar: Google's consistent, global authorization system. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 33–46, Renton, WA, July 2019. USENIX Association.

[25] T. Rabl, S. Gómez-Villamor, M. Sadoghi, V. Muntés-Mulero, H.-A. Jacobsen, and S. Mankovskii. Solving big data challenges for enterprise application performance management. *Proc. VLDB Endow.*, 5(12):1724–1735, Aug. 2012.

[26] M. Raynal and M. Singhal. Logical time: Capturing causality in distributed systems. *Computer*, 29(2):49–56, Feb. 1996.

[27] Amazon aurora serverless. https://aws.amazon.com/rds/aurora/serverless/.

[28] M. Shahrad, J. Balkind, and D. Wentzlaff. Architectural implications of function-as-a-service computing. In *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, pages 1063–1075, New York, NY, USA, 2019. ACM.

[29] M. Singhal and A. Kshemkalyani. An efficient implementation of vector clocks. *Inf. Process. Lett.*, 43(1):47–52, Aug. 1992.

[30] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. M. Faleiro, J. Gonzalez, J. M. Hellerstein, and A. Tumanov. Cloudburst: Stateful functions-as-a-service. *ArXiv*, abs/2001.04592, 2020.

[31] C. Wu, J. Faleiro, Y. Lin, and J. Hellerstein. Anna: A kvs for any scale. *IEEE Transactions on Knowledge and Data Engineering*, 2019.

[32] C. Wu, V. Sreekanti, and J. M. Hellerstein. Autoscaling tiered cloud storage in anna. *Proceedings of the VLDB Endowment*, 12(6):624–638, 2019.

[33] M. Zawirski, N. Preguiça, S. Duarte, A. Bieniusa, V. Balegas, and M. Shapiro. Write fast, read in the past: Causal consistency for client-side applications. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, pages 75–87, New York, NY, USA, 2015. ACM.