CausalMesh: A Causal Cache for Stateful Serverless Computing

Haoran Zhang
University of Pennsylvania
Philadelphia, PA, USA
haorz@seas.upenn.edu

Shuai Mu
Stony Brook University
Stony Brook, NY, USA
shuai@cs.stonybrook.edu

Sebastian Angel
University of Pennsylvania
Philadelphia, PA, USA
sebastian.angel@cis.upenn.edu

Vincent Liu
University of Pennsylvania
Philadelphia, PA, USA
liuv@seas.upenn.edu

ABSTRACT

Stateful serverless workflows consist of multiple serverless functions that access state on a remote database. Developers sometimes add a cache layer between the serverless runtime and the database to improve I/O latency. However, in a serverless environment, functions in the same workflow may be scheduled to different nodes with different caches, which can cause non-intuitive anomalies. This paper presents CausalMesh, a novel approach to causally consistent caching in serverless computing. CausalMesh is the first cache system that supports coordination-free and abort-free read-/write operations and read transactions when clients roam among multiple servers. CausalMesh also supports read-write transactional causal consistency in the presence of client roaming, but at the cost of abort-freedom. Our evaluation shows that CausalMesh has lower latency and higher throughput than existing proposals.

PVLDB Reference Format:
Haoran Zhang, Shuai Mu, Sebastian Angel, and Vincent Liu. CausalMesh: A Causal Cache for Stateful Serverless Computing. PVLDB, 17(13): 4599-4613, 2024.
doi:10.14778/3704965.3704969

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/eniac/causalmesh.

1 INTRODUCTION

Serverless functions allow clients to run their applications on cloud providers without needing to manage or operate servers, load balance requests across VMs / containers, scale resources up or down based on load, or deal with failures. This paradigm has proven to be popular, with all large cloud providers offering a range of options for serverless execution.

One remaining sticking point is how to deal with stateful functions that need to access shared and often persistent states. Existing solutions [9, 26, 28, 48, 63] take a straightforward approach: ensure the serverless functions are stateless (so they can be scheduled anywhere without constraints) and, instead, store the state in a set of backend databases. The stateless function can then query these databases to retrieve the necessary state on every execution, perform its operations, and update the databases as needed.

Given that accessing remote databases is expensive [23, 45] (e.g., 10–20 ms to read or write to DynamoDB), recent works [34, 49] ask whether serverless functions can use caches to keep the state closer to these functions. Proposals here include having (i) a large cache or multiple caches with a cache coherence protocol, which provides strong consistency but does not scale, or (ii) a cluster of caches such as Amazon DynamoDB Accelerator (DAX) that scales well but provides only weak (eventual) consistency.

Weak consistency is problematic in the common scenario where developers use *workflows*, which are directed graphs of functions that collectively implement the application's logic. To see this issue, imagine a social media website that uses a serverless workflow that contains two serverless functions that run one after the other and that access the same state: the first function marks a user as 'not interested' in a particular video, and the second function generates recommendations based on users' likes/dislikes. If these two functions are scheduled on different machines (a likely outcome, given that functions are supposed to be stateless and ought to be schedulable anywhere), they may end up accessing different eventually consistent caches. In such a case, the workflow would not be able to even read its own writes (i.e., the second function will not see the effect of the first). This is a violation of basic session consistency and just one example of why weakly consistent caches can make writing workflows exceedingly difficult (see Section 2.2 for more details). To better characterize this issue, we implement a minimal serverless workflow on AWS Lambda and DynamoDB DAX. The workflow consists of two serverless functions that access the same state, where the first function writes to the state and the second function reads from it. As shown in Figure 1, we observe that in this simple example, the anomaly probability can be as high as 14.2% when there are 8 cache nodes in DAX.

Recent works, in particular HydroCache [59] and FaaSTCC [39], aim to address this issue by introducing a *causal cache*: a set of caches that collectively guarantee causal consistency. Both HydroCache and FaaSTCC provide transactional causal consistency [3, 36] which they adopt from traditional causal databases. The main technical challenge present in serverless computing that is addressed by them and other prior works [41, 62] is dealing with *client mobility*: a client, or serverless workflow in our context, can access one cache during one function, and then a completely different cache in another function within the same workflow.

Proceedings of the VLDB Endowment, Vol. 17, No. 13 ISSN 2150-8097.
doi:10.14778/3704965.3704969

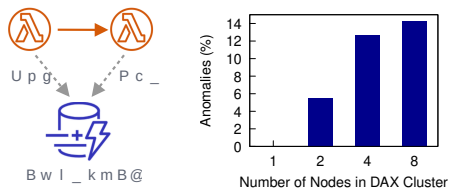

Figure 1: Anomalies rate of a two-function workflow, where the second function reads the data written by the first. The workflow runs on AWS Lambda and using DynamoDB Accelerator (DAX) as the cache. There are no anomalies when utilizing a single cache node, but it lacks scalability.

All prior works handle client mobility by introducing expensive coordination and aborts that significantly reduce the benefits of introducing a cache in the first place. In particular, HydroCache requires cache servers to coordinate with each other before execution to fetch necessary versions of data items. It also requires aborts and retries of the entire workflow when the transactions fail to commit. Both can introduce significant overheads to applications.

To improve the performance of stateful serverless and ensure that workflows work as intended, we present *CausalMesh*, a novel cache architecture for stateful serverless functions that supports client mobility. CausalMesh has several features:

(1) *Per-workflow causal consistency.* CausalMesh ensures that any data accessed by a serverless function observes the effects of prior serverless functions in the same workflow, even if they run on different servers and access different caches.
(2) *Coordination-free reads and writes.* In CausalMesh, a cache never has to synchronize with peers to process an operation.
(3) *No aborts.* All functions in a workflow that use CausalMesh always read from a causally consistent snapshot, so they never need to abort due to inconsistencies.
(4) *High throughput and low latency.* CausalMesh achieves high throughput and its latency is low and stays nearly constant as we vary the number of caches.

To simultaneously achieve all of these features, CausalMesh needs to: (i) ensure that caches can process read and write operations independently, without aborts or blocking coordination with other caches; (ii) ensure that read operations return state from a causally consistent snapshot; and (iii) make new writes to one cache visible in other caches in a timely manner. To address (i) and (ii), CausalMesh introduces a novel data structure—the *dual cache* (§4)—and an asynchronous protocol that we call *dependency integration* (§4). A dual cache represents two caches, one serving read requests and the other serving write requests. Dependency integration updates the dual cache from time to time and makes the subcache serving read requests always contain clients' dependencies so that it does not require communication with other servers to fetch the missing versions of some data items. To address (iii), CausalMesh connects servers into a series of *causally consistent chains* (§5.2), where each server simultaneously serves as the head, intermediate, and tail node in the chain. Within a chain, writes are stored initially in the head and are propagated towards the tail; the tail then reveals it to the client.

To make the benefits of CausalMesh broadly applicable, we build a library that exposes an intuitive interface similar to that of a traditional key-value store (§6). Developers can use this library to build their serverless applications. We also describe a variant of CausalMesh, CausalMesh-TCC (§7), that provides support for arbitrary read/write transactions across multiple serverless functions, although this comes at the cost of losing the abort-free property.

We implement CausalMesh and CausalMesh-TCC on top of Nightcore [27], a serverless runtime platform. We then use the key-value store interface provided by CausalMesh's client library to write several microbenchmarks (§9.2) and real-world applications consisting of workflows with 13 serverless functions (§9.5) to evaluate the performance.

To put our results in context, we compare CausalMesh and CausalMesh-TCC with HydroCache [59] and FaaSTCC [39], recent caches for serverless workflows that aim to play a similar role. In a nutshell, CausalMesh is significantly faster: we observe an up to 59% reduction in median latency, up to a 97% reduction in tail latency and 1.3–2× higher throughput. Furthermore, caches in CausalMesh do not need to coordinate with other caches or abort (whereas HydroCache and FaaSTCC must do one of the two). When we extend the comparison to the transactional variant of CausalMesh, CausalMesh-TCC, we observe that CausalMesh still achieves 1.35–1.6× higher throughput and comparable latency.

In summary, the contributions of this work are:

(1) CausalMesh, a cache system that provides causal+ consistency. To our knowledge, CausalMesh is the first general causal cache system that supports coordination-free reads and writes in the presence of client roaming, which is critical to the performance of serverless computing.
(2) A lock- and coordination-free read transaction protocol that allows developers to get a causally consistent view across multiple keys within a single serverless function.
(3) CausalMesh-TCC, an extension of CausalMesh that supports transactional causal consisteny across serverless functions.
(4) Demonstrating experimentally that CausalMesh has low latency and high throughput.
(5) A formal specification of CausalMesh in TLA+ and the corresponding model checking effort to provide evidence for the correctness of our algorithms.

2 BACKGROUND AND GOALS

We begin by providing context on serverless execution models as well as our target consistency levels.

2.1 Serverless Architecture

When deploying a traditional, serverful application to the cloud, users allocate VMs and deploy their software to the resulting instances. While the cloud handles the management of the physical infrastructure, users remain responsible for many tasks before their applications can execute, e.g., requesting a batch of VMs from the cloud provider, specifying their resource profiles, choosing their base VM images, setting permissions/firewall rules, deploying dependencies, and monitoring the application as it runs, among others.

Serverless computing promises to free users from all of the above concerns. Instead, users supply the cloud provider with a function that executes their application logic, and the provider handles all

provisioning, scaling, load balancing, and management of the execution instances. The functions can even be composed into *workflows*, which are graphs of serverless functions that collectively perform the logic of an application. Two aspects of this architecture are particularly salient to the design and necessity of CausalMesh:

(1) Provisioning and scheduling. Unlike in traditional execution environments, one of the core responsibilities of cloud providers in serverless is managing function workers and assigning requests to those workers, all of which are done out of the view of users.

At a high level, the typical strategy operates as follows. When a request for a function arrives and finds that all existing instances of that function are busy, the provider will deploy a new instance of the function to handle the request, i.e., a cold start. After handling the request, the instance will be kept warm (provisioned) for some time before being reclaimed—up to 1 hr in the case of AWS Lambda [32]. Requests are generally handled in FIFO order and routed to random instances among the set of unsaturated, pre-warmed instances when possible.

For a workflow that has a few functions, each function can be assigned to a different worker. We say a workflow *migrates* to a new worker when a function in the workflow is allocated to a different worker than its predecessor. Note that a workflow can migrate to multiple workers concurrently if it has a fan-out structure.

In reality, the workers that execute a workflow are typically located close to each other, e.g., in the same data center or availability zone, because a cluster often defines the management boundary for workloads. Once a workload is deployed to a cluster, it is typically not moved to another cluster because each cluster usually has its own isolated control plane [52]. In AWS Lambda, to improve cache locality, enable connection re-use, and amortize the costs of moving and loading customer code, events for a single function are sticky-routed to as few workers as possible [2].

(2) State management. A side effect of the above approach is that users must carefully manage any state that should persist across function executions, as the number of underlying instances and the routing of requests to instances is opaque to users. There is no guarantee or method to enforce that two requests will be executed in the same instance, whether the requests are for the same function or different functions in the same workflow. For these so-called Stateful Serverless Functions (SSFs), external storage services, e.g., relational databases or key-value stores, are standard solutions for persisting application state. Of course, access to these remote services can incur high latency and block critical path execution.

(3) Caching. To reduce the latency of accessing remote storage services, a cluster of cache nodes is deployed between the application and the remote storage. Taking Amazon's DynamoDB Accelerator (DAX) as an example, using the write-through mode, a write request is first directed to the primary cache and then replicated to other cache nodes. This replication is eventually consistent and can take up to 1 second to complete. Consequently, two clients may obtain different values when accessing the same key from the same DAX cluster, depending on the node that each client accesses.

2.2 Consistency Goals

One potential solution to the high overhead (particularly high latency) of state management is to cache the remote state at each provisioned instance, allowing functions to access the state immediately if the data is in the cache. Unfortunately, to maintain consistency across an entire workflow, traditional caches generally either need to block and confirm that they have the latest state by synchronizing with other caches, or they must proceed speculatively but then abort if an inconsistency is ever detected (as is the case in systems like HydroCache [59] and classic cache coherency protocols). This results in higher latency, particularly at the tail. Another approach altogether is to ignore strong consistency in favor of weaker guarantees (as is the case in AWS's DAX service [14]), but as we alluded to in the introduction, writing serverless workflows with weak consistency is very challenging. To strike a balance between excellent performance and meaningful consistency semantics, we settle on *causal+ consistency* (CC+). Recent work [40] has shown that no model stronger than causal consistency is achievable with high availability, making it ideal for our coordination-free goal. CC+ can summarized as:

(1) **Client-side dependency.** If an operation is issued after another operation is completed by the same client, the latter operation must observe the effects of the former. In the context of serverless, a client is a workflow.
(2) **Read-write dependency.** If a read operation reads the effect of a write operation, then we say there is a read-write dependency between the two operations. This clause itself is not a guarantee, but together with the following rule, it restricts the system's behaviors.
(3) **Transitive dependency**. The above two dependencies are transitive. Transitive dependencies must be respected by the execution. For example, if a read operation transitively depends on a write operation, the write must be reflected in the read results.
(4) **State convergence**. Different replicas of the same data will eventually converge to the same state. This is also known as causal+ in the context of causal consistency.

Providing causal consistency in the cache can greatly simplify programming in serverless workflows and make them less error-prone. A simple example is that it can avoid the anomaly discussed in Section 1. In a more complex example, consider a serverless workflow that implements a Twitter-like social media service. This example was previously implemented in serverless by Beldi [63] and was ported from the microservice library DeathStarBench [15]. When Alice replies to Bob's post, a serverless function will store the reply in the database's reply table and notification table; it also stores the id of the reply in the database's post table as foreign keys. When Bob receives the notification and interacts with this serverless application, a serverless function will fetch the post content and all its replies to render the page. There is a dependency between the notification and the post's replies, and without causal consistency, when the serverless function returns the page to Bob with the rendered post, it might not contain the reply that triggers the notification. Another common example includes applications

whereby a user sets permission (e.g., removes a user from an access control list), and then posts a sensitive file. Without causal consistency, the removed user may see the sensitive file [11, 36, 43].

2.3 Challenges

Serverless environments present a unique challenge to maintaining causal consistency. One key reason is the extremely fine-grained resource provisioning and autoscaling that is central to the serverless approach—low latency is achieved by routing functions preferentially to instances that are provisioned and available, even if they access a cold cache. Spinning up a new instance on the local machine or allowing longer queues for local execution might improve cache hit rates (e.g., using a technique like [1]), but without extremely restrictive scheduling policies, there is no guarantee of correctness. It also potentially comes at the cost of the provider resources that are not included in current pricing models.

Traditional causal cache systems like Bolt-on [5] cannot be directly applied in this scenario. Bolt-on uses a background thread that subscribes to the database and periodically merges new updates into the cache. In a *sticky* setup like Bolt-on, where the client always communicates with the same cache server, this approach is efficient and correct because the data in the cache are monotonic and never roll back, ensuring clients never read an older version than the one they previously read. However, in a serverless environment, different functions within workflow could be scheduled to different instances and interact with different cache servers. Even though each cache is monotonic, a cache server might provide a version older than one accessed from another cache server. This means that clients may fail to find a version they previously read if they migrate to another server.

3 CAUSALMESH OVERVIEW

The goal of CausalMesh is to provide a high-performance, resilient, and causally consistent cache for serverless platforms that addresses the challenge of maintaining consistency while supporting the mobility of serverless workflows.

3.1 Architecture

Figure 2 illustrates the architecture of CausalMesh. The architecture consists of four components:

Serverless Platform. The serverless platform acts as the runtime environment for user applications. It orchestrates the workflow and dispatches functions to available workers.

Databases. The database stores user data. CausalMesh supports any database, as long as the database allows custom conflict resolution policies to resolve concurrent updates (e.g., Azure CosmosDB [12], Couchbase [13], and MongoDB [42]).

CausalMesh. CausalMesh is a middleware that sits between the serverless platform and the backend databases. It contains two components, cache servers and a client library. The user functions interact with the cache servers using the client library. Cache servers communicate with each other via remote procedure calls (RPCs). The messages between cache servers follow a FIFO order but can experience arbitrary delays. CausalMesh plays a similar role to DynamoDB DAX or HydroCache. In our setup, CausalMesh's cache

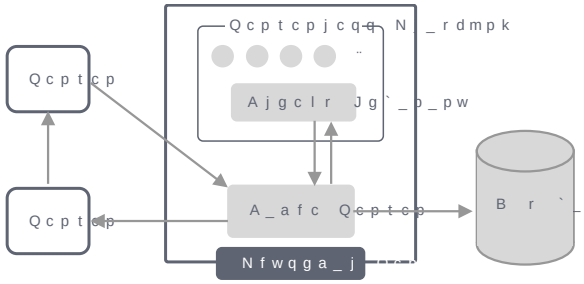

Figure 2: Architecture of CausalMesh in a three-server setup.

server runs in a 1:1 correspondence with physical machines, each physical machine runs a serverless worker and a cache server. All requests are routed to the cache server on the same machine. This setup provides the best locality. However, other configurations are also possible. For example, machines in the same rack may be assigned to the same cache server. Cache servers are managed and configured by a fault-tolerant coordinator (e.g. Zookeeper [24]).

Using the above components, the journey of a stateful serverless workflow proceeds as follows:

(1) The workflow is triggered by an event, e.g., a request from a browser or some service arriving at a gateway.
(2) The scheduler in the serverless platform dispatches the first function in the workflow to a worker machine based on resource usage, hardware requirements, and other factors.
(3) The function accesses state by using CausalMesh's client library to communicate with the cache on the same machine.
(4) After the function finishes, the scheduler gathers the result before dispatching the subsequent function or functions (in the context of a fan-out workflow) to potentially different worker machine(s) than the previous one.
(5) Repeat until the workflow is complete.

3.2 CC+ in CausalMesh

CausalMesh uses vector clocks and dependencies to enforce CC+.

Vector Clocks (VC). Vector clocks [50] are used to identify different versions of an object and capture the happens-before relation [33] between them. We use version and vector clock interchangeably in the paper. A vector clock VC is a set of ⟨server id, timestamp⟩ pairs; each server maintains its corresponding timestamp and increments it as needed. For simplicity, we assume that given N servers, each server is assigned an id from 0 to N so that a VC can be represented using a list of timestamps where $VC[i]$ is the timestamp of server i.

We define the union of two VCs, $VC_1 \cup VC_2$, as their element-wise maximum, e.g., $[1, 0] \cup [0, 1] = [1, 1]$. By using vector clocks, we can implement a custom conflict resolution policy to ensure state convergence in CC+. Informally, new versions overwrite old versions; if two versions are concurrent, we merge the vector clocks and pick one of the values as the new value in a deterministic way. In the implementation, we break ties by picking the value of the larger version by lexicographical ordering.

Dependencies (deps). Dependencies are used to track causal relationships across different keys. They are stored as a map from a

```
1  # self is a cache server
2  def integrate(self, deps):
3      all_deps = ∀[key, vc] that are transitive
        predecessors of deps (inclusive)
4      for k, vcs in all_deps.items():
5          consistent_versions =
6            self.Inconsistent[k].remove(
7              filter(vc ∈ vcs)
8            )
9          self.vc.merge_all(vcs)
10         self.Consistent[k].merge_all(
11           consistent_versions
12         )
```

Figure 3: Pseudo-code for Dependency integration.

key to the vector clocks of the writes that it depends on.

$$deps \quad := \quad \{Key \mapsto VC\}$$

To reduce the size of metadata, it only contains the nearest dependencies, meaning that if $x \rightarrow y$ (x happens before y, y depends on x) and $y \rightarrow z$, z's dependency will contain y but not x. Dependencies can be merged using the same mechanism as vector clocks.

4 THE DUAL CACHE

A core component of CausalMesh is its *dual cache*. The dual cache is what makes CausalMesh coordination-free. Each cache server maintains an instance of this dual cache, which is essentially two subcaches, a *Consistent cache* (C-cache), and an *Inconsistent cache* (I-cache).

C-cache is a hash map from keys to values and their corresponding versions; it acts like a single-version key-value store. As its name suggests, all versions in C-cache are guaranteed to be synchronized on all cache servers and, therefore, visible to clients.

I-cache, on the other hand, is a hash map from keys to a tuple (Value, VC, Deps). It acts like a multi-version key-value store and stores versions that the cache server is unsure whether they have been synchronized to all servers. As a result, I-cache is unsafe to reveal to clients.

$$\begin{aligned} \textit{C-cache} \quad &:= \quad \{\ \textit{Key} \mapsto (\ \textit{Value}, \textit{VC}\ )\ \} \\ \textit{I-cache} \quad &:= \quad \{\ \textit{Key} \mapsto [\ (\ \textit{Value}, \textit{VC}, \textit{deps}\ )\ ]\ \} \end{aligned}$$

C-cache and I-cache have different functionalities. All read requests are served by C-cache, and all write requests are served by I-cache, then moved to C-cache when they are safe to be revealed to clients through a procedure called *dependency integration* (or *integration*).

Dependency Integration. *Integration* is triggered whenever the cache server wants to make a version visible, i.e., when it receives a read from a client (§5.3) or when it determines a write exists on all servers (§5.2). The pseudo-code for this procedure is shown in Figure 3 and follows the steps below:

(1) Iterate over the dependencies.
(2) For each key-version pair in the dependencies, check if this version has already been merged into C-cache.
(3) If not, search I-cache for this version, remove it from I-cache (Lines 6–8), and merge it into C-cache (Lines 9–11) using the same procedure of merging two versions. Note that unlike I-cache, C-cache has no dependency metadata; the dependencies are automatically dropped when merged into C-cache.

As previously mentioned, a writer's dependencies only consist of their nearest dependencies. Therefore, it is necessary to recursively integrate the dependencies of these dependencies as well (Figure 3 Line 3). It's worth noting that *integration* is a purely local operation on the data structure and does not require blocking on any communication.

The purpose of integration is to ensure that, when updating C-cache, it is always a *strict causal cut*, or simply a cut. Informally, this means that the dependencies for each write in the cut should either be in the cut or should happen before a write to the same key that is already in the cut. The formal definition is as follows.

Definition 1 (Strict Causal Cut)**.** A set of writes S is a strict causal cut $\iff \forall x \in S, \forall y \in x.deps, \exists y' \in S \mid y.key = y'.key \wedge (y' = y \vee y' \rightarrow y)$

When evicting a key k, all keys that depend on it are also evicted so that C-cache remains a cut.

5 CAUSALMESH PROTOCOL

This section describes how CausalMesh works internally. We will begin by introducing CausalMesh's APIs and then describe how read and write requests are processed, followed by how read transactions are implemented, and end with an intuitive explanation on how CausalMesh achieves CC+.

5.1 CausalMesh APIs

CausalMesh APIs include a client API and a server API. The client API is used by developers; the server API is used internally and opaque to developers.

Client API. CausalMesh's client library offers an intuitive interface for developers similar to a traditional key-value store, with the added functionality of the `ReadTxn` operation, which returns a consistent view of multiple keys. The client API is as follows:

(1) `Read(key)` → `value`
(2) `Write(key, value)`
(3) `ReadTxn(keys)` → `values`

Server API. Server API is used by the client library to communicate with the cache servers, or by the cache servers to communicate with each other. Figure 4 lists all server API functions. The first three operations correspond to those in the client library's API, with additional metadata including VC, *deps* and *local*. *local* contains the client's own writes in a map from keys to their corresponding value, vector clocks, and dependencies.

5.2 Write Path

Clients' writes are always firstly saved in the server's I-cache because they only exist in one server. When saving it to the I-cache, CausalMesh first integrates carried writes (described at the end of this subsection), then assigns a version based on the server's global vector clock to the client's new write.

Global Vector Clock (GVC). Each cache server maintains its own *GVC*, which records its view of version clocks on all servers. When receiving a write, the server increments the corresponding index in its *GVC* to create a unique version for the write. For example, in a three-server setup, the *GVC* for server S_0 is $[7, 5, 2]$, and when it

CausalMesh Server API	Description
`ClientRead(key, deps)` → `value, vc`	client's read request with a key and its dependencies, return the value and version.
`ClientWrite(key, value, deps, local)` → `vc`	client's write request with the key, value, dependencies and the client's own writes, return the version.
`ClientReadTxn(keys, deps)` → `values, vcs`	client's read transaction request with keys and their dependencies, return values and their versions.
`ServerWrite(key, value, vc, deps)`	write request from another server with the key, value, version and dependencies.

Figure 4: CausalMesh's Server APIs. The first three APIs (`Client*`) are used in CausalMesh's client library. `ServerWrite` is called by other CausalMesh servers via RPC to propagate writes. Note that users do not interact directly with these functions.

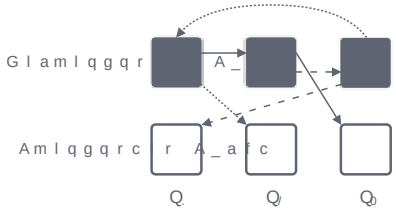

Figure 5: Propagation chain in a three-server setup. The cache servers S_0, S_1, and S_2 respectively serve as the head nodes for the solid, dashed, and dotted chains.

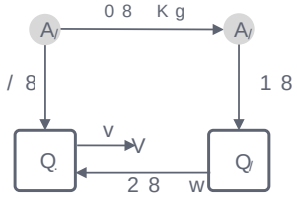

Figure 6: Without carrying and sending its own write to the cache, the client may fail to read causally consistent values. The text on the arrows contains the step number and the data being transferred.

receives a new write from a client, the assigned vector clock will be $[8, 5, 2]$. The value in the corresponding index of the *GVC*, namely $GVC[0]$, is used as a unique identifier for the writes received by S_0. The rest of *GVC* represent the newest visible versions that S_0 is aware of for other servers. In the previous example, 5 in the *GVC* indicates that among all vector clocks in C-cache, the largest value in the second index is 5, even if S_0 has some values that are larger than 5 in the second index in its I-cache, they do not contribute to the *GVC* until they are integrated.

After assigning a new version to a write, the cache server adds the write to the I-cache and flushes it to the database. The server can then safely return an acknowledgment to the client. To speed up write path, the cache server can opt to store the writes to a local persistent log and flush them asynchronously; it weakens durability guarantees but it does not affect the casual consistency nor the read performance. However, at this point, the new value is not yet visible to other clients. To make the value visible, the server will notify its peer servers by asynchronously sending the new write to its successor in the propagation chain.

Propagation Chain. Our data propagation design is inspired by Chain Replication [54] in distributed systems. In CausalMesh, each $S_i \rightarrow S_{(i+1) \mod N} \rightarrow S_{(i+2) \mod N} \cdots \rightarrow S_{(i+N-1) \mod N}$ forms a chain. Thus, in a three-server system, there are three chains in total: $S_0 \rightarrow S_1 \rightarrow S_2$, $S_1 \rightarrow S_2 \rightarrow S_0$ and $S_2 \rightarrow S_0 \rightarrow S_1$. For each chain, cache servers can take on one of three roles: head, intermediate, or tail; however, every cache server serves all three roles, just for different chains.

Writes are forwarded to the head of the chain and propagated until they reach the tail in a FIFO manner. When an intermediate cache server receives a write from its predecessor, it adds the write to its I-cache and forwards it to its successor. When a tail cache server receives a write, it integrates the write (and the dependencies) to its C-cache. Figure 5 illustrates the propagation chains of a system with three servers. This propagation takes place asynchronously after the server responds to the client and is not on the critical path.

The tail of the chain has the option to disseminate the write to other servers, asking them to integrate the write into their C-caches as well. The decision whether or not to inform others is a trade-off between network cost and visibility and does not affect the correctness of the system.

Integrating carried writes. After a client migrates to a new server, it will piggyback its *local* on its first write request. Unless the versions in the *local* have been previously received, the cache server will append them to I-cache before processing the client's current write. To see why this is necessary, consider the scenario shown in Figure 6. In a two-server setup where the connection S_0 to S_1 is very slow, suppose a client c_1 first writes x to S_0, then migrates to S_1 and writes y that depends on x. As the connection S_0 to S_1 is slow, y appears on S_1 before x arrives at S_1. However, if another client c_2 now reads y followed by x on S_1, it can see y but not x, violating causal consistency. To solve this problem, CausalMesh's client library *carries* its local writes. The carried writes are stored in the workflow context, and the scheduler will pass along the context to the subsequent functions within the same workflow. This design decision mirrors other systems that consider client roaming [59, 62].

As a result, when performing a write w, the client attaches its *deps* and *local*. The cache server iterates over all writes in *local*, and adds those not seen before into its I-cache (Figure 7 Lines 9–11). In this example, when the client migrates from S_0 to S_1, it also carries the previous write x to S_1, so that c_2 can see both x and y at S_1.

5.3 Read Path

The read request includes its dependencies, and the server will integrate the dependencies and return the value and its version from C-cache, shown in Figure 7. The integration ensures that the client never reads an older version than its dependencies. For example, a client reads y_1 in the past, where $x_1 \rightarrow y_1$, when it reads x later, it will get a version at least as new as x_1.

If the requested key does not exist in the cache, the client has to read directly from the underlying storage. The cache server, in the background, will add the result to I-cache as if it were written by a client. This value will follow the same propagation chain as a write.

```
# self is a cache server
def ClientRead(self, key, deps):
    self.integrate(deps)
    return self.Consistent[key]


def ClientWrite(self, key, value, deps, local):
    self.vc[self.id] += 1;
    for k, (v, vc, k_deps) in local.items():
        if not self.has_seen(k, vc):
            self.Inconsistent[k].add((v, vc, k_deps))
        deps.add(k, vc)
    self.Inconsistent[key].add((
        self.vc, value, deps
    ))
    self.successor.ServerWrite(
        key, self.vc, value, deps
    )
    return self.vc

def ServerWrite(self, key, vc, value, deps):
    if self is tail:
        self.vc.merge(vc)
        self.integrate(deps)
        self.Consistent[key].merge(
          (vc, value, deps)
        )
    else:
        if not self.has_seen(key, vc):
            self.Inconsistent.merge_all(local)
        self.successor.ServerWrite(
            key, vc, value, deps
        )
```

Figure 7: Pseudo-code for CausalMesh's Server.

```
1  # self is a client
2  def read_txn(self, keys):
3     values, vcs = ClientReadTxn(keys, self.deps)
4     res = []
5     for k, v, vc in zip(keys, values, vcs):
6         if k in self.local \
7           and not (self.local[k] <= vc):
8             return None
9         res.append(v)
10        self.deps[k].merge(vc)
11    return res
```

Figure 8: Pseudo-code for read transaction in CausalMesh's client library. It shows read transaction may fail (Line 8) if keys in the transaction contains the client's own writes.

Each cache server serves read requests independently without consulting other servers or going through the propagation chain, making reads in CausalMesh coordination-free.

5.4 Read Transactions

CausalMesh supports causally-consistent read transactions. A transactional read request includes a set of keys. When the cache server receives such requests, it integrates the dependencies before reading each key from C-cache. As previously mentioned, all versions in C-cache naturally form a consistent view because it is a cut. Furthermore, CausalMesh's read transactions do not communicate with other servers or wait for a specific version to arrive.

Figure 8 presents the pseudo-code for the read transaction in the client library. If the client reads a key that it has written before, similar to read operations, the client library merges the result from the server with its own write. However, the client's own writes may not be part of the same causal cut as the other keys in the request. In this case, the transaction has to abort unless the returned value is at least as new as the one in *local* (Figure 8 Lines 5–8). Aborts are handled by the client library by notifying the scheduler and are opaque to the users. Aborts can only occur in cases where read transactions include keys that have been previously written by the same client, such as when a client writes x and then reads x and y within a transaction. To prevent aborts, developers can rearrange the order of operations by placing writes after read transactions if their keys happen to overlap.

5.5 Achieving Causal+ Consistency

This section provides a rough explanation of how CausalMesh guarantees CC+. We have a full proof and a TLA+ model which is publicly available alongside our code.

In a CC+ system, clients read from a monotonic cut, where the new cut covers the old cut.

Definition 2 (Cut Coverage)**.**

$$S_j \text{ covers } S_i \triangleq \forall k \in S_i, k \in S_j \wedge S_i[k].vc \leq S_j[k].vc$$

Informally, this means that a key can only change from an old version to a new version, or from non-existent to existent. Bolton [5] does not explicitly point out this requirement because it is trivial in their setting of a single cache with no client roaming. In contrast, we have to deal with the fact that it is difficult to create a monotonic cut across multiple machines without coordination because each server processes write requests from clients independently. However, CausalMesh can still ensure that the client sees a monotonic cut with respect to all keys it has accessed.

CausalMesh integrates dependencies before performing a read, which ensures that C-cache covers its dependencies. Those dependencies can always be found on the same cache server without communicating with other servers because the propagation chains guarantee a write can only be visible to clients other than its writer once it reaches the tail of the chain, which implies it has been propagated to all servers. On the other hand, its own writer can always read it, but excludes it from dependencies when sending it to the server, so that it will not be integrated at the server.

Figure 9 shows a concrete example. A client c_1 writes $x \rightarrow y$ to S_0, x and y get propagated from $S_0 \rightarrow S_1 \rightarrow S_2$ and become visible at S_2. Another client c_2 reads y at S_2 and migrates to S_1. When c_2 tries to read x, the cache server integrates y's dependencies, namely x, to S_1's C-cache. On the other hand, when c_1 reads x at S_0, S_0 simply returns an old value without integrating it from I-cache. Then the client library merges it with its local writes.

6 CLIENT LIBRARY

In serverless computing, a client refers to a workflow made up of multiple functions. When a workflow starts, the client library creates two maps, *local* and *deps*. These two maps track the client's own writes and dependencies, respectively, and are carried along the workflow during migration. The client library proxies reads and

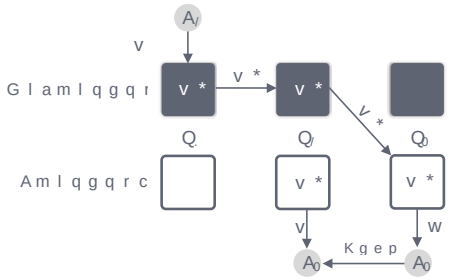

Figure 9: Example of two client c_1 and c_2 in a three-server setup. The arrows between servers are data propagation. The arrows from and to the server are reads and writes, respectively. Figure shows c_2 read c_1's y at S_2 and integrates both x and y after migrating to S_1.

```
1  # self is a client
2  def read(self, key):
3      value, vc = ClientRead(key, self.deps)
4      self.deps.merge(key, vc)
5      if key in self.local:
6          return merge(self.local[key],
7                  {value, vc}).value
8      return value
9
10 def write(self, key, value):
11     vc = ClientWrite(key, value,
12                      self.deps, self.local)
13     self.local[key].merge(value, vc)
```

Figure 10: Pseudo-code for CausalMesh's Client Library.

writes, interacting with the cache server via RPC and providing the necessary metadata. Figure 10 shows the pseudo-code.

Read. When a client performs a read operation, it sends a `ClientRead` request to the designated cache server along with the *deps* map. The cache server responds with w_{cached}, containing the value and its corresponding vector clock. Subsequently, the client library checks its *local* map to determine if it has previously written to the same key. If there has been no prior write, the client library returns the value received from the cache server. However, if there has been a prior write, the client library returns the value by merging the value obtained from the cache server and the value stored in the *local* map, $w_{cached} \cup w_{local}$. Finally, the client library adds the returned version to the *deps* map.

Write. When a client writes, it attaches both the *deps* and *local* maps to the `ClientWrite` request that is sent to the cache server. The cache server then returns a vector clock assigned to the write. The client library adds this vector clock to its *local* map.

7 CAUSALMESH-TCC

CausalMesh only supports read transactions within a single serverless function. However, certain workloads necessitate read-write transactions, specifically when dealing with access-control lists (ACLs), as well as read transactions across multiple serverless functions. To tackle this limitation, we propose an extension of CausalMesh called CausalMesh-TCC, which provides *Transactional Causal Consistency* (TCC) [3, 36] as a stronger consistency level. In CausalMesh-TCC, each workflow is treated as a transaction.

```
1  # self is a cache server
2  def integrate_tcc(self, deps):
3      all_deps = ∀[key, vc] that are transitive
        predecessors of deps (inclusive)
4      for k, vcs in all_deps.items():
5          consistent_versions =
6            self.Inconsistent[k].remove(
7              filter(vc ∈ vcs)
8            )
9          self.vc.merge_all(vcs)
10         new_version = self.Consistent[k].merge_all(
11           consistent_versions
12         )
13         self.Consistent[k].append(new_version)
14
15 def ClientReadTCC(self, key, deps):
16     self.integrate_tcc(deps)
17     for v in self.Consistent[key]:
18         if deps \/ v \/ v.deps is Cut:
19             return (v.value, v.vc)
20     return None
```

Figure 11: Pseudo-code for CausalMesh-TCC's Server.

TCC and CC+ differ in two key aspects. First, TCC ensures atomicity of writes, meaning that all writes from a transaction are either fully visible or not visible at all. In contrast, CC+ does not offer such atomicity guarantees. Second, TCC enforces that all reads within a transaction must originate from the same causal cut. For example, if a client reads $x = 0$, all subsequent reads of x within the same transaction will also return 0. On the other hand, CC+ allows for the possibility of reading newer values in subsequent read operations by reading from a monotonic cut.

To enforce atomic writes, CausalMesh-TCC's client library saves writes in a buffer and returns to the client immediately. The writes are then sent to the server in a batch at the end of the workflow. If the workflow has multiple leaves, it will add a dummy sink function that joins all leaves. During dependency integration, the cache server will integrate all writes in the same batch atomically.

To make all reads come from the same cut, CausalMesh-TCC extends C-cache to be a map from a key to a list of tuples that includes the value, VC, and deps.

$$\textit{C-cache} \quad := \quad \{\textit{Key} \mapsto [\ (\textit{Value}, \textit{VC}, \textit{Deps})\ ]\}$$

Figure 11 shows CausalMesh-TCC's pseudo-code on the server side. During dependency integration, rather than updating the value in C-cache in place as CausalMesh does, the cache server in CausalMesh-TCC creates a new version and appends it to the list so that the list contains multiple versions for each key (Figure 11 Lines 10–12). Upon receiving a read request, the cache server returns the oldest version from this list that, when combined with the previous read set, forms a cut—thus adhering to TCC (Figure 11 Line 18). If no such version is found, the workflow has to be aborted and retried. In the case of multiple parallel functions within the workflow, CausalMesh-TCC runs a validation phase that checks if the union of the read sets from these functions forms a cut. If it does not, the workflow is aborted and retried, which, similar to read transactions in CausalMesh, is opaque to developers.

We utilize a ring buffer to implement the version list, where the size of the buffer is a tunable parameter which determines the trade-off between the abort rate and memory usage.

	Consistency	Unk. ReadSet	Coordination Cost	Read / Write	Abort Free	Visibility
CausalMesh	CC+	Yes	0 RTT	0 RTT / 1 RTT to DB	Yes	N × RTT
CausalMesh-TCC	TCC	Yes	0 RTT	0 RTT / 1 RTT to DB	No	N × RTT
HydroCache-Con	TCC	No	2 RTTs	0 RTT / 1 RTT to DB	Yes	refresh period
HydroCache-Opt	TCC	No*	0 RTT ~ 2*N* RTT	0 RTT / 1 RTT to DB	No	refresh period
FaaSTCC	TCC	Yes	0 RTT ~ 2*N* RTT	0 RTT / 1 RTT to DB	No	refresh period

Figure 12: Comparison between CausalMesh, CausalMesh-TCC, HydroCache-Con, and HydroCache-Opt. *N* is the number of servers. Unknown ReadSet means that the read set does not need to be known ahead of time, which is needed for supporting dynamic workflows. HydroCache-Opt's Unknown ReadSet field is No* because it supports partially dynamic workflows (§11). In HydroCache and FaaSTCC, writes become visible after a refresh period, set to 100ms and 50ms in the original papers.

8 IMPLEMENTATION

We implemented CausalMesh, CausalMesh-TCC, and two baselines.

Baselines: HydroCache and FaaSTCC. HydroCache [59] is currently the state-of-the-art serverless caching system. It guarantees Transactional Causal Consistency (TCC). It has two versions: a conservative version (HydroCache-Con) and an optimistic version (HydroCache-Opt). In HydroCache-Con, prior to execution, a centralized scheduler distributes the read set to all candidate cache servers and blocks until it has received their responses with their respective snapshots of the read set. The scheduler then uses these responses to construct a consistent causal cut and send it back to all cache servers. In HydroCache-Opt, the scheduler checks for causal violations between two serverless functions during execution. If a violation is detected, the entire workflow is aborted and retried. FaaSTCC improves HydroCache-Opt by replacing dependency metadata with a *snapshot interval* which is a time frame where reads are valid, however, it requires the underlying storage system to provide a global timestamp. Figure 12 shows the full comparison between them.

Prototype. We have an implementation of CausalMesh for serverless applications that are written in Go. The cache server for CausalMesh is written in Rust. As HydroCache is not open-source, we also implement it in Rust and ensure that it achieves the same or better performance as the results given in the HydroCache paper [59]. The cache system in total consists of roughly 5K lines of Rust. The client library for CausalMesh, written in Go, consists of approximately 400 lines. All systems use grpc [21] for communication.

HydroCache and FaaSTCC have a background thread that uses double-buffered hash tables [19] to refresh the cache: the background thread updates one table while read handlers in the main thread read the other; an atomic pointer swap exposes new writes ensuring that the refresh does not affect readers in the critical path.

9 EVALUATION

CausalMesh helps serverless developers rely on caches without the complexity of weak consistency semantics. To see how well CausalMesh works, we answer the following questions:

- What is CausalMesh's performance on micro-benchmarks and how does it compare to prior serverless cache systems? (§9.2)
- What overhead does CausalMesh introduce? (§9.3)
- How does CausalMesh scale with server count? (§9.4)
- What are the latency and throughput of representative applications running on CausalMesh? (§9.5)
- How long does it take for a write in CausalMesh to become visible on other servers? (§9.6)

9.1 Experimental Setup

In our evaluation, we used CloudLab [10] m510 machines with 8-core 2.0 GHz CPUs, 64GB of RAM, 256GB NVMe SSDs, and 10GB NICs. The typical round-trip time (RTT) between servers is 0.15ms. We use Nightcore [27] as the serverless runtime. It uses two cores and 8 workers per machine for all experiments except for the experiments that evaluate the real-world applications (§9.5) which use five cores and 16 workers. We run a Redis [46] server in *append-only* mode as our underlying storage, with a custom conflict resolution layer on top of it. We add an artificial latency of 5ms to the Redis server using `netem` to emulate remote storage so that it has similar latency to those found in public clouds (e.g. AWS Lambda with DynamoDB). For all evaluations except the scalability evaluation, we used a setup with three workers, consisting of five machines: one machine running Redis as the database, one machine running a client and a scheduler, and three machines each running a Nightcore instance and a cache (either CausalMesh or HydroCache).

CausalMesh uses a single thread. The ring buffer size in CausalMesh-TCC is set to one to have a comparable memory footprint. Both versions of HydroCache and FaaSTCC use an additional thread to run a background task that refreshes the cache by merging new updates from the database. The refresh period was set to 100ms and 50ms, respectively, as in the original papers. Unless otherwise specified, the caches were pre-warmed to remove the overhead of data retrieval from the persistent database.

We use wrk2 [56], which is a constant-load HTTP workload generator and measurement tool to obtain the latency and throughput numbers. Each workload runs for a total of 90 seconds, with the first 30 seconds serving as a warm-up period. The results of the subsequent 60 seconds are reported.

9.2 Micro-Benchmark

In our micro-benchmark, we evaluate a three-function serverless workflow that aligns with the one described in the HydroCache paper [59]. The first two functions in the workflow read three keys, while the last function writes to a single key. The keys are sampled from a pool of 1,000,000 keys, following a Zipfian distribution with a coefficient of 1.0. The value is an 8-byte string.

Results. Figure 13 shows the results of our micro-benchmark. Compared to HydroCache and FaaSTCC, CausalMesh's throughput is

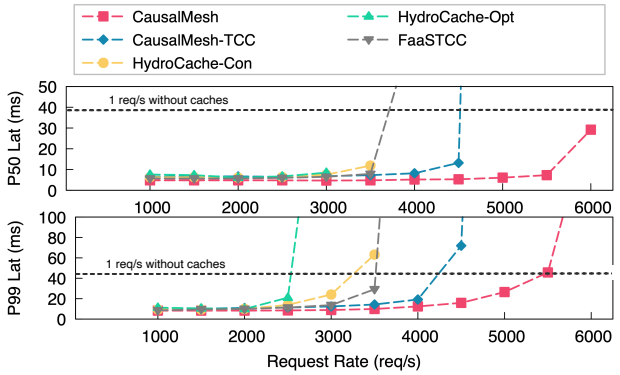

Figure 13: Median and tail response time and throughput of CausalMesh, CausalMesh-TCC, HydroCache-Con, and HydroCache-Opt in the micro-benchmark.

1.57×–2.2× higher. In terms of median latency, CausalMesh's is 7%–59% lower than HydroCache-Con, 15%–44% lower than HydroCache-Opt and 5%–38% lower than FaaSTCC. Regarding tail latency, CausalMesh achieves up to 85%, 97% and 54% lower latency than HydroCache-Con, HydroCache-Opt and FaaSTCC, respectively. CausalMesh-TCC achieves comparable latency to HydroCache and FaaSTCC but much better throughput: up to 1.35× and 1.6× higher compared to HydroCache and FaaSTCC. For comparison, Figure 13 also shows a lower bound on the latency of the workflow when accessing the database directly without caches (horizontal dotted line).

Takeaway. Caches are critical in keeping the latency of stateful serverless functions low (up to 4× lower than the baseline without caches shown as a dotted horizontal bar). CausalMesh-TCC provides the same consistency guarantees and supports the same applications as state-of-the-art serverless caches but achieves considerably higher throughput. CausalMesh further achieves lower latency at the tail if the workflow does not require read/write transactions within a function or cross-function transactions.

9.3 Resource Overhead

To quantify the cost of running CausalMesh, we send requests to the serverless workflow at a constant rate of 1000 requests/second, using the same workload as that in the micro-benchmark, and we analyze the overhead from two sources: CPU and memory usage. For memory, we differentiate between the total usage of the cache server and the size of internal metadata (i.e., how much additional data is required to ensure correctness). In CausalMesh, metadata includes the contents of the I-cache, while in CausalMesh-TCC, it includes the I-cache and any dependencies in the C-cache. In FaaSTCC, it includes the interval timestamps. In HydroCache, it includes all dependencies of any element in the cache. Note that, absent a separate garbage collection protocol, HydroCache's metadata will grow infinitely (§11); we measure the size of HydroCache's metadata after 1 minute.

Result. Figure 14 shows that CausalMesh has 57% lower CPU consumption compared to HydroCache-Con, 23% lower compared to HydroCache-Opt and 26% lower compared to FaaSTCC. For CausalMesh-TCC, the CPU consumption is 46% lower than HydroCache-Con; it is similar to HydroCache-Opt and FaaSTCC. In terms of

	CausalMesh / -TCC	HydroCache-Con/Opt	FaaSTCC
CPU (%)	**42.0** / 53.3	98.3 / 54.4	52.9
Memory (MB)	**54.4** / 57.4	69.3 / 69.6	61.0
Metadata (KB)	35.1 / 69.9	45.3 / 99.8	**28.8**

Figure 14: CPU and memory usage of cache servers in the micro-benchmark. The request rate is 1000 requests per second. Our system uses a single thread, while HydroCache and FaaSTCC use two threads. Metadata is the additional data required by each protocol to ensure correctness.

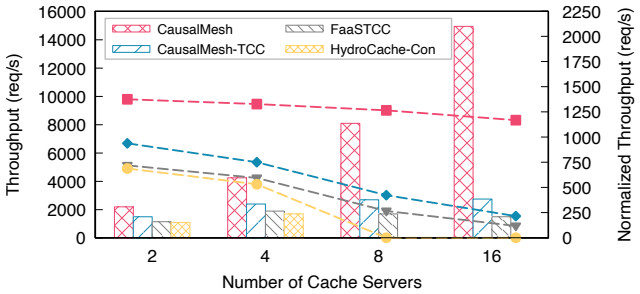

Figure 15: The histogram with y-axis on the left depicts the throughput as we vary the number of servers. The line plot with y-axis on the right shows the normalized throughput by dividing the throughput by the number of servers.

memory, CausalMesh consumes up to 20% less memory than HydroCache and FaaSTCC, while CausalMesh-TCC uses up to 17% less memory. The size of metadata in CausalMesh and CausalMesh-TCC are 2.2% and 4.5% of the total data set, respectively.

Takeaway. CausalMesh(-TCC) incurs less CPU and memory overhead compared to HydroCache and FaaSTCC. The metadata of CausalMesh and FaaSTCC stays low and stable while HydroCache's metadata grows over time. FaaSTCC reduces a great amount of metadata by delegating the job of assigning versions to the underlying storage system.

9.4 Effect of the Number of Caches

To evaluate how CausalMesh scales with the the number of servers, we conduct experiments with 2–16 servers (the same order of magnitude as AWS DAX's maximum of 11 cache nodes). More servers result in more concurrent clients. We issue requests in increments of 50 req/s until the system is nearly saturated, which we determine by observing a tail latency longer than 10ms. Each function randomly reads two keys and writes one key.

Results. We normalize the results by dividing the raw throughput by the number of servers. Figure 15 includes a histogram illustrating the raw throughput and a line plot depicting the normalized throughput. It shows that CausalMesh's normalized throughput is nearly constant, which means CausalMesh scales almost linearly with respect to the number of servers. On the other hand, CausalMesh-TCC reaches saturation at around 2800 request/second due to increased contention. FaaSTCC experiences throughput degradation as the number of servers increases. CausalMesh-TCC achieves 1.3×–1.8× higher throughput than FaaSTCC. Both HydroCache-Opt and HydroCache-Con perform worse than both FaaSTCC and

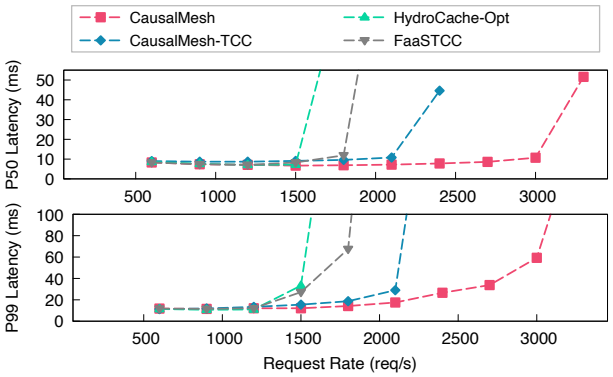

Figure 16: Comparison of CausalMesh, HydroCache, and FaaSTCC in terms of median and tail response time and throughput in a mixed workload that has contention between reads and writes.

CausalMesh-TCC; HydroCache does not scale to 8 servers or beyond because the cost of coordinating between those servers and pulling dependencies is far too high. HydroCache-Opt performs even worse, which is why we do not include it in the figure.

Takeaway. Developers should use CausalMesh whenever allowed, as it has better performance when there's more cache servers; developers should only use CausalMesh-TCC when read transactions across multiple serverless functions or read-write transactions are necessary. We discuss scalability further in Section 11.

9.5 Movie Review Service

We evaluate CausalMesh's performance on the movie review service described in DeathStarBench [15, 20]. In this service, users create accounts, read reviews, view the plot and cast of movies, and write movie reviews. We use Beldi's implementation of this app [63] which is a V-shape workflow of 13 serverless functions.

We evaluate a mixed workload, consisting of 50% ComposeReview and 50% ReadReview. ComposeReview generates a review for a random user and movie, and then saves the review ID to the profiles of both the movie and the user. ReadReview involves two functions. First, it reads the profile of a movie to retrieve all associated review IDs. Then, it reads the contents of the reviews using those IDs. It is worth noting that HydroCache-Con cannot support this type of workload as it requires prior knowledge of the keys.

Results. Figure 16 shows that both HydroCache-Opt and FaaSTCC start experiencing high tail latency at around 1,500 req/s. In contrast, CausalMesh achieves 2× higher throughput while reducing median latency by up to 10% and tail latency by up to 64% before HydroCache-Opt and FaaSTCC become saturated. CausalMesh-TCC achieves up to 1.35× higher throughput and similar latency.

Takeaway. Both CausalMesh and CausalMesh-TCC outperform HydroCache and FaaSTCC in throughput for real-world applications. As Causal+ consistency is sufficient for many applications, including the movie review service above, CausalMesh significantly reduces the latency when compared to the others.

Num Servers	3	4	5	6	7	8
Inconsistency Window (μs)	1259	1586	2030	2360	2799	3138
Marginal (μs)	-	327	444	330	439	339

Figure 17: Relationship between the number of servers and visibility, as measured by the inconsistency window [7]. Marginal inconsistencies indicate the increase in delay that the system experiences when an additional server is added.

9.6 Visibility

To evaluate the visibility of CausalMesh, we use the concept of observed inconsistency window [7]. We compute the inconsistency window using the following steps:

(1) Create a timestamp, t_1.
(2) Write to a server and save the version received from it.
(3) Create another timestamp, t_2.
(4) Poll the result from the tail server until it sees the saved version.
(5) Log the elapsed time from the mean of t_1 and t_2.

We vary the number of servers and record the effect this has on the inconsistency window. Additionally, we calculate the marginal inconsistency window by subtracting the inconsistency window of N servers from that of $N-1$ servers.

Results. Figure 17 shows that the marginal inconsistency window remains stable between 300 and 450 μs, indicating that the visibility is nearly proportional to the number of servers in the system.

Takeaway. In CausalMesh and CausalMesh-TCC, the inconsistent window grows as the number of servers increases. CausalMesh exhibits a significantly lower inconsistency window (e.g., 3ms in an eight-server configuration). In contrast, HydroCache, FaaSTCC, and other systems with background refreshing can have an inconsistency window up to the refresh period, which is 100ms in HydroCache and 50ms in FaaSTCC.

9.7 How CausalMesh-TCC Outperforms Others

CausalMesh-TCC's performance advantages stem from several key factors. Figure 12 lists the characteristics of CausalMesh, HydroCache, and FaaSTCC. One visible advantage of CausalMesh-TCC is the absence of coordination costs, along with better data freshness. This is a crucial aspect, as delays in data visibility increase the likelihood of missing versions when a client migrates to a new server. Another significant factor is that CausalMesh eliminates the need for a background thread that periodically updates data, a feature present in both HydroCache and FaaSTCC. The background thread is used to subscribe to the underlying database and apply new writes to the cache data structure. The new writes must be applied atomically or in a causal order to maintain correctness, thus creating contention with the request-serving threads of a cache server. We managed to considerably reduce this contention by implementing a double-buffered hash table (§8) that improved upon the published designs. However, despite our efforts, when the request rate is high, our optimized versions of prior work are still hamstrung by significant overheads.

10 RELATED WORK

Causal consistency. There is a large body of work on causally consistent systems [3–6, 17, 18, 36, 37, 41, 44, 53, 60, 62]. Several systems (such as Bolt-on [5], SwiftCloud [62], and Occult [41]) have explored the idea of reading safe but stale data to achieve causally consistent plus (CC+) guarantees. However, these systems are not designed to guarantee CC+ across multiple caches. They either do not support client roaming or if one deploys them in a setting with client roaming they must block when dependencies are not satisfied. As a result, they are not suitable for serverless computing where mobility is a common case. CausalMesh, on the other hand, is specifically designed to support client roaming and guarantee CC+ across multiple caches.

Chain Reaction [4] introduces the idea of using chain replication for causal consistency, but we have applied this concept in a different manner. In their single-datacenter setting, keys are sharded to different chains using consistent hashing, and reads are forwarded to the server that holds the key. In their multi-datacenter setting, the heads of the chains in different datacenters are connected. If the required version is missing, the request will be forwarded to other datacenters or blocked until the version becomes available. In contrast, CausalMesh allows a server to serve requests without coordinating with other servers, and the chains are interleaved (one's head to another's tail) rather than parallel (one's head to another's head), which enables coordination-free reads.

Serverless caching. Other systems have also noted the high cost of remote data access in serverless computing. HydroCache [59] and FaasTCC [39] are the most closely related works. One difference between CausalMesh and these works is that CausalMesh is non-blocking and has advantages such as supporting fully dynamic workflows (§11), while these works must block when a specific version is missing, which nullifies many of the benefits of caching. In terms of requirements, CausalMesh and HydroCache can run on top of existing databases, whereas FaaSTCC needs the underlying storage to assign a causal timestamp for each write, which is not supported by most production databases.

Also related is Faa$T [47], which offers strong consistency but requires validating the version at the remote storage to ensure that the cached data is up to date, introducing high latency. Cloudburst [51] is a serverless platform that supports a causal cache by using lattice data types provided by Anna KVS [57, 58] with its custom API. Other ephemeral storage and caches, like Pocket [31], InfiniCache [55], and Locus [45] are designed for data-intensive serverless apps like data analytics and lack consistency guarantees.

Serverless scheduling and orchestration. Finally, we note that CausalMesh is complementary to recent work on more efficient scheduling for serverless computing (e.g., Kaffes [29], Fifer [22], Pheromone [61], Unum [35], Cypress [8], Hermod [30], Palette [1], and Caerus [64]). In particular, CausalMesh adds an extra layer to the efficiency of function placement that scheduling algorithms can leverage to improve the performance of workflow execution while ensuring causal consistency. Integration may differ slightly for different schedulers, but we leave a full exploration of the optimal co-design of the end-to-end caching, instance provisioning, and request scheduling infrastructure to future work.

11 DISCUSSION

Dynamic workflows. Recent analyses [25, 38] show that workflows and operation sets tend to be highly dynamic. For example, consider a function that reads the value of key k_1 and then uses the value of k_1 as the key for a subsequent read operation. HydroCache cannot support this type of function. HydroCache-Con only supports static workflows because it requires knowledge of the read set before execution. HydroCache-Opt can support partially dynamic workflows, but it still requires knowledge of the read set of each function before execution and then performs a validation phase to check for causal violations. In contrast, both CausalMesh and CausalMesh-TCC support arbitrary dynamic workflows.

Metadata and garbage collection. In dependency-tracking systems, the accumulation of dependency metadata can cause system slowdown over time. To mitigate this issue, HydroCache implements periodic garbage collection (GC) using a background consensus protocol to clear the dependency metadata. In contrast, CausalMesh and CausalMesh-TCC clear unnecessary metadata seamlessly while processing requests, without the need for dedicated GC processes. Specifically, In CausalMesh, dependencies are discarded during dependency integration. In CausalMesh-TCC, C-cache is a ring buffer that automatically removes both old values along with their associated dependencies when it is full.

Scale to more cache nodes. The usage of vector clocks can potentially lead to performance issues if the vector clock becomes huge, e.g., supporting over 1000 cache nodes. However, it is generally not encountered in serverless setups where multiple workers can be routed to the same cache server. For example, DynamoDB DAX is designed to allow up to 11 cache nodes; CosmosDB's Integrated Cache allows a maximum of 5 cache nodes. Should the system go to an extreme scale in the future and the vector clock becomes a bottleneck, we expect to utilize a garbage collection scheme to practically trim the vector clock, such as Dynamo did [16].

Cache eviction strategies. Cache eviction is an orthogonal problem and thus is not the focus of our work. The design of dual cache allows it to benefit from any eviction policies. The only additional requirement is upon the eviction of a key, all keys that depend on it are also evicted so that C-cache remains a cut.

12 CONCLUSION

This paper presents CausalMesh, the first cache system to support coordination-free and abort-free causal read/write operations when clients (workflows) move from server to server. It also presents CausalMesh-TCC that supports transactional causal consistency within a workflow. They enable developers to build applications that take advantage of both the scalability of serverless computing and the low latency of a local cache. Our evaluation shows that CausalMesh(-TCC) achieves significantly better performance than the current state-of-the-art of consistent caches and is a great addition to this ecosystem.

ACKNOWLEDGMENTS

We thank the reviewers for their thoughtful feedback. This project was funded in part by NSF grants CCF 2124184, CNS 2107147, CNS 2321726, CNS 2130590, CNS 2238768, and CNS 2321725.

REFERENCES

[1] Mania Abdi, Samuel Ginzburg, Xiayue Charles Lin, Jose Faleiro, Gohar Irfan Chaudhry, Inigo Goiri, Ricardo Bianchini, Daniel S Berger, and Rodrigo Fonseca. Palette load balancing: Locality hints for serverless functions. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2023.

[2] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[3] Deepthi Devaki Akkoorath, Alejandro Z Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno Preguiça, and Marc Shapiro. Cure: Strong semantics meets high availability and low latency. In *International Conference on Distributed Computing Systems (ICDCS)*, 2016.

[4] Sérgio Almeida, João Leitão, and Luís Rodrigues. Chainreaction: a causal+ consistent datastore based on chain replication. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2013.

[5] Peter Bailis, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Bolt-on causal consistency. In *Proceedings of the ACM SIGMOD Conference (SIGMOD)*, 2013.

[6] Nalini Moti Belaramani, Michael Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. Practi replication. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2006.

[7] David Bermbach and Stefan Tai. Eventual consistency: How soon is eventual? an evaluation of amazon s3's consistency behavior. In *Workshop on Middleware for Service Oriented Computing (MW4SOC)*, 2011.

[8] Vivek M Bhasi, Jashwant Raj Gunasekaran, Aakash Sharma, Mahmut Taylan Kandemir, and Chita Das. Cypress: input size-sensitive container provisioning and request scheduling for serverless platforms. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, 2022.

[9] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S Meiklejohn. Durable functions: semantics for stateful serverless. *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2021.

[10] Cloudlab. https://cloudlab.us.

[11] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2008.

[12] cosmosdb. https://azure.microsoft.com/en-us/products/cosmos-db/.

[13] couchbase. https://www.couchbase.com/.

[14] Dax and dynamodb consistency models. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/DAX.consistency.html.

[15] DeathStarBench. https://github.com/delimitrou/DeathStarBench/.

[16] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[17] Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, 2013.

[18] Jiaqing Du, Călin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, 2014.

[19] evmap: A lock-free, eventually consistent, concurrent multi-value map. https://github.com/jonhoo/evmap.

[20] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2019.

[21] grpc. https://grpc.io/.

[22] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan C Nachiappan, Mahmut Taylan Kandemir, and Chita R Das. Fifer: Tackling resource underutilization in the serverless era. In *Proceedings of the ACM/IFIP/USENIX International Middleware Conference (Middleware)*, 2020.

[23] Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. In *Conference on Innovative Data Systems Research (CIDR)*, 2019.

[24] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2010.

[25] Darby Huye, Yuri Shkuro, and Raja R Sambasivan. Lifting the veil on meta's microservice architecture: Analyses of topology and request workflows. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2023.

[26] Zhipeng Jia and Emmett Witchel. Boki: Stateful serverless computing with shared logs. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[27] Zhipeng Jia and Emmett Witchel. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[28] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, 2017.

[29] Kostis Kaffes, Neeraja J Yadwadkar, and Christos Kozyrakis. Centralized core-granular scheduling for serverless functions. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, 2019.

[30] Kostis Kaffes, Neeraja J Yadwadkar, and Christos Kozyrakis. Hermod: principled and practical scheduling for serverless functions. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, 2022.

[31] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[32] How long does AWS Lambda keep your idle functions around before a cold start? https://acloudguru.com/blog/engineering/how-long-does-aws-lambda-keep-your-idle-functions-around-before-a-cold-start.

[33] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Communications of the ACM*, 2019.

[34] Zijun Li, Linsong Guo, Jiagan Cheng, Quan Chen, BingSheng He, and Minyi Guo. The serverless computing survey: A technical primer for design architecture. *ACM Computing Surveys*, 2022.

[35] David H. Liu, Shadi Noghabi, Sebastian Burckhardt, and Amit Levy. Doing more with less: Orchestrating serverless applications without an orchestrator. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2023.

[36] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[37] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[38] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, 2021.

[39] Taras Lykhenko, Rafael Soares, and Luis Rodrigues. Faastcc: efficient transactional causal consistency for serverless computing. In *Proceedings of the ACM/IFIP/USENIX International Middleware Conference (Middleware)*, 2021.

[40] Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin. Consistency, availability. Technical report, and convergence. Technical Report TR-11-22, Univ. Texas at Austin, 2011.

[41] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. I can't believe it's not causal! scalable causal consistency with no slowdown cascades. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[42] mongodb. https://www.mongodb.com/.

[43] Ruoming Pang, Ramon Caceres, Mike Burrows, Zhifeng Chen, Pratik Dave, Nathan Germer, Alexander Golynski, Kevin Graney, Nina Kang, Lea Kissner, et al. Zanzibar: Google's consistent, global authorization system. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2019.

[44] Karin Petersen, Mike J Spreitzer, Douglas B Terry, Marvin M Theimer, and Alan J Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 1997.

[45] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[46] redis. https://redis.io/.

[47] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. Faa$t: A transparent auto-scaling cache for serverless applications. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, 2021.

[48] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J Yadwadkar, Raluca Ada Popa, Joseph E Gonzalez, Ion Stoica, and David A Patterson. What serverless computing is and should become: The next phase of cloud computing. *Communications of the ACM*, 2021.

[49] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. Serverless computing: a survey of opportunities, challenges, and applications. *ACM Computing Surveys*, 2022.

[50] Mukesh Singhal and Ajay Kshemkalyani. An efficient implementation of vector clocks. *Information Processing Letters*, 1992.

[51] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E Gonzalez, Joseph M Hellerstein, and Alexey Tumanov. Cloudburst: stateful functions-as-a-service. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2020.

[52] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, et al. Twine: A unified cluster management system for shared infrastructure. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[53] Misha Tyulenev, Andy Schwerin, Asya Kamsky, Randolph Tan, Alyson Cabral, and Jack Mulrow. Implementation of cluster-wide logical clock and causal consistency in mongodb. In *Proceedings of the ACM SIGMOD Conference (SIGMOD)*, 2019.

[54] Robbert Van Renesse and Fred B Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[55] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. Infinicache: Exploiting ephemeral serverless functions to build a cost-effective memory cache. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*,

2020.

[56] wrk2: A constant throughput, correct latency recording variant of wrk. https://github.com/giltene/wrk2.

[57] Chenggang Wu, Jose M Faleiro, Yihan Lin, and Joseph M Hellerstein. Anna: A kvs for any scale. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2019.

[58] Chenggang Wu, Vikram Sreekanti, and Joseph M Hellerstein. Autoscaling tiered cloud storage in anna. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2019.

[59] Chenggang Wu, Vikram Sreekanti, and Joseph M Hellerstein. Transactional causal consistency for serverless computing. In *Proceedings of the ACM SIGMOD Conference (SIGMOD)*, 2020.

[60] Haifeng Yu. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2000.

[61] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. Restructuring serverless computing with data-centric function orchestration. arXiv:2109.13492, 2021. https://arxiv.org/abs/2109.13492.

[62] Marek Zawirski, Nuno Preguiça, Sérgio Duarte, Annette Bieniusa, Valter Balegas, and Marc Shapiro. Write fast, read in the past: Causal consistency for client-side applications. In *Proceedings of the ACM/IFIP/USENIX International Middleware Conference (Middleware)*, 2015.

[63] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. Fault-tolerant and transactional stateful serverless workflows. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[64] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. Caerus: Nimble task scheduling for serverless analytics. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021.