CausalMesh: A Causal Cache for Stateful Serverless Computing

Haoran Zhang* Shuai Mu† Sebastian Angel* Vincent Liu*
*University of Pennsylvania †Stony Brook University

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.

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 state. Existing solutions [16, 27, 34, 14, 4] 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 [11, 26] (e.g., 10–20 ms to read or write to DynamoDB), recent works [28, 18] 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.

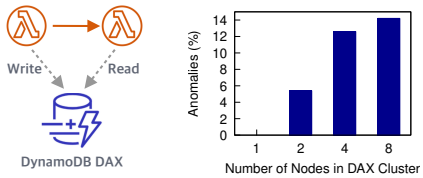

Figure 1: Anomalies rate of a two-function workflow, where the second function reads the data written by the first. The workflow runs on AWS Lambda and using DynamoDB Accelerator (DAX) as the cache. There are no anomalies when utilizing a single cache node, but it lacks scalability.

More so than in traditional applications and data stores, in serverless applications, 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 workflow that contains two serverless functions that run one after the other and that access the same state: the first function blocks the author of a post, and the second function regenerates a timeline for the user that should exclude the blocked sources.

Because these functions themselves are supposed to be stateless, cloud providers will frequently schedule these functions on different machines. Normally, this flexibility is a major advantage to scheduling efficiency and resource provisioning, but here, it means that the functions 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), and the user may see posts from the blocked source. 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 [32] and FaaSTCC [21], aim to address this issue by introducing a *causal cache*: a set of caches that collectively guarantee causal consistency. Both HydroCache and FaaSTCC provide trans-

 This is a simplified version of the paper entitled CausalMesh: A Causal Cache for Stateful Serverless Computing, published in PVLDB, Vol. 17, No. 13, ISSN 2150-8097. DOI: https://doi.org/10.14778/3704965.3704969

actional causal consistency [2, 19] which they adopt from traditional causal databases. The main technical challenge present in serverless computing that is addressed by them and other prior works [33, 23] 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.

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 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*—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 [15], a serverless runtime platform. We then use the key-value store interface provided by CausalMesh's client library to implement real-world applications consisting of workflows with 13 serverless functions (§10.2) to evaluate the performance.

To put our results in context, we compare CausalMesh and CausalMesh-TCC with HydroCache [32] and FaaSTCC [21], recent caches for serverless workflows that aim to play a similar role. In a nutshell, CausalMesh is faster: up to 59% reduction in median latency, up to 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), CausalMesh still achieves 1.35–1.6× higher throughput and comparable latency than HydroCache and FaaSTCC.

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 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 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. Three aspects of this architecture are particularly salient to CausalMesh and differentiate serverless from traditional applications interacting with traditional databases and data stores:

(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. In cases when a request for a function arrives and finds that all existing instances of that function are busy, the provider deploys 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. 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 [30]. 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 [1].

(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 that two requests will be executed in the same worker, whether the requests are for the same function or different functions in the same workflow. To handle stateful serverless functions, external storage services, such as 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) in write-through mode as an example, 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 seconds to complete. Consequently, two functions may obtain different values when accessing the same key from the same DAX cluster, depending on which cache node each function accesses.

2.2 Consistency Goals

As we describe above, caching the remote state at each provisioned instance is critical to reduce latency, as it allows functions to access the state immediately if the data is in the cache. But to maintain consistency across an entire workflow, traditional caches 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 [32] 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 [8]), but as we alluded to in the introduction, writing serverless workflows with weak consistency is hard. To strike a balance between performance and meaningful consistency semantics, we settle on *causal+ consistency* (CC+). Recent work [22] has shown that no model stronger than causal consistency is achievable with high availability, making it ideal for our coordination-free goal.

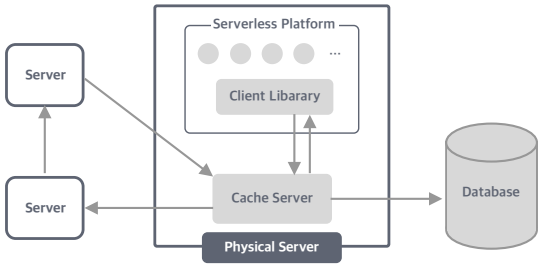

Figure 2: Architecture of CausalMesh with three servers.

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 [34] and was ported from the microservice library DeathStarBench [9]. 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 [5, 19, 25].

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 application data. CausalMesh supports any database that admits custom conflict resolution policies to resolve concurrent updates (e.g., Azure CosmosDB [6], Couchbase [7], and MongoDB [24]).

CausalMesh. CausalMesh is a middleware that sits between the serverless platform and the backend databases. It con-

tains 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 servers run in a 1-to-1 correspondence with physical machines. All database requests from functions that originate from a given machine 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 [12]).

End-to-end procedure. 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 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 via CausalMesh's library.
4. When a function calls another function, the scheduler gathers the state updates 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. When a function returns, its state updates are also collected by the scheduler.
5. Repeat until the workflow is complete.

3.2 CC+ in CausalMesh

CausalMesh provides *causal consistency with convergent conflict handling* [19] or CC+ with two components:

Vector Clocks (VC) [29]. Used to identify different versions of an object and capture the happens-before relation [17] 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). Used to track causal relationships across different keys. They are stored as a map from a 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 $x \to y$ (x happens before y, y depends on x) and $y \to z$, z's dependency will contain y but not x. Dependencies can be merged using the same mechanism as vector clocks.

```
1  # self is a cache server
2  def integrate(self, deps):
3      all_deps = {∀[k, {vc₁, vc₂, ...}] | [k, vc_i] is
        transitive predecessor 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: Pseudocode for dependency integration.

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 functions as a multi-version key-value store, storing versions that the cache server is unsure have been synchronized to all servers. As a result, the contents of I-cache are not safe 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 processed 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 updat-

ing 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* $\iff \forall x \in S, \forall y \in x.deps, \exists y' \in S \mid y.key = y'.key \wedge (y' = y \vee y \rightarrow y')$

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 interface 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 first 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, if the GVC for server S_0 is $[7, 5, 2]$, then when it 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 values 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. 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, but is different from, Chain Replication [31]. Each $S_i \rightarrow S_{(i+1) \bmod N} \rightarrow ... \rightarrow S_{(i+N-1) \bmod N}$ 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 are 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 asynchronous propagation happens after the server responds to the client and is off the critical path.

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 `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_0 before x arrives at S_1. However, if another client c_2 now reads y on S_0, 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 [32, 33].

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, if a client reads y_1 in the past, where $x_1 \rightarrow y_1$, then 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.

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

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 developers do not use these functions.

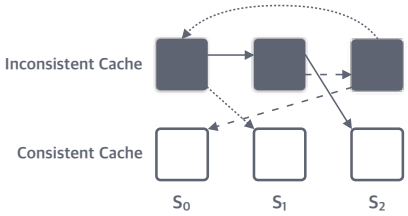

Figure 5: Propagation chain in a three-server setup. The cache servers S_0, S_1, and S_2 respectively serve as the head nodes for the solid, dashed, and dotted chains.

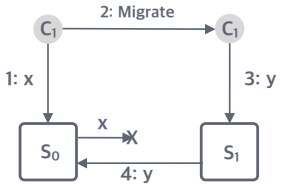

Figure 6: Without carrying and sending its own write to the cache, the client may fail to read causally consistent values. The text on the arrows contains the step number and the data being transferred.

```
# 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: Pseudocode for CausalMesh's server.

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.

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`. 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.

6 Client Library

In serverless, the "client" in our setting is 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 writes, interacting with the cache server via RPC and providing the necessary metadata. Figure 8 shows the pseudocode.

Read. Issues a `ClientRead` request to the designated cache server along with `deps`. The cache server responds with w_{cached}, containing the value and its corresponding vector clock. Subsequently, the client library checks `local` 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 `local`, $w_{cached} \cup w_{\texttt{local}}$. Finally, the client library adds the returned version to `deps`.

Write. Attaches `deps` and `local` to the `ClientWrite` request 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 supports read transactions within a single serverless function. However, certain workloads need read-write transactions (e.g., when dealing with access-control lists)

```
# self is a client
def read(self, key):
    value, vc = ClientRead(key, self.deps)
    self.deps.merge(key, vc)
    if key in self.local:
        return merge(self.local[key],
                {value, vc}).value
    return value

def write(self, key, value):
    vc = ClientWrite(key, value,
                     self.deps, self.local)
    self.local[key].merge(value, vc)
```

Figure 8: Pseudocode for CausalMesh's client library.

and read transactions across multiple serverless functions. This raises a question: what kind of transactional isolation semantics should CausalMesh provide that satisfy our goals of high performance and at least causal consistency. Two potential options are *Transactional Causal Consistency* (TCC) [2, 19] and *Snapshot Isolation* (SI) [3].

TCC can be seen as an extension of CC+ to the transactional context. 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 reads by reading from a monotonic cut.

Compared with Snapshot Isolation (SI), TCC may lead to reading stale data whereas SI ensures that reads always reflect all committed transactions. Additionally, SI prevents write-write conflicts. It has a first-committer-wins feature [3] to abort concurrent writes and serialize all committed writes. In contrast, TCC allows concurrent writes and resolves conflicts by merging versions. Consequently, SI is a stronger isolation level but prior works have shown that it is an order of magnitude slower than TCC [32], with notably higher tail latencies. Consistent with our goal of obtaining good performance, we therefore settle on using TCC to support transactional semantics in CausalMesh.

In particular, we implement a variant of CausalMesh that we call CausalMesh-TCC. In CausalMesh-TCC, each workflow is treated as a transaction. 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; to do this, we add a dummy sink function at the end of the workflow. 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.

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

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. 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. If no such version is found, the workflow aborts and retries. 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 aborts and retries.

8 Correctness

The full version of this paper [36] and our codebase [35] contain proofs and TLA+ models of CausalMesh.

9 Implementation

We implemented CausalMesh, CausalMesh-TCC, and two baselines: HydroCache [32] and FaaSTCC [21]. HydroCache has two versions: a conservative version (HydroCache-Con) and an optimistic version (HydroCache-Opt). Figure 9 summarizes the properties of these systems.

10 Evaluation

CausalMesh improves performance without the headaches of weak consistency. To see how well CausalMesh works, we answer these questions (our full paper covers more):

- How does CausalMesh scale with server count? (§10.1)
- What are the latency and throughput of representative applications running on CausalMesh? (§10.2)

10.1 Effect of the Number of Caches

To evaluate how CausalMesh scales with 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 10 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 versions of HydroCache perform worse than both CausalMesh-TCC and FaaSTCC; 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.

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 our full paper [36].

10.2 Movie Review Service

We evaluate CausalMesh's performance on the movie review service described in DeathStarBench [10, 9], where users create accounts, read reviews, view the plot and cast of movies,

	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 ∼ 2N RTT	0 RTT / 1 RTT to DB	No	refresh period
FaaSTCC	TCC	Yes	0 RTT ∼ 2N RTT	0 RTT / 1 RTT to DB	No	refresh period

Figure 9: 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.

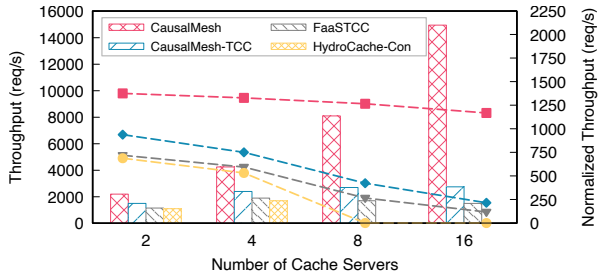

Figure 10: The histogram with y-axis on the left depicts the throughput as we vary the number of servers. The line plot with y-axis on the right shows the normalized throughput by dividing the throughput by the number of servers.

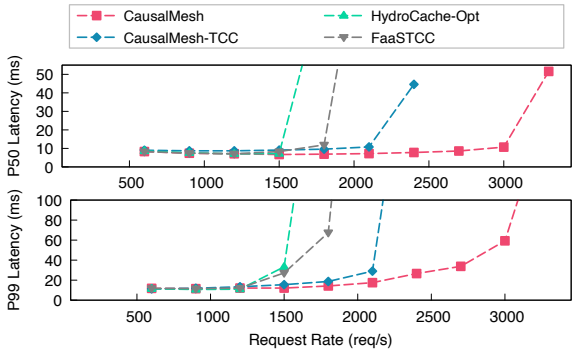

Figure 11: Comparison of CausalMesh, HydroCache, and FaaSTCC in terms of median and tail response time and throughput in a mixed workload that has contention between reads and writes.

and write movie reviews. We use Beldi's implementation [34] 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 11 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.

11 Discussion

Dynamic workflows. Workflows can be highly dynamic, such as reading one object and using its result to decide which other objects to read [20, 13]. HydroCache cannot support this, while both CausalMesh and CausalMesh-TCC do.

Metadata and garbage collection. The accumulation of dependencies can slow the system over time. CausalMesh and CausalMesh-TCC clear unnecessary metadata seamlessly while processing requests, without the need for dedicated GC processes. 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.

Cache eviction. We consider cache eviction an orthogonal problem. The design of dual cache allows it to benefit from any eviction policy. 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. Since C-cache drops dependencies, the system will need to include a coarse-grained dependency tracking method, e.g., a hashtable that indicates which keys should be evicted together. We can improve accuracy by adding more detailed dependency tracking in the C-cache at the cost of more memory.

12 Conclusion

This paper presents CausalMesh, the first cache system to support coordination-free and abort-free causal read/write operations when clients 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 state-of-the-art and is a great addition to the serverless ecosystem.

Acknowledgments

We thank Phil Bernstein and the VLDB reviewers for their thorough feedback, which significantly improved this paper. This project was funded in part by NSF grants CCF 2124184, CNS 2107147/2321726/2130590/2238768/2321725.

13 References

[1] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa. Firecracker: Lightweight virtualization for serverless applications. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[2] 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 *International Conference on Distributed Computing Systems (ICDCS)*, 2016.

[3] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. *ACM SIGMOD Record*, 24(2), 1995.

[4] S. Burckhardt, C. Gillum, D. Justo, K. Kallas, C. McMahon, and C. S. Meiklejohn. Durable functions: semantics for stateful serverless. *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2021.

[5] 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. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2008.

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

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

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

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

[10] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. 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)*, Apr. 2019.

[11] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu. Serverless computing: One step forward, two steps back. In *Conference on Innovative Data Systems Research (CIDR)*, 2019.

[12] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2010.

[13] D. Huye, Y. Shkuro, and R. 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.

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

[15] Z. Jia and E. 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.

[16] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, 2017.

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

[18] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He, and M. Guo. The serverless computing survey: A technical primer for design architecture. *ACM Computing Surveys*, 2022.

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

[20] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, and C. Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, 2021.

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

[22] P. Mahajan, L. Alvisi, and M. Dahlin. Consistency, availability. Technical report, and convergence. Technical Report TR-11-22, Univ. Texas at Austin, 2011.

[23] 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 *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

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

[25] R. Pang, R. Caceres, M. Burrows, Z. Chen, P. Dave, N. Germer, A. Golynski, K. Graney, N. Kang, L. Kissner, et al. Zanzibar: Google's consistent, global authorization system. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2019.

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

[27] J. Schleier-Smith, V. Sreekanti, A. Khandelwal, J. Carreira, N. J. Yadwadkar, R. A. Popa, J. E. Gonzalez, I. Stoica, and D. A. Patterson. What serverless computing is and should become: The next phase of cloud computing. *Communications of the ACM*, 2021.

[28] H. Shafiei, A. Khonsari, and P. Mousavi. Serverless

computing: a survey of opportunities, challenges, and applications. *ACM Computing Surveys*, 2022.

[29] M. Singhal and A. Kshemkalyani. An efficient implementation of vector clocks. *Information Processing Letters*, 1992.

[30] C. Tang, K. Yu, K. Veeraraghavan, J. Kaldor, S. Michelson, T. Kooburat, A. Anbudurai, M. Clark, K. Gogia, L. 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.

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

[32] C. Wu, V. Sreekanti, and J. M. Hellerstein. Transactional causal consistency for serverless computing. In *Proceedings of the ACM SIGMOD Conference (SIGMOD)*, 2020.

[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 ACM/IFIP/USENIX International Middleware Conference (Middleware)*, 2015.

[34] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu. Fault-tolerant and transactional stateful serverless workflows. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[35] H. Zhang, S. Mu, S. Angel, and V. Liu. Causalmesh. https://github.com/eniac/causalmesh.

[36] H. Zhang, S. Mu, S. Angel, and V. Liu. Causalmesh: A causal cache for stateful serverless computing. *VLDB*, 2025.