FaaSTCC: Efficient Transactional Causal Consistency for Serverless Computing

Taras Lykhenko
INESC-ID, Instituto Superior Técnico,
Universidade de Lisboa
Portugal
taras.lykhenko@tecnico.ulisboa.pt

Rafael Soares
INESC-ID, Instituto Superior Técnico,
Universidade de Lisboa
Portugal
joao.rafael.pinto.soares@tecnico.ulisboa.pt

Luis Rodrigues
INESC-ID, Instituto Superior Técnico,
Universidade de Lisboa
Portugal
ler@tecnico.ulisboa.pt

Abstract

In this paper we study mechanisms that permit to augment the FaaS middleware with support for Transactional Causal Consistency (TCC). At first glance, it may seem that offering TCC to FaaS applications can trivially be achieved, given that the FaaS paadigm does not prevent applications from selecting the storage service with the properties they need. Unfortunately, most TCC storage services ensure consistency only to individual client processes, while a FaaS application is executed by multiple, independent, worker processes. Therefore, there is the need to coordinate the workers, a task that can be a significant source of overhead. We propose a novel architecture to support TCC in FaaS, named FaaSTCC, that significantly reduces the coordination overhead. FaaSTCC achieves this goal by augmenting the workers with a caching layer and by implementing novel mechanisms that maximize the cache usage. First, our storage layer offers to the caching layer a *promise*, that sets a horizon where the versions retrieved by the cache are guaranteed to be consistent. Second, in FaaSTCC, functions coordinate using *snapshot intervals*, that support the lazy identification of the read snapshot, increasing the chances of using the cached values. We have implemented and experimentally evaluated FaaSTCC. Our results show that FaaSTCC achieves up to 5*x* lower average latency and 6*x* lower tail latency than previous work.

CCS Concepts: • **Computer systems organization** → **Cloud computing**; • **Software and its engineering** → **Consistency**.

Keywords: Distributed Systems, Function as a Service, Transactional Causal Consistency

ACM Reference Format:
Taras Lykhenko, Rafael Soares, and Luis Rodrigues. 2021. FaaSTCC: Efficient Transactional Causal Consistency for Serverless Computing. In *22nd International Middleware Conference (Middleware '21), December 6–10, 2021, Virtual Event, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3464298.3493392

1 Introduction

Serverless computing, in particular the Function-as-a-Service (FaaS) abstraction, has gained significant attention [12, 27, 28, 36, 39]. Systems that implement this abstraction, such as AWS Lambda or Google Cloud Functions, allow programmers to execute functions in the cloud without having to provision or maintain servers. Applications can be constructed by composing multiple functions [4], that are executed independently, possibly on different machines, managed by a run-time that automatically deals with load balancing, fault tolerance, and varying resource requirements. FaaS platforms achieve flexible autoscaling by disaggregating the computation and storage layers, so they can scale independently [24].

Most FaaS implementations are optimized for performance and low cost. Many AWS Lambda use cases [3] use DynamoDB and Amazon S3 as storage layer. This weakens the consistency provided to the users [27]. For instance, Amazon S3 offers linearizable writes but no atomic multi-writes (and even Read Your Writes guarantees are only available since 2020). This means that applications can observe intermediate or mutually inconsistent results that may take the system to unwanted states [20, 40], with potential severe consequences, such as capital loss [26]. These disadvantages can be mitigated if the programmer develops compensation functions to deal with consistency anomalies, a strategy initially proposed in the Sagas system [23]. However, compensation actions do not prevent clients from observing inconsistent states and may significantly increase the code complexity [16, 29].

This justifies the search for mechanisms that can offer stronger guarantees in FaaS, a trend that has been observed in the past for other services. In fact, is possible to find several examples of services that started by offering weak consistency (mainly for performance reasons), but gradually introduced support for stronger consistency levels, due to the large benefits that this bring to application developers. For instance, in [2] one can find the motivation for increasing the consistency model provided by Amazon S3, which includes requirements from large customers such as Dropbox and Salesforce. Similarly, DynamoDB [19] was initially proposed as a weakly consistent NoSQL database but, today, supports different consistency levels, including ACID transactions [1]. It is therefore credible that techniques to support stronger consistency levels in this context will be needed in the future.

In this paper we study mechanisms that permit to augment the FaaS middleware with support for Transactional Causal Consistency (TCC) [11]. Although other alternative consistency levels could be considered, there is a strong appeal in selecting TCC in this context. On one hand, CC can be implemented using non-blocking algorithms as it does not require locks nor the execution of a consensus protocols to coordinate participants, circumventing the CAP theorem [25]. Therefore, TCC has the potential to be offered without imposing significant performance impairments on the FaaS infrastructure. On the other hand, TCC is powerful enough to avoid relevant anomalies and, furthermore, it has been

Middleware '21, December 6–10, 2021, Virtual Event, Canada
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8534-3/21/12.
https://doi.org/10.1145/3464298.3493392

shown that it is possible to verify automatically if a given transactional program, when executed over a causally consistent database, exhibits serializable behaviour [18, 38].

At first glance, it may seem that offering TCC to FaaS applications can trivially be achieved given that the application can select the storage service with the properties it needs. For instance, instead of selecting Amazon S3, as some of the use case described in [3], the application could select a storage layer that offers TCC, among the several designs that have emerged in recent years, such as Cure [11], Wren [41] and Occult [35]. However, the TCC storage alone is not enough, because all these storage services ensure TCC to *individual* client processes. Namely, TCC storage services require the client processes to keep metadata that captures the consistent snapshot they are using; this metadata is key to enforce consistency. In FaaS, an application is composed of multiple functions, potentially executed by different workers. Therefore, there is the need to ensure TCC across *different* workers, which requires the metadata to be shared, introducing additional sources of overhead. Furthermore, in order to ensure that consistent versions are returned, workers may be forced to contact the storage layer, preventing the effective use of caches, a key ingredient to achieve performance.

HydroCache [45] is a first attempt at providing TCC to FaaS. For better performance, HydroCache uses a caching layer, that runs in compute nodes, and that mediates the access to the storage layer [45]. To ensure that caches serve values from a consistent snapshot [22], HydroCache passes per-key dependencies from function to function. As we will show in the evaluation, the amount of information that needs to be transferred from site to site can be a significant impairment to performance. Furthermore, because HydroCache uses a weakly consistent storage layer, at each cache, several rounds of communication with the storage backend may be needed in order to obtain a version of the data that belongs to the snapshot used by the client, a phenomenon that is more likely to occur in face of skewed workloads, with a negative impact on the tail latency [14]. This may add a latency penalty in the data access, leading to long tail latencies in the execution of functions.

In this paper, we propose a novel architecture to support TCC in FaaS, with the aim of reducing both the average and the tail latency of transaction execution. Our approach, that we have named FaaSTCC, combines, in a synergistic way, a multi-site cache with a TCC storage-layer. The key idea is that different workers can coordinate in an efficient manner by using a small amount of metadata that is passed from function to function, in the form of a *snapshot interval*, two timestamps that capture the versions that can be read by the application. Furthermore, when the storage layer serves a cache, it offers a consistency *promise*, that sets a horizon, in the timeline of transactions, where the returned values are guaranteed to still be consistent. As we show in the evaluation, our layered approach, using these coordination services, can significantly reduce both the amount of information that needs to be exchanged among multiple sites and the number of times workers are required to access the storage.

We have implemented FaaSTCC and experimentally evaluated it against HydroCache [45]. Our results show that our mechanisms are highly efficient. First, by using snapshot intervals to coordinate caches, FaaSTCC can outperform HydroCache even with the cache disabled, since it uses much less metadata to ensure consistent reads. Second, the promises provided by our storage layer to the caching layer can avoid many unnecessary accesses to be backend, even in challenging scenarios, such as in settings where the read-set and the write-set are not known *a priori*. As a result, FaaSTCC achieves up to 5*x* lower average latency and 6*x* lower tail latency than HydroCache for dynamic transactions.

2 Related Work

Several systems have proposed the use of caches to improve both the consistency guarantees and to improve performance. However, most of these systems do not consider transactions that span multiple caches or do not enforce transactional causal consistency. SwiftCloud [37] and Bolt-on Causal Consistency [17] ensures that clients always read from a causally consistent snapshot; however it does not support transactions or consistency across caches. A number of recent works focus on offering caching systems for FaaS [42, 43]; however, these works either do not ensure consistency across multiple functions or offer only eventual consistency guarantees.

There is an extensive literature on the implementation of transactions in weakly consistent stores. Relevant works include Orbe [21], GentleRain [22], and Cure [11] that use physical clocks to order the transactions, and Wren [41] that uses hybrid clocks. These systems are decentralized, and use a "stabilization" procedure, where partitions exchange information to know whose timestamps are in the past of all other nodes, to check when it is safe to apply updates. Other systems, rely on a centralized component that can totally order read transactions with regard to write transactions. A system that implements this strategy is ChainReaction [13]. This centralized component is a bottleneck that limits scalability. Systems like COPS [32] and Eiger [33] use explicit dependency checking, a technique that requires clients to keep track of all versions in their causal past, to ensure consistency. Unfortunately, the latter strategy can force clients to carry large amounts of metadata.

To the best of our knowledge, HydroCache [45] is the only previous work that offers transactional causal consistency to FaaS applications. HydroCache is based on a caching layer that is kept mutually consistent using techniques inspired by systems such as COPS. In HydroCache, when a transaction writes an object, it stores with the object the versions of the objects that are in the causal past of that transaction, together with the version of the other objects written by the transaction (this metadata is denoted the *dependency list*). When a transaction reads an object it also extracts the dependency list. Furthermore, if a transaction reads an object that is already in the dependency list, the worker that performs the read operation ensures that a fresh copy is returned (i.e, a copy with version number equal or greater than the one stored in the dependency list). Due to the asynchrony of the system, the worker may need to contact the storage multiple times before it manages to obtain a fresh version. The performance of HydroCache is penalized by the amount of metadata it needs to coordinate through multiple caches and by the fact that it may need to read multiple times from storage to obtain a consistent version. Thus, the performance of HydroCache differs significantly in function of the workload skew and, even when the skew is large (the most favorable case for HydroCache), it exhibits a large tail latency.

One could consider an alternative approach to offer TCC to FaaS that would consist in avoiding the use of a cache and, instead, use directly a TCC storage layer that would serve reads with constant metadata [34]. However, as we show in our evaluation, the advantages that a caching layer can bring to FaaS are significant, thus, eliminating the cache is not efficient. On the other hand, finding the most recent snapshot that is stable in every cache (as done in [34]) is not trivial, as it requires clients to know which versions are already visible to avoid serving stale data, something that is infeasible in a dynamic environment such as FaaS.

3 Background

In this section, we briefly introduce the system model that we have considered in the design of FaaSTCC. We also define causal consistency and its extensions, which are essential for the understanding of the paper.

3.1 FaaS-based Applications

We assume that applications are defined as a composition of stateless functions, which are written using off-the-shelf programming languages such as Python. Functions may return results, that are passed to other functions, but may also read from and/or write to stable storage. Compositions of functions can be modeled by a graph, with one root function, and one or more sink functions. A function can have one or more child functions and passes its results to all its children. In a graph where a function has multiple children, child functions can be executed in parallel. A function that has multiple parents, merges the results from its parents before executing. If a function is invoked before another, during the execution of the graph, the former is denoted to be *upstream* and the latter to be *downstream*. In this paper we treat each composition as a single transaction, for which we provide Transactional Causal Consistency guarantees (discussed below).

Transactions can be classified according to the set of data objects read and written by the functions. If the set of objects is fixed and can be known *a priori* (either because it is explicitly declared by the programmer, or because it can be extracted using static analysis of the code), the transaction is said to be *static*. It has been shown that prior knowledge regarding the read-set and the write-set of transactions can be used to optimize distributed transactions [9, 45]. However, obtaining this knowledge is not always possible. In this work we are particularly interested in supporting general (also denoted *dynamic*) transactions, where the read and write-set cannot be known a priori.

Note that in our experiments, compositions are limited to Directed Acyclic Graphs (DAGs) because our prototype builds on previous work, namely Cloudburst [42], that has this constraint. However, FaaSTCC can also work with graphs that have cycles (in fact, it is agnostic to the fact that a given function is called just once or multiple times in a graph). Our implementation also assumes that each DAG has a single sink function, with no further downstream function; the results of the sink function are returned to the user and/or written in the storage. Graphs that have multiple sink functions can be automatically extended to satisfy this constraint, by adding a dummy sync function that does nothing more that aggregate the outputs of the multiple sinks in the original graph before writing the aggregated results to stable storage.

3.2 System Model

We consider the generic and widely adopted FaaS architecture, where stateful storage is decoupled from the function execution.

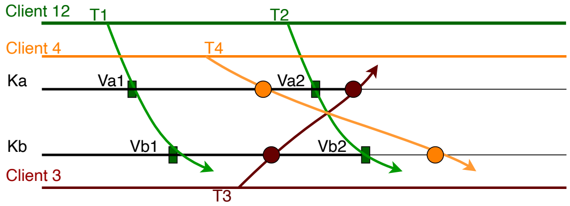


Figure 1. T_3 and T_4 violate TCC

Functions are executed by *executor threads* deployed in several *compute nodes*, and each compute node is responsible for one or more functions, that can be replicated across a different number of compute nodes. Each compute node has a pool of processing threads. The storage layer is, in itself, a distributed component. Typically, the storage layer is implemented by a distributed key-value store, that partitions the data in multiple shards, and each shard is stored in a different set of machines. In turn, each shard may also be replicated. Each function invocation is assigned to a given compute node by a *function scheduler layer* that receives all requests and routes them to worker threads based on heuristics that take into account different criteria such as CPU utilization and data locality. Our work is agnostic to the way functions are assigned to computing nodes.

3.3 Causal Consistency

The causal dependencies of an operation are determined by the *happened-before* relation ($\rightsquigarrow$) originally defined by Lamport [31] for message-passing systems and subsequently adapted to shared memory [10]:

- **Thread of Execution**. If a and b are two operations executed by the same function execution, then $a \rightsquigarrow b$ if a is executed before b.
- **Reads From**. If a is an update operation and b is a read operation that reads the value written by a, then $a \rightsquigarrow b$.
- **Transitivity**. If $a \rightsquigarrow b$ and $b \rightsquigarrow c$, then $a \rightsquigarrow c$.

In the following, we adopt the notation of [8]. Let $w_1(k_a, v_1)$ and $w_2(k_b, v_2)$ be two write operations, by different processes, p_1 and p_2, on k_a and k_b respectively. Let $r_3(k_a, v_1)$ and $r_3(k_b, v_2)$ be two read operations by the same process p_3, where $r_3(k_a, v_1)$ is executed before $r_3(k_b, v_2)$ (note that $r_3(k_a, v_1)$ returns the value written by $w_1(k_a, v_1)$ and $r_3(k_b, v_2)$ returns the value written by $w_2(k_b, v_2)$). This execution satisfies *Causal Consistency* (CC) if there was no write $w_x(k_b, v_x)$ such that $w_2(k_b, v_2) \rightsquigarrow w_x(k_b, v_x) \rightsquigarrow w_1(k_a, v_1)$.

3.4 Transactional Causal Consistency

Many applications include sequences of write operations, named transactions, whose effects are expected by the programmer to become visible atomically. However, causal consistency is defined for individual operations, regardless of how these operations are tied in the application logic. Thus, causal consistency allows sequences of read and write operations to produce unexpected outcomes.

Consider the example depicted in Figure 1, with two write transactions T_1 and T_2, executed by the same client, such that $T_1 \rightsquigarrow T_2$, where transaction $T_1 = \langle w_1(k_a, v_{a1}), w_1(k_b, v_{b1})\rangle$ and transaction $T_2 = \langle w_2(k_a, v_{a2}), w_2(k_b, v_{b2})\rangle$. Each write transaction creates a new consistent snapshot of the key-value store. Consider also two read transactions $T_3 = \langle r_3(k_b, v_{b3}), r_3(k_a, v_{a3})\rangle$ and $T_4 = \langle r_4(k_a, v_{a4}), r_4(k_b, v_{b4})\rangle$. CC guarantees that in case that T_3 reads $v_{b3} = v_{b2}$ written by T_2 then, later it should read $v_{a3} = v_{a2}$ also written by T_2 (and not the previous value v_{a1} written by T_1). This guarantee results from the fact that $w_1(k_a, v_{a1}) \rightsquigarrow w_1(k_b, v_{b1}) \rightsquigarrow w_2(k_a, v_{a2}) \rightsquigarrow w_2(k_b, v_{b2})$ being independent from the way that operations are ordered in a transaction. However, CC allows that T_3 reads v_{b1} (from T_1) and subsequently reads v_{a2} (from T_2), as depicted in the figure. Also, CC allows T_4 to read v_{a1} (from T_1) and subsequently to read v_{b2} (from T_2). Neither of these sequences violates causal consistency, but they introduce unexpected behaviors, as illustrated be the following example:

Example: Consider a social network application, where the friendship relations are symmetric, and one needs to ensure the following invariant. If user u_1 is visible in the friends list of u_2, then u_2 needs to also be visible in the friends list of u_1. Consider that k_i stores the friends list of u_i and that the transaction T_1 establishes a new relation of friendship between u_a and u_b, and that T_2 erases that relation. In this case, both T_3 and T_4 would read a state that would violate the application's invariant.

Transactional Causal Consistency (TCC) extends CC with additional guarantees, that are enforced on operations that are part of the same transaction. More precisely, TCC combines the following guarantees, as introduced in [11]: i) Transactions read from a causally consistent snapshot, which represents a view of the data store that includes the effects of all transactions that causally precede it. ii) A transaction updating multiple objects respects atomicity, i.e., all updates occur and are made visible simultaneously, or none does. Applying updates in this fashion creates a new causally consistent snapshot of the store.

TCC avoids the anomalies described above by ensuring that transactions read from a causal snapshot, and that updates are atomically visible. A causal snapshot is a set of item versions such that all causal dependencies of those versions are also included in the snapshot. For any two write operations, if $w_1(k_a, v_{a1}) \rightsquigarrow w_2(k_b, v_{b2})$ and both v_{a1} and v_{b2} belong to the same causal snapshot, then there is no $w_3(k_a, v_{a3})$, such that $w_1(k_a, v_{a1}) \rightsquigarrow w_3(k_a, v_{a3}) \rightsquigarrow w_2(k_b, v_{b2})$. Informally, a causal snapshot includes the most recent value of each key before the logical time that defines the snapshot.

Atomic visibility guarantees that either all items written by a transaction are visible to other transactions, or none are. Updates are atomically visible if a transaction T_2 writes $w_2(k_a, v_{a2})$ and $w_2(k_b, v_{b2})$, then any snapshot visible to other transactions either includes both v_{a2} and v_{b2} or neither one of them. Note that TCC is weaker that Snapshot Isolation [15] and allows concurrent transactions to write on the same set of keys. Still, atomic visibility is strong enough to avoid the anomaly illustrated by T_3 and allows to maintain symmetric relations between entities. Reading from a causal snapshot also avoids the anomaly of T_4.

4 FaaSTCC

FaaSTCC augments the FaaS middleware with support for Transactional Causal Consistency. Previous work achieves this goal via a caching layer alone [45], without any TCC support from the storage system. However, as we will show, this approach makes it hard to use the cache efficiently. An alternative design, could use instead a TCC storage without the caching layer but, as we will also show in the evaluation, this approach prevents significant gains to be achieved.

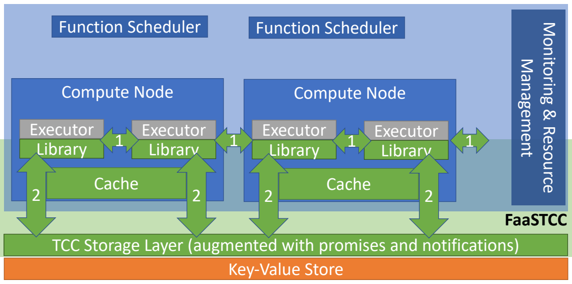


Figure 2. FaaSTCC Architecture.

FaaSTCC aims at combining the advantages of both approaches: it uses a caching layer to avoid unnecessary accesses to storage, and a TCC storage layer, augmented with mechanisms to ease the job of checking the consistency of the cache. In short, FaaSTCC consists of four main components, as illustrated in Figure 2: ***Client library:*** used by the functions to read and write from persistent store (via the caching layer); ***Caching layer:*** keeps objects accessed in the past and, additionally, pro-actively stores new versions of objects recently read; ***Augmented TCC Storage layer:*** offers TCC storage augmented with helper services that simplify the task of coordinating multiple workers/caches; ***Coordination layer:*** implements the required coordination between client libraries executing functions on behalf of the same transaction, such that all functions read from the same snapshot (arrows labeled "1" in Fig. 2) and the coordination between the client libraries and the storage layer, to ensure that different functions read mutually consistent versions (arrows labeled "2" in Fig. 2).

Note that the architecture presented in Figure 2 does not represent all possible FaaS architectures. As we will discuss in Section 5, this architecture matches the architecture of the tools we use to build our prototype, namely Cloudburst [42], and it can easily be compared with the architecture used by related work, namely HydroCache [45].

4.1 Rationale

In the following, we motivate why the layered approach proposed by FaaSTCC is advantageous.

We start by illustrating the limitations of attempting to achieve TCC in a caching layer, while accessing an eventually consistent store. Consider a FaaS transaction T that needs to read mutually consistent versions of two objects k_1 and k_2 that are stored in different storage partitions. T reads a version v_1 of k_1 from one partition and version v_2 of k_2 from another partition. Assume that v_2 belongs to a more recent snapshot than v_1. T needs to read k_1 again. Unfortunately, it can now read a version v_1' that belongs to a more recent snapshot than v_2, forcing T to read k_2 again. This sequence can be repeated indefinitely. In practice, if the workload is skewed and k_1 and k_2 are frequently accessed data items, the cache may be forced to access the data store multiple rounds. This heavily affects the tail latency of transactions. HydroCache is vulnerable to this pattern (in our experiments, we have observed sequences of up to 23 accesses for reading one item). If TCC is supported at the storage layer, one can prevent such long sequence of accesses to the data store from occurring, as it is possible to ensure that clients can read from stable storage a consistent snapshot in a bounded, and small, number of rounds [11, 33, 41].

Given the challenges in using the cache efficiently, one could consider to eliminate the cache and use a TCC storage instead. But, in this case, some form of coordination among workers is needed. In fact, all TCC storage services require the client to maintain some metadata to observe a consistent state. In FaaS, where an application is executed by multiple workers, this metadata needs to be exchanged among functions. If the metadata is large, as in COPS [32] and HydroCache, this metadata may be significant source of overhead. If the metadata is small, such as in system like PORT [34], clients may not be served fresh versions of data to avoid blocking. Furthermore, not using a cache at all may result in poor performance.

FaaSTCC aims at combining the advantages of the use of a caching layer, namely the possibility of serving reads without accessing the storage layer, with the advantages of using a TCC storage layer, namely being able to offer TCC semantics with small metadata. Still, the combination of these two layers is far from trivial. For instance, systems like PORT require clients to keep track of all storage partitions to capture the stable snapshot; these mechanisms cannot be easily extended for the caches, because in FaaS, the number of workers may be very large and highly dynamic.

FaaSTCC proposes novel coordination mechanisms that simplify the inter-cache coordination and the coordination of the caching layer with the storage layer. The FaaSTCC TCC storage layer offers to the caching layer a *promise*, that sets a horizon where the versions retrieved by the cache are guaranteed to be consistent. Furthermore, inter-cache coordination is achieved using metadata of constant size, more specifically, a *snapshot interval*, composed of just two timestamps, that captures the set of versions that a given function can read without violating consistency. Snapshot interval allows the read snapshot to be lazily evaluated, and increase the chances of finding consistent values in the cache. The combined use of snapshot intervals and promises ensures that function can often read safely from the cache, avoiding unnecessary accesses to the storage layer.

4.2 FaaSTCC Storage Layer

The FaaSTCC storage layer exports read and write operations, namely *TCC_WriteTX* and *TCC_ReadTX*. Note that read/write transactions can be implemented by executing one or more *TCC_ReadTX* operations followed by one final *TCC_WriteTX* operation.

The interface assumes that every write operation is assigned a *write timestamp t* that is stored together with the value. Thus, for each value written in the key-value store, a tuple $\langle k, v, t^v \rangle$ is maintained. Objects written by the same transaction are assigned exactly the same write timestamp. This is possible because, when executing a write transaction, all writes are buffered and applied atomically by the sink function. Timestamps are partially ordered, and if $T^1 \rightsquigarrow T^2$, then $t^2 > t^1$. Also, let two values, for different objects, be $\langle k, v, t^v \rangle$ and $\langle k', v', t^{v'} \rangle$: if $t^v \not> t^{v'}$ and $t^{v'} \not> t^v$, then values v and v' have been written by concurrent transactions.

The current version of the FaaSTCC Storage Layer has been implemented on top of Wren [41] that uses hybrid clocks [30]. However, we have designed our cache coordination algorithms such that timestamps are treated as abstract data types. This would simplify the work of implementing alternative versions of the FaaSTCC Storage Layer on top of other TCC storage systems, such as [11, 33], if needed.

The *TCC_WriteTX* operation is responsible for writing to store multiple keys: it receives as input a key value map, and returns a commit timestamp assigned to that transaction. *TCC_WriteTX* guarantees atomic visibility, i.e., either all items written by a transaction are made visible to other transactions, or none is.

The *TCC_ReadTX* operation returns the values of a set of keys at a given snapshot time. The primitive receives as input a key set and a timestamp $t_{snapshot}$. It returns mutually consistent values: for each key k in the read-set, the primitive returns the corresponding value v_k with the associated timestamp t^v_k and a $promise^v_k$. The value timestamp t^v_k corresponds to the most recent version of that key such that $t^v_k \leq t_{snapshot}$. The $promise^v_k$ sets a horizon for that version, i.e., the value of the most recent snapshot where that version is guaranteed to still be valid. This is either the timestamp of the next version (if there is any) or the timestamp of the last committed transaction (that may have touched different keys, other than key k): all future values of that key are guaranteed to have a timestamp greater than $promise^v_k$.

To avoid aborting or retrying read transactions multiple times, we assume that the FaaSTCC storage layer may return old values for a given key, even if more recent versions beyond the read interval already exist. This requires the storage system to maintain multiple versions for the same key. In practice, this is not a limitation, as all the storage systems that offer TCC do implement Multi Version Concurrency Control (MVCC) for better performance [11, 33, 41]. Naturally, any implementation needs to garbage collect older version at some point. If no version in the requested interval exists, *TCC_ReadTX* returns null, and the client will be forced to abort and restart the transaction from a recent snapshot. In fact, our algorithms also ensure correctness even if the storage layer only keeps a single version of each item (the most recent) but, in this case, long lived read transaction have an increased chance of aborting.

Finally, the FaaSTCC storage layer implements a publish/subscribe service that allows the caching layer to subscribe to updates to a set of keys. When the subscribed keys are updated, the new versions are periodically pushed to the subscribing caches. A cache can unsubscribe from a set of keys at any point and will stop receiving notifications for those keys.

4.3 Caching Layer

The FaaSTCC cache is implemented as a local in-memory key-value store that keeps a copy of the most recent read/written value of keys accessed in the past by multiple function executors on each node. The size of the cache is a parameter that can be tuned by the system administrator. In the current implementation, we use a Least Recently Used (LRU) policy to replace entries in the cache, but our algorithms are oblivious to the cache replacement strategy.

Each entry in the cache is a tuple ⟨*key*, *value*, *t*, *promise*⟩, where *key* is the item unique identifier, *value* is the last known value for that object, *t* is the timestamp of the write transaction that wrote that value and, as described before, *promise* is a guarantee, provided by the storage service, that any future update to the *key* will have a timestamp no smaller than promised timestamp. The *promise* field is extensively used to avoid calling the storage service unless strictly needed. For instance, assume that some function executor needs to access the most recent version of object k with timestamp lower than 100. If the cache entry for k has content $\langle k, v, 80, 120\rangle$, one can infer that the cached value can be safely returned to the function, as there is a promise that any future update to k will have a timestamp greater than 120.

To increase the chance that the cache is up-to-date when a request is received from a function executor, the FaaSTCC subscribes updates for all the caches keys from the storage system. Updates are received asynchronously, and there may be a delay between the time the update is stored at the persistent store and the time it is received at a given cache. Furthermore, because the system is asynchronous, two different caches, at different nodes, may receive the same update at different moments.

When a cache receives an update $\langle key, value, t, promise\rangle$, for a given key, it checks if the update is fresh and applies or discard the update accordingly (due to the notification service's asynchrony, an update can be received late). Although the update mechanism helps in keeping the content of the cache up-to-date, explicit cache invalidation mechanisms are not required by FaaSTCC. This happens because TCC does not requires a transaction to read from the most recent snapshot, but only from a snapshot that is consistent with both causality and atomic visibility.

4.4 Client Library

Every time a function needs to read or write an object, it invokes the client library. The library keeps a copy of all objects read or written by the function. Thus, even if other transactions update the objects, the function keeps observing a consistent snapshot of the storage. This is particularly relevant to prevent read-only transactions from aborting unnecessarily. The client library interacts with the FaaSTCC caching layer maintained at each compute node.

4.5 Read Consistency and Snapshot Interval

Before describing the operation of the client library, we discuss the notion of *snapshot interval*, a key concept in FaaSTCC to maximize the probability of getting a hit in the caching layer. *Snapshot intervals* allow for the lazy identification of a transaction read snapshot, based on the actual content of the cache. Consider a sequential DAG with just two functions f_x and f_y, executed in this order. Assume that f_x read a single key k and that a value of k is found in the cache of worker that executes f_x with value $\langle k, v, 80, 120\rangle$. Assume that function f_y is executed by a different worker with its own cache, and that f_y needs to read some other key k'. Assume that a version of k' is found in the cache of f_y's worker. Several scenarios can occur:

Case 1: The value of k' in the cache is $\langle k', v', 50, 60\rangle$. In this case, the value is only guaranteed to be consistent up to snapshot 60. There may exist a version of k' with timestamp in the interval $]60, 80[$ in the past of the value $\langle k, v, 80, 120\rangle$ already read by the transaction. In this case, the value of k' is not guaranteed to be consistent, and FaaSTCC needs to refresh the cache from the storage system, updating the promise or obtaining a newer version.

Case 2: The value of k' in the cache is $\langle k', v', 50, 90\rangle$. In this case, thanks to the information provided by the promise, it is known that there is no other version of k' in the interval $]80, 90[$. Therefore, the value of k' is consistent with the value of k and can be returned from the cache. Note that a single function, reading directly from the storage using, for instance, $t_{snapshot} = 85$ would read exactly these versions of k and k'.

Case 3: The value of k' in the cache is $\langle k', v', 90, 130\rangle$. In this case, although this version of k' has been written in the future of the version of k, from the promise of k we know that the two versions are mutually consistent. This is because we know that there are no other version of k in the interval $]90, 120[$. In fact, a single function, reading directly from the storage using, for instance, $t_{snapshot} = 100$ would read exactly these versions of k and k'.

Case 4: The value of k' in the cache is $\langle k', v', 130, 140\rangle$. In this case, the version is not guaranteed to be consistent, because there may some other version of key k with timestamp larger than 120 (the promise for k) in the past of the version of k' (that has timestamp 130). In this case, FaaSTCC needs to read an older, consistent version from the storage system.

In order to maximize the use of the cache, we need to pass from f_x to f_y metadata that is: i) as small as possible; ii) prevents the inconsistent values, such as the ones in Cases 1 and Case 4 above, to be read; iii) allows all consistent values, such as the ones in Cases 2 and Case 3 above, to be read without forcing an access to the storage layer.

Note that one could select a *single* snapshot value to perform inter-cache coordination. While this would ensure correctness (as reading from a single snapshot from a TCC store yields a consistent view) it would also perform poorly, because it would miss many opportunities to use the cache. For instance, if all functions would read from snapshot 80, in the Case 3 above the value in f_y's cache would not be used (although it is consistent). If all function would read from snapshot 120, in the Case 2 above the value in f_y's cache would not be used (although it is consistent). By using the *snapshot interval* to coordinate functions, we ensure that the cache of f_y can be used both in Case 2 and Case 3.

Instead of deciding a fixed read snapshot interval when the transaction starts, we use a *snapshot interval*, a range of snapshot values that can be used when performing a read. In the example above, the snapshot interval passed from f_x to f_y would be $[80, 120]$, where 80 is the snapshot that generated the value of k and 120 is the promise know for k at the time the value was written. Reading a value of k' from any snapshot in this interval is guaranteed to be consistent. By using *snapshot intervals*, the actual read snapshot is evaluated lazily, depending on the actual content of the caches.

Assume that a function f_x is executing with snapshot interval $[s_{low}, s_{high}]$ and obtains the tuple $\langle l, v_l, t_l^v, promise_l^v\rangle$ when reading some object l. This reading is consistent with the snapshot interval if the following inequalities hold:

$$(promise_l^v \geq s_{low}) \wedge (t_l^v \leq s_{high}) \qquad (1)$$

When a reading is consistent with $[s_{low}, s_{high}]$, the read-set of f_x is updated, i.e., the new value will be added to R_x. Furthermore, the snapshot interval is updated as follows:

$$s_{low} = \textsc{max}(s_{low}, t_l) \wedge s_{high} = \textsc{min}(s_{high}, p_l) \qquad (2)$$

4.6 Processing Reads

When a function executor needs to read an object, it issues a read request to the client library. At this point, two scenarios can occur. If the invoking function has previously accessed the object, the client library already has a copy of that object. In this case, the version stored in the library is returned. Otherwise, the client library forwards the read request to the caching layer. The snapshot interval $[s_{low}, s_{high}]$ is provided as an input parameter to the read operation. As noted before, the fact that we use an interval instead of a single fixed snapshot maximizes the chances of finding an appropriate hit in the cache. If there is a *Cache Miss*, i.e., the object is not cached, the cache forwards the read operation to the storage layer, and creates a new entry when the reply is received. If the object is already cache, there is a *Cache Hit*. Unfortunately, having the object in the cache is not enough to ensure the correctness of the application. One needs to check if the cached version is consistent with the interval specified by the client library. Assume that the cached version has the following content $\langle k, v_k, t_k^v, promise_k^v \rangle$. Three different situations may occur:

The cached version is consistent: The cache uses Eq. 1 to check if the value stored locally is consistent with the snapshot interval specified by the client library. If the value in the cache is consistent, it can be returned to the function.

The desired version has been replaced ($s_{high} < t_k^v$): In this case, the function needs an older version that is no longer in the cache. The request is forwarded to the storage layer using $t_{snapshot} = s_{high}$ and the reply is returned back to the client library without updating the cache.

The current version is stale ($promise_k^v < s_{low}$): In this case, the cache may be stale and a refresh is needed. A read request is sent the storage layer with $t_{snapshot} = s_{high}$. When the storage layer returns, the cache is updated accordingly and the new value is returned.

4.7 Processing Writes

When a function executor needs to update a set of objects, it does so via the client library. The client library keeps a copy of the written items. If the function subsequently reads the written items, the version kept by the library is returned to the function. This ensures that transactions see their own writes. When a function terminates, the written values are passed to the next function(s) in the DAG and the versions kept by the library are deleted.

When the transaction commits at the sink function, the client library forwards the write request directly to the storage layer. The storage layer assigns a write timestamp to the transaction. The values committed by the transaction are not cached immediately. Instead, they may be cached later, only if a transaction needs them. By avoiding to update the cache eagerly, FaaSTCC increases the chances that other transactions read from stable snapshots.

4.8 Multi-Site Coordination

Given that a user level transaction is executed in multiple compute nodes, executors must coordinate to ensure that all reads are made on the same consistent snapshot. In FaaSTCC this coordination is ensured by the client library. This coordination is achieved by passing the snapshot interval and the write-set from upstream to downstream functions, and by updating the snapshot interval as a result of the values read by each function along the path. In detail:

i) A downstream function f_d receives from the upstream function f_u a *snapshot interval* $SI_d == [u_{low}, u_{high}]$, for the values read by f_u, and a write-set, with all the values written by upstream function. The function sets the initial value of its snapshot interval $SI_d = [d_{low}, d_{high}]$ to $SI_d = SI_u$. A function f_d with multiple parents $f_u, f_{u'}, \ldots$ receives multiple snapshot intervals $SI_u, SI_{u'}, \ldots$ and multiple write-sets. In this case, it merges the write-sets and initiates $[d_{low}, d_{high}]$ as follows:

$$d_{low} = \text{MAX}(u_{low}, u'_{low}, \ldots) \wedge d_{high} = \text{MIN}(d_{high}, d'_{high}, \ldots) \quad (3)$$

If, as a result, $d_{low} > d_{high}$, this means that the upstream functions have read from inconsistent snapshots and the transaction is aborted.

ii) If the function f_d needs to read one or more keys, it attempts to read versions that are consistent with the snapshot interval inherited from the upstream function(s). Depending on the versions that are read by f_d, the snapshot interval may be narrowed as a result of applying Eq. 2. If the function f_d cannot read a consistent version, the transaction is aborted.

iii) Items written by f_d are not sent to stable storage (nor to the cache) and, therefore, are not made visible to other transactions. They are added to the transaction's write-set.

iv) The function f_d passes the updated snapshot interval SI_d (a possibly narrowed version of the snapshot interval SI_u inherited from the upstream function(s)) and the write-set (possibly extended with new values) to the next downstream function.

The root and the sink of functions have a slightly different behavior as explained below:

- The root function f_{root} has no parent. It sets the snapshot interval to $SI_{root} = [-\infty, +\infty]$.
- The sink function, when and if it commits, stores the write set of the transaction in the stable storage.

4.9 Algorithm Pseudo-Code

We provide a detailed description of the algorithms used to implement FaaSTCC. The description is organized according to the logical layers of the FaaSTCC design. We do not describe the implementation of the TCC storage layer, given that this is orthogonal to the main contributions of this paper.

The client library layer, whose pseudo-code is captured in Algorithm 1, is a layer that helps in ensuring that a transaction reads consistent values and, also, reads its own writes. The library keeps some transient state while a function is executed. Some of this state called the DAG *context*, is passed to all children functions when a function terminates. The remaining state is simply discarded.

When a DAG starts executing a *context* is created. The context stores the snapshot interval (that is updated as the transaction executes) and the write-set of the transaction (writes are applied to the storage only when the DAG execution is concluded and the transaction commits). This context is passed from a function to all its children. If a child function has more than one parent (i.e., if the DAG includes functions that are executed in parallel), the context inherited from the parents needs to be merged before the child function starts execution (Alg. 1, Line 7). In this case, it may happen that different parents read from incompatible snapshots; in this case the transaction needs to be aborted (Alg. 1, Line 11).

In addition to the context, that is passed from function to function, the library also keeps a copy of the values that have been previously read while executing function at the local worker (Alg. 1, Line 16). If a function needs to read the same key again while executing some DAG, the values previously returned is simply fetched from the read-set. Note that, because the read-set may be large, it is not passed explicitly in the context from one worker to another worker. Each worker builds its own copy of the read-set, based on the read operations performed locally. Since the snapshot interval

Algorithm 1 Client Library Code

```
 1: ▷ Context_Init is executed once, for each DAG, before the first function is called
 2: function CLIENT_CONTEXT_INIT
 3:     context.s_interval ← $[-\infty, +\infty]$
 4:     context.write_set ← $\emptyset$
 5:     ▷ The context is passed to child function when a function terminates
 6:
 7: ▷ Called before invoking a function after receiving context from parent(s)
 8: function MERGE_CONTEXTS(pc: set of contexts from parents)
 9:     context.s_interval ← [MAX(pc.s_interval.$t_{\text{low}}$), MIN(pc.s_interval.$t_{\text{high}}$)]
10:     if $t_{low} > t_{high}$ then
11:         ABORT
12:     context.write_set ← $\bigcup$ pc$_i$.write_set
13:
14: ▷ Local_Init initializes transient state that is only kept while a function is executed
15: function CLIENT_LIBRARY_LOCAL_INIT
16:     library.read_set ← $\emptyset$
17:
18: function LIBRARY_WRITETX(wset: set of ⟨key, value⟩)
19:     context.write_set ← context.write_set ∪ wset
20:
21: function LIBRARY_READTX(keys, in_interval = $[t_{low\_in}, t_{high\_in}]$)
22:     key_value_set ← $\emptyset$
23:     read_interval = $[t_{low}, t_{high}]$ ← in_interval
24:     for all k ∈ keys do
25:         if $\exists\langle k, v\rangle$ ∈ context.write_set then
26:             key_value_set ← key_value_set ∪ $\{\langle k, v\rangle\}$
27:         else if $\exists\langle k, v\rangle$ ∈ library.read_set then
28:             key_value_set ← key_value_set ∪ $\{\langle k, v\rangle\}$
29:         else
30:             $(\{\langle k, v\rangle\}, [t_{low\_k}, t_{high\_k}])$ ← CACHE_READTX($\{k\}$, read_interval)
31:             if $\{\langle k, v\rangle\} = \perp$ then
32:                 ABORT
33:             else
34:                 read_interval ← [MAX($t_{low}, t_{low\_k}$), MIN($t_{high}, t_{high\_k}$)]
35:                 library.read_set ← library.read_set ∪ $\{\langle k, v\rangle\}$
36:                 key_value_set ← key_value_set ∪ $\{\langle k, v\rangle\}$
37:     return (key_value_set, read_interval)
38:
39: ▷ Called after the sink function terminates
40: function COMMIT
41:     TCC_WRITETX(context.write_set)
```

is passed from one worker to another, different workers are ensured to obtain the same value for every key that is read during the execution of the DAG.

The client library is also in charge of keeping all writes in the transaction write-set, which is stored in the context. The write-set needs to be passed from worker to worker because written values only become visible when the transaction commits. If a function reads a value previously written, the value is read directly from the transaction context (Alg. 1, Line 25). If the key value is not found in the write-set or in the cached read-set, the client library requests the value from the worker's node cache (Alg. 1, Line 30). Note that every time the transaction performs a read operation, the snapshot interval may be narrowed as explained in Section 4.5 (Alg. 1, Line 34). In some rare cases, when old values are garbage collected prematurely from storage, the read may fail to find a value that is consistent with the snapshot interval, and the transaction is aborted (Alg. 1, Line 32).

The transaction write-set, kept in the context, is made persistent upon commit (Alg. 1, Line 19). After the sink function of the DAG executes, the write-set of the transaction is written to stable storage (Alg. 1, Line 39). Note that values are written directly to the storage and are not cached immediately. This allows other transactions to read from snapshots that are stable (even if older), increasing the chances that these transactions execute more efficiently.

Algorithm 2 Caching Layer Code

```
 1: ▷ Init is executed once, for all DAGs, at each worker
 2: function CACHE_INIT
 3:     cache.values ← $\emptyset$
 4:
 5: function CACHE_READTX(k, read_interval = $[t_{low}, t_{high}]$)
 6:     if $\nexists$⟨k, v, t, p⟩ ∈ cache.values: $p \geq t_{low} \wedge t \leq t_{high}$ then
 7:         ⟨k, v, t, p⟩ ← TCC_READTX(k, $t_{high}$)
 8:         if ⟨k, v, t, p⟩ = ⊥ then
 9:             ABORT
10:         else
11:             cache.values ← cache.values ∪ {⟨k, v, t, p⟩}
12:     return ⟨k, v, t, p⟩ ∈ cache.values: $p \geq t_{low} \wedge t \leq t_{high}$
13:
14: function CACHE_NOTIFY(updates: set of ⟨key, value, timestamp, promise⟩)
15:     cache.values ← cache.values ∪ updates
```

The pseudo-code of the caching layer is provided in Algorithm 2. The caching layer executes at each worker node and keeps the most recent versions of keys that have been read by functions that execute locally. The goal of the caching layer is to avoid unnecessary calls to the storage layer. The caching layer keeps a set of cached values, along with the corresponding timestamps and promises provided by the storage layer (see Section 4.1 for the rationale behind keeping promises). The caching uses the publish-subscribe interface provided by the storage layer to obtain updates to any of the cached values (Alg. 2, Line 14). We omit the subscription tasks from the pseudo-code, as these are trivial. When the client library needs a value, the cache first checks if it has a version that is consistent with the snapshot interval used by that client (Alg. 2, Line 6). If it exists, the library is served from the cache. If not, the storage layer is contacted to refresh the cache (Alg. 2, Line 7). As noted above, in some cases, the desired version may no longer be available and the transaction is aborted. Otherwise, the new version is added to the cache (Alg. 2, Line 11).

For clarity, in the pseudo-code of the client library layer and of the caching layer, we read one value at the time (and make multiple calls to the caching layer or to the storage layer, respectively); in the actual implementation, for efficiency, a single call is made in both cases to read all the missing values.

4.10 Correctness Argument

Consider a DAG execution and denote $[s_{low}, s_{high}]$ the snapshot interval used by the sink function s. The correctness of FaaSTCC derives from the following observations.

Observation 1. *If all functions in the DAG read directly from the storage, using the same snapshot timestamp $t_{effective}$, the application reads from a consistent snapshot.*

Basis: This derives directly from fact that FaaSTCC uses a TCC storage layer and that, by definition , the storage always return values that are consistent at $t_{effective}$. □

Observation 2. *Reading a key value from FaaSTCC cache with timestamp $t_{effective}$ is equivalent to reading directly from the storage with timestamp $t_{effective}$.*

Basis: This derives from the fact that FaaSTCC only reads a version v from the cache if the associated promise is larger than $t_{effective}$. Therefore, the version in the cache is guaranteed to be the same as the one that would be returned by the storage. □

Observation 3. *The execution of the DAG, is equivalent to an execution where all functions read from a single snapshot* $t_{effective} = s_{high}$.

Basis: Assume some function p in the DAG that has executed with snapshot interval $[p_{low}, p_{high}]$ and let c be a child of function p that executes with snapshot interval $[c_{low}, c_{high}]$. The result is equivalent to executing both functions with a single snapshot timestamp $t_{effective} = c_{high}$. This derives from the fact that $c_{high} \leq p_{high}$ since we take MIN of all promises when snapshot intervals are adjusted, and that $c_{low} \geq p_{low}$ since we take MAX of all version timestamps when snapshot intervals are adjusted. Therefore, $[c_{low}, c_{high}]$ is included in $[p_{low}, p_{high}]$, and thus, the values read by p when using p_{high} are the same as the vales that p would read if it would have used c_{high}. This argument can be used inductively, starting with the sink node and moving upwards in the DAG to the root node to show that all functions have obtained the same result as if they would have used $t_{effective} = s_{high}$. □

Argument 1. *When using FaaSTCC, DAGs read from a consistent snapshot.*

Basis: From Observation 3 the execution of the DAG using the set of snapshot intervals is equivalent to an execution using a single timestamp $t_{effective} = s_{high}$. From Observation 2, reading at $t_{effective}$ from the cache is equivalent from reading directly from the storage at $t_{effective}$. From Observation 1, if all functions read from the TCC storage using $t_{effective}$, all functions obtain a mutually consistent state. □

5 Implementation

We have implemented FaaSTCC using the same C++ code-base as HydroCache, making the appropriate changes to use the new coordination mechanisms, and the use of the FaaSTCC TCC storage layer. This ensures that the comparison with HydroCache, that we present in the evaluation, is fair. For the Function-as-a-Service platform, we used Cloudburst [42]; this allows us to compare FaaSTCC with HydroChace in the same codebase (Hydrochache is also based on Cloudburst). We use Anna [44] as the underling cloud store, on top of which the FaaSTCC Storage layer is built. The FaaSTCC storage layer has been implemented as a variant of Wren [41] that supports our promise mechanism. More precisely, we have modified Wren's interface to return the stable time (the timestamp of the most recent transaction known to be visible at every partition), that is used as the promise. This allowed us to extend Anna with TCC guarantees. We use ZeroMQ [7] with Protocol Buffers [6] for communication, and NTP [5] to synchronize the hybrid clocks used by Wren. We used Anna's notification mechanism to implement our storage layer notification service.

The caching layer was implemented over the Cloudburst cache, which is an in-memory key-value store. The executor threads use the FaaSTCC library to communicate with the cache by Inter-Process Communication (IPC). All the storage layer requests are forwarded through the caching layer. Thus, the caching layer is responsible for managing the request to the storage. In our implementation, the FaaSTCC client library is tightly coupled with the cache layer, such that if two or more transactions require the same version of a read-only object, only one copy is stored at the node.

All our code is open source and available at https://github.com/FaaSTCC.

6 Evaluation

In this section, we presents an extensive experimental evaluation of FaaSTCC. We compare FaaSTCC in terms of latency with HydroCache [45], the state-of-the-art approach to offer TCC guarantees in FaaS settings.

6.1 Experiment Setup and Workloads

All experiments are run on the Grid'5000 experimental platform using dedicated servers. Each server consists of 1 Intel Xeon Gold 5220 CPU, with 18 cores, 96GB RAM, and 480 GB of SSD storage. Nodes are connected through shared *25Gbps* switches. The measured latency within the cluster was approximately *0.15 ms*. Clocks were synchronized through an NTP server running within the cluster. We deployed the two systems in a Kubernetes cluster. For our testbench we allocated 18 nodes, used to run multiple virtual machines with the following physical machine distribution: 4 machines for Anna replicas, for a total of 16 Anna partitions, 10 machines for Cloudburst nodes, deploying a total of 40 computing pods. Each pod consists of 3 executor threads and 1 cache thread. We used 2 machines for clients, deploying 16 clients threads, 1 machine reserved for system managers, such as the Cloudburst Scheduler, Anna's Routing and Monitoring and 1 machine for kubernetes control plane's components.

Each client sequentially issues 1000 DAG execution requests (effectively, we more than double the amount of load used in the original HydroCache evaluation). The cache refresh period for snapshot updates was set to 50ms. Our dataset was 100000 keys, each with an 8-byte payload. When comparing with the original HydroCache evaluation, we opted to use smaller objects and also a smaller number of objects to reduce the time needed to populate the database, which was consuming a large fraction of the time required to run the experiments; given that we have limited access to the experimental testbed, this was instrumental to run the large number of experiments reported here. Cache sizes are unbounded and were pre-warmed to remove the data retrieval overheads from the storage layer.

As in HydroCache, our benchmarks evaluate sequential DAGs, which are chains of six sequentially executed functions. The result of each upstream function is passed in as an argument to the downstream function. The arguments are references to two keys in the storage (drawn from a Zipfian distribution). At the end of a DAG, the sink function writes its result into a key (drawn from a Zipfian distribution).

Transactions can be executed in two distinct modes: static and dynamic. With static transactions, all the keys that will be read in all functions are known *a priori*, and this read-set does not change during the execution. With dynamic transactions, the read-set of a function is unknown until the function executes.

6.2 Contribution of FaaSTCC Mechanisms

We start by showing that the combination of the use of promises and snapshot intervals is relevant to maximize the use of the cache. In Figure 3 we show the normalized transaction latency for three different configurations of FaaSTCC. The first, is a configuration that uses a fixed snapshot and where promises are not available; in this case, unless the exact version that matches the fixed snapshot is found in the cache, the system must always read from the storage. The second configuration uses the promises to validate the content

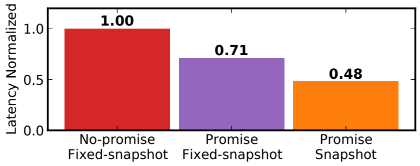
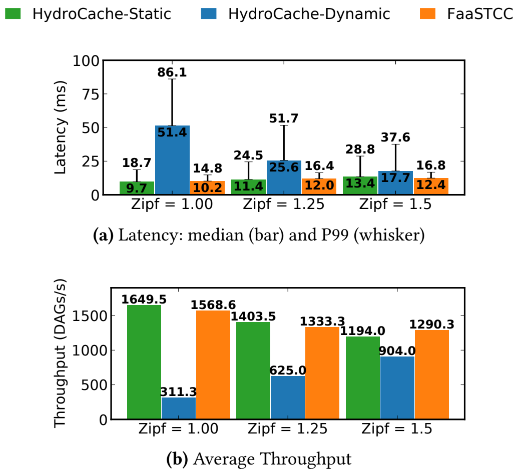


Figure 3. Impact of promises and snapshot intervals.



(a) Latency: median (bar) and P99 (whisker)

(b) Average Throughput

Figure 4. FaaSTCC vs HydroCache executing

of the cache, but still reads from a fixed snapshot. The last uses both the promise and the snapshot interval. When using a fixed snapshot, we have selected the promise of the first key read by the DAG as the value to be used during the entire execution. As it can be seen, the use of promises is responsible for 29% of the gains achieved by FaaSTCC and snapshot intervals bring an additional 23% reduction in latency. This shows that both mechanisms are relevant and should be used in combination. In the rest of the evaluation, we only show values for the full version of FaaSTCC, with both mechanisms active.

6.3 Latency and Throughput

We now compare the performance of FaaSTCC against HydroCache for both static and dynamic transactions. For these experiments, we varied the skew in the keys accessed by the DAGs functions (i.e., some keys are more accessed than others) and measured its effect on the DAG's latency and system throughput. We modeled the data access pattern using a Zipfian distribution. The results are presented in Figure 4. Note that FaaSTCC has the same performance with both static and dynamic transactions because our mechanisms do not require prior knowledge of the read-set, and work similarly in both cases.

It can be observed that the advantages of FaaSTCC, when using static transactions are relatively small. This happens because, for static transactions, both systems have advantages and disadvantages. On one hand, FaaSTCC leverages the storage layer to read from a consistent snapshot and avoid aborts, while with HydroCache almost 9% of the transactions abort. On the other hand, HydroCache metadata is more detailed, and therefore HydroCache performs slightly better at avoiding false conflicts: 70% of DAG functions can be executed only using the cache with HydroCache for Zipf = 1.0, while only 60% of the transactions can avoid reading from storage with FaaSTCC. As a result, FaaSTCC and HydroCache present almost the same median latency with static transactions. Still, FaaSTCC exhibits a lower tail latency (50% lower for Zipf = 1.5 and 20% lower for Zipf = 1.0).

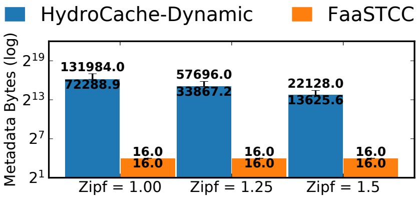


Figure 5. Metadata size: median (bar) and P99 (whisker).

The differences between FaaSTCC and HydroCache are striking with dynamic transactions. Because the read-set of functions is unknown beforehand, HydroCache needs to carry large amounts of metadata from function to function to ensure that it reads from a consistent snapshot. This information needs to be passed, because it is impossible to guess which dependencies are going to be needed downstream. FaaSTCC uses less metadata and has much lower overhead in this scenario. Furthermore, HydroCache often needs multiple accesses to the store when there is a cache miss while, by leveraging the storage layer, FaaSTCC is always able to fetch the right version in a single access to the storage. These effects combined allow FaaSTCC to offer 5x lower average latency and approximately 6x lower tail latency than HydroCache with dynamic transactions.

As expected, the throughput increases proportionally with the decrease in latency. Because clients are working in a closed loop (and start a new transaction as soon as the previous transaction finishes), they can execute more transactions when the average latency is smaller. This is clear when comparing Figures 4a and 4b.

6.4 Metadata Size

As was noted before, the high latency of running dynamic transactions in HydroCache is partially caused by the size of the metadata carried from function to function. We have measured the amount of metadata exchanged between functions, by both HydroCache and FaaSTCC, when running the experiments used to obtain the results presented in Figure 4. The average size of the metadata, measured in bytes, is depicted in Figure 5. The size of the metadata used by FaaSTCC is constant and limited to the two timestamps that define the snapshot interval ($[u_{low}, u_{high}]$). We can observe that HydroCache metadata size is tightly coupled with the skew of the workload: with Zipf = 1, median metadata size transferred between functions is *72KB*, 4500× larger than FaaSTCC. This effect decreases when the workload skew increases, because more skewed workloads tend to have fewer objects in the causal past. However, for Zipf = 1.5, HydroCache still uses more than *13KB*, i.e., 850× more metadata than FaaSTCC.

6.5 Accessing the Storage

Figure 6 and Figure 7 show, for different Zipfian distributions, the number of communication rounds required and the number of bytes transferred, when satisfying a read request from the storage. Thanks to the use of the TCC storage layer, FaaSTCC requires

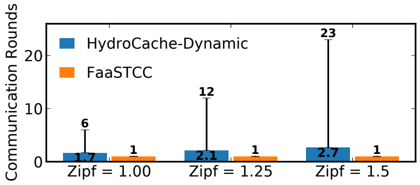


Figure 6. Number of rounds needed to read a consistent snapshot from the storage: median (bar) and P99 (whisker).

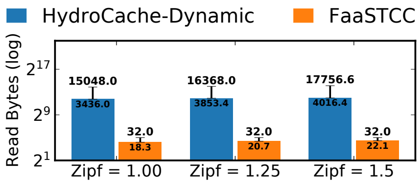


Figure 7. Bytes needed to read a consistent snapshot from the storage: median (bar) and P99 (whisker).

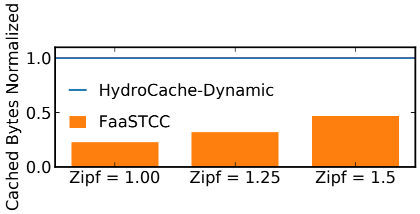


Figure 8. Cache Consumption (normalized).

one single access to the storage to retrieve a TCC consistent snapshot. We can observe that the number of communication rounds in HydroCache varies significantly with the workload skew. In fact, due to the phenomenon previously described in Section 4.1, as the skew increases, so does the number of communication rounds required by HydroCache to fetch a consistent version. We have observed that, with Zipf = 1.5, some functions have used up to 23 rounds of communication with the storage layer. Figure 7 shows that this increase not only impacts the latency but also induces more bandwidth consumption, due to the metadata used in HydroCache. FaaSTCC needs fewer data to be transferred due to smaller metadata size, and most of the requests to the storage send only metadata to update the promise.

6.6 Cache Consumption

We now compare the amount of cache memory used by FaaSTCC in comparison with HydroCache. For this purpose, we measured the size of the cache at the end of each run. We have performed this experiment with varying Zipfian distributions. The results are depicted in Figure 8, normalized using the size of the HydroCache as the reference. We can observe that for moderate skews (Zipf = 1.0), HydroCache needs to store more keys and those keys have more metadata associated. Storing the dependencies of the dependencies in the cache is essential to ensure consistency. However, it has the associated cost of storing a large number of keys and the associated metadata that may never be accessed. FaaSTCC only stores keys that have been previously accessed by the executor functions,

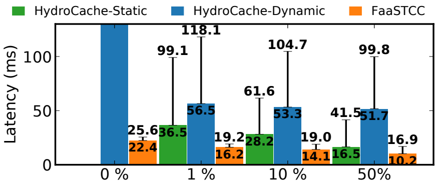


Figure 9. Latency for different cache sizes: median (bar) and P99 (whisker).

thus reducing the number of stored keys significantly in the cache compared with HydroCache's approach. Moreover, HydroCache's metadata requires more space as it uses explicit dependencies to track causality. These dependencies grow depending on the skew of the workload, as a lower skewed workload means that the write transactions are dependent on a broader set of keys.

6.7 Bounded Cache Size

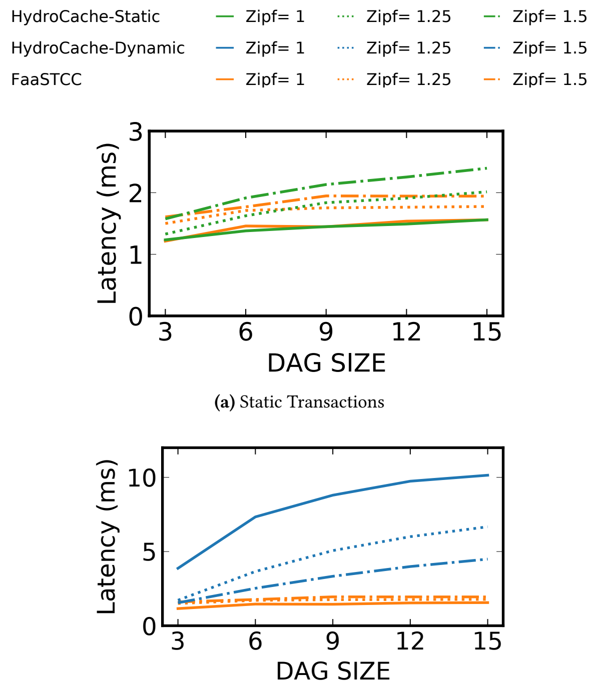
We now discuss the impact of the size of the cache in the performance of FaaSTCC. Typically, FaaS providers offer consumption-based pricing, meaning that storing keys that are rarely accessed comes at a financial cost. Therefore, it may be relevant to impose limits on the memory consumed by the caching layer. To assess how the systems perform with a bounded cache, we have run experiments with different cache sizes. In the experiments, we set the cache size as a percentage of the full size of the cache if the same workload was executed with no limits. We use a LRU policy to replace cache entries. This experiment also shows the performance of FaaSTCC with the cache disabled (cache size 0%).

Figure 9 shows the end-to-end DAG execution latency for FaaSTCC and HydroCache for both static and dynamic transactions when using a moderate workload skew (Zipf = 1.0). For small cache sizes, FaaSTCC has 2.2x lower median latency and 5.1x lower tail latency than HydroCache. When the cache is small, functions need to access the storage more often, and this is particularly expensive in HydroCache. Also, accessing the storage increases the chances that the required version was already overwritten, requiring the DAG to abort. Note that the values for FaaSTCC are *the same* for static and dynamic transactions: this is due to the fact that the operation of FaaSTCC is oblivious to this parameter, and runs exactly the same algorithm in both cases. HydroCache, instead, performs a number of optimizations with static transactions that cannot be implemented with dynamic transactions.

These results also highlight the importance of the combined effect of the mechanisms implemented by FaaSTCC. On one hand, the advantages of the use of small metadata, are clear since even with the cache disable, FaaSTCC can outperform or approximate the performance of HydroCache with the cache enabled. On the other hand, the results also show that the caching layer is key to improve performance: the FaaSTCC latency with the cache enabled is reduced to approximately half of the latency without cache.

6.8 DAG Size

We now assess how the DAG size affects the execution of the individual functions that belong to a given transaction. For this purpose we have executed FaaSTCC and HydroCache with different DAG sizes and different workload skews. Ideally, the average function execution time should be the same, regardless of the size of the



(a) Static Transactions

(b) Dynamic Transactions

Figure 10. Median latency as a function of the DAG size.

DAG of the enclosing transactions. However, there are two phenomena that contribute to an increase in the function execution time when longer DAGs are used. In HydroCache the metadata size grows significantly with the size of the DAG. In FaaSTCC, longer DAGs are associated with larger read-sets which, in turn, reduce the width of the snapshot interval. Figure 10a and Figure 10b report the average execution time of *each function* in the DAG, i.e., the time it takes to execute the entire transaction divided by the number of functions in the DAG. As it can be seen, with both systems, the average execution time of each individual function increases with the DAG size. However, as it can be observed, this effect is much more pronounced on HydroCache.

For static transactions, because the read-set is known before the transaction starts, HydroCache optimizes the metadata transferred from function to function to ensure consistency. In this case, the observed increase in the function execution time is mainly dominated by cache-misses. Because FaaSTCC is guaranteed to get the correct version in a single access to the storage, it is less affected by this factor.

For dynamic transactions, HydroCache requires significantly more metadata to be transferred between functions to ensure coordination than FaaSTCC. The coordination overhead in HydroCache is so large that the execution time of each function increases sharply for larger DAGs. This is particularly noticeable for workloads that have lower skew and tend to access a larger number of objects. In this setting, the time it takes for HydroCache to execute a function in the context of a long DAG can be roughly 5*x* larger than the time it takes to execute the same function in a short DAG. FaaSTCC, instead, is almost immune to this problem.

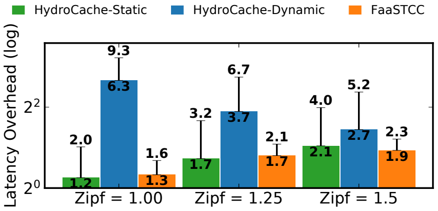


Figure 11. Latency overhead w.r.t. eventual consistency

6.9 Latency Overhead w.r.t. Eventual Consistency

Finally, we evaluate the overhead of both HydroCache and FaaSTCC when compared with the base Cloudburst implementation. Figure 11 shows the normalized latency of both systems w.r.t. Cloudburst. It can be observed that FaaSTCC is on-par with HydroCache for static transactions, with an overhead ranging from 1.2*x* to 2*x* slower than Cloudburst, depending on the workload skew. Note that Cloudburst only offers eventual consistency and has no transactional support. Furthermore, FaaSTCC has the same performance with static and dynamic transactions unlike HydroCache, where the overhead increases substantially (up to 6.3*x*) with dynamic transactions.

7 Conclusions and Future Work

In this paper we have proposed a novel architecture to offer transactional causal consistency to applications implemented using the FaaS paradigm. We combine the use of a TCC storage service with two key mechanisms to improve the cache usage: promises, that help in assessing if the cached values are still valid, and snapshot intervals, that support the lazy evaluation of the read snapshot in order to maximize the cache usage. Our design brings two significant advantages: i) it requires the exchange of significantly less information between caches and ii) it avoids long sequences of accesses to the storage system. While the performance of HydroCache significantly drops for dynamic transactions and less skewed workloads, the performance of FaaSTCC is oblivious to the static/dynamic nature of the workload and relatively stable across different skews. Also, with skewed workloads (the most favorable scenario for HydroCache), FaaSTCC offers much lower tail latency. In fact, we have shown that FaaSTCC can achieve up to 5*x* lower average latency and 6*x* lower tail latency than HydroCache.

Although we have focused on Transactional Causal Consistency, we believe that the mechanisms introduced here, namely promises and snapshot intervals, can be useful to support other consistency criteria. In particular, we are planning to extend our architecture to support stronger semantics, such as Snapshot Isolation.

Acknowledgements: We thank the anonymous reviewers for their help in improving the final version of the manuscript. Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr). This work was partially supported by Fundação para a Ciência e Tecnologia (FCT) under project UIDB/50021/2020 and grant 2020.05270.BD, and via project COSMOS (via the OE with ref. PTDC/EEI-COM/29271/2017 and via the "Programa Operacional Regional de Lisboa na sua componente FEDER" with ref. Lisboa-01-0145-FEDER-029271).

References

[1] [n.d.]. Amazon DynamoDB FAQs. https://aws.amazon.com/dynamodb/faqs/

[2] [n.d.]. Amazon S3 Strong Consistency. https://aws.amazon.com/s3/consistency/

[3] [n.d.]. AWS Lambda Customer Case Studies. https://aws.amazon.com/lambda/resources/customer-case-studies/

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

[5] [n.d.]. The network time protocol. http://www.ntp.org.

[6] [n.d.]. Protocol Buffers. https://developers.google.com/protocol-buffers.

[7] [n.d.]. ZeroMQ. http://zeromq.org/.

[8] Atul Adya, Barbara Liskov, and Patrick O'Neil. 2000. Generalized isolation level definitions. In *Proceedings of 16th International Conference on Data Engineering (ICDE))*. San Diego (CA), USA, 67–78.

[9] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. 2009. Sinfonia: A New Paradigm for Building Scalable Distributed Systems. *ACM Trans. Comput. Syst.* 27, 3, Article 5 (Nov. 2009).

[10] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. 1995. Causal memory: definitions, implementation, and programming. *Distributed Computing* 9, 1 (1995).

[11] Deepthi Devaki Akkoorath, Alejandro Z Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno Preguiça, and Marc Shapiro. 2016. Cure: Strong Semantics Meets High Availability and Low Latency. In *Proceedings of the 36th IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE, Nara, Japan.

[12] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *Proceedings of the Usenix Annual Technical Conference (ATC)*. Usenix, Boston (MA), USA, 923–935.

[13] Sérgio Almeida, João Leitão, and Luís Rodrigues. 2013. ChainReaction: A Causal+ Consistent Datastore Based on Chain Replication. In *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, Prague, Czech Republic.

[14] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-Scale Key-Value Store. *ACM SIGMETRICS Performance Evaluation Review* 40, 1 (2012).

[15] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. 2013. Highly Available Transactions: Virtues and Limitations. In *Proceedings of the 39th International Conference on Very Large Data Bases*. Trento, Italy.

[16] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2015. Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. Melbourne, Victoria, Australia, 1327–1342.

[17] Peter Bailis, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. 2013. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. New York (NY), USA, 761–772.

[18] Sidi Mohamed Beillahi, Ahmed Bouajjani, and Constantin Enea. 2021. Robustness Against Transactional Causal Consistency. arXiv:1906.12095 [cs.PL]

[19] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*. Santa Clara, CA.

[20] Nicola Dragoni, Ivan Lanese, Stephan Thordal Larsen, Manuel Mazzara, Ruslan Mustafin, and Larisa Safina. 2017. Microservices: How to make your application scale. In *Proceedings of the International Conference on Perspectives of System Informatics*. Moscow, Russia.

[21] Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. 2013. Orbe: Scalable Causal Consistency Using Dependency Matrices and Physical Clocks. In *Proceedings of the 4th ACM Annual Symposium on Cloud Computing*. ACM, San Jose (CA), USA.

[22] Jiaqing Du, Călin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. 2014. GentleRain: Cheap and Scalable Causal Consistency with Physical Clocks. In *Proceedings of the 5th ACM Annual Symposium on Cloud Computing (SOCC)*. ACM, Seattle (WA), USA.

[23] Hector Garcia-Molina and Kenneth Salem. 1987. Sagas. In *Proceedings of the 1987 ACM SIGMOD international conference on Management of data (SIGMOD)*. San Francisco (CA), USA.

[24] Pedro García-López, Aleksander Slominski, Simon Shillaker, Michael Behrendt, and Barnard Metzler. 2020. Serverless End Game: Disaggregation enabling Transparency. arXiv:2006.01251 [cs.DC]

[25] Seth Gilbert and Nancy Lynch. 2002. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *SIGACT News* 33, 2 (June 2002), 51–59.

[26] Grygoriy Gonchar. 2018. Data Consistency in Microservices Architecture. Voxxed Days Microservices Conference Talk.

[27] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2019. Serverless computing: One step forward, two steps back. In *Proceedings of the 9th Biennial Conference on Innovative Data Systems Research (CIDR)*. Asilomar (CA), USA.

[28] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2016. Serverless computation with openlambda. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Cloud Computing*. Usenix, Denver (CO), USA.

[29] Gowtham Kaki, Kartik Nagar, Mahsa Najafzadeh, and Suresh Jagannathan. 2018. Alone Together: Compositional Reasoning and Inference for Weak Isolation. In *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. Los Angeles (CA), USA.

[30] Sandeep S. Kulkarni, Murat Demirbas, Deepak Madappa, Bharadwaj Avva, and Marcelo Leone. 2014. Logical Physical Clocks. In *Proceedings of the 18th International Conference on Principles of Distributed Systems (OPODIS)*. Cortina d'Ampezzo, Italy.

[31] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978).

[32] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don'T Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. Cascais, Portugal.

[33] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. 2013. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*. Usenix, Lombard (IL), USA.

[34] H. Lu, S. Sen, and W. Lloyd. 2020. Performance-Optimal Read-Only Transactions. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. online.

[35] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. 2017. I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation*. Boston (MA), USA.

[36] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *Proceedings of the Usenix Annual Technical Conference (ATC)* (Boston (MA), USA).

[37] Nuno Preguica, Marek Zawirski, Annette Bieniusa, Sergio Duarte, Valter Balegas, Carlos Baquero, and Marc Shapiro. 2014. SwiftCloud: Fault-Tolerant Geo-Replication Integrated all the Way to the Client Machine. In *Proceedings of the 33rd IEEE International Symposium on Reliable Distributed Systems Workshops*. Nara, Japan.

[38] Kia Rahmani, Kartik Nagar, Benjamin Delaware, and Suresh Jagannathan. 2019. CLOTHO: Directed Test Generation for Weakly Consistent Database Systems. *Proceedings ACM Program. Lang.* 3, OOPSLA, Article 117 (Oct. 2019), 28 pages. https://doi.org/10.1145/3360543

[39] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. 2019. Architectural Implications of Function-as-a-Service Computing. In *Proceedings of the 52nd Annual International Symposium on Microarchitecture* (Columbus, OH).

[40] Jacopo Soldani, Damian Andrew Tamburri, and Willem-Jan Van Den Heuvel. 2018. The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software* 146 (2018).

[41] Kristina Spirovska, Diego Didona, and Willy Zwaenepoel. 2018. Wren: Nonblocking reads in a partitioned transactional causally consistent data store. In *Proceedings of the 48th IEEE International Conference on Dependable Systems and Networks*. Luxembourg City, Luxembourg.

[42] Vikram Sreekanti, Chenggang Wu Xiayue Charles Lin, Jose M Faleiro, Joseph E Gonzalez, Joseph M Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful functions-as-a-service. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2438–2452.

[43] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. 2020. InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies*. Santa Clara, CA.

[44] Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. 2019. Autoscaling Tiered Cloud Storage in Anna. *Proceedings of the VLDB Endowment* 12, 6 (2019).

[45] Chenggang Wu, Vikram Sreekanti, and Joseph M Hellerstein. 2020. Transactional Causal Consistency for Serverless Computing. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD)* (Portland (OR), USA).