2025 IEEE International Symposium on High-Performance Computer Architecture (HPCA)

Concord: Rethinking Distributed Coherence for Software Caches in Serverless Environments

Jovan Stojkovic, Chloe Alverti, Alan Andrade, Nikoleta Iliakopoulou, Hubertus Franke†,
Tianyin Xu, Josep Torrellas
University of Illinois at Urbana-Champaign †IBM Research
{jovans2, xalverti, alansa2, nmi4, tyxu, torrella}@illinois.edu, frankeh@us.ibm.com

***Abstract*—Costly accesses to global storage substantially limit the performance of serverless functions. To mitigate this overhead, data can be cached in the memory of the nodes where functions are executed. Existing caching schemes either (1) restrict a data item to be cached in a single node, causing frequent remote reads or (2) allow a data item to be cached in multiple nodes concurrently, adding substantial overhead to maintain cache coherence. Unfortunately, current approaches are suboptimal for the access patterns present in serverless workloads, which are characterized by frequent reads to small data items, strong temporal locality, and a small number of nodes that concurrently execute functions of the same application.**

Driven by these insights, we propose *Concord*, a distributed software caching system tailored to serverless environments. Concord allows multiple copies of the same data item to be cached in different nodes concurrently, allowing each cache to satisfy local reads. To maintain coherence across software caches, Concord proposes a directory-based distributed coherence protocol. The protocol is inspired by hardware cache coherence, and is enhanced to minimize coherence traffic, reduce contention points, and be robust to node failures and frequent coherence domain changes. Further, with the Concord coherence protocol, we unlock two new capabilities in serverless environments: transactional storage accesses and transparent data-aware function placement. Compared to state-of-the-art severless caching schemes, Concord running on a 16-node cluster speeds-up execution by 2.4× and improves throughput by 1.7×, while using only 6.2MB of otherwise idle application memory (*i.e.*, 4.8% of the total application memory).

I. Introduction

Serverless computing or Function-as-a-Service (FaaS) is an emerging paradigm adopted by all major cloud providers [8], [9], [29], [37], [39], [62]. It offers application flexibility, fine-grained billing, and high resource utilization [40], [56]. With FaaS, users upload their code and providers secure the necessary dependencies (*e.g.*, libraries and runtimes) and hardware resources (*e.g.*, memory and cores) to run the code. The unit of execution is a function deployed in a container, and applications are composed of workflows of functions.

For high availability and fast scalability, functions are commonly implemented as *stateless* services [12], [58], which means that all the data of a function is discarded from a node once the function is unloaded from the node. Hence, any durable data must be stored in global storage, such as the Azure Blob Storage service [59]. This results in inefficient data reuse: subsequent function invocations must reload their data from global storage. In addition, for security reasons, cloud providers do not allow direct communication between functions. As a result, data items must be passed through the global storage [52], [55], [81], [83].

We measure that applications spend 35-93% of their end-to-end response time on storage reads/writes. Such operations are typically implemented as Remote Procedure Calls (RPCs). To mitigate these costs, data can be cached locally in the memory of the nodes where functions execute. However, distributed software caches add a new challenge to the FaaS infrastructure: how to keep these caches coherent while avoiding the high cost of frequent inter-node communication. In this paper, we show that prior proposals [46], [65], [68], [70], [75], [82] address this challenge in sub-optimal ways for FaaS environments.

Most schemes [46], [65], [68], [82] cache a data item in the memory of only a single node, called the data item's *home node*. Function invocations running on nodes that are not the data item's home always access the item from the home. These schemes eliminate any need for coherence, as there is at most one cached copy of the data item. However, they work well only when the function invocation runs on the node that is the home of the data items that the function accesses; otherwise, remote accesses are needed. In practice, multiple function instances running concurrently on different nodes need to access the same data item. Consequently, these caching schemes are not effective. We measure that data movement due to remote reads/writes still accounts for up to 82% of the total application response time.

Faa$T [70] allows a data item to be cached in multiple nodes and keeps caches coherent via a software protocol. It uses a *versioning protocol* that associates a version number with each data item. Data items have a home node, which caches the latest data value and version number. When a non-home node reads the data item, it first fetches the item's version number from the home, even if it caches the data item locally. Then, it compares the version number in the home with the locally-cached version number. If the two numbers match, the invocation accesses the data item directly from the local cache. Otherwise, it fetches it from the home. Further, when a non-home node writes the data item, the update is propagated to the home, where it updates both data and version number.

This protocol works well for relatively large data items, where fetching only the small version number is substantially cheaper than fetching the entire data item. Moreover, it is designed to scale to many sharing nodes and frequent updates, as it avoids any invalidation messages. However, it is not optimized for FaaS access patterns: we measure that accessing

and checking versions in our applications can cost up to 78% of the application response time.

The reason is that the majority of storage accesses in FaaS are *reads to small data items*. Production-level Azure functions [70] reveal that 77% of the storage accesses are reads, and 80% of the data items are no larger than 12KB. These facts make versioning protocols suboptimal, since: (1) the time to fetch the version number is comparable to the time to fetch the data item, and (2) the majority of version comparisons are unnecessary, since there are no writes between reads.

This motivates us to re-visit *invalidation-based* distributed coherence for FaaS. Invalidation-based protocols, though commonly used for hardware cache coherence [20], [21], [49], [66], have been disregarded in distributed software environments [24], mainly because: (1) coherence directories introduce fault tolerance concerns and (2) invalidation messages may scale poorly with increasing numbers of nodes. However, we argue that invalidation-based protocols can be a good match for FaaS. The reasons are: (1) functions are designed to be stateless [12] and are thus more robust to failures, and (2) the total number of nodes sharing the same data item is typically less than a few 10s [72], [77], [87], which limits the coherence traffic due to invalidation operations.

Given these insights, this paper proposes *Concord*, a novel distributed caching system for FaaS environments. Concord maintains a software data cache per FaaS application and distributes the cache across the multiple nodes where function instances of that application execute. Concord leverages the extensively-studied area of hardware cache coherence and proposes an invalidation-based distributed coherence protocol in software.

Concord tailors the protocol to a distributed environment in three ways. First, it organizes the distributed caches and coherence support on a *per-application* basis. Second, it designs the coherence protocol to be resilient to failures. Third, it minimizes coherence traffic by modifying the scheduling of function invocations to route them to nodes that likely cache the needed data. To make the protocol resilient to failures, Concord employs write-through software caches and a distributed coordination service that monitors nodes' health. In case of node crashes, the coordination service redistributes the data items homed in the crashed node. Overall, Concord achieves high performance while ensuring data safety.

We also use Concord to provide *transparent support* for transactional storage accesses and communication-aware function placement. Specifically, for transactional accesses, Concord relies on its coherence protocol to detect and recover from transaction races—instead of requiring application rewriting [13] or extensive storage logging [86]. It buffers the speculative state in local caches before committing it to global storage, and relies on coherence messages to detect conflicts. Furthermore, for function placement, Concord exploits coherence messages to transparently learn over time which functions frequently communicate with each other. Later, it uses the collected data to intelligently co-locate such functions on the same nodes, reducing network overheads.

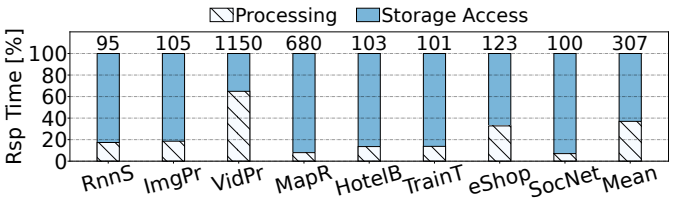


Fig. 1: Breakdown of applications' response time into processing and storage access. The numbers on top of the bars indicate the absolute response time in ms.

We implement Concord in the OpenWhisk [1] serverless platform. We use a 16-node cluster running a diverse set of serverless benchmarks. Compared to state-of-the-art baselines [65], [70], Concord speeds-up execution by 2.4× and improves throughput by 1.7×, while using only 6.2MB of otherwise idle application memory (*i.e.*, 4.8% of the total application memory). Overall, this paper makes the following contributions:

- We analyze distributed FaaS software cache designs and the design space of coherence protocols for them.
- We introduce the Concord caching system, which includes a high-performance and fault-tolerant directory-based distributed coherence protocol.
- We use Concord to provide transactional storage accesses and communication-aware function placement.
- We evaluate Concord and its features.

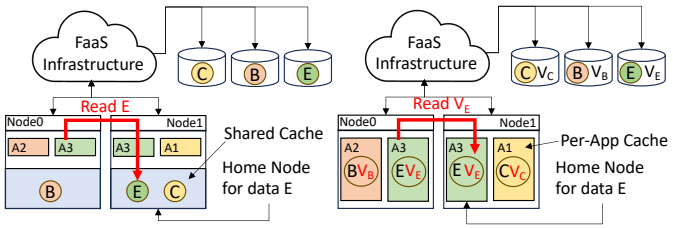
II. Background and Motivation

A FaaS platform [1]–[3], [36] is composed of multiple modules. First, a frontend checks the integrity of incoming function requests and forwards the requests to the load balancer. The load balancer distributes the requests across nodes in the cluster for load balance. The requests forwarded to a node are handled by a node controller. The node controller encapsulates the function code with all dependencies inside a container (or a micro VM), and invokes the function's handler with the amount of memory and number of cores that the function is allowed to use, as given by the user.

A. Need and Opportunity for Data Cache Designs

Figure 1 breaks down the response time of popular FaaS applications into time spent reading/writing data from/to storage and time processing the data. We will describe the applications in Section V. Our experimental results corroborate past studies [46], [65], [70] showing that global storage access time dominates FaaS response time. The graph shows that such time accounts for 35.1-93.0% of the end-to-end function response time, with an average of 63.1%.

A long line of research [46], [65], [70], [75] uses in-memory caching to mitigate these overheads. Most works propose per-node caches shared by all the applications co-located on the same node (Figure 2a) [46], [65], [68], [75], [82]. However, real-world data from Azure production [70] shows that strong data re-use is mainly observed among invocations of the same application. For example, 30% of Azure applications access



(a) Shared caches. (b) Per-application caches.

Fig. 2: State-of-the-art in-memory FaaS caching schemes.

the exact same data objects across all of their invocations, and 99.7% of data objects stored in global storage are not shared across applications at all. Thus, caches should be managed and maintained per application (Figure 2b), rather than being shared across applications. This design also enables discarding a cache instance from a node's memory when the corresponding local application instances are shut down.

Using main memory to cache data objects creates the challenge of properly reserving the necessary physical resources and charging for them. Faa$T [70] allocates memory in a node for an application's cache and charges users extra per cached byte access. In reality, we find that it is unnecessary to allocate such extra memory. The reason is that the majority of the memory already allocated for FaaS functions remains unused. Indeed, a recent trace from Huawei [42] suggests that users request 5× more memory than needed for 50% of the functions, which adds-up to 100s of MBs per node. The unused memory of one application can be transparently and dynamically re-purposed into a cache for that same application, improving performance for free.

Insight #1. Accesses to global storage limit the performance of FaaS functions. Per-application data caches can mitigate these costs transparently and, if designed properly, for free—by utilizing applications' allocated but unused memory.

B. Data Coherence and FaaS Trends

Reads on small data items dominate FaaS accesses. An analysis of Azure public traces [60] performed by Faa$T [70] showed that 80% of data items are no larger than 12KB, 77.3% of accesses are reads, and a large fraction of accesses to data items are bursty (even burstier than Poisson). We observe similar trends with IBM traces [25]. The high fraction of reads and high burstiness can result in high local cache hit rates in distributed software cache designs for FaaS.

Data is shared across nodes. We use Azure production traces for storage accesses [70] to benchmark our FaaS applications running on a 16-node cluster. We will describe the cluster in Section V. We measure the number of nodes accessing the same data, *i.e.*, the data *sharers*. Table I reports the average number of sharers under a low, medium, and high load of requests across all data items. We find that even under low load, there are multiple sharers per data item. For high performance, distributed software caches should allow caching

TABLE I: Average/Maximum number of node sharers measured while running on a 16-node cluster.

	Low Req. Load	Medium Req. Load	High Req. Load
HotelB [28]	1.7/6	2.2/9	3.4/12
TrainT [4]	1.3/5	1.6/6	2.2/10
eShop [80]	1.1/4	1.2/5	1.4/6
SocNet [28]	2.7/11	3.6/14	5.0/15
Average	1.7/6.5	2.2/8.5	3.0/10.8

the same object in multiple nodes. Of course, for correctness, the caches of sharers must be kept coherent.

Insight #2. FaaS distributed software caches require coherence, and the protocol should be optimized for read operations on small data items that commonly hit in local caches.

The maximum number of sharers is often modest. Table I also reports the maximum number of sharers measured, and shows that such number never reaches the total number of nodes, even under high request load. This is attributed to the modest data sharing across function instances of the same application, and the high degree of function instance co-location in a node. This allows us to revisit protocols for distributed systems that were traditionally discarded because of the potential need to support many sharers.

Functions are robust to failures. On today's serverless platforms, when a function fails, the platform re-executes the function and tolerates the failure [11], [30], [61]. To use this retry-based approach, functions must be idempotent, *i.e.*, functions must exhibit the same behavior when they are re-executed. To make functions idempotent, storage APIs for serverless functions are also designed to be idempotent, typically using a form of key-value interface.

Insight #3. The observed number of sharers per data object and the inherent robustness of serverless functions on failures allow us to revisit coherence protocols originally tailored for hardware directory protocols.

C. Prior Art on Software Caching Schemes

We detail the operation of two closely related works: Faa$T [70] and OFC [65]. In Section VI, we quantitatively compare them to our proposal.

In **OFC** [65], each node provides a cache shared by all locally-running applications. Each data item has a home node and can be cached only in the cache of its home. All caches in the system can be accessed by all applications. Figure 2a shows an example of this system. Applications *A2* and *A3* are co-located on Node 0, while *A1* and a second instance of *A3* are co-located on Node 1. Node 1 is the home for data *E*, as determined by the hash of *E*'s address, and all reads/writes to E go to the cache of Node 1. This design has no need for cache coherence because there is no data replication. However, it results in frequent remote reads/writes because a data item can only be cached in its home node.

Faa$T [70] introduces per-application caches and allows multiple cached copies of the same data across nodes. A cached

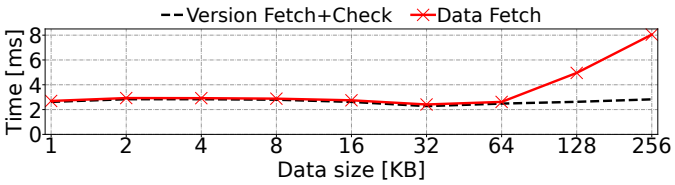


Fig. 3: Time to fetch and check a version number vs time to fetch a data item.

copy consists of the data value and its version number. Each data item has a home node. The cache in the home contains the data value that is consistent with the global storage and the latest version number of the data. When a non-home node reads a data item, it first fetches the item's version number from the home, even if it caches the data locally. Then, it compares the number with the locally-cached version number. If the two numbers match, the function accesses the data directly from the local cache. Otherwise, the data is fetched from the home. When a non-home node writes the data item, the write is propagated to the home, where it updates both data and version number, and then to the global storage. The home returns the new version number, and the writing node updates both data and version number locally. Writes by the home node update both version and value on the data item, both locally and in global storage. No invalidation is sent.

Figure 2b shows an example of this system with the same application and data layout as in Figure 2a. Note that Applications *A1*, *A2*, and *A3* have separate caches. However, *A3* has caches in the two nodes. Each data item has a version number, shown with the letter *V*. In the figure, Node 1 is the home for *E* and *E* is also cached in Node 0. When *A3* running in Node 0 reads *E*, Faa$T first fetches V_E from Node 1 and compares it with the local V_E. If version numbers are the same, *A3* consumes the local data. Otherwise, *A3* fetches *E*'s data and version number from Node 1, and stores them locally.

This design suffers from traffic induced by fetching version numbers. Figure 3 compares the time it takes to fetch and check a version number from the home node to the time it takes to fetch the actual data from the home, as we increase the data size in our cluster. We use dual-port Intel X520-DA2 10Gb NICs (PCIe v3.0, 8 lanes) and gRPC for reading/writing remote data. The nodes are connected with 10 Gbps full-duplex ethernet. The total time for fetching the data and its sequence number also includes gRPC overheads, such as serialization and RPC-encoding. We see that the cost of version fetch and check is comparable to the cost of data fetch for objects of 64KB or less; it is lower only for larger objects. In FaaS environments, the data size is typically no larger than 12KB. As a result, Faa$T adds coherence messages that could potentially be avoided.

Insight #4. Prior cache designs are suboptimal for FaaS, as they induce remote accesses to either data or metadata.

III. Concord: High-Performance Caching for FaaS

Driven by our insights, we propose *Concord*, a distributed software caching system for FaaS environments. Concord achieves high performance with support for fault tolerance. It exploits unused memory resources that users are already charged for. Hence, it does not introduce any extra monetary costs. Finally, it requires no changes to the applications.

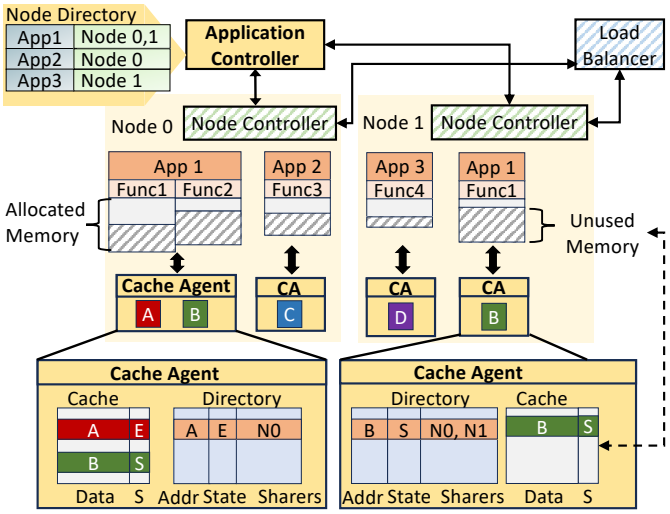


Fig. 4: Architecture overview of the Concord FaaS platform.

A. Concord Overview

Figure 4 overviews the Concord system. Concord takes a fraction of the memory allocated to local functions that is temporarily unused, and re-purposes it to act as *local cache instances*. It then binds a cache instance to each application that includes a local function. In a cluster, a function may have multiple function instances located on the same or different nodes. Function instances from the same application that are co-located on a node share a cache instance. For example, in Figure 4, instances of *Func1* and *Func2* from *App1* in Node 0 share the same cache instance. A given application typically has function instances in multiple nodes of the cluster, as different invocations of the application may be executing on different nodes. Consequently, multiple nodes may have cache instances of the same application. All these cache instances together form the distributed cache of the application. In the figure, the cache instances of *App1* on Nodes 0 and 1 form *App1*'s cache.

Since multiple applications may be co-located on a node, individual nodes typically host multiple cache instances. However, as data sharing occurs only within an application, caches of different applications are isolated from each other. Cache instances, like function instances, are ephemeral: once all function instances sharing a cache instance are removed from a node, the cache instance is discarded.

Concord allows copies of the same data item to reside in cache instances in multiple nodes, and keeps these copies coherent—*e.g.*, in Figure 4 data item *B* is cached in cache instances in Nodes 0 and 1. As Concord binds caches to applications, writes from one application do not affect the caches of other applications.

Concord is tailored to the needs of a distributed FaaS environment. Compared to systems that are kept coherent with conventional hardware schemes, FaaS environments have larger scale, suffer longer-latency communication and higher network contention, and are more prone to failures. Thus, Concord (1) minimizes contention, (2) reduces the number of coherence messages, and, (3) provides fault tolerance.

First, to minimize contention, Concord has per-application caches, and the directory for an application is sharded only across the nodes that contain caches of that application. Each data item can be cached in multiple nodes, but it is assigned one home node. The home provides the data item to other nodes and, through the directory, maintains the cache instances coherent for that data. The home of a data item is decided via *consistent hashing* [44]. For a given application, when a new cache instance is created or an old one is removed, the home of some data items of the application may dynamically change.

Second, Concord schedules function invocations via a new coherence-aware algorithm, which tries to place invocations that operate on the same data on the same node. In this way, such invocations often use the same cache instance, minimizing the need for coherence messages.

Third, Concord designs the protocol for fault tolerance. Caches are write-through and, therefore, global storage is always up-to-date. Further, Concord uses a distributed coordination service to detect a failed cache instance and then embeds a recovery mechanism in the protocol.

B. Concord Architecture

Concord has an organization similar to existing FaaS platforms [1], [19], with enhanced load balancer and node controllers. Concord adds an *Application Controller* and, in each node, a *Cache Agent* (CA) for each cache instance. Figure 4 overviews the architecture.

Cache Agent. Each cache instance is managed by a cache agent, which has three roles. First, storage requests issued by a function are transparently intercepted by the function's runtime and forwarded to the corresponding cache agent. The cache agent can either satisfy the request locally, forward it to a remote cache agent, or forward it to the global storage. Second, to maintain cache coherence, the cache agent manages a *Data Directory* for the data items homed locally. The directory entry for a given data item stores the list of remote cache agents that have the data item in their local caches (*sharers*). When a node modifies the data item, the cache agent in the data item's home invalidates all other sharers. Finally, the cache agent re-purposes the allocated unused memory of all the co-located instances of functions that belong to an application into that application's local cache instance. The agent monitors the memory use of the functions and dynamically adjusts the size of the cache instance based on the total unused memory. When the cache instance needs to shrink, the agent evicts some data.

Node Controller. On every node, the node controller connects function instances with the appropriate cache agent. When a function instance is created, the node controller checks if there is a corresponding cache instance. If there is no such instance, the node controller creates it, together with its cache agent.

Application Controller. It stores a list of the nodes that host a cache instance of each application in a *Node Directory*. When a new cache instance is created or an old instance is removed, it informs other cache instances of the same application.

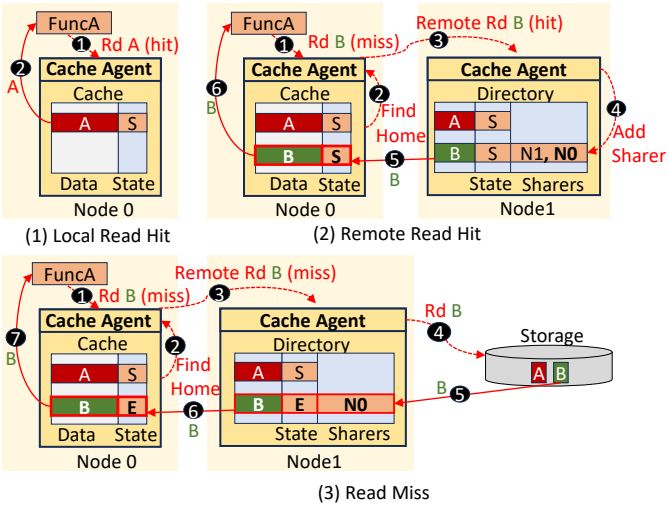
C. Distributed Directory-Based Coherence Protocol

1) Data Directory: The main software structure for maintaining cache coherence in Concord is the *Data Directory*. The data directory of an application is distributed and managed by the application's cache agents. Each entry in the directory corresponds to a data item, which is a blob of potentially different sizes accessed by its key. A directory entry stores the list of cache instances that currently cache the data item (*i.e.*, the data sharers), and whether the data item in the cache instances is in state Shared (S) or Exclusive (E) (in which case, there is a single sharer). The data item in a cache instance can be in one of three possible states: Exclusive (E), Shared (S) or Invalid (I). E and S states indicate that the data item is cached in a single cache instance or in potentially multiple cache instances, respectively, and that the data is coherent with storage. The cache instance caching the data in state E is the *data owner*. We use a MESI protocol without the M state; the latter is removed to enhance reliability.

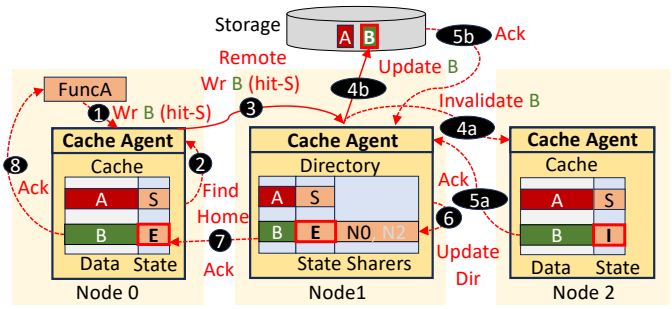
The directory of an application is distributed across all the nodes that have cache instances of that application. A given data item has its directory entry in its home node, where its *home cache agent* manages its entry. For example, in Figure 4, the homes and directory entries of data items A and B of application $App1$ are in Node 0 and Node 1, respectively.

The home cache agent of a data item is determined via consistent hashing [44]. Consistent hashing is a common technique used in distributed systems to shard the keys uniformly across a cluster of nodes. The goal is to minimize the number of keys that need to be moved when nodes are added or removed from the cluster, thus reducing the impact of these changes on the overall system. In Concord, we use it for when the cache of a give application expands into more nodes or shrinks into fewer nodes. Specifically, in Concord, all the cache agents of an application form a consistent hashing ring. The hash of a cache agent ID determines the position of the cache agent in the ring. The hash of the address of a data item determines the position of the data item in the ring. The home of a given data item is the first cache agent that appears in the ring while traversing the ring clockwise starting from the data item's position. Thus, when cache agents are added to or removed from the ring, some data items may change homes. Note that this is a departure from conventional hardware schemes, where the location of the directory entry for a data item is fixed over time, as long as the number of nodes remains constant.

2) Coherence Operations: There are six coherence operations in Concord's cache coherence protocol. As we describe them, note that, to minimize traffic, when a cache instance evicts a data item, it does not inform the home.



(a) Read operations.



(b) Local write hit on a data item in S state.

Fig. 5: Coherence operations in Concord. Dashed lines indicate control messages, while full lines indicate data transfers.

Local read hit. It occurs when a read finds the data item in the local cache instance (Figure 5a-1). The local cache instance provides the data item.

Remote read hit. When a read does not find the data item in the local cache instance (Figure 5a-2), the cache agent forwards the read to the home cache agent of the data item (1-3). If the home has a directory entry for the data item, this is a remote read hit. The directory entry can be in state S or E. Assume that it is in state S. In this case, the home cache agent gets the data item either from the local cache instance (if the home cache instance has it) or from the global storage (otherwise). The directory then marks the requesting cache agent as a sharer (4), and forwards the data item to it (5). The requesting cache instance loads the data item in state S and passes it to the function process (6).

Assume, instead, that the data item is in state E in the directory. The home cache agent gets the data item from the owner cache instance, which downgrades its state to S. Both the requesting cache agent and the previous owner are marked in the directory as sharers in state S. The data item is then forwarded to the requesting cache instance, which loads it in state S. Note that, because of a cache eviction, the owner cache agent may respond that it does not have the data item anymore. In this case, the home cache agent gets the data item from storage, marks the requesting cache agent as sharer in state E and forwards the data item to the requester, which loads it in state E.

Read miss. When a read misses in the local cache and, on reaching the home, finds that the directory has no entry for the data item (Figure 5a-3), this is a read miss (1-3). The home cache agent fetches the data item from global storage (4), creates a directory entry for the data item, sets the requesting cache agent as sharer in state E (5), and sends the data item to the requesting cache instance (6), which loads the data item in state E and passes it to the function process (7).

Local write hit. When a write finds the data item in the local cache, this is a local write hit. The data item can be in state E or S. If it is in E state, the write updates the local cache and propagates to global storage, bypassing the home. The local cache agent does not accept external requests for the data item until the storage acknowledges the update.

If the data item is in S state (Figure 5b), the write updates the local cache and is propagated to the home cache agent (1-3). The home cache agent checks the corresponding directory entry, sends invalidations to any sharer cache instance (4a), and propagates the update to global storage (4b). All sharer cache instances invalidate their copies and then send acknowledgments to the home (5a). When the home cache agent receives all acknowledgments (including one from the storage), it updates the directory to mark only the requesting cache agent as owner in state E (6). Then, the home responds to the requesting cache agent (7), which marks the state of the local copy as E and informs the function process (8).

Remote write hit. When a write does not find the data item in the local cache instance, the cache agent forwards the update to the home cache agent of the data item. If the home has a directory entry for the data item, this is a remote write hit. Then, the transaction consists of the home cache agent sending invalidations to the current sharers and propagating the update to global storage. The action is slightly different depending on whether the directory entry has an owner in state E or multiple sharers in state S. In the first case, the home cache agent needs to send the invalidation to the owner and receive the acknowledgment before sending the update to global storage; in the second case, the home cache agent can send the invalidations to all the sharers and the update to global storage in parallel. In either case, when the home cache agent has received all the acknowledgments (including the one from the storage), the transaction follows steps 6, 7, and 8 of the local write hit.

Write miss. When a write misses in the local cache and, on forwarding the update to the home, finds that the directory has no entry for the data item, this is a write miss. The home cache agent propagates the update to global storage. On receiving the acknowledgment from the storage, the home cache agent creates a directory entry for the data item, sets the requesting cache agent as owner in state E, and sends an acknowledgment

to the requesting cache instance, which marks the entry in the local cache in state E and informs the function process.

We have seen that, when a node writes to a data item that is in state E in its local cache instance, the update propagates to storage directly while bypassing the home. This design is faster than having to go through the home. It exploits the common case when a node keeps updating the same data item while no other node is accessing the data item. It is also race free because any future read or write to the data item by another node must access the cache instance of the owner node before reading the data item or updating storage, respectively.

In all other writes, the home cache agent is informed of the update and acts as the point of serialization for writes to that data item. If multiple nodes want to update the data item concurrently, the home processes one write operation at a time, and waits until all nodes acknowledge the invalidation and the global storage is updated, before signaling that the write operation is completed. This eliminates any data races.

In theory, sending invalidations can slow down write transactions in Concord. However, in a write-through protocol (used in both Concord and the state-of-the-art [70]), the overall write latency is typically dominated by the update to the global storage and its acknowledgment. Indeed, except in the case when there is a single sharer in state E, storage update and its acknowledgment happen in parallel with sending invalidations and receiving acknowledgments from the sharers—therefore hiding the cache invalidation latency. If the number of sharers is so high that the invalidations to S nodes and their acknowledgments are on the critical path, Concord could potentially fall back to a versioning coherence protocol [35], [70], but we have not evaluated it. We evaluate the cost of invalidations in Section VI.

3) Support for External Reads/Writes: Concord allows other cloud workloads to share data with serverless functions via global storage, through what we call external reads/writes. When users deploy their FaaS applications to Concord, they specify which storage locations are going to be used by the applications (*e.g.*, folders in Azure Blob Storage [59]). Then, the system registers a *listener* FaaS function that is invoked on every storage update (including external writes) on these locations [63]. When the listener is invoked, it first checks if the write was triggered by a FaaS function or from an external application. If the latter, the listener forwards the update to the application controller. The controller forwards the update to the correct home cache agent, which handles the external write as a local write. Thus, even with external writes, functions never operate on stale data.

D. Dynamic Coherence Domains

We call the set of cache instances of an application a *coherence domain*. In Concord, the coherence domain of an application changes over time. This is because a cache instance is ephemeral: it is created when the first function instance of the application is loaded into the node, and it is destroyed when all the instances of all the functions of the application are removed from that node. A node controller removes a function instance from a node when its grace period expires (*e.g.*, after 10 minutes without being used [72]) or when there is no space for the instance in the node. To ensure correctness, all cache instances must know what is their current domain and how to compute the homes of all the data items in their application.

Consistent hashing enables cache instances to enter and leave a domain with minimal disruption [44]. Consider a cache instance leaving a domain. The instance first synchronizes with all the other cache instances in the domain, informing them about its departure. Then, such instances remove from their local directories any sharer pointer pointing to the node of the departing cache instance. In addition, the instances recompute the new home for all the data items that were homed in the departing cache instance. Note that, with consistent hashing, all nodes can compute the new data item homes in a distributed, decentralized manner. Then, the departing cache instance sends the directory entries of all the data items homed locally to the corresponding new home, and waits for an acknowledgment from the new home. Note that all the data items are re-homed into the same node—i.e., the next node clockwise on the hash ring. Finally, after all the data items have been moved, the remaining cache instances synchronize again. By using this two-phase commit protocol, Concord avoids data races. Concord takes similar steps when the domain expands after a new cache instance is created.

E. Memory Use in Concord

Concord does not reserve extra memory for the software caches. The size of the cache for an application is dynamically adjusted by leveraging the unused memory from all the co-located containers that belong to the same application. As a result, the cache does not increase the application's memory footprint. Large objects are cached only if sufficient unused memory is available, and they are evicted if their memory is needed for regular application operations. When the cache size reduces dynamically, some data items and directory entries may be evicted. However, no data item changes homes and, therefore, there is no rehashing of the data.

If caches were to increase an application's memory footprint, we could run the risk of having to evict containers due to lack of memory space. The result could be more container cold starts, which are more expensive than remote storage accesses.

F. Fault Tolerant Distributed Coherence Protocol

As core failures happen rarely in multicore processors, hardware coherence protocols do not typically include features for fault tolerance [26]. However, the software-based coherence protocol in a distributed environment must be robust to unexpected node failures. Hence, Concord is equipped with mechanisms to detect failed nodes and recover from them. Additionally, Concord ensures that the data remains consistent on a node failure. We consider each of the two aspects in turn. Note also that serverless functions have inherent resilence to node failures due to their idempotent design principle.

First, to detect node failures, Concord uses a distributed coordination service that manages the membership of the

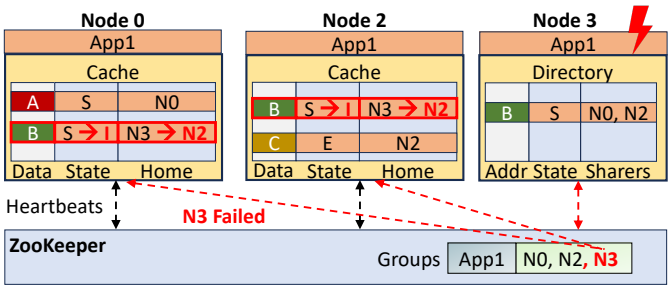


Fig. 6: Failure detection and recovery in Concord.

nodes. Figure 6 shows the mechanism. The coordination service periodically sends heartbeats to all the cache agents of each of the active applications. When a node does not respond to heartbeats, Concord assumes that the node failed. Then, it informs all the cache instances in the coherence domain(s) of the application(s) in the failed node. We use ZooKeeper [38] in our implementation. As ZooKeeper provides hierarchical namespaces, each application in Concord is treated as a separate group, and ZooKeeper manages the membership of cache instances per individual application. For example, in Figure 6, when Node 3 fails, ZooKeeper informs only the other cache instances of App_1, which happen to be in Nodes 0 and 2. If Nodes 0 and 2 also run other applications, the cache instances of those applications are not informed.

When the cache instances of an application receive a notification that one of the cache instances in their domain was in a node that failed, they first check if they locally cache any data homed in the failed cache instance. As caches are write-through, it is guaranteed that the data item in the global storage is up-to-date, and the only lost information is the directory. Hence, the cache agents in the domain evict from their caches all the data items homed in the failed cache instance. Then, they form a new consistent hash ring as described in Section III-D. In the example of Figure 6, Nodes 0 and 2 detect that locally cached data item B was homed in a cache instance in failed Node 3. Thus, they evict B from their caches. After recomputing the consistent hash ring, the figure assumes that B is now homed in Node 2.

Second, Concord ensures that the data remains consistent on a node failure. Node failures during reads are simple to handle, as reads at most change the directory state. During recovery, if the updated directory was lost, all sharers evict the data items in their caches that were homed in the failed node; if the reader node was lost, the directory removes the reader node from the list of sharer nodes for all the data items.

Writes are more complex, as they additionally modify the state of data items. The critical case is when the home node fails while processing a write that could have updated global storage but failed to invalidate all the cached copies. In this case, Concord must prevent the case of some cache instances reading the new value of the data item while others can still read the old value of the data item. This is prevented as follows: no cache instance is allowed to read the global storage for a data item that was homed in the failed node (and therefore potentially read a new value) until the recovery is complete. During the recovery, as nodes discover the failed node, they evict from their caches the data items homed in the failed node. By the time the recovery is over, all the old versions of the data item in cache instances are explicitly invalidated and a new home is declared. The next read will necessarily go through the new home, which will read the latest data item value from storage. All subsequent reads will see the new value while no read will see the old value. No data inconsistency will occur.

G. Coherence-Aware Invocation Scheduling

In conventional systems, function invocations are typically scheduled on a node that has a warm container of the function to minimize cold start overheads [43]. In high-load environments, a given function may have concurrent instances on several nodes. In this case, existing systems schedule an invocation randomly on any of the nodes that have an instance of the function. These invocations may operate on the same or different data, typically determined by the invocation's inputs, which *are visible* to the provider. With random scheduling, invocations operating on the same data may be scheduled on different nodes. This results in frequent remote accesses, increased number of cache instances sharing data and, consequently, a larger number of coherence invalidations.

Concord proposes *coherence-aware* invocation scheduling. When the load balancer receives a function invocation, it picks the function instance to send it to as follows. It computes the hash of the invocation inputs and uses the result to pick one of the nodes that has an instance of the function. By doing this, the system maximizes the chances of local cache hits.

It is possible that the chosen node is overloaded. In this case, the load balancer does not send the invocation to it. Instead, it tries with another hash function and picks the resulting node. If multiple tries picked overloaded nodes, the load balancer picks a random non-overloaded node. Overall, Concord densely packs the invocations of a function operating on the same data, to minimize data transfer overheads and coherence traffic.

H. Verification of the Software-Based Concord Protocol

During fault-free operation, the software-based Concord protocol follows the well-established ESI protocol from hardware schemes. The corner cases occur on node failure, system recovery, and coherence domain expansion/reduction. The actions taken in these cases are described in Sections III-D and III-F. To verify all cases, we use the TLA+ formal specification and verification language, and model-check the protocol in TLC [48]. With TLA+, we first specify all the states and possible actions from all the states. Specifically, we use Exclusive, Shared, and Invalid for the cache states, Active and Failed for the node states, Sharers+Ownership for the directory states, and ActiveInstances for the set of nodes that participate in a coherence domain. We model the events {Local/Remote}{Read/Write}Hit, {Read/Write}Miss, DataEvict, NodeFail, RecoverOnFail, and DomainChange.

We check for no deadlock and no livelock concurrency conditions, and prove that they always hold. The checks are performed under fault-free operation and under node failure. We next describe two corner cases and show how Concord ensures forward progress.

First, consider a Node A waiting for an invalidation acknowledgment from a failed or unreachable Node B. In Concord, Node A will not wait forever. If Node B has failed, Concord's coordination service (*i.e.*, ZooKeeper) will detect it through its heartbeats. If, instead, Node B is simply unreachable from A, Node A will timeout and inform the Application Controller to delete the cache instance on Node B. In both cases, the coordination service then notifies all the nodes in the same coherence domain(s) that the cache instances in B left the domain(s). Then, Node A cancels the waiting request and all the nodes perform the actions described in Section III-F

Second, consider that a read from Node A misses in the local cache, is forwarded to the home Node B and, in the meantime, the cache instance in Node B leaves the coherence domain. In this case, Node B cannot respond to the read request. In Concord, Node A does not wait forever. Node B initiates the creation of a new consistent hash ring as explained in Section III-D. In the process, all the nodes check if they have any outstanding operations with Node B. If a node, such as A does, it cancels the read, recomputes the correct new home, and reissues the operation. All these operations are performed in software.

We also check two data consistency invariants. The first one is that the coherence states in all the caches are correct. The second one is that a read to a valid cache location returns the value last written to it.

IV. Unlocking New Capabilities with Concord

We further leverage Concord's coherence protocol to unlock two new capabilities in FaaS environments: transactions and communication-aware function placement.

A. Support for Transactions

Transactions could be useful in many FaaS applications, such as banking and online shopping. However, current FaaS systems do not inherently support transactions. To execute sections of code atomically, users need to write the code in a manner that guarantees atomicity. For example, with AWS Saga patterns [13], users write additional functions to detect transaction violations and roll back to the correct state. State-of-the-art proposals for transactions log all the storage accesses that happen within a transaction to enable safe rollbacks (*e.g.*, Beldi [14]) or use global storage locks. Both practices can penalize performance due to the logging overheads and the reduced storage availability due to the locks.

The Concord coherence protocol can automatically and user-transparently ensure transaction atomicity, improving programmability while delivering high performance. The user only needs to specify the beginning and end of the transaction. A transaction can be a piece of a function, a whole function, or multiple functions of an application. Then, the Concord caching layer monitors data accesses and determines if accesses from any function conflict with the accesses of the transaction.

In Concord, while a process P_1 is executing a transaction, every data item that it reads or writes is recorded in the local cache instance as Speculatively Read or Speculatively Written, respectively, and marked with the ID of the process. No other process P_2 executing the same application, either locally (and, therefore, accessing the same local cache instance) or remotely (and, therefore, accessing a remote cache instance that is kept coherent with the Concord protocol) is allowed to conflict with P_1. Specifically, if P_2 attempts to write a data item that has been speculatively read by P_1, or attempts to read or write a data item that has been speculatively written by P_1, the transaction in P_1 is squashed. Conflicts between two local processes are trivially detected on access to the local cache instance; conflicts between a local and a remote process are detected through the Concord cache coherence protocol: a locally-cached speculatively read data item receives an external invalidation or a locally-cached speculative written data item receives an external read or an invalidation.

Speculatively written data items remain buffered in the local cache instance and are not propagated to global storage. If a transaction is squashed, all its speculatively written data items in the local cache instance are discarded, and their Speculative bits and process ID field are cleared. The transaction can then be re-executed.

If a transaction completes, it proceeds to commit. For this, the runtime grabs a global lock to ensure that commits are serialized. In addition, it locks the directory entries for the data items accessed in the transaction. Then, the runtime forwards all the updates of the transaction to the global storage, and clears the corresponding Speculative bits and process ID fields. After this, the directory entries are unlocked and the global lock is released.

To ensure livelock freedom, forward progress, and fairness, Concord uses known techniques from software transactional memory [34]. They include exponential back-off on conflict and priority increases after multiple squashes.

Users must ensure that a transaction execution has no side effects beyond storage accesses and invocations of functions, such as HTTPs or other system calls. One could automatically detect these side effects and prevent them from being globally visible while the transaction is in progress, using techniques similar to those proposed in SpecFaaS [78]. We leave this exploration for future work.

B. Communication-Aware Function Placement

Conventional FaaS systems place different functions on nodes in the cluster independently from each other. This means that two functions that interact in a producer-consumer manner may well be placed on different nodes. In this case, performance suffers due to communication overheads. A higher-performance solution would co-locate both functions on the same node, to reduce network overhead and even

allow them to communicate via shared memory instead of via network RPCs [55], [81], [83]. Unfortunately, providers cannot easily co-locate producer-consumer functions since, for privacy reasons, they do not know which opaque-box functions communicate with each other.

Concord proposes *Communication-Aware Function Placement* to reduce communication overhead while still maintaining function privacy. The idea is to monitor coherence messages and use this coherence traffic to transparently identify functions that frequently interact with each other. These functions are then co-located in the same node. For example, if one function keeps writing to certain locations and a second one keeps reading these locations, we have identified a producer-consumer pattern.

Concord monitors coherence messages and builds a *Producer-Consumer Table (PCT)*, which lists small sets of functions that frequently communicate with each other. We call them *Paired* functions. Concord consults the PCT to decide where to place function instances. Specifically, when a cluster receives a new invocation of function F, Concord checks if there is already an available instance of F to serve it. If so, the instance is re-used, avoiding a cold start. Otherwise, Concord uses the PCT to place the new instance of F in the cluster. For this, it first checks if the cluster has an instance of a function paired with F. If so, Concord places the new instance of F on the same node to reduce communication overhead. If no such instance exists, Concord anticipates the resource needs of a Paired function and places the new instance of F on a node that can accommodate it plus a Paired function instance.

Overall, Concord uses both communication-aware function placement and coherence-aware invocation scheduling (Section III-G) to minimize communication overheads.

V. Evaluation Setup

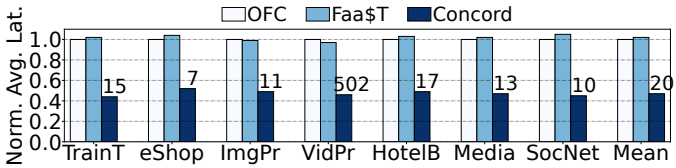
We evaluate Concord on OpenWhisk [1] in a 16-node cluster. Each node is an Intel Xeon Silver server with 20 cores, 192GB DRAM, and a 128MB LLC. It runs Ubuntu 20.04.

Evaluated Systems. We compare Concord to the state-of-the-art *Faa$T* [70] and *OFC* [65] designs (Section II). For all systems, we use write-through caches and an LRU replacement policy. We use an optimized Faa$T implementation that caches the version numbers of data items in the home [70], rather than having to fetch them from storage. All systems run on an optimized OpenWhisk implementation that supports the open-source MXFaaS [77] serverless framework. Hence, the cold start and scheduling overheads are minimized.

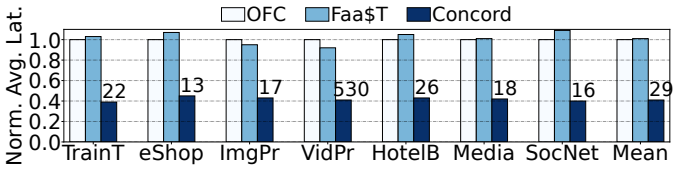
Evaluated Applications. We evaluate Concord with the 7 multi-function applications shown in Table II. Each function is deployed with the minimum amount of memory allowed with OpenWhisk (128MB), and all functions use less than this amount of memory throughout their whole execution. We use Azure Blob Storage [59] as the storage service for all the functions. The distribution of storage accesses is 80% reads and 20% writes with 5% read-only objects, which is the same as in Azure [70].

TABLE II: Serverless applications used in the evaluation.

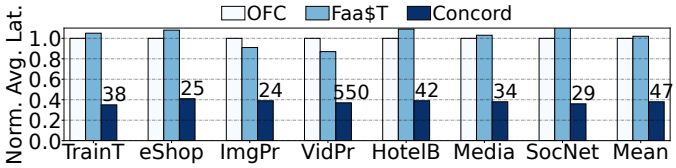
Application	Description
TrainT [4]	Book, cancel, or get remaining train tickets.
eShop [80]	Web e-commerce to browse and buy items.
ImgProc [85]	An image thumbnail generator pipeline.
VidProc [65]	Distributed video processing benchmark.
HotelBook [28]	A hotel reservation application.
MediaServ [28]	Review, rate, rent, and stream movies.
SocNet [28]	Social network application.



(a) Low request load.



(b) Medium request load.



(c) High request load.

Fig. 7: Average application request latency in OFC, Faa$T, and Concord normalized to OFC. The numbers on top of the Concord bars are the absolute request latencies in ms.

We evaluate Concord under low, medium, and high load levels, corresponding to an average of 500 requests per second (RPS), 1250 RPS, and 2000 RPS, respectively, received by the cluster. These load levels are chosen based on load testing: the low, medium, and high loads drive the CPU utilization of the cluster to about 25%, 50%, and 70%, which is representative [22], [33], [53], [69]. We use the Poisson distribution to model the request inter-arrival time [7], [18], [32], [71], [74], [76], [79], [87].

VI. Evaluation

In this section, we evaluate Concord's performance, scalability, memory consumption, robustness to coherence domain changes, sensitivity to available cache size, transactional support, and communication-aware function placement.

A. Concord Performance

We measure the applications' average request latency and throughput. The latency is measured end-to-end, from when the client sends a request until the result is received.

Application Latency. Figure 7 shows the average application request latency in OFC, Faa$T, and Concord with different

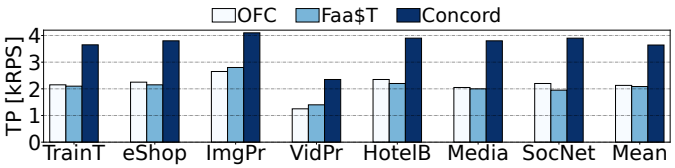


Fig. 8: Cluster throughput of OFC, Faa$T, and Concord measured in kilo requests per second (kRPS).

system loads normalized to OFC. On top of the Concord bars, we show the absolute latencies in ms. We see that OFC and Faa$T have similar latencies. While Faa$T allows a data item to be cached in multiple caches, it does not reduce the average latency over OFC because it has significant coherence overheads. On the other hand, Concord's optimized caching protocol minimizes network overheads and results in a much lower latency for all applications and across all loads. On average, Concord reduces the average application latency over OFC by 2.1×, 2.4×, and 2.6× for low, medium, and high load, respectively, and over Faa$T by 2.2×, 2.5×, and 2.7× for the same loads. Concord attains higher reductions for applications that frequently read small data, such as TrainT and SocNet; in these cases, the impact of the Concord techniques is more notable. Also, Concord's latency reductions increase with higher system loads.

Cluster Throughput. Concord's reduced request latencies result in improved overall cluster throughput. We define cluster throughput as the rate of requests that the cluster can process before violating the applications' Service Level Objectives (SLO). Similar to prior art [23], [51], [64], we define SLO as five times the application latency on an unloaded cluster. Figure 8 shows the cluster throughput of OFC, Faa$T, and Concord. On average, Concord improves the throughput over OFC and Faa$T by 1.7× and 1.8×, respectively.

Characterization of Concord Operations. To understand the performance of Concord, we now characterize some of its operations. Unless otherwise indicated, the data corresponds to the average of low, medium, and high load conditions. First, we note that Concord enables fast reads. Recall that a read request in Concord can result in a local hit, a remote hit, or a remote miss. It can be shown that, on average, a local hit, a remote hit, and a remote miss for a read in Concord take 1.6ms, 3.1ms, and 32ms, respectively.

Table III shows the distribution of read accesses per application for (i) Concord without coherence-aware invocation scheduling (Section III-G) and (ii) the complete Concord. In both cases, the techniques of Section IV are not included. We can see that, in Concord, on average 83% of read requests are local hits. Even without the proposed coherence-aware invocation scheduling, on average 75% of read requests are local hits. With so many accesses satisfied with low latency, Concord delivers high performance.

Write requests are less frequent, and can be slightly slower due to the invalidation messages sent to sharers. Figure 9 shows the average and maximum number of invalidations sent

TABLE III: Distribution of read operations in Concord without coherence-aware invocation scheduling (*C-NoCAS*) and in Concord (*C*).

	Local Hit [%] (C-NoCAS — C)	**Remote Hit** [%] (C-NoCAS — C)	**Remote Miss** [%] (C-NoCAS — C)
TrainT	72 — 84	21 — 9	7 — 7
eShop	68 — 75	23 — 16	9 — 9
ImgProc	76 — 85	18 — 9	6 — 6
VidProc	73 — 81	19 — 11	8 — 8
HotelBook	82 — 90	13 — 5	5 — 5
MediaServ	79 — 88	15 — 6	6 — 6
SocNet	73 — 81	21 — 13	6 — 6
Average	75 — 83	18 — 10	7 — 7

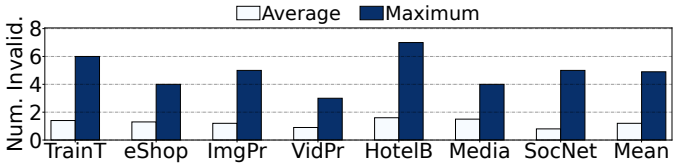


Fig. 9: Average and maximum number of invalidation messages sent per write operation in Concord.

per write operation throughout the execution of the evaluated applications. Averaged across all applications, an average write operation causes 1.2 invalidations, while the maximum number of invalidations per write is 4.9. Recall that our platform has 16 nodes.

Finally, Figure 10 compares the average request latency for Concord without coherence-aware invocation scheduling (Concord No CAS) and Concord for the different applications. The bars are normalized to Concord No CAS and the Concord bars are annotated with the average request latency. Concord No CAS tries to co-locate invocations of the same function on the same subset of nodes. Thus, it already captures some locality. However, it does not consider the specific data that these invocations operate on. Co-locating invocations that potentially operate on the same data increases data reuse, which results in higher local cache hits. As we see in the figure, coherence-aware invocation scheduling reduces the average request latency by 11%.

B. Write Operation Scalability

In this experiment, we measure the latency of write operations to shared data in Concord as we change the cluster size from 1 to 30 nodes. All nodes first load the data item in their caches and then one of them writes, invalidating all the other nodes. The home node sends the invalidations and receives the acknowledgments in parallel with propagating the write to global storage and receiving the acknowledgment from global storage. As a reference, a round trip to storage takes around 30ms, while the round trip of an invalidation to another node and its acknowledgment takes around 2ms.

Figure 11 shows the average latency of these writes across all the evaluated applications for Concord and Faa$T. Recall that a write in Faa$T sends the update to the home and then to the global storage, but does not invalidate the sharers. For

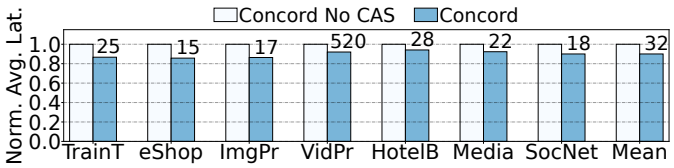


Fig. 10: Normalized average request latency in Concord without coherence-aware invocation scheduling (Concord No CAS) and in Concord. The numbers on top of the Concord bars are the absolute latencies in ms.

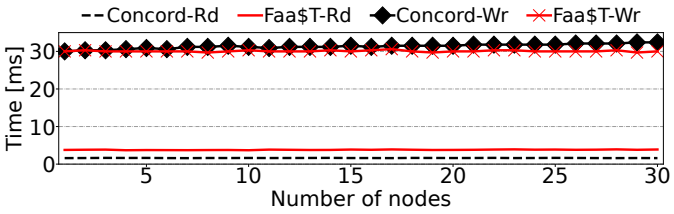


Fig. 11: Average time to perform a write operation on a data item shared by all nodes and a read operation that hits in the local cache for different numbers of nodes in *Faa$T* and *Concord*.

comparison, the figure also shows the latency of a read hit to the local cache in both Concord and Faa$T.

The figure shows that, with few nodes, a write in both Concord and Faa$T has the same latency of 30ms—which is determined by the access to the global storage. As the number of nodes increases, a write in Faa$T maintains the same latency, since no node is invalidated. In Concord, the latency increases, reaching 32.4ms for 30 nodes. The reason is that more invalidations are being sent. However, the latency increase is modest because invalidations are sent in parallel with the access to global storage.

The figure also shows the latency of read hits, which does not change with the number of nodes. It is 3.8ms in Faa$T and 1.6ms in Concord. The latency is Faa$T is higher because even a local read hit needs to perform an access to the home to check the version number. Overall, Concord speeds-up the frequent read operations by more than 2× while slowing down the less frequent write operations by at most 8%.

C. Concord Memory Consumption

Concord re-purposes the unused memory pre-allocated by the functions of an application into the application cache. Hence, users are not charged extra for the caches. However, the amount of cache memory is limited by the amount of unused memory. Fortunately, the unused memory is typically significantly larger than the amount of memory needed by the caches in Concord. Figure 12 shows the average and maximum amount of memory consumed by a single cache instance. Across all applications, the average and maximum sizes of a cache instance are 6.2MB and 12.6MB, respectively. On the other hand, the average unused memory per application in a node is 56.8MB. Hence, a cache instance typically uses a bit more than one tenth of the unused application memory.

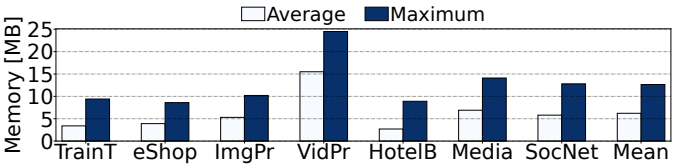


Fig. 12: Average and maximum memory consumed by a cache instance of an application throughout the application's execution in Concord.

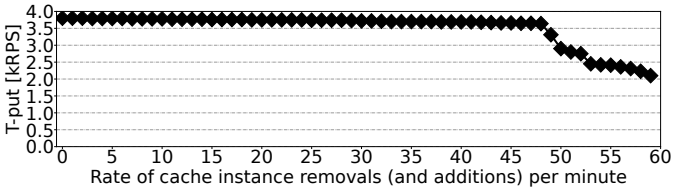


Fig. 13: Throughput of the SocNet application when varying the rate of cache instance removals (and additions).

D. Concord Coherence Domain Changes

Concord removes and adds cache instances to a coherence domain transparently to the application. While such operations take some time, they are not blocking. For example, on average, the latency of removing a cache instance from a 16-node coherence domain and adding a new one is about 120ms, but all the nodes beyond the one being removed or added do not stall unless they try to access a data item that moves homes. Figure 13 considers the SocNet application and shows the cluster throughput as we change the rate of cache instance removal (and subsequent addition). We use a coherence domain that extends across 16 nodes and randomly select instances to evict and add them back. The figure shows that Concord maintains high throughput until a removal rate as high as 48 removals (and additions) per minute. The other applications show similar performance trends.

E. Sensitivity Analysis

Available Node Cache Size. Figure 14 shows the speedup of Concord over OFC with different node cache sizes at medium load. We define speedup as reduction in average latency, and show the results averaged across all applications. With very small cache sizes (tens of KBs), Concord provides little benefit due to frequent cache evictions. As the cache size increases, the speedup increases, reaching 2.5. Once the cache captures the application's working set at about 6-7MB, further increases in size provide little benefit.

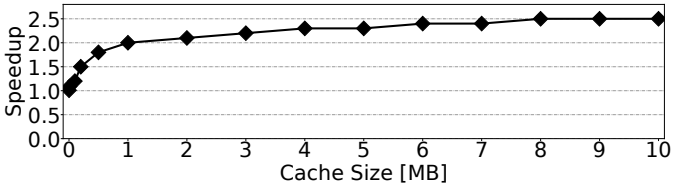


Fig. 14: Speedup of Concord over OFC with different cache sizes at medium load, averaged across all applications.

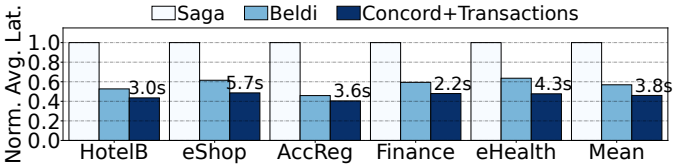


Fig. 15: Average application latency in Saga, Beldi, and Concord with transactions. The numbers on top of the bars are the absolute latency.

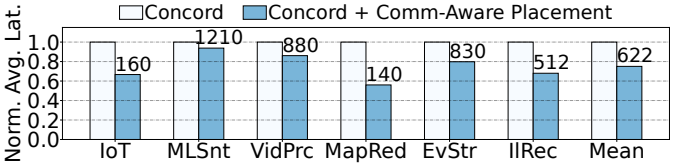


Fig. 16: Normalized average application latency with Concord and with Concord plus our communication-aware function placement policy. The numbers on top of the bars are the absolute latencies in ms.

Remote Node Access Latency. The cluster used in our experiments has whole-stack internode round trip latencies of around 2ms. This is a typical latency in current datacenters. If this round-trip latency is reduced to a few μs, the gains of Concord over OFC or Faa$T decrease—since the additional access to the home node required for most read operations in OFC and Faa$T becomes cheaper. Such scenario could be the case with rack-level CXL deployments.

F. Transaction Support

To evaluate transactions, we use five applications from AWS samples [10]: Hotel Booking, Online Shopping, Account Registration, Online Banking, and Online Health Records. These applications have large transactions: a transaction encloses a sequence of 6-8 functions plus the scheduling and FaaS platform overheads. Figure 15 shows the average application latency when using transactions implemented with Saga [13], Beldi [86], and Concord.

Concord outperforms the other schemes for two main reasons. First, it detects transaction conflicts much faster, thanks to using coherence messages; the other schemes detect conflicts by re-reading the data (or logs) from the storage. Second, Concord does not require the execution of additional functions to clean-up an aborted state; it rolls back to the correct state by just flushing its software caches. Overall, Concord with transactions reduces the average application latency by 54% and 20% over Saga and Beldi, respectively.

G. Communication-Aware Function Placement

To evaluate communication-aware function placement, we cannot use the applications from Table II because they do not have frequent producer-consumer patterns. Instead, we use a new set of applications with such patterns from open-source projects [45], [52]. These applications are: IoT Sensor Data Collection, ML Sentiment Analysis, Video Processing, Map Reduce, Event Streaming, and Illegal Recognizer. Figure 16 shows the normalized average latency of these applications with Concord and with Concord plus our communication-aware function placement policy. The numbers on top of the bars are the absolute latencies in ms. We see that co-locating the functions that communicate frequently with each other can significantly reduce application latency. On average, communication-aware function placement reduces the applications' average latency by 25%. The reductions are higher for the applications with shorter execution times, as the network overheads dominate.

VII. Comparison to a Fault-Tolerant Protocol

Apta [67] is a hardware-based cache coherence protocol targeting CXL that uses write-through hardware caches for fault tolerance. In this section, we create a software version of Apta's protocol using software caches, run it in our cluster of Section V, and compare its performance to Concord. Apta uses separate compute and memory nodes, and places the directory in the memory nodes. Apta is described without explicitly considering persistent storage (unlike Concord).

Apta introduces *lazy invalidations*, where the invalidations issued by the directory on a write are moved out of the critical path of the write—allowing the write to complete while some caches may hold stale data items for a short window of time. Apta then enforces *coherence-aware scheduling*, where the memory nodes tell the scheduler not to schedule functions that might use such data items, on the nodes that temporarily have stale values of such data items.

Some additional differences between the extended implementation of Concord and Apta are that Concord introduces: 1) *coherence-aware invocation scheduling* (where a hash of the function inputs determines the node where to schedule the function, hoping to reuse state left by the same function with the same inputs), 2) *transaction execution* (easy to support because all accesses in Concord are recorded in software), and 3) *communication-aware function placement* (where Concord learns producer-consumer function pairs and schedules them together). On the other hand, Apta introduces *locality-aware scheduling* (where a function is scheduled on the node where its predecessors executed).

Due to the limited size of our cluster and to consider the worst case for Concord, we compare Concord with 15 compute nodes to the software version of Apta with 15 compute and 15 memory nodes in two cases. First, in *Apta-Az* and *Concord-Az*, both systems need to propagate updates to Azure Blob Storage (like in our Concord design). Second, in *Apta-Mem* and *Concord-Mem*, updates are propagated only to memory nodes (like in the Apta paper); in this case, we add 15 memory nodes in *Concord-Mem* as well. Figure 17 shows the average application latency at medium load in all four environments normalized to Apta-Az. The Concord bars include the communication-aware function placement optimization. The numbers on top of the Concord-Mem bars are the absolute application latency values in ms.

On average across applications, Concord-Az and Concord-

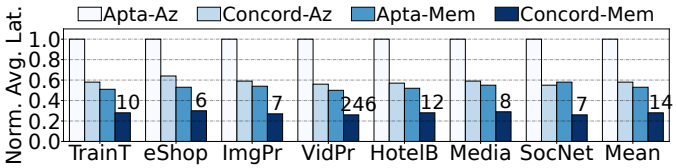


Fig. 17: Average application latency in Apta and Concord normalized to Apta-Az. The numbers on top of the Concord-Mem bars are the absolute application latency values in ms.

Mem reduce the average application latency over Apta-Az and Apta-Mem by 41.2% and 47.4%, respectively. Focusing on the *Mem* environments, there are three reasons why Concord is faster. First, Concord allows all nodes to continue scheduling new function invocations without interruption, unlike Apta's approach with coherence-aware scheduling. On average, Apta's scheduler has only 8.9 out of 15 compute nodes available for scheduling new invocations at a time.

Second, Apta adds scheduling overheads, as on every function invocation, the scheduler contacts all memory nodes and checks which compute nodes are currently unavailable. Then, it schedules the request on an available compute node. On average, Apta increases the scheduler's response time by 2.8×.

Finally, Concord's coherence-aware scheduling and communication-aware placement optimizations are more effective than Apta's locality-aware scheduling optimization.

In the *Az* environments, the fact than writes may have fewer hops in Concord than in Apta directly affects performance. Indeed, Apta's write operations travel to both the memory nodes (to check the directory) and to the Azure Storage; in Concord, they go directly to the storage if they update a datum homed locally or a datum in E state. This is the reason for the E state in Concord's protocol. On average, Concord reduces the number of hops per write by 28.6%.

VIII. Related Work

Distributed Caching. Researchers have proposed various caching schemes in distributed systems [16], [17], [41], [50], [54], [57], [84]. These proposals target environments where the cache is an independent software system used by long-lived web services. In contrast, Concord targets serverless environments where caches are tightly coupled with the highly-dynamic ephemeral function instances. Some prior works target FaaS [46], [65], [68], [70], [73], [75], [82]. Most proposals do not provide coherent caches [46], [65], [68], [73], [82]. Also, some designs introduce custom APIs and make applications responsible for their data coherence management [73]. We compare Concord to closely related work, namely Faa$T [70] and OFC [65], throughout our paper. If the developer annotates all read-only objects, Faa$T can avoid version checks for such objects, potentially improving performance. However, in the Azure traces [60], only 5% of the objects are read-only. Thus, Concord still outperforms Faa$T with annotations in this configuration.

Producer-Consumer Optimizations. SAND [7] and Faastlane [47] optimize producer-consumer patterns across the functions of an application workflow. They do not optimize the storage accesses to persistent objects. SAND [7] uses a hierarchy of local and global per-function queues used for local and remote communication, respectively, leading to multiple overheads: (i) communication via queues is serialized, creating bottlenecks in high-load scenarios, (ii) the publish-based messaging is slower than RPCs, and (iii) the hierarchy introduces multiple hops to read/write data when the communicating functions are located on different nodes. Our experiments show that, with conventional scheduling, Concord reduces the average latency of SAND by 8% for workflows with producer-consumer patterns. When Concord adds communication-aware function placement, it reduces the average latency of SAND by 31%.

Faastlane [47] consolidates all functions of an application into a single container, enabling communication through loads and stores with Intel's Memory Protection Keys (MPK). This approach performs similarly to Concord when all accesses hit in the cache. However, it complicates resource management and scaling since the functions within a container can have diverse hardware requirements and software dependencies.

Leases have been extensively used in distributed systems [6], [15], [31], [35]. A lease gives its holder cache the right to read or read-write a cached object for a limited period of time. When the lease expires, the cached object is self-invalidated and the cache must renew the lease to cache the object again. All lease designs induce lease renewal overheads that commonly include communication with the storage server that grants them.

Scheduling. Proposals on locality-aware FaaS scheduling [27], [43] focus on scheduling same-function invocations on the same nodes to minimize cold-starts. Palette [5] co-schedules functions that operate on the same data, providing a new API for users and a load balancer that takes this hint into account. Concord *transparently* prioritizes the co-location of function invocations that have the same input parameters.

IX. Conclusion

This paper proposes *Concord*, a distributed software caching system for FaaS environments. Concord proposes a directory-based distributed coherence protocol for software caches. The protocol minimizes coherence traffic, reduces contention points, and is robust to failures and frequent creation/removal of application cache instances. Concord on average speeds-up FaaS execution by 2.4× and improves throughput by 1.7×, while using 6.2MB of idle application memory.

Acknowledgments

This work was supported in part by NSF under grants CNS 1956007 and CCF 2107470; by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA; and by the IBM-Illinois Discovery Accelerator Institute.

REFERENCES

[1] "Apache OpenWhisk," https://openwhisk.apache.org/.

[2] "Knative," https://knative.dev/docs/.

[3] "OpenFaaS," https://docs.openfaas.com/.

[4] "Serverless Train Ticket," https://github.com/FudanSELab/serverless-trainticket.

[5] M. Abdi, S. Ginzburg, X. C. Lin, J. Faleiro, G. I. Chaudhry, I. Goiri, R. Bianchini, D. S. Berger, and R. Fonseca, "Palette load balancing: Locality hints for serverless functions," in *Proceedings of the 18th European Conference on Computer Systems (EuroSys)*, 2023.

[6] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer, "FARSITE: Federated, available, and reliable storage for an incompletely trusted environment," in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, 2002.

[7] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "SAND: Towards High-Performance Serverless Computing," in *2018 USENIX Annual Technical Conference (USENIX ATC '18)*, 2018.

[8] Alibaba, "Function Compute," https://www.alibabacloud.com/product/function-compute.

[9] Amazon AWS, "AWS Lambda," https://aws.amazon.com/lambda/.

[10] Amazon AWS, "AWS Samples: AWS Serverless Workshops," https://github.com/aws-samples/aws-serverless-workshops/.

[11] Amazon AWS, "Developing for retries and failures," https://docs.aws.amazon.com/lambda/latest/operatorguide/retries-failures.html.

[12] Amazon AWS, "Implementing statelessness in functions," https://docs.aws.amazon.com/lambda/latest/operatorguide/statelessness-functions.html.

[13] Amazon AWS, "Saga pattern," https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-data-persistence/saga-pattern.html.

[14] Amazon AWS, "Implement the serverless saga pattern by using AWS Step Functions," https://docs.aws.amazon.com/prescriptive-guidance/latest/patterns/implement-the-serverless-saga-pattern-by-using-aws-step-functions.html, 2023.

[15] T. E. Anderson, M. Canini, J. Kim, D. Kostić, Y. Kwon, S. Peter, W. Reda, H. N. Schuh, and E. Witchel, "Assise: Performance and availability via client-local NVM in a distributed file system," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, 2020.

[16] B. Berg, D. S. Berger, S. McAllister, I. Grosof, S. Gunasekar, J. Lu, M. Uhlar, J. Carrig, N. Beckmann, M. Harchol-Balter, and G. R. Ganger, "The CacheLib Caching Engine: Design and Experiences at Scale," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, 2020.

[17] D. S. Berger, B. Berg, T. Zhu, S. Sen, and M. Harchol-Balter, "RobinHood: Tail Latency Aware Caching – Dynamic Reallocation from Cache-Rich to Cache-Poor," in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, 2018.

[18] V. M. Bhasi, J. R. Gunasekaran, P. Thinakaran, C. S. Mishra, M. T. Kandemir, and C. Das, "Kraken: Adaptive Container Provisioning for Deploying Dynamic DAGs in Serverless Platforms," in *Proceedings of the 12th ACM Symposium on Cloud Computing (SoCC '21)*, 2021.

[19] M. Brooker, M. Danilov, C. Greenwood, and P. Piwonka, "On-demand Container Loading in AWS Lambda," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '23)*, 2023.

[20] L. Censier and P. Feautrier, "A new solution to coherence problems in multicache systems," *IEEE transactions on computers*, vol. 100, no. 12, pp. 1112–1118, 1978.

[21] D. Chaiken, J. Kubiatowicz, and A. Agarwal, "LimitLESS Directories: A Scalable Cache Coherence Scheme," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '91)*, 1991.

[22] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms," in *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*, 2017.

[23] C. Delimitrou and C. Kozyrakis, "Amdahl's Law for Tail Latency," *Commun. ACM*, vol. 61, no. 8, jul 2018.

[24] Engineering at Meta, "Cache made consistent," https://engineering.fb.com/2022/06/08/core-infra/cache-made-consistent/, 2024.

[25] O. Eytan, D. Harnik, E. Ofer, R. Friedman, and R. Kat, "It's Time to Revisit LRU vs. FIFO," in *Proceedings of the 12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '20)*, 2020.

[26] R. Fernández-Pascual, J. M. García, M. E. Acacio, and J. Duato, "Fault-Tolerant Cache Coherence Protocols for CMPs: Evaluation and Trade-Offs," in *Proceedings of the 15th International Conference on High Performance Computing (HiPC '08)*, 2008.

[27] A. Fuerst and P. Sharma, "Locality-Aware Load-Balancing For Serverless Clusters," in *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing (HPDC '22)*, 2022.

[28] 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 Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*, 2019.

[29] Google, "Google Cloud Functions," https://cloud.google.com/functions.

[30] Google Cloud, "Enable event-driven function retries," https://cloud.google.com/functions/docs/bestpractices/retries.

[31] C. Gray and D. Cheriton, "Leases: An efficient fault-tolerant mechanism for distributed file cache consistency," in *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, ser. SOSP '89, 1989.

[32] J. R. Gunasekaran, P. Thinakaran, N. C. Nachiappan, M. T. Kandemir, and C. R. Das, "Fifer: Tackling Resource Underutilization in the Serverless Era," in *Proceedings of the 21st International Middleware Conference (Middleware '20)*, 2020.

[33] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, L. Mao, and Y. Bao, "Who Limits the Resource Efficiency of My Datacenter: An Analysis of Alibaba Datacenter Traces," in *Proceedings of the International Symposium on Quality of Service (IWQoS '19)*, 2019.

[34] T. Harris, J. Larus, and R. Rajwar, *Transactional Memory*, 2nd ed. Morgan and Claypool Publishers, 2010.

[35] T. Haynes and P. Noveck, "Network file system (NFS) version 4 protocol," https://datatracker.ietf.org/doc/html/rfc7530.

[36] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless Computation with OpenLambda," in *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing (HotCloud '16)*, 2016.

[37] Huawei, "Cloud functions," https://developer.huawei.com/consumer/en/agconnect/cloud-function/.

[38] 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 (USENIX ATC '10)*, 2010.

[39] IBM, "IBM Cloud Functions," https://cloud.ibm.com/functions/.

[40] Jason Polites and Aparna Sinha, "The next big evolution in serverless computing," https://cloud.google.com/blog/products/serverless/the-next-big-evolution-in-cloud-computing.

[41] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "NetCache: Balancing Key-Value Stores with Fast In-Network Caching," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, 2017.

[42] A. Joosen, A. Hassan, M. Asenov, R. Singh, L. Darlow, J. Wang, and A. Barker, "How Does It Function? Characterizing Long-Term Trends in Production Serverless Workloads," in *Proceedings of the 14th ACM Symposium on Cloud Computing (SoCC '23)*, 2023.

[43] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis, "Hermod: Principled and Practical Scheduling for Serverless Functions," in *Proceedings of the 13th Symposium on Cloud Computing (SoCC '22)*, 2022.

[44] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web," in *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing (STOC '97)*, 1997.

[45] J. Kim and K. Lee, "FunctionBench: A Suite of Workloads for Serverless Cloud Function Service," in *Proceedings of the IEEE 12th International Conference on Cloud Computing (CLOUD '19)*, 2019.

[46] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic Ephemeral Storage for Serverless Analytics," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI '18)*, 2018.

[47] S. Kotni, A. Nayak, V. Ganapathy, and A. Basu, "Faastlane: Accelerating Function-as-a-Service Workflows," in *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC '21)*, 2021.

[48] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman, 2022.

[49] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor," in *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA '90)*, 1990.

[50] J. Li, J. Nelson, E. Michael, X. Jin, and D. R. K. Ports, "Pegasus: Tolerating Skewed Workloads in Distributed Storage with In-Network Coherence Directories," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, 2020.

[51] Z. Li, L. Zheng, Y. Zhong, V. Liu, Y. Sheng, X. Jin, Y. Huang, Z. Chen, H. Zhang, J. E. Gonzalez, and I. Stoica, "AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving," in *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI '23)*, 2023.

[52] Z. Li, Y. Liu, L. Guo, Q. Chen, J. Cheng, W. Zheng, and M. Guo, "FaaSFlow: Enable Efficient Workflow Execution for Function-as-a-Service," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*, 2022.

[53] Q. Liu and Z. Yu, "The Elasticity and Plasticity in Semi-Containerized Co-Locating Cloud Workload: A View from Alibaba Trace," in *Proceedings of the 9th ACM Symposium on Cloud Computing (SoCC '18)*, 2018.

[54] Z. Liu, Z. Bai, Z. Liu, X. Li, C. Kim, V. Braverman, X. Jin, and I. Stoica, "DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching," in *17th USENIX Conference on File and Storage Technologies (FAST '19)*, 2019.

[55] A. Mahgoub, K. Shankar, S. Mitra, A. Klimovic, S. Chaterji, and S. Bagchi, "SONIC: Application-aware Data Passing for Chained Serverless Applications," in *2021 USENIX Annual Technical Conference (USENIX ATC '21)*, 2021.

[56] M. Maximilien, D. Hadas, A. Danducci, and S. Moser, "The future is serverless," https://developer.ibm.com/blogs/the-future-is-serverless/.

[57] S. McAllister, B. Berg, J. Tutuncu-Macias, J. Yang, S. Gunasekar, J. Lu, D. S. Berger, N. Beckmann, and G. R. Ganger, "Kangaroo: Caching Billions of Tiny Objects on Flash," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*, 2021.

[58] Microsoft, "Serverless architecture considerations," https://learn.microsoft.com/en-us/dotnet/architecture/serverless/serverless-architecture-considerations.

[59] Microsoft Azure, "Azure Blob Storage," https://azure.microsoft.com/en-us/products/storage/blobs.

[60] Microsoft Azure, "Azure Public Dataset," https://github.com/Azure/AzurePublicDataset.

[61] Microsoft Azure, "Designing Azure Functions for identical input," https://learn.microsoft.com/en-us/azure/azure-functions/functions-idempotent.

[62] Microsoft Azure, "Microsoft Azure Functions," https://azure.microsoft.com/en-gb/services/functions/.

[63] Microsoft Azure, "Create a function in Azure that's triggered by Blob storage," https://learn.microsoft.com/en-us/azure/azure-functions/functions-create-storage-blob-triggered-function, 2024.

[64] A. Mirhosseini and T. Wenisch, "μSteal: A Theory-Backed Framework for Preemptive Work and Resource Stealing in Mixed-Criticality Microservices," in *Proceedings of the ACM International Conference on Supercomputing (ICS '21)*, 2021.

[65] D. Mvondo, M. Bacou, K. Nguetchouang, L. Ngale, S. Pouget, J. Kouam, R. Lachaize, J. Hwang, T. Wood, D. Hagimont, N. De Palma, B. Batchakui, and A. Tchana, "OFC: An Opportunistic Caching System for FaaS Platforms," in *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys '21)*, 2021.

[66] V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood, *A primer on memory consistency and cache coherence. Second edition*. Springer Nature, 2020.

[67] A. Patil, V. Nagarajan, N. Nikoleris, and N. Oswald, "Āpta: Fault-tolerant object-granular CXL disaggregated memory for accelerating FaaS," in *Proceedings of the 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '23)*, 2023.

[68] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure," in *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*, 2019.

[69] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis," in *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC '12)*, 2012.

[70] F. Romero, G. I. Chaudhry, I. Goiri, P. Gopa, P. Batum, N. Yadwadkar, R. Fonseca, C. Kozyrakis, and R. Bianchini, "Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications," in *Proceedings of the 12th ACM Symposium on Cloud Computing (SoCC '21)*, 2021.

[71] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural Implications of Function-as-a-Service Computing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52)*, 2019.

[72] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider," in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*, 2020.

[73] S. Shillaker and P. Pietzuch, "Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing," in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*, 2020.

[74] A. Singhvi, A. Balasubramanian, K. Houck, M. D. Shaikh, S. Venkataraman, and A. Akella, "Atoll: A Scalable Low-Latency Serverless Platform," in *Proceedings of the 12th ACM Symposium on Cloud Computing (SoCC '21)*, 2021.

[75] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst: Stateful Functions-as-a-Service," *Proceedings of the VLDB Endowment*, 2020.

[76] J. Stojkovic, N. Iliakopoulou, T. Xu, H. Franke, and J. Torrellas, "EcoFaaS: Rethinking the Design of Serverless Environments for Energy Efficiency," in *Proceedings of the 51st Annual International Symposium on Computer Architecture (ISCA'24)*, Jun. 2024.

[77] J. Stojkovic, T. Xu, H. Franke, and J. Torrellas, "MXFaaS: Resource Sharing in Serverless Environments for Parallelism and Efficiency," in *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23)*, 2023.

[78] J. Stojkovic, T. Xu, H. Franke, and J. Torrellas, "SpecFaaS: Accelerating Serverless Applications with Speculative Function Execution," in *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA '23)*, 2023.

[79] A. Tariq, A. Pahl, S. Nimmagadda, E. Rozner, and S. Lanka, "Sequoia: Enabling Quality-of-Service in Serverless Computing," in *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC '20)*, 2020.

[80] The vHive Ecosystem, "vSwarm - Serverless Benchmarking Suite," https://github.com/vhive-serverless/vSwarm.

[81] D. Ustiugov, S. Jesalpura, M. B. Alper, M. Baczun, R. Feyzkhanov, E. Bugnion, B. Grot, and M. Kogias, "Expedited Data Transfers for Serverless Clouds," *CoRR*, vol. abs/2309.14821, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2309.14821

[82] A. Wang, J. Zhang, X. Ma, A. Anwar, L. Rupprecht, D. Skourtis, V. Tarasov, F. Yan, and Y. Cheng, "InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache," in *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)*, 2020.

[83] M. Wawrzoniak, I. Müller, G. Alonso, and R. Bruno, "Boxer: Data Analytics on Network-enabled Serverless Platforms," in *Proceedings of the 11th Annual Conference on Innovative Data Systems Research (CIDR '21)*, 2021.

[84] J. Yang, Y. Yue, and K. V. Rashmi, "A Large-Scale Analysis of Hundreds of In-Memory Key-Value Cache Clusters at Twitter," *ACM Transactions on Storage*, vol. 17, no. 3, 2021.

[85] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen, "Characterizing Serverless Platforms with ServerlessBench," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC '20)*, 2020.

[86] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu, "Fault-tolerant and transactional stateful serverless workflows," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, 2020.

[87] Y. Zhang, Goiri, G. I. Chaudhry, R. Fonseca, S. Elnikety, C. Delimitrou, and R. Bianchini, "Faster and Cheaper Serverless Computing on Harvested Resources," in *Proceedings of the International Symposium on Operating Systems Principles (SOSP '21)*, 2021.