UniCache: The Next 700 Caches for Serverless Computing

Jovan Stojkovic, Tianyin Xu, Hubertus Franke†, Josep Torrellas
University of Illinois at Urbana-Champaign †*IBM Research*

ABSTRACT

Different caches for serverless computing are typically effective for their target use cases. However, they do not generalize well to the diverse range of serverless applications and their workloads. To address this issue, we present the first taxonomy of data caching designs for serverless systems. The taxonomy dissects the complex design space into six dimensions, each with multiple design choices. The taxonomy enables us to understand the impact of individual design decisions on the overall cache performance. Based on the taxonomy, we introduce the *Unified Cache (UniCache)*—a general-purpose caching system for serverless environments. UniCache dynamically reconfigures itself by intelligently selecting the cache organization for each application and thus specializes itself to the application requirements and workloads using a hierarchy of reinforcement learning models. We implement UniCache on two serverless platforms and evaluate it with diverse applications. Compared to a state-of-the-art baseline, UniCache on average speeds-up application execution by 5.7× and improves throughput by 4.6×.

1 INTRODUCTION

Motivation. The rapid growth of serverless computing or function-as-a-service (FaaS) is due to its ability to provide application flexibility, fine-grain billing, and high resource utilization [9, 14]. With FaaS, users only need to upload their application code to the cloud provider, which is responsible for securing all the necessary resources needed to execute the code. The execution unit is a function, which runs in a container created and scheduled on demand in an event-driven manner. Serverless applications are then composed of multiple standalone functions that communicate with each other. All major cloud providers offer FaaS environments, such as AWS Lambda [4], Microsoft Azure [16] and IBM Cloud Functions [8].

Due to their ephemeral nature, serverless functions were expected to be stateless services [5, 17]. This property enables high availability, fast scalability, and fault tolerance. However, the only way to preserve state across invocations of a given function is via global storage. In addition, serverless functions in an application often cannot communicate with each other arbitrarily; instead, they communicate indirectly via global storage [13]. Therefore, the interactions between functions and global storage are on the critical path and often determine the overall application performance.

Related work. To mitigate the overhead of storage accesses, a popular approach is to cache data. Substantial prior art explored various aspects of data caching in distributed systems (e.g., [6, 12, 23]). These techniques are typically very effective at speeding-up application execution. However, their focus is on long-lived applications and, thus, they are unlikely to handle the highly-dynamic ephemeral nature of FaaS environments efficiently. Recently, researchers have proposed caching schemes designed for FaaS systems [11, 18, 19, 21]. However, majority of these schemes require user-configurations [11, 21] or transparently set the static cache configuration that remains unmodified throughout the lifetime of applications [18, 19].

These schemes are highly optimized for the specific use case they target, e.g., ML workloads [19]. On the other hand, heterogeneous serverless workloads have high diversity with large fluctuations over time, e.g., the size of the data used by a function can range from a few bytes to a few GBs. Further, the data can be accessed by a single function invocation or by hundreds of concurrent invocations. Finally, the data usage can vary from read-only across all functions of the application to frequently updated by a single function. As a result, designing a general, broadly usable high-performance caching system for FaaS environments is challenging.

Our work. To address this challenge, this paper presents the first *taxonomy* of data caching designs for serverless environments. The taxonomy systematically dissects the design space in six dimensions, each with multiple decision choices. These dimensions define how basic cache operations, such as data coherence, replacement, and replication are performed. The taxonomy enables us to understand the impact of individual design decisions on the overall cache performance, identify the relevant metrics for each design category, and map application- and system-level characteristics to the right cache configuration. The taxonomy sheds light on overlooked design aspects due to specializations of prior art.

Based on the principles derived from the taxonomy, we develop *Unified Cache (UniCache)*, the first comprehensive caching framework designed for FaaS environments. Caches in UniCache are organized into independent *cachelets*, i.e., cache instances associated with a given function or application. Each cachelet has its own configuration representing a set of values for the six taxonomy dimensions. This modular architecture enables effective integration of new protocols for any of the taxonomy dimensions, facilitating a plug-and-play paradigm that augments the caching system's capabilities without disrupting its operational integrity. As an example, a new cache replacement or cache coherence algorithm can be easily integrated without affecting the rest of the UniCache system.

UniCache is an automatically reconfigurable distributed caching scheme. It continuously monitors the application characteristics, such as data sizes and read/write ratio, and the system state, such as the load and network bandwidth. Then, the system periodically uses the collected information to recompute the best cachelet configuration using a set of reinforcement learning (RL) models. RL models are organized into a hierarchy: individual taxonomy dimensions have independent models, and a top-level model serves as a moderator, integrating the outputs of the individual models to determine the optimal configuration for each cachelet. By leveraging RL models, UniCache improves its robustness compared to the fine-tuned rigid heuristics. Moreover, new applications and new taxonomy categories protocols can be supported without the need to re-profile the system, making the scheme highly evolvable.

Results. UniCache can be integrated with existing serverless platform via minimal modifications. We implement UniCache on top

of OpenWhisk [1] and KNative [3]. UniCache does not require any hardware support, OS support, or changes to user-provided functions. We evaluate UniCache with a diverse set of serverless applications from FunctionBench [10], TrainTicket [2], ServerlessBench [24] and vSwarm [22]. UniCache on average speeds-up application execution by 5.7×, reduces the P99 tail latency by 4.3×, and improves throughput by 4.6× over a state-of-the-art baseline [19].

Contributions. This paper makes the following contributions:

- A first taxonomy of cache designs for serverless computing;
- UniCache, a generic cache framework which enables dynamic, intelligent reconfiguration of caches to specialize them for different severless applications;
- An implementation and evaluation of UniCache.

2 TAXONOMY OF CACHING SCHEMES

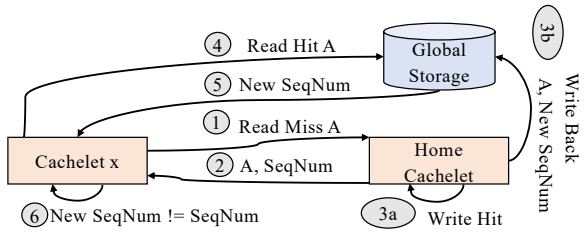
The optimal organization of a distributed cache in FaaS environments at a given time depends on both the application and system-level characteristics. To identify the optimal organizations, we start by analyzing the design space. Specifically, we propose *the first taxonomy* of distributed caching organizations for FaaS environments. Researchers and practitioners can use the taxonomy to *(1)* understand the impact of individual design decisions on overall cache performance, *(2)* devise metrics of interest for each design point, and *(3)* map an application's characteristics to the best cache organization. Table 1 shows the taxonomy. It has six dimensions. For each dimension, the table shows the design points and some metrics that determine the optimal design point. In the following, we define *Cachelet* as a cache instance with its own independent configuration associated with a single function instance or a group of instances co-located in the same node.

1. Cachelet Scope. Concurrently active containers on a given node can be instances of the same function, instances of different functions within the same application, or instances from different applications. Hence, a cachelet can be (1) *Per-Instance*—associated with a single instance, (2) *Per-Function*—shared among all the instances of the same function in the node, (3) *Per-Application*—shared among all the instances of all the functions of the same application in the node, or (4) *Shared*–shared among all the instances in the node. Depending on the data sharing patterns, different scopes may work best. The more sharing in a cachelet, more cache space is available and cross-instance data prefetching can be exploited. However, there is a higher chance of memory utilization unfairness and security implications (e.g., side channels) across users. Since *Shared* cachelets allow sharing across applications from different users, they are insecure and, hence, excluded from further discussion.

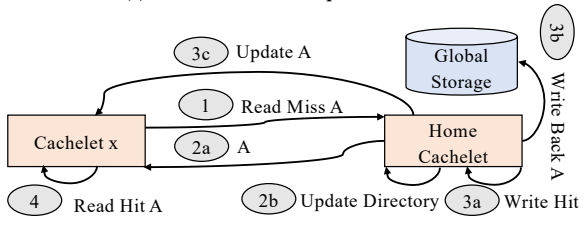
2. Cachelet Write Policy. On a data update, the cachelet can employ two strategies to update global storage. First, *Write-Through* caches immediately propagate the update to global storage, and wait for the global update to complete before sending an acknowledgment to the user. Second, *Write-Back* caches buffer the update, send an acknowledgment to the user right away, and propagate the update to global storage only when the dirty data is evicted from the cachelet. Write-Through caches provide higher fault tolerance and are simpler to implement. Write-Back caches typically improve the performance, especially at high loads and large data sizes.

3. Cachelet Replacement Policy. When a cache runs out of space and needs to evict an entry, it can use different policies, including evicting (1) the oldest data (*FIFO*), (2) the least recently used data (*LRU*), or (3) the least recently used data of the lowest priority (*Priority-LRU*). The priority can be determined, e.g., by the data size or the level of burstiness. FIFO caches are the easiest to implement, while LRU caches typically have higher performance. Priority-LRU caches take into account that objects have various sizes and different access patterns (e.g., bursty ones). Prior art has explored LRU [19] and FIFO [7] policies, but has not considered the non-uniform data sizes and access patterns commonly found in serverless applications. Thus, Priority-LRU has not been proposed. Other replacement policies, such as *random* or *clock*, can also be integrated with UniCache as new independent modules.

4. Cachelet Coherence Protocol. When multiple cachelets *in the same or in different nodes* operate on shared data, the system needs to keep the cachelets coherent. To maintain cache coherence, the system can use: (1) metadata (i.e., *SequenceNumbers*), or (2) directories using invalidations (*DirectoryInvalidate*) or updates (*DirectoryUpdate*). Figure 1 shows an example of protocol operation when using sequence numbers (Figure 1a) or directories with updates (Figure 1b). The example uses two coherent cachelets: Cachelet x and the Home Cachelet (i.e., the one responsible to keep the metadata or the directory for the coherence domain). In the example, Cachelet x first read misses on datum A, then the Home Cachelet writes A, and then x re-reads A. We use a write-through policy.



(a) Protocol based on sequence numbers.



(b) Protocol based on directories and updates.

Figure 1: Two cachelet coherence protocols.

With sequence numbers, *SeqNum*, (Figure 1a), x sends a read-miss request to the home cachelet (1), which responds with A's value and (*SeqNum*) (2). Later, the home updates A (3a) and writes back A's new value and new *SeqNum* to global storage (3b). After this, as x re-reads A, it hits locally. However, to ensure that x has the correct version, x has to send a request for A's *SeqNum* to global storage (or home) (4), which provides it (5). Then, x compares the received *SeqNum* to the one it has, finds that they are different (6), and sends a read miss request to the home to get A's correct value.

With directories and updates (Figure 1b), x sends the read-miss request to the home (1), which responds with A's value (2a) and

Table 1: Taxonomy of distributed data caching designs in serverless environments.

Category	Some Design Points			Metrics of Interest
Cachelet Scope	Per-Instance	Per-Function	Per-Application	Sharing across instances; Sharing across functions
Cachelet Write Policy	Write-Through	Write-Back		System Load; Write Portion; Data Size; Fault Tolerance
Cachelet Replacement Policy	FIFO	LRU	Priority-LRU	Data Popularity; Data Size; Number of Cache Elements
Cachelet Coherence Protocol	DirectoryInvalidate	DirectoryUpdate	SeqNumbers	Data Size; Num. Sharers; Write Portion; Write Run
Replication Policy	ReplData	ReplCoh	ReplData&Coh	Fault Tolerance; System Load; Frequency of Changes
Coherence Domain Organization	Flat	Hierarchical		Sharing across Nodes; Num. Sharers and Cache Elems

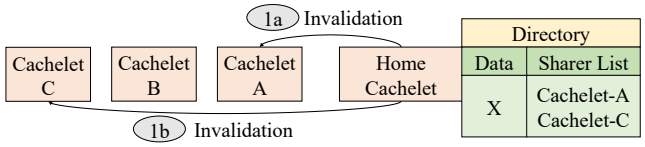
updates the directory (2b) adding x as a sharer. Later, the home updates A (3a) and writes back A's new value to global storage (3b). As part of the transaction, the home also updates all the sharers, including x (3c). After this, as x re-reads A, it satisfies the request locally (4). The data present in x is never stale.

Neither using sequence numbers nor using directories is better in all cases. Using sequence numbers is undesirable when data sizes are small (getting the sequence number already costs the same as getting the data), when data is read-only (the sequence number does not change) or when data is frequently written (sequence numbers always mismatch). In these cases, directories are superior. Directories are relatively less attractive when (1) there are many sharers of the data and, therefore, frequent updates or invalidations need to be sent—especially if the data size is large and the number of writes is moderate, and (2) the available network bandwidth is low. Similar to cache replacement policies, UniCache can integrate additional coherence protocols, e.g., leases, with rest of the system.

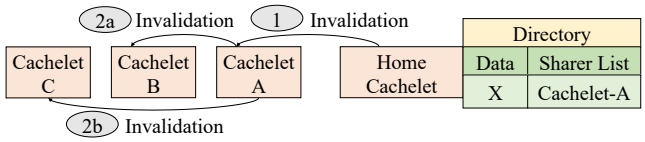
5. Coherence Domain Organization. A cache coherence domain can be *Flat* or *Hierarchical*. In the text we explain the directory organization. However, the same principles apply to other coherence protocols, such as sequence numbers or leases. In a flat directory, only the home cachelet can send invalidation or update messages to other cachelets. Further, when a cachelet requires data, it only communicates with the home cachelet. In a hierarchical directory, the coherence domain is organized in *clusters* of cachelets, and each cluster has a *Leader* cachelet. The home cachelet has information of which clusters hold copies of the data, and sends invalidations or updates only to their leaders. Then, leaders propagate the message to *all* the nodes in the cluster as they do not have information of which local cachelets have the data. When a cachelet requires data, it communicates with its cluster leader, which either immediately provides the data or requests it from the home cachelet.

Figure 2 shows a transaction where the home cachelet sends an invalidation to all the sharers. Suppose that Cachelets A and C have the data and Cachelet B does not have it. In the flat directory (Figure 2a), the home sends an invalidation to both A (1a) and C (1b). In the hierarchical directory (Figure 2b), the home only knows that the cluster led by A has the data. Hence, it sends an invalidation to A (1), which then forwards it to both B (2a) and C (2b). *Hierarchical* exploits data sharing across cachelets, reduces directory size, and mitigates a network bottleneck in the home. *Flat* reduces the number of coherence messages and the data access time when there is no data sharing across cachelets.

6. Replication Policy. The last axis involves trading off performance for fault tolerance through the use of state replication. We consider three cases: data replication, coherence replication, and



(a) Flat directory.



(b) Hierarchical directory.

Figure 2: Directory organizations in distributed caching.

both data and coherence replication. The first case applies to write-back cachelets. We can choose to replicate dirty data in multiple cachelets of the same coherence domain (*ReplData*). The second case applies to coherence protocols. We can choose to replicate the coherence state in multiple cachelets (*ReplCoh*). Finally, in write-back caches we can replicate both dirty data and coherence state across cachelets (*ReplData&Coh*). Replication increases fault tolerance: a cachelet can be lost and its dirty data or coherence state can still survive. However, replication reduces performance, as it requires more coherence messages and storage space.

3 UNICACHE DESIGN

With the insights from the taxonomy, we design *UniCache*, the first automatically reconfigurable caching scheme for serverless environments. UniCache achieves high performance by automatically and transparently selecting optimal cachelet configuration based on serverless applications. Every cachelet utilizes currently unused memory of the constituting function instances (hence, its size dynamically changes), and has its own configuration of each dimension of the taxonomy. Each node in UniCache has one *Server Caching Agent (SCA)* and as many *Client Caching Agents (CCAs)* as cachelets. The SCA connects the local function Invoker of the FaaS platform to the caching subsystem, while the CCAs handle cache accesses from the function instances and manage cached data. The SCA and CCAs monitor the application characteristics and system state, and use the information and a set of RL models to continuously recompute optimal configuration for each cachelet.

Server Caching Agent. When the local function Invoker creates a new function instance, it informs the SCA. The SCA checks if there is an existing cachelet that can serve the request by the newly created instance. If so, the SCA allows the new instance to share the existing cachelet, and increases the size of the cachelet by the

amount of the unused memory of the new instance. Otherwise, it creates a new cachelet for the instance. All global storage requests issued by the new instance will be transparently intercepted by the function's runtime and redirected to the chosen cachelet. Similarly, when the local Invoker removes a function instance from the node, it informs the SCA. If there are other instances served by the cachelet of the removed instance, the SCA just reduces the size of the cachelet to exclude the memory of removed instance. Otherwise, the SCA removes the cachelet along with the instance. To maintain coherence, a cachelet needs to be aware of all the other cachelets from the same coherence domain. The SCA broadcasts a cachelet creation message to the SCAs in all the other nodes. When an SCA receives such a message, it forwards it to the cachelets of the given coherence domain. These cachelets now add the newly created cachelet in the list of in-domain cachelets.

Client Caching Agent. A CCA is assigned to a cachelet. It communicates with the function instances associated with the cachelet, other CCAs from the same coherence domain, and the local SCA. When a function instance issues a *GET* or a *SET* request to the global storage, the function runtime transparently intercepts the request and forwards it to the associated CCA. For the *GET* requests, the CCA first checks if the data is present locally. If not, it sends a *READ-HOME* message to the home cachelet. Once the home responds, the cachelet inserts the data in the local cache and returns it to the requestor function instance. When the home cachelet receives a *READ-HOME* request, it provides the cached data or fetches it from the global storage if the data is not present locally. The home cachelet also maintains the information needed for cache coherence, such as the sharer list or per-data sequence numbers. For the *SET* requests, the CCA always sends a *WRITE-HOME* message to the home cachelet and updates the value locally once it receives an acknowledgment. The *WRITE-HOME* message triggers the home cachelet to perform operations of the cache coherence and fault tolerance protocols. The home cachelet updates the sequence number or sends update/invalidate messages to the sharers as needed, and updates the data in the global storage if necessary. Finally, the home responds to the requestor agent along with the potentially updated sequence number.

3.1 Cachelet Reconfiguration Principles

UniCache uses a set of Reinforcement Learning (RL) models to dynamically tune cachelet configuration for each application based on system conditions. We consider each category in Table 1 at a time: the RL model decisions for a given category depend on the *Metrics of Interest* for that particular category. UniCache continuously collects these metrics of interest while the application is running. Each of the metrics affects one or more taxonomy categories, with each category being influenced by at least two metrics. To generate the datasets and understand the design space we have performed extensive sensitivity studies running over 15,000 experiments with our serverless applications. We describe the intuitions derived from these experiments, and explain how UniCache organizes its RL models to automatically tune configuration for each cachelet.

3.1.1 **Cachelet Coherence Protocol.** To compute the optimal coherence protocol for a given cachelet, UniCache collects the data size, number of sharers, portion of writes, and write-run (i.e., the number of reads/writes following a single write). We vary the data size between 1KB-512MB in powers of two, the number of sharers between 2-64 in powers of two, the write portion between 0%-90% in increments of 10%, and the write-run between 1-10 in increments of one. Out of the resulting 12,000 experiments, we highlight the four environments (Exp1-Exp4) in Figure 3.

- ***Exp-1***. When the data is small (32KB), fetching the sequence number from global storage is expensive. Hence, Directories outperform SequenceNumbers by 78.9× on average.
- ***Exp-2***. When the data is small and every write is always followed by 2-3 reads of the same data by a different cachelet, updating the data in other cachelets reduces the number of cache misses. Thus, DirectoryUpdate outperforms DirectoryInvalidate by 34.3×.
- ***Exp-3***. When data is small and every write is followed by 2-3 writes to the same cachelet before a single read of the same data from another cachelet, invalidating the data in other cachelets reduces network bandwidth consumption of redundant updates. DirectoryInvalidate outperforms DirectoryUpdate by 2.3×.
- ***Exp-4***. When the data is large (8MB), there are many sharers (32), and the fraction of writes is moderate (10%), fetching the sequence number is less expensive than sending the coherence messages and waiting on the acknowledgments. Thus, SequenceNumbers outperform Directories by 4.1×.

Principle 1: If the average data size is larger than a given threshold *and* the average number of sharers is larger than a given threshold *and* the write fraction is within a given range, choose *SequenceNumbers*. If one of the conditions is not met and the write run is short, choose *DirectoryUpdate*. Otherwise, choose *DirectoryInvalidate*.

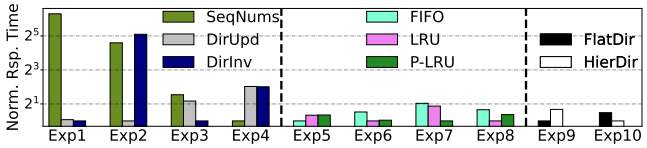


Figure 3: Normalized response time with different cachelet organizations in different environments.

3.1.2 **Cachelet Replacement.** To compute optimal cachelet replacement policy, UniCache learns over time the number of elements in the cache, data-size distribution, and data-popularity distribution (i.e., number of concurrent requests for a given data). We vary the number of cached elements between 10-10K in increments of 200, the data size distribution, and the data popularity distribution, both with uniform and Zipfian distributions with α values 0-1 in 0.2 increments. This results in 2450 different experiments, and we highlight 4 of them in Figure 3 (Exp5-Exp8).

- ***Exp-5***. When there are many small elements in the cache (10K of 100B), maintaining the access bits is expensive. Thus, FIFO outperforms LRUs by 1.3× on average.
- ***Exp-6***. When there are few larger elements (10 of 256KB) accessed with temporal locality, the access bits are able to exploit the locality and reduce the number of cache misses. Thus, LRUs outperform FIFO by 1.4× on average.
- ***Exp-7***. When there are few moderately sized elements (10 of 128KB) and one large element (1MB) repetitively accessed in

bursts, prioritizing the large element reduces the cache miss penalty. Thus, Priority-LRU outperforms LRU by 1.8×.
- ***Exp-8***. When there are few frequently re-used moderately sized elements and one large rarely used element, prioritizing the large element negatively impacts the cache hit rate. Thus, LRU outperforms Priority-LRU by 1.3×.

Principle 2: If the number of cached elements is larger than a given threshold, choose *FIFO*. Otherwise, if the data size *or* popularity distribution is not uniform *and* the low priority data is not frequently reused, choose *Priority-LRU*. Otherwise, choose *LRU*.

3.1.3 **Coherence Domain Organization.** UniCache monitors the average sharer list length and the amount of sharing between the cachelets located on neighboring nodes. We vary the number of the data sharers between 2-64 in powers of 2, and the number of clusters that the sharers belong to from 1 to the number of sharers. This results in 126 different experiments, and we highlight two of them in Figure 3 (Exp9, Exp10).

- ***Exp-9***. When there are few sharers per data (5), all from different clusters, sending all coherence messages from the home node is fast and does not create a network bottleneck, thus, Flat outperforms Hierarchical by 1.6× on average.
- ***Exp-10***. When there are many sharers per data (32), broadcasting a large number of coherence messages creates a network bottleneck in the home node. In addition, as those sharers belong to a small number of clusters (5), one can exploit data sharing across nodes. Thus, Hierarchical outperforms Flat by 1.3×.

Principle 3: If the average number of sharers per data and the average number of sharers per cluster are lower than some thresholds, choose *Flat*. Otherwise, choose *Hierarchical*.

3.1.4 **Write and Replication.** Write and replication policies make trade-offs between performance and fault tolerance. For the applications that require a high degree of fault tolerance, UniCache needs to provide Write-Through or replicated cachelets. For the applications that do not need any fault tolerance guarantees, UniCache always employs Write-Back schemes with no replication for optimal performance. The slowdown of Write-Through depends on the write intensity of the workload, the data size, and the load of the system. The slowdown of replication depends on the load of the system and on the frequency of directory or dirty data changes.
Principle 4: If the application does not have strict fault tolerance requirements, use *Write-Back* with *NoRepl* for both dirty data and directories. Otherwise, if the load is lower than a threshold *and* the fraction of writes is below a threshold *and* the data size is less than a threshold, use *Write-Through* with *NoRepl*; else, use *Write-Back* with *ReplData* for the dirty data. In addition, if the coherence protocol is *Directories*, use *ReplDir* to replicate the directory.

3.2 Reinforcement Learning for Configuration

Each taxonomy category has its own expert RL model that collects the identified metrics of interests and periodically recomputes the optimal value for a given category. The RL model uses (1) the possible values of a given category and the current statistics of the collected metrics as its states, (2) the transition to a new category value as its action, and (3) the improvement (or degradation) of the cache response time as a reward. The models rely on Q-learning. The expected cumulative reward by taking an action A in a given state S is defined as the Q-value of the state-action pair using the SARSA [20] algorithm. These models operate independently of each other and do not take into account decisions made by the other models. This can result in a suboptimal total performance. For example, a specific combination of replacement policy and coherence protocol may not yield the expected results. Hence, in UniCache, the models are organized into a hierarchy. Individual experts send their outputs (a few ranked suggestions) to the moderator model. The moderator combines the outputs and makes the final decision on the optimal cachelet configuration. The moderator is a simple model that assigns weights to the individual downstream models and selects one of the suggested values for each of the models that does not negate the chosen parameter of another higher ranked model. Thus, it takes into account the potential interactions between different categories and the overheads of transforming the cachelet from one configuration to another.

4 EVALUATION

Methodology. We evaluate UniCache on top of OpenWhisk in a 15-node cluster. Each node is an Intel Xeon Silver server with 20 cores, 192GB DRAM and 128MB LLC. Every node runs Ubuntu 20.04. We use Azure Blob Storage [15] as the storage service for all the evaluated functions. To serve as baseline, we emulate the state-of-the-art Faa$T [19] on top of OpenWhisk. Every cachelet has the same configuration: Per-Instance, Write-Through, LRU, Sequence-Numbers and Single-Home. We evaluate UniCache under low, medium, and high load levels, corresponding to an average of 450 requests per second (RPS), 1000 RPS, and 1800 RPS, respectively.
Response Time and Speedups. We measure the end-to-end application response time, from the moment the client sends a request to the point it receives the result. Using this response time, we compute the speedup of UniCache over the baseline. Figure 4 shows the average and tail speedup of each application for different loads.
Average Speedups. UniCache effectively reduces the response times and, as a result, delivers speedups across all applications and load levels, as shown in Figure 4 (up). The speedups range from 1.3× (MLTrain, low load) to 9.8× (HotelR, high load), with the average of 5.7×. The speedup is particularly high for the applications whose optimal configuration diverges significantly from the baseline's configuration. For example, TrainTicket applications operate on small data, thus, maintaining the coherence via sequence numbers is expensive. In addition, the data is shared across all instances that belong to the same application, thus, having Per-Instance cachelets reduces the sharing opportunity. In high loads, UniCache speeds up these applications by 9.4× on average.
Tail Latency Speedups. UniCache significantly reduces the tail latency of the application requests. Figure 4 (down) shows the P99 tail latency speedups in UniCache for different loads normalized to that in the baseline. On average across all benchmarks and loads, UniCache reduces the P99 latency by 4.3×. As the load increases, the reduction also increases. The main sources of tail latency in the baseline are global storage accesses: for the data or sequence numbers. UniCache improves the cache hit rate and minimizes the coherence cost, thus, effectively reduces the tail latency overheads.
Optimal Configurations Table 2 shows the configurations that UniCache sets for each of the applications. For all applications we

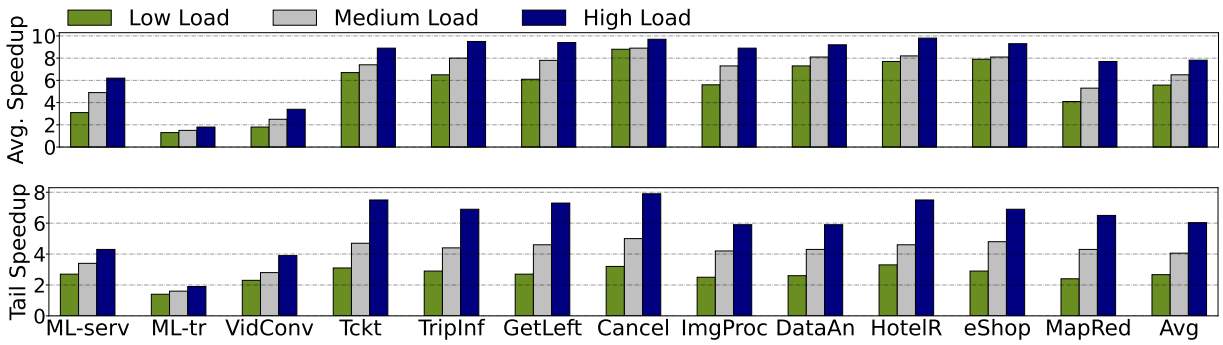


Figure 4: Average (up) and tail (down) speedup of UniCache over the baseline across various loads and applications.

assume strong fault tolerance requirements for the cached data, thus, the system uses Write-Through cachelets (data is never lost). For the other five taxonomy categories, each application requires different cachelet organization to achieve the best performance. UniCache manages that automatically, without users involvement, and dynamically, without offline profiling.

Table 2: Per-benchmark configuration set by UniCache.

Workloads	Configuration
ML-serv	Per-Func, LRU, DirInv, Flat
ML-tr	Per-Func, LRU, SequenceNumbers
VidConv	Per-Func, LRU, SequenceNumbers
TrainTicket	Per-App, FIFO, DirUpd, Hierarchical
ImgProc	Per-Func, Prio-LRU, DirInv, Flat
DataAn	Per-App, Prio-LRU, DirUpd, Flat
HotelR	Per-App, FIFO, DirUpd, Hierarchical
eShop	Per-Func, FIFO, DirUpd, Hierarchical
MapRed	Per-App, LRU, DirUpd, Flat

System Throughput. The reduced network bandwidth, increased cache hit rate and reduced overall data access latency result in improved system throughput. We show the maximum throughput in Table 3 for each of the tested applications. UniCache increases the throughput by 1.4-5.4×, with the average of 4.6×. In addition, we measure the throughput of a system that does not have any caching mechanism. Faa$T significantly increases the throughput of such a system, with the average of 9.3×, by reducing the data access latency and the pressure on the remote storage and network bandwidth. UniCache further improves the throughput by properly adjusting the cache organization to the application needs and system state.

Table 3: System throughput in UniCache and in the baseline.

Workloads	Baseline (Req/s)	UniCache (Req/s)	Improvement (Times)
FunctionBench	1916.7	6133.3	3.2
TrainTicket	3550.0	18050.0	5.1
ServerlessBench	2900.0	13875.0	4.8
vSwarm	2600.0	11916.7	4.6
Average	2795.8	12841.7	4.6

5 CONCLUSION

We propose the first comprehensive taxonomy of distributed data caching schemes in serverless environments. Using the taxonomy, we introduce UniCache, a novel distributed caching scheme that automatically selects the best cache organization for each application without any user intervention. Compared to a state-of-the-art baseline, UniCache speeds up the average execution by 5.7×, reduces P99 tail latency by 4.3×, and improves throughput by 4.6×.

REFERENCES

[1] Apache OpenWhisk. https://openwhisk.apache.org/, 2024.
[2] FaaS Train Ticket. https://github.com/FudanSELab/serverless-trainticket, 2024.
[3] Knative. https://knative.dev/docs/, 2024.
[4] Amazon AWS. AWS Lambda. https://aws.amazon.com/lambda/, 2024.
[5] Amazon AWS. Implementing statelessness in functions. https://docs.aws.amazon.com/lambda/latest/operatorguide/statelessness-functions.html, 2024.
[6] Berg, B., Berger, D. S., McAllister, S., Grosof, I., Gunasekar, S., Lu, J., Uhlar, M., Carrig, J., Beckmann, N., Harchol-Balter, M., and Ganger, G. R. The CacheLib Caching Engine: Design and Experiences at Scale. In *OSDI* (2020).
[7] Eytan, O., Harnik, D., Ofer, E., Friedman, R., and Kat, R. It's Time to Revisit LRU vs. FIFO. In *HotStorage* (2020).
[8] IBM. IBM Cloud Functions. https://cloud.ibm.com/functions/, 2024.
[9] 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, 2024.
[10] Kim, J., and Lee, K. FunctionBench: A Suite of Workloads for Serverless Cloud Function Service. In *CLOUD* (2019).
[11] Klimovic, A., Wang, Y., Stuedi, P., Trivedi, A., Pfefferle, J., and Kozyrakis, C. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *OSDI* (2018).
[12] Liu, Z., Bai, Z., Liu, Z., Li, X., Kim, C., Braverman, V., Jin, X., and Stoica, I. DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching. In *FAST* (2019).
[13] Mahgoub, A., Shankar, K., Mitra, S., Klimovic, A., Chaterji, S., and Bagchi, S. SONIC: Application-aware Data Passing for Chained Serverless Applications. In *ATC* (2021).
[14] Maximilien, M., Hadas, D., II, A. D., and Moser, S. The future is serverless. https://developer.ibm.com/blogs/the-future-is-serverless/, 2024.
[15] Microsoft. Azure Blob Storage. https://azure.microsoft.com/en-us/products/storage/blobs, 2024.
[16] Microsoft. Azure Functions. https://azure.microsoft.com/en-gb/services/functions/, 2024.
[17] Microsoft. Serverless architecture considerations. https://learn.microsoft.com/en-us/dotnet/architecture/serverless/serverless-architecture-considerations, 2024.
[18] Mvondo, D., Bacou, M., Nguetchouang, K., Ngale, L., Pouget, S., Kouam, J., Lachaize, R., Hwang, J., Wood, T., Hagimont, D., De Palma, N., Batchakui, B., and Tchana, A. OFC: An Opportunistic Caching System for FaaS Platforms. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys '21)* (2021).
[19] Romero, F., Chaudhry, G. I., Goiri, I., Gopa, P., Batum, P., Yadwadkar, N., Fonseca, R., Kozyrakis, C., and Bianchini, R. Faa$T: A Transparent Auto-Scaling Cache for Serverless Apps. In *SoCC* (2021).

[20] Rummery, G., and Niranjan, M. On-line Q-learning using connectionist systems. In *University of Cambridge, Department of Engineering* (1994).
[21] Shillaker, S., and Pietzuch, P. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *ATC* (2020).
[22] The vHive Ecosystem. vSwarm - Serverless Benchmarking Suite. https://github.com/vhive-serverless/vSwarm, 2024.
[23] Yang, J., Yue, Y., and Rashmi, K. V. A Large-Scale Analysis of 100s of In-Memory Key-Value Cache Clusters at Twitter. In *ACM Trans. Stor.* (2021).
[24] Yu, T., Liu, Q., Du, D., Xia, Y., Zang, B., Lu, Z., Yang, P., Qin, C., and Chen, H. Characterizing Serverless Platforms with ServerlessBench. In *SoCC* (2020).