InfiniStore: Elastic Serverless Cloud Storage

Jingyuan Zhang
George Mason University
jzhang33@gmu.edu

Ao Wang
George Mason University
Alibaba Group
shenlan.wa@alibaba-inc.com

Xiaolong Ma
University of Nevada, Reno
xiaolongm@nevada.unr.edu

Benjamin Carver
George Mason University
bcarver2@gmu.edu

Nicholas John Newman
George Mason University
nnewman7@gmu.edu

Ali Anwar
University of Minnesota
aanwar@umn.edu

Lukas Rupprecht
IBM Research
lukas.rupprecht@ibm.com

Vasily Tarasov
IBM Research
vtarasov@us.ibm.com

Dimitrios Skourtis
Redpanda Data
skourtis@soe.ucsc.edu

Feng Yan
University of Houston
fyan5@central.uh.edu

Yue Cheng*
University of Virginia
mrz7dp@virginia.edu

ABSTRACT

Cloud object storage such as AWS S3 is cost-effective and highly elastic but relatively slow, while high-performance cloud storage such as AWS ElastiCache is expensive and provides limited elasticity. We present a new cloud storage service called ServerlessMemory, which stores data using the memory of serverless functions. ServerlessMemory employs a sliding-window-based memory management strategy inspired by the garbage collection mechanisms used in the programming language to effectively segregate hot/cold data and provides fine-grained elasticity, good performance, and a pay-per-access cost model with extremely low cost.

We then design and implement InfiniStore, a persistent and elastic cloud storage system, which seamlessly couples the function-based ServerlessMemory layer with a persistent, inexpensive cloud object store layer. InfiniStore enables durability despite function failures using a fast parallel recovery scheme built on the auto-scaling functionality of a FaaS (Function-as-a-Service) platform. We evaluate InfiniStore extensively using both microbenchmarking and two real-world applications. Results show that InfiniStore has more performance benefits for objects larger than 10 MB compared to AWS ElastiCache and Anna, and InfiniStore achieves 26.25% and 97.24% tenant-side cost reduction compared to InfiniCache and ElastiCache, respectively.

PVLDB Reference Format:
Jingyuan Zhang, Ao Wang, Xiaolong Ma, Benjamin Carver, Nicholas John Newman, Ali Anwar, Lukas Rupprecht, Vasily Tarasov, Dimitrios Skourtis, Feng Yan, and Yue Cheng. InfiniStore: Elastic Serverless Cloud Storage. PVLDB, 16(7): 1629 - 1642, 2023.
doi:10.14778/3587136.3587139

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ds2-lab/infinistore.

*Corresponding author

Proceedings of the VLDB Endowment, Vol. 16, No. 7 ISSN 2150-8097.
doi:10.14778/3587136.3587139

1 INTRODUCTION

Public clouds free tenants from the tedious tasks of IT infrastructure planning and maintenance and allow tenants to focus on application development. These offerings are driving the adoption of public clouds for hosting massive-scale, data- and compute-intensive applications, such as Internet-scale web applications [4, 11, 19, 40].

Although cloud providers can simplify the allocation and scaling of compute resources, there is an excessively wide range of cloud *storage* services with various persistence, performance, pricing, and capacity characteristics to choose from. This choice complicates resource management and application deployment. For example, AWS ElastiCache [6] is an AWS-managed memory cache service, but ElastiCache does not provide data persistence by default. AWS S3 [9] is a ubiquitous object store service, which offers data durability and persistence, but S3 is much slower than ElastiCache.

This choice is further complicated by the varying memory and storage requirements of heterogeneous cloud workloads. For example, a production object store workload for accessing Docker container images [4] exhibits a frequently changing working set size (WSS) having a wide range of object sizes and strong temporal reuse patterns—most object reuse happens within one or two hours of the object's previous access. In addition, emerging serverless function applications require large, short-term, elastic storage capacity that scales based on WSS and object sizes [40]. Similar dynamic I/O patterns are observed in enterprise network file system workloads [30], big data analytics workloads [29, 47], and data warehouse workloads [49], among others [37, 48].

We argue that today's clouds are missing an elastic, performant, and cost-effective cloud storage solution that can fulfill the heterogeneous and dynamic storage requirements of a wide variety of applications. Durable cloud object store solutions such as AWS S3 [9] and Google Cloud Storage [21] are inexpensive but cannot provide memory-store-level performance. Faster storage/caches such as AWS FSx [7] and ElastiCache [6] offer high-bandwidth and low-latency data access, but these solutions are expensive and lack the capability to automatically and rapidly grow and shrink storage capacity in response to changing application demands.

The Function-as-a-Service (FaaS) model is well suited to fill this gap. FaaS applications are structured as a collection of *functions*

managed by the service provider in terms of resource scaling and management. The combination of instant scaling, fast access, and pay-per-use pricing makes FaaS platforms an appealing foundation for an elastic, performant, and cost-effective storage service.

InfiniCache is a distributed memory caching system that exploits the properties of FaaS [51]. InfiniCache uses many serverless functions whose function-memory is used collectively for object storage with PUT/GET APIs for accessing the objects stored. Serverless functions are transient and may be reclaimed after a short period by the FaaS provider. Therefore, InfiniCache implements a primary/backup replication protocol to increase data durability at the serverless function-memory level.

While InfiniCache has demonstrated the feasibility of using serverless functions for data caching, InfiniCache has several limitations. (1) The serverless-function-based cluster deployed in InfiniCache is fixed, and therefore, lacks elasticity: InfiniCache randomly assigns old and new data objects to function instances; if were to be scaled out, this data mapping strategy may lead to excessive data migration, which we will show later. (2) InfiniCache provides best-effort data durability via erasure coding and replication at the serverless function level; however, the FaaS provider may reclaim a function instance and its memory at any time, which causes cache misses and impacts application performance. (3) Data cached in InfiniCache is replicated twice, which doubles memory resources and doubles cost.

In this paper, we introduce a new storage service named *ServerlessMemory*. ServerlessMemory uses the collective function-memory as a storage medium to construct a continuous memory space. When allocating memory, new functions are invoked and function-memory are added to memory space. Inspired by the mark-compact garbage collection (GC) algorithms in programming languages [25], we apply a fully-automatic, sliding-window-based function-memory management scheme to separate live/unused (hot/cold) data and leverage FaaS provider's function reclaiming mechanism for garbage collection. Together, the ServerlessMemory achieves high storage elasticity at function granularity. Differentiated with a cache, which would typically require an expensive offline tuning process to construct the miss ratio curve to find the optimal cache size for a certain workload, the ServerlessMemory service can automatically capture the application's working set. *To the best of our knowledge, the ServerlessMemory is the first cloud service that leverages the desirable FaaS properties to achieve fine-grained elasticity in disaggregated memory management.*

We build InfiniStore, an elastic, fault-tolerant cloud storage on top of the ServerlessMemory service. InfiniStore has two layers: a *ServerlessMemory* layer that exposes durable serverless function-memory to serve application I/Os, and an inexpensive *object store* layer that uses a cloud object store for data persistence. We implement durability via fast *parallel recovery* and lightweight *insertion logs*. When a function instance is reclaimed unexpectedly, InfiniStore launches a group of pre-selected peer (recovery function) instances for parallel data recovery. Each recovery function instance replays a portion of its assigned insertion log and downloads lost data from InfiniStore's object store layer.

This paper makes the following contributions:

- We introduce a new ServerlessMemory cloud service that is elastic and pay-per-access at the memory storage level. The ServerlessMemory is the *first* cloud service that exploits the FaaS properties to automatically capture the working set of a stateful data-intensive application.
- We design and implement InfiniStore, an elastic, cost-effective, high-performance, and fault-tolerant cloud storage system that combines the ServerlessMemory layer with a persistent but inexpensive object store layer.
- We perform extensive evaluations using YCSB microbenchmark stress testing and two practical applications: an IBM container registry workload and an Azure Functions blobs workload.

Experimental results show that InfiniStore represents a novel performance-$cost tradeoff in today's cloud storage landscape. It is worth noting that InfiniStore is a memory storage while reducing cost by 26.25% compared to InfiniCache, 97.24% compared to AWS ElastiCache, and offering better performance for large object requests than Anna [53], AWS S3, and FSx. InfiniStore is pay-per-access, which means it incurs a cost proportional to the number of GET and PUT requests it serves, with a small cost overhead of 26.00%.

2 MOTIVATION

Data-intensive applications have dynamic and heterogeneous workloads that benefit from storage elasticity. This section performs a detailed workload analysis on two representative storage workloads: a cloud object store workload of IBM container registry [4, 31] and a serverless application workload of Azure Functions blob accesses [40]. To understand the dynamic characteristics of these workloads, we will focus along two dimensions: (1) working set size (WSS) and throughput; and (2) temporal access patterns. WSS here means the aggregate footprint of the data accessed, re-accessed, and modified by the application in a given time interval.

2.1 Dynamic WSS and Throughput

We begin our workload analysis by asking these questions:

(1) *Does WSS/aggregate throughput dynamically change?*
(2) *If so, what is the magnitude of this change?*

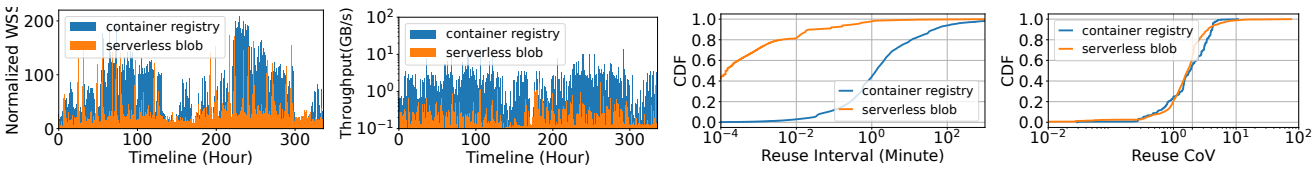
The answers to these questions will help us understand the elasticity requirement for InfiniStore.

Both workloads exhibit a highly variable WSS with a maximum size over 209× and 173× larger than the minimum size for the container image workload and serverless application workload, respectively (Figure 1(a)). Additionally, the WSS is shifting every minute. Such significant temporal WSS variance suggests that a high degree of elasticity at the storage level is required to be able to serve the objects in the working set efficiently without excessive storage overprovisioning.

The container image workload also demonstrates bursty I/O characteristics. Figure 1(b) shows that the aggregate throughput spikes to 15 GB/s with an average of 1 GB/s. This throughput variability poses a significant challenge for tenants trying to provision storage resources that meet the application's performance requirements.
Implication 1: *The WSS variance requires instantaneous provisioning of large amounts of storage resources to satisfy the high throughput and low latency requirements.*

2.2 Temporal Access Pattern

We study the temporal access patterns by asking:



(a) Working set size timeline. (b) Aggregate I/O throughput timeline. (c) Reuse interval distribution. (d) IAT CoV distribution (#*reuse* ≥ 10).

Figure 1: The container registry and serverless blob workload characteristics: dynamicity (a, b) and temporal (c, d) behaviors.

(1) *What is the time interval between two successive accesses to the same data object?*
(2) *What is the request inter-arrival time (IAT) pattern for reused objects?*

The answers to these questions will guide the design of InfiniStore's elastic data placement strategy, i.e., the mapping between data objects and serverless functions.

We first study the temporal patterns by measuring the time interval between two successive accesses of the same data object (the temporal reuse interval). Figure 1(c) shows that for the first 50 hours of the container image workload, approximately 80% and 94% of all reuses happen within 10 minutes and 100 minutes, respectively. The serverless application workload shows a similar reuse pattern but with much shorter reuse intervals: approximately 98% of requests revisit the same object within one minute. These patterns suggest that much of the data accessed is very likely to be reused within a short time interval.

We next study the IAT pattern by quantifying the coefficient of variation (CoV) of reused data objects. A CoV of 1 suggests that a re-access may arrive at an arbitrary time, where the IAT follows a Poisson distribution. A CoV > 1 indicates a more bursty arrival pattern than a Poisson distribution. We filter out objects with less than 10 reuses, which excludes 16% and 2% of the requests from the container image and serverless blob workloads, respectively. Figure 1(d) shows that about 80% of reused objects have a CoV greater than 1, indicating that both workloads are very bursty.

Implication 2: *A multi-layer storage system with a fast elastic layer and a slower persistence layer would be beneficial. Most data become cold quickly and can be discarded from the faster storage layer if all data are durably stored in a persistence layer. This also suggests that hot data must be tracked and hot/cold data segregation would be beneficial in this context.*

3 WHY USE FAAS FOR DATA STORAGE?

3.1 FaaS Properties

The above implications motivate us to rethink the design and implementation of cloud storage systems. The main question we seek to answer is: *How can a cloud storage system be designed to offer elasticity for dynamically changing workloads while simultaneously providing high performance and high durability at low cost?* In this section, we first introduce the unique properties of FaaS and explain how storage services that are built on cloud functions can take advantage of these properties. We then discuss InfiniCache and its limitations.

Quick Startup and Pay-per-Use. Unlike VMs, which can take minutes to launch, thousands of cloud function instances can be provisioned in a fraction of a second, without advance notice, via an HTTP API. Storage services can use fast and massive instance provisioning to effectively handle bursty and dynamically-changing storage access patterns. More importantly, FaaS providers [10, 20, 23] charge tenants on a fine-grained per-use basis. For example, AWS Lambda bills on a per-invocation basis ($0.02 per 1 million invocations) and charges for a CPU+memory bundle at a rate of $0.0000166667 per second (rounding up to the nearest millisecond) for each GB of bundled memory. This means accessing objects stored in the memory of function instances will be billed on a per-access basis.

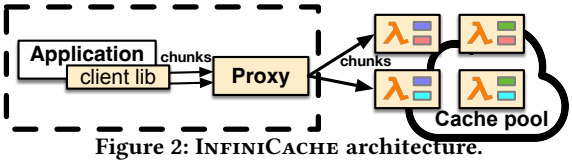


Figure 2: InfiniCache architecture.

Exploiting Opportunistic and Elastic Function Memory for Data Storage. Recent studies [51, 52] report that FaaS providers support limited, short-term caching of function instances by keeping the instance states "warm" in memory to mitigate cold start penalties [44]. Idle function instances that are not invoked after a period of time can be reclaimed by the provider. This period of time varies, ranging from tens of minutes to hours for AWS Lambda. A later invocation to the same instance can extend the function's lifetime. The memory of a cluster of function instances can be aggregated to enable opportunistic storage services. Furthermore, fast function startup enables low-latency resource scaling when more functions need to be provisioned, which provides an elastic storage foundation for highly dynamic workloads.

3.2 InfiniCache

Table 1: Summary of terminologies used in InfiniCache.

Terminology	Definition
Chunk	A partition of an erasure-coded data object
Function	A deployment of AWS Lambda
Instance	A Lambda instance w/ bundled compute-memory resources
Function-memory	The memory resource in a function instance

InfiniCache [51] builds on the above insights by exploiting the collective memory of serverless functions to cache data objects. Figure 2 shows the architecture of InfiniCache. InfiniCache exposes `GET`/`PUT` API via a client library. To `PUT` an object, the client library sends erasure-coded ***chunks*** of the object to a proxy. The proxy randomly maps chunks to Lambda ***function deployments*** in the cache pool and streams data to invoked ***function instances*** for chunk storage. The chunk-function mapping is stored in the proxy for serving `GET` requests. The terms are summarized in Table 1.

InfiniCache periodically invokes all functions in the cache pool to keep idle function instances from being reclaimed by AWS. InfiniCache uses two fault tolerance techniques, primary/backup

delta-replication and erasure coding, to maximize data durability at the cache level. The delta-replication technique replicates all chunks twice between two instances of the same function deployment. The replication provides instantaneous failover if one of the two replica instances is failed. Erasure coding complements replication by providing object-level fault tolerance if no more than a configurable fraction of an object's chunks are lost.

InfiniCache's Limitations:

- Lack of elasticity: InfiniCache maintains a static, fixed cache pool of functions whose collective function-memory is typically larger than the active WSS of the workload. This strategy may cause massive data movement if the cache pool were scaling out and data rebalancing is required.
- Mixed hot/cold object placement and no hot data tracking: InfiniCache arbitrarily assigns new data objects to functions in the cache regardless of whether the data will remain hot or become cold. Hot data must be tracked and migrated if a function were selected to be removed from the cache on scaling-in events.
- Durability is not guaranteed: Function instances reclamation causes InfiniCache to suffer data loss at the memory cache level. InfiniStore allows no data loss as a memory storage.

4 SERVERLESS MEMORY

We propose a new cloud storage service called *ServerlessMemory*, which combines the memory of a cluster of serverless function instances and exposes this memory to applications as fast, elastic, inexpensive, and pay-per-access cloud storage. To provide durability, InfiniStore combines ServerlessMemory with a persistent backing object store. In this section, we provide an overview of ServerlessMemory.

In ServerlessMemory, cloud function instances and their supporting function-memories are automatically allocated when the tenant application calls the PUT API. This abstraction is analogous to programming languages that provide automatic memory management, e.g., Java. Data objects are inserted into the memory of function instances and accessed by re-invoking functions.

When an application's storage demand surges, more ServerlessMemory functions can be invoked. The function-memory of the new function instances joins the distributed memory pool instantly without requiring any manual effort for launching and scaling servers that are otherwise hosted by cloud VMs. A function F's function-memory that stores data object O is only billed when O is accessed by a GET or PUT operation of F, i.e., when an instance of F is executing. This pay-per-access property is naturally utilized by ServerlessMemory. Hot data are tracked by the ServerlessMemory and compacted into a collection of active function instances. If F is not in the collection and F's function-memory holds cold data, F is rarely invoked, and the storage for the data objects in F's function-memory is rarely billed. No explicit garbage collection is required for ServerlessMemory as storage is implicitly garbage collected by the serverless provider when the instance of F becomes inactive for a prolonged period and is reclaimed.

5 INFINISTORE DESIGN

InfiniStore is a co-designed cloud storage system that tightly couples a serverless, function-based ServerlessMemory store (SMS) layer and a persistent cloud object store (COS) layer. All data is stored in SMS for fast access and copied to COS for durability. SMS is highly configurable and implements an adaptive, sliding-window-based data management mechanism to hold the current working set of a workload. In this section, we present the design of InfiniStore and its SMS and COS components.

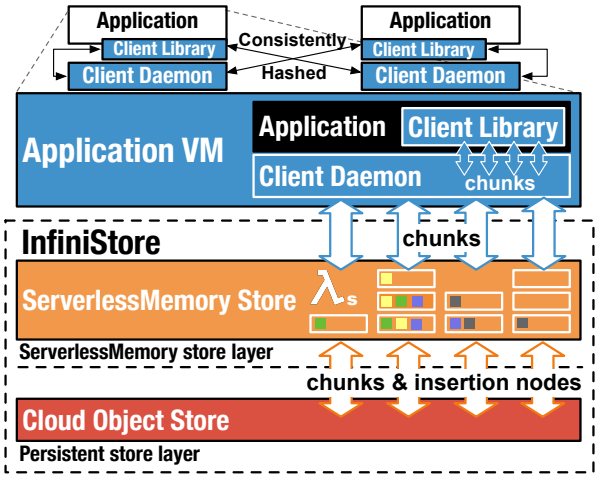


Figure 3: InfiniStore architecture.

5.1 SMS Design Challenges

ServerlessMemory poses two design challenges: elasticity-optimized data placement and data durability.

Data Placement. The strategy that assigns data objects to functions is critical as it impacts the elasticity and cost-effectiveness of the storage system. Randomly placing data in function instances using conventional (serverful) strategies, such as consistent hashing [26], may lead to unnecessary expenses. As cold data would be co-located with hot/new data in the same function instance, function warmups would be needed for the entire ServerlessMemory instance pool, which increases monetary cost. Hence, we need to design an elasticity-optimized data placement strategy for ServerlessMemory to minimize function-warmup overhead.

Data Durability in Functions. As mentioned in §3.2, InfiniCache does not guarantee durability and best-effort data recovery impacts cost-effectiveness. ServerlessMemory must handle data durability issues transparently without noticeably impacting the application's performance and the monetary cost. To effectively meet this objective, we combine SMS with an inexpensive COS.

5.2 Design Overview

Next, we present InfiniStore's design. We use AWS Lambda as an example to illustrate the design of InfiniStore. In the remainder of this section, all instances are assumed to be Lambda function instances. It is worth noting that the design of InfiniStore is generic and can be easily ported to other cloud platforms. InfiniStore consists of four components: a InfiniStore client library, a client daemon, an SMS layer, and a persistent COS layer, see Figure 3.

Applications interact with InfiniStore via a **client library** that communicates with the client daemon managing the SMS. The library exposes `GET(key)/PUT(key, value)` as read/write APIs to the application and is responsible for (1) transparently encoding and decoding data objects with Reed-Solomon erasure coding (EC), and (2) load balancing incoming application requests across a distributed

set of client daemons. InfiniStore offers strong consistency. The PUT API uses versioning provided by the client daemon to support object updates. Versions of objects are read-only after PUTs return.

Co-located with the application, a **client daemon** stores data objects' metadata with the versioning information in an in-memory data structure called metadata table, orchestrates the function instances of SMS, and serves as a rendezvous point for streaming EC-encoded object chunks between the client library and SMS. The metadata table can be persisted to the local disk for fault tolerance. In the distributed application setting, we adopt a multi-VM deployment in which a client daemon is deployed on each VM that hosts the application and manages a separate SMS with shared access among application clients.

SMS consists of a collection of function instances. Unlike InfiniCache, in SMS, each function instance does not have a peer replica. The scaling of SMS is driven by the workload's working set. SMS manages the active data objects in function-memory and serves all requests sent from the client daemon (§5.3, §5.4).

COS forms the persistence layer of InfiniStore. COS stores all data objects and critical metadata (insertion logs) for SMS data recovery (§5.5). To persistently store an object, InfiniStore's client library first determines the destination client daemon (and therefore its corresponding SMS) using consistent hashing. The client library then streams EC-encoded chunks to SMS via the daemon. Without compromising the durability and strong consistency, we design a persistent buffer (§5.3.2) to allow the client application to receive a response once all chunks are successfully inserted into SMS but before all chunks are fully persisted to COS. The function instances in SMS will not return until chunks are fully written to COS.

5.3 ServerlessMemory Store Management

InfiniStore's SMS management design is based primarily on the workload characteristics described in §2 and the requirement to provide elasticity, performance, and durability (§3.2). Instead of a static approach, InfiniStore's client daemon uses a novel, highly adaptive, sliding-window-based SMS management mechanism inspired by the garbage collector designs used in programming languages. The ServerlessMemory is regarded as a continuous memory space of functions, where each function is identified by a global unique ID_λ. FaaS platforms offer virtually infinite memory capacity and new memory can be allocated by simply invoking more functions. In a garbage-collection-based (GC-based) programming language, a GC procedure is invoked once every fixed time interval to release the allocated memory that is no longer referenced. In InfiniStore settings, "no longer referenced" data is cold data that has not been accessed for a specified time period H. Cold data's memory can be released by changing the function management policy (e.g., by stopping invoking corresponding functions and leaving the instances to be eventually reclaimed by the FaaS provider). The client daemon organizes the memory space at the function granularity. New data is always appended to functions newly added to memory space. The daemon adds new functions when needed (e.g., out of function-memory, §5.3.1). Data re-accessed within H is marked and compacted to newly added functions, too (§5.3.3). These functions added during the same GC interval construct a new *GC-bucket*. With compaction, a GC-bucket contains only cold data after H and the memory-functions within is released by the GC.

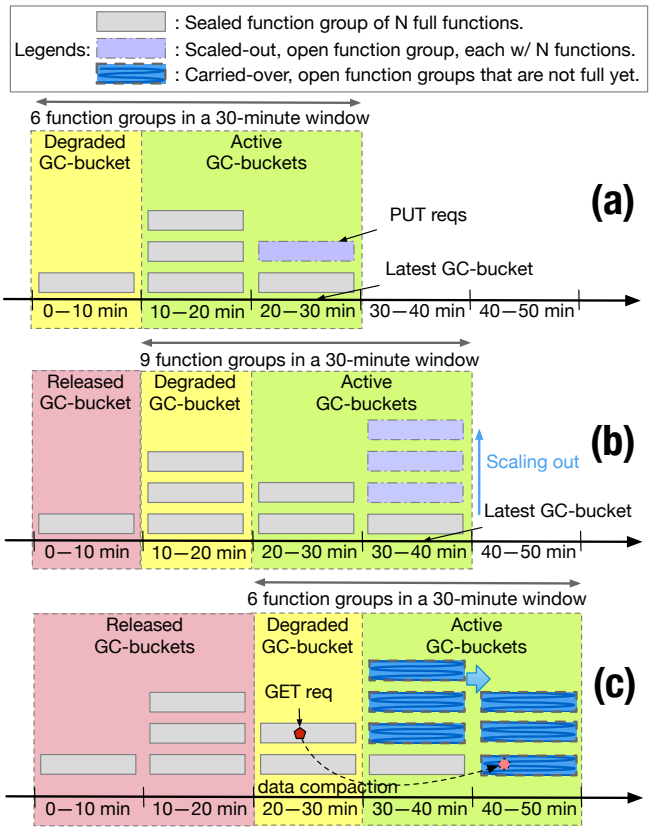


Figure 4: A sliding-window example of GC execution. The example shows a configuration in which GC executes every 10 minutes, and the active (M) and degraded (N) window are configured as 20 and 10 minutes respectively. H in this case is 30 minutes. Functions are shown as FGs. (a) SMS has been running for 20+ minutes, with its latest GC-bucket in the third time slot. (b) Time runs into the `30-40 min` slot. The GC releases the oldest GC-bucket. More functions are added to the latest GC-bucket due to scaling out. (c) At `40-50 min`, The GC releases one more GC-bucket. Functions not fully populated keep open in the new GC-bucket.

For a smoother function management policy (FMP) changing across the function lifespan, the client daemon divides functions added within H into active and degraded GC-buckets, each lasting for M and N GC intervals, respectively[1]. Figure 4 uses a sliding-window example to illustrate how InfiniStore's GC works. For functions added to memory within M GC intervals in active GC-buckets, we apply an active FMP that extends instance lifespan using a no-op heartbeat message (warmup) sent periodically by the daemon. On executing the GC procedure, (1) the oldest active GC-bucket created M intervals ago becomes *degraded*, which is then applied a degraded FMP with a reduced warmup frequency, (2) the oldest degraded GC-bucket created H intervals ago becomes *released* and all functions in released GC-bucket are immediately removed from memory space. Any function in a degraded GC-bucket will be removed if failures are detected (§5.5.1).

[1] M and N are configurable. Empirically, M and N can be configured based on the $mean$ and $mean + stdev$ of objects' reuse interval, respectively.

```
1 func PlaceChunk(chunk *Chunk):
2   // Initialize a func pointer using the object's chunk ID.
3   funcPtr := chunk.ID
4   // Ensure at least funcPtr open functions are available.
5   functions := GetOpenFuncs(funcPtr)
6   for:
7     if funcPtr >= len(functions): // Needs to scale out?
8       // Scaling out by ensuring funcPtr open functions available.
9       functions, funcPtr = GetOpenFuncs(funcPtr)
10    else if !TestAndPlace(functions[funcPtr], chunk):
11      funcPtr += chunk.FGSize // Increment func pointer by one FG.
12    else: return
```

Figure 5: Chunk placement algorithm.

Function Group. Within a GC-bucket, the client daemon manages functions by *function groups* (FGs). An FG is a logical scaling unit that consists of N functions, where N is determined by the number of chunks of an object. Note that this abstraction is essential to support objects with various numbers of chunks. The client daemon scales out the latest GC-bucket at FG granularity.

For clarification, the client daemon serves PUT/GET requests at the chunk granularity. All chunks of an object are served in parallel.

5.3.1 Serving PUT*s.* As shown in Figure 4(a), PUTs are served by the latest GC-bucket. FGs in the current GC-bucket that actively serve PUTs are referred to as *open* FGs. When the workload surges, e.g., the WSS or the number of concurrent PUT requests increases, the daemon scales out the latest GC-bucket by launching more FGs (Figure 4(b)). The client daemon keeps track of the memory consumption of each function in the latest GC-bucket by reconciling the memory statistics piggybacked on the response of each function invocation request. When the memory consumption of an FG in the latest GC-bucket exceeds a predefined HARDCAP threshold[2], the daemon starts the scale-out process and all functions in that FG are sealed (read-only). Note that the degraded GC-buckets along with the functions therein are also *sealed*.

Data Placement under PUTs. InfiniStore's SMS management employs a simple and highly-efficient data placement algorithm, see Figure 5. To PUT a new object into SMS, the client daemon calls `PlaceChunk()` in parallel for each of the N chunks (`chunk.FGSize`) in order to determine in which FG the chunk should be stored. `PlaceChunk()` tests FGs using a function pointer (`funcPtr`). Starting from the $ID_{chunk}{}^{th}$ function (line 3), the algorithm first ensures that there are at least `funcPtr` functions open for placement (line 5). The algorithm then tries to place the chunk in the function identified by `funcPtr` using `TestAndPlace()` atomically (line 10, first iteration). If failed, the algorithm advances `funcPtr` by N (line 11) to ensure that at least `funcPtr` functions open (line 9 in later iterations), auto-scales if needed, and probes the next FG (line 10) until it succeeds. The `funcPtr` is advanced by FGs to ensure that InfiniStore never places any two chunks of an object on the same function for two reasons: (1) to minimize the chances of multiple chunks becoming unavailable due to a single function reclamation; and (2) to parallelize the I/Os across all function instances that will store the object chunks. To balance the load and minimize network contention, `TestAndPlace()` validates (1) if the memory HARDCAP of the function is reached, and (2) if this function's request queues (§5.3.4) are full; Otherwise, the function serves the PUT request. `PlaceChunk()` uses a greedy policy for choosing an available FG within the latest GC-bucket. Specifically, the algorithm always tries to use the oldest open FGs for inserting new data. This design is because older FGs holding relatively cold data are likely to be first to reach their memory HARDCAP and thus are likely to be sealed earlier than newer FGs. All the *open* FGs are carried over to the new GC-bucket during GC, as depicted in Figure 4(c).

Auto-Scaling under PUTs. InfiniStore support customized auto-scaling policies in `GetOpenFuncs()` to handle PUT spikes (e.g., one can implement a more aggressive policy that doubles the number of functions each time an auto-scaling is triggered [18]). InfiniStore's current linear auto-scaling policy works well since deploying and invoking new functions is fast; a more aggressive auto-scaling policy may result in lower SMS capacity utilization with a higher monetary cost.

5.3.2 Daemon-side Versioning and Persistent Buffer. A persistent buffer is a stream buffer that intercepts the chunk streaming on the data path of PUT requests and temporarily buffers the intercepted data on the daemon-local disk. InfiniStore uses the daemon-side persistent buffer to accelerate PUT requests without compromising durability and strong consistency. Since object chunks written to SMS and COS are read-only, the client daemon uses versioning to support updates. InfiniStore uses the consistency increasing algorithm proposed in SCFS [12] to achieve strong consistency atop an eventually consistent COS, e.g., AWS S3. Strong consistency requires a PUT request to return after object chunks have been stored in both SMS and COS. Since InfiniStore uses an inexpensive, slow COS layer, the tail latency of PUT requests is not guaranteed if the application must wait for the data to be fully stored in COS. With the persistent buffer, a PUT request can return immediately after being stored in SMS. The client daemon guarantees that the chunks will be successfully stored in COS later by retrying the request using the data stored in the persistent buffer. After the data is stored in COS, its persistent buffer can then be released. For performance, a read-after-write GET request can be directly served from the persistent buffer if the chunk's buffer has not been released. Details of the versioning algorithm can be found in the appendix of the full report [57].

5.3.3 Serving GET*s.* GET requests are routed from the daemon to SMS and served by the functions storing chunks. The client daemon marks chunks of a hot object that has been re-accessed and may trigger asynchronous migration that compacts the chunks to the latest GC-bucket. This compaction is performed by loading chunks from the COS into SMS' latest GC-bucket. A GET hits on a chunk in a degraded GC-bucket must trigger the data migration, see Figure 4(c). The client daemon performs the compaction in multiple rounds: in each round, the daemon randomly picks a subset (e.g., 50%) of all marked chunks and migrates the subset to the latest GC-bucket until all chunks are migrated. The daemon is configured to bound the maximum compaction interval for each compaction operation to prevent compaction bursts from consuming too many SMS resources. There is still a chance that a GET request may hit on an object with all or part of its chunks residing in a removed function. In this case, the daemon performs synchronous, on-demand migration, which restores cold chunks from COS to the latest GC-bucket. The daemon updates the mapping table after completed the migration operation.

[2]The HARDCAP is defined by excluding the Lambda function's program and runtime overhead (around 100 MB) and a fraction of the total function-memory reserved for data recovery (§5.5.2).

Auto-Scaling under GETs. Elastic demand caching is triggered when a function's request queue is full and the function can not serve more requests. GET-triggered auto-scaling follows the algorithm in Figure 5 and launches more functions—called cache functions—if the GET throughput surges. In TestAndPlace(), the client daemon makes an on-demand cache request with the following steps: (1) check if the function's request queues are full; (2) check if the memory HARDCAP of the function is reached; (3) if these two checks return true, the function caches the requested chunk from COS into function-memory and returns the chunk. Demand-cached data under this temporary caching scheme can be evicted to make room for later caching operations or regular object chunks inserted via PUT. Function-memory space management is discussed in §5.4.

5.3.4 Request Queues and Handling Large Objects. InfiniStore's client daemon adopts a two-queue scheme for each function, where each queue is associated with a network connection that connects the client daemon with a function instance. This scheme separates small requests from large requests in order to avoid the convoy effect in which large requests block small requests.

InfiniStore supports large objects by splitting objects into smaller fragments no larger than 200 MB. We implement a data streaming protocol to pipeline object fragments transfer, similar to HTTP/2's multiplexing [22], to improve transmission efficiency.

5.4 Function-Memory Space Management

As mentioned earlier, the demand for extra cache functions is temporary and driven by bursty requests. The cache functions will not be fully utilized for storing regular object chunks until all older FGs become full and sealed. Simply discarding cache functions from SMS would impact performance, as a burst of requests may arrive again anytime. To utilize the memory space of the cache functions, InfiniStore divides the memory of a function into two partitions, a *storage partition* and a *cache space*: regular object chunks are stored in the storage partition, while the cache space is designated for buffering demand-cached chunks temporarily. There are two categories of demand-cached chunks: hot chunks cached for serving bursty GETs (§5.3.3); and cold chunks from released function-memory (e.g., GC-bucket releasing) temporarily cached for serving potential out-of-working-set GETs. The size of chunks in the cache space is not included in a function's memory consumption, and therefore, chunks can be evicted if the function's memory is inadequate to serve a new PUT. The cache space is also persisted to COS and will be recovered if a cache function is reclaimed (§5.5.2).

5.5 Fault Tolerance and Data Recovery

While AWS provides transient function caching that allows a function instance to be used for multiple invocations [8], it does not guarantee an instance will be cached and a cached instance can be reclaimed anytime. Hence, InfiniStore needs to be robust against frequent reclamations of function instances.

One possible approach to enable durability in SMS is to replicate each object in the function-memory [51]. However, this approach would significantly increase the monetary cost (more than 50% of the monetary cost of InfiniCache is for maintaining replica instances) and reduce the effective memory capacity (half of the memory is used for storing replicas when using 2-way replication). Furthermore, in the highly unreliable environment that is created when users are given no direct control over AWS' internal reclamation policy, durability is still not guaranteed when replicas are used, since all instances that store replicas may get reclaimed by AWS.

InfiniStore takes a different approach to handle frequent reclamation of function instances: all objects are backed up in an inexpensive, persistent COS layer. Upon an invoked instance detecting a partial or complete data loss (e.g., an instance has been reclaimed), this instance recovers, from COS, all of the objects previously stored in the reclaimed instance. Recovering all objects can be slow given instance's bandwidth limit. Inspired by RAMCloud's fast recovery mechanism [35], InfiniStore by default maintains a single copy for each chunk (replication may exist in cache space) in SMS and uses tens of recovery functions, each recovering a portion of all the lost chunks in parallel. The unique challenge and the differences from RAMCloud are discussed at the end of §5.5.2.

5.5.1 Failure Detection. In order to initiate a data recovery, InfiniStore needs to establish that a previously invoked instance has been reclaimed since its last invocation. InfiniStore does this by using an *insertion log* to keep track of PUT operations for each individual function. The log is used to determine whether an instance's memory state is up-to-date.

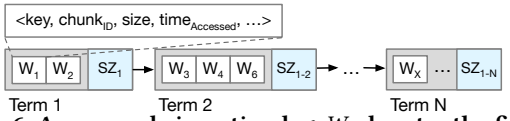


Figure 6: An example insertion log. W_1 denotes the first PUT record stored under Term 1.

Insertion Log. The insertion log contains one or more insertion nodes, each of which is a COS object that records the PUT operations served by a function instance during a single invocation of the function instance. Each PUT results in the creation of an insertion node that is stamped with a monotonically increasing counter value called *term*. Upon a PUT, the object chunks are inserted into SMS and then pushed to COS; in the meantime, the function consolidates and seals concurrent PUT records (received at roughly the same time in a time window, whose duration is configurable) generated during its current invocation into an insertion log node and persists the log node to COS before returning. Figure 6 shows an example insertion log and the information recorded in each PUT request. On returning, the function instance creates a *snapshot* of the chunks it currently stores to speed up recovery downloading. The snapshot is also persisted in COS. The most up-to-date insertion node information is piggybacked onto the GET/PUT response payload sent to the client daemon. This information includes the term of the insertion node, a hash that is computed from the insertion node, the diff_rank, which is discussed in the next paragraph, the size of the last insertion node stored in COS, and a copy of snapshot information containing all of the aforementioned fields. The client daemon maintains up-to-date insertion information for each function it manages.

Triggering a Recovery. On a function invocation for a GET/PUT, the invoked instance N uses the insertion log information passed in the invocation parameter to determine whether data recovery is required. AWS Lambda does not guarantee that an invocation of a function constantly reuses the function instance P previously invoked. If P is not reused, the function-memory of N will not contain the most up-to-date object chunks, and these missing chunks will have to be recovered from COS.

An invoked function instance determines whether its objects are up-to-date by performing a consistency check that compares the local `term` and `hash` values that it has recorded with the corresponding values passed from the client daemon. If the values are inconsistent, the instance is considered to have *failed*.

When a function instance fails, InfiniStore needs to decide on an appropriate recovery strategy. Specifically, each instance maintains a `diff_rank` that indicates how many object chunks will be recovered since the first `term`, which equals the number of all `PUT`s including deleted chunks. A difference is calculated by subtracting the local `diff_rank` from the `diff_rank` received from the client daemon. If the difference is significantly larger than N, where N equals to the number of recovery functions assigned to the function, the failed instance notifies the client daemon that a *parallel recovery* is required (§5.5.2). Otherwise, the lost data can be recovered locally, and no further action from the client daemon is required.

InfiniStore uses the number of chunks rather than the aggregated chunk size as the indicator for parallel recovery because the recovery process can take advantage of the available massive parallelism of recovery functions only if the number of missing chunks is large enough. Regardless of whether a parallel recovery is triggered, the failed function instance will perform recovery locally. The failed instance recovers data by first downloading an operation manifest of the function from COS, which is a combination of a chunk list covered by the last snapshot and operations in the insertion nodes constructed since the latest snapshot, if any. The instance then replays all operations to find objects to be recovered and downloads objects from COS. Though `GET` requests will be blocked until the object has been downloaded, the request latency will not be affected, as discussed in §5.5.2. To guarantee `PUT` consistency, the being-recovered storage instance still serves `PUT` requests, and any successive read-after-write `GET`s that request the chunks inserted by these new `PUT`s during the recovery process.

5.5.2 Parallel Recovery. If parallel recovery is required, the failed function instance notifies the client daemon to start the parallel recovery process. The process involves three phases: recovery group selection, chunk recovery, and service resumption.

Phase 1: Recovery Group Selection. Each function in the SMS is assigned one of two roles. The role of a *storage* function is to store data objects in function-memory. The role of a *recovery* function is to mitigate the recovery phase's impact by recovering a part of all data missing from the failed storage function, in massive parallel, and delegating `GET` requests before the restoration of failed function. Each storage function is initialized with a group of recovery functions, which are randomly chosen from all the functions in the SMS (§5.3). The client daemon guarantees that each function may only serve as a recovery function for one storage function at any time. The daemon maintains a non-recovering function pool within the active GC-buckets. Upon starting a parallel recovery process, if any previously initialized recovery function is not available (i.e., serving another storage function), a new function will be selected from the pool to serve as the recovery function.

Phase 2: Chunk Recovery. Each recovery instance is assigned a unique ID i and is responsible for recovering a portion of all the chunks to be recovered, which are the chunks with hashed key j if j modulo the size of the recovery group equals i. Recovery instances execute the same recovery routine as described for the failed instance with the exception that they only download objects they are responsible.

To serve `GET` requests for chunks that are being recovered, the client daemon reroutes requests to the corresponding recovery functions. In §6.4, we show that parallel recovery instances will recover a 3,008 MB function within 1.18 s on average. With erasure coding, the client daemon can tolerate the loss/delay of up to the number of p parity chunks, which greatly reduces the possibility that the instance reclamation impacts the latency of `GET` requests. `PUT` requests are served by the failed instance, as discussed previously.

Phase 3: Service Resumption. Once the failed storage function instance finishes recovering all missing chunks, it notifies the client daemon, which then seamlessly redirects all further `GET` requests back to the recovered storage instance. The recovered chunks in the recovery function instances are retained for a certain period before being freed in case a parallel recovery for the same storage function is triggered again in the near future.

Note RAMCloud [35] assumes normal datacenter server failure rates, spreads recovered data across all the recovery nodes, and serves recovered data from recovery nodes permanently. In contrast, InfiniStore deals with a more dynamic and unpredictable FaaS environment with higher failure frequencies. To prevent cascading parallel recoveries caused by recovery functions being reclaimed during the recovery of a storage function instance, a recovery function instance only stores recovered data objects temporarily to mitigate the impact of storage instance recovery on request latency.

6 EVALUATION

In this section, we evaluate InfiniStore on AWS Lambda.

Implementation. We have implemented a production-quality prototype of InfiniStore atop InfiniCache by modifying and adding 21,397 lines of Go code [2] using about two person-years: 1,171 LoC for the client library, 10,296 for the client daemon, 6,429 for the Lambda runtime, and 3,501 for utilities shared across components.

Client Setup. Unless otherwise specified, we deployed the workload replayer and microbenchmarks together with a InfiniStore client daemon on a `c5n.4xlarge` EC2 VM instance.

Goals. We answer the following questions in the evaluation:

- How well does InfiniStore elastically adapt to real-world production storage workload changes (§6.1)?
- Is InfiniStore cost-effective and pay-per-access (§6.1.1)?
- How does InfiniStore perform under YCSB [17] stress testing compared to state-of-the-art cloud storage systems (§6.2)?
- How fast does InfiniStore scale out and react to throughput changes compared to state-of-the-art cloud storage systems (§6.3)?
- How fast can InfiniStore recover from function failures (§6.4)?
- How much do different design options contribute to InfiniStore's cost-effectiveness and latency improvement (§6.5)?

6.1 Applications

6.1.1 IBM Container Registry Workload. We now evaluate InfiniStore's elasticity, cost, and performance using the production IBM container registry workload[3]. The original workload contains a 75-day request trace collected from 7 geographically distributed datacenters. We use the first 50 hours of the workload from the

[3] §2 describes workload statistics in detail.

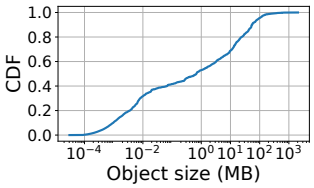


Figure 7: IBM container registry workload.

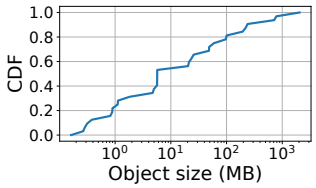


Figure 8: Azure Functions blob workload.

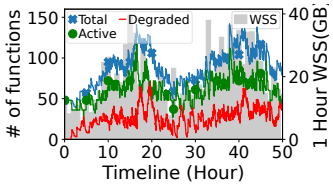


Figure 9: Number of functions managed by InfiniStore. Total=active+degraded.

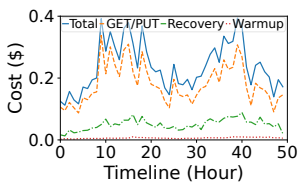


Figure 10: Timeline of hourly cost. Total=I/O+recovery+warm.

Table 2: Workload's memory-level read hit ratio achieved by InfiniStore (IS), InfiniCache (IC), ElastiCache as a storage (EC S), and ElastiCache as a cache (EC C). InfiniStore's read hit ratio is defined as the ratio of the number of object chunks read directly from the SMS layer against the total object chunk requests.

Workload	IS SMS	IC	EC S	EC C
IBM container registry	95.8%	95.4%	99.96%	93.3%

Dallas datacenter, which features the highest load (with an average throughput of 3,654 GET requests per hour). The distribution of object sizes is shown in Figure 7. Among all objects, 31% are larger than 10 MB. We compare our results with InfiniCache [51] and AWS ElastiCache for Redis [6]. Following workload configurations are applied to both InfiniCache and InfiniStore:

- Parse all the GET requests reading a container image layer.
- Include all objects (both small and large objects > 10 MB).
- Use a Reed-Solomon EC configuration of (10 + 2).
- Use Lambda function instances with 1,536 MB memory.

For InfiniCache, we use a fixed cluster of 400 Lambda functions and a warmup interval of 1 minute. InfiniStore applies a warmup interval of 1 minute for Lambda functions in active GC-buckets and a longer warmup interval of 5 minutes for degraded GC-buckets. The GC interval is set to 10 minutes. The number of active (M) and degraded (N) GC-buckets are set to 6 and 12, respectively, based on the average reuse interval and the average one-hour WSS.

Elasticity. We start by evaluating InfiniStore's ability to adapt to working set size changes (Figure 1(a)). Figure 9 shows the number of functions managed by InfiniStore over the 50-hour workload. Each curve corresponds to the number of functions managed in InfiniStore's active GC-buckets (green) and degraded GC-buckets (red) at each 1-minute time interval, regardless of whether the function instances are invoked or not. Note that the number of total functions matches 1 hour WSS changes, which are capped at 40 GB out of an aggregate 888 GB of workload's 50 hours WSS. The measured SMS-level hit ratio is 95.8% (see Table 2), which indicates that 95.8% of requests are directly served from SMS without loading data from COS. The high SMS-level hit ratio also indicates that InfiniStore can automatically capture application's working set.

Cost-effectiveness. Next, we evaluate the overall monetary cost of InfiniStore in comparison with InfiniCache and AWS ElastiCache for serving the container registry workload (Figure 11). For ElastiCache, we provisioned two Redis clusters: a cluster of 12 `cache.r6g.2xlarge` instances with 633.64 GB aggregate memory to provide full in-memory storage and a cluster of 8 `cache.m6g.large` instances with 51.04 GB aggregate memory to provide in-memory caching. For fairness, the cache cluster uses S3 as the backing store and has reduced hit ratio (Table 2). Both ElastiCache clusters have up to 10 Gbps network bandwidth per instance.

InfiniCache costs a total of $18.4, of which 43.48% is spent on the backup scheme. ElastiCache costs $492.6 (as a storage) and $61.02 (as a cache with S3 as the backing store), 36.30× and 4.50× more expensive than InfiniStore as it is statically provisioned. InfiniStore has the lowest cost of $13.57, of which $2.49 is used for parallel recoveries, $0.31 for warmup, and $1.68 for COS (S3) storing 1053 GB of data (including overhead of using erasure coding) in 50 hours. Note that InfiniStore's warmup cost is considerably lower than InfiniCache, which maintains a fixed-sized function pool with replicas, whereas InfiniStore uses a sliding-window scheme to dynamically adjust the number of functions in SMS.

Pay-per-Access. To find out whether InfiniStore delivers pay-per-access, we breakdown the cost per hour over the entire 50 hours (Figure 10). We observe a clear trend in that the total cost is proportional to the cost of serving GET/PUT requests. InfiniStore's parallel recovery scheme incurs a small portion (18.34%) of the overall cost, which is the recovery cost for handling 1,083 function instance failures during the workload.

InfiniStore's cost overhead over an ideally pay-per-access scheme is 26.00%. This result is calculated using the ratio of the aggregate cost of recovery + warmup (i.e., the extra activities required to maintain data durability) to the aggregate cost of serving GET/PUT requests + S3 cost (i.e., access and storage cost). This overhead is significantly lower than InfiniCache, which adds 106.51% for backup and warmup.

Latency Performance. Figure 12 compares InfiniStore's latency performance against InfiniCache, ElastiCache as a storage, and S3. The results of using ElastiCache as a cache are similar to use as a storage, which are not shown in plots. InfiniStore achieves lower latency than InfiniCache for more than 40% of the requests. This is because InfiniStore's two-queue request scheme effectively mitigates the convoy effect of large requests against small requests. InfiniStore is significantly faster than S3 for more than 70% of the requests. For objects larger than 10 MB, InfiniStore is two orders of magnitude faster than S3. For objects smaller than 1 MB, InfiniStore suffers from the overhead of Lambda function invocation and thus is lower than S3. InfiniStore exhibits comparable performance to ElastiCache for around 50% of requests reading objects over 10 MB (Figure 13). For the rest of the requests accessing smaller objects, InfiniStore does not see a benefit compared to ElastiCache, again because of the high invocation overhead. Overall, InfiniStore offers *a novel performance-$cost tradeoff* in the space of general-purpose cloud storage services.

6.1.2 Azure Functions Blob I/O Workload. The full Azure Functions trace contains 14 days of blob accesses in 855 serverless function

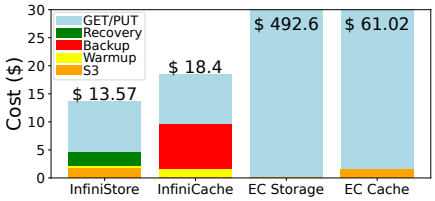


Figure 11: $ cost. EC: ElastiCache.

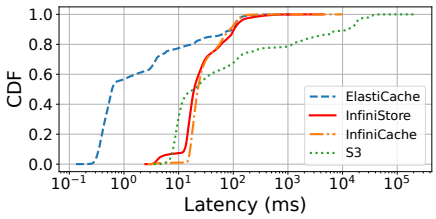


Figure 12: Latency comparison (all obj).

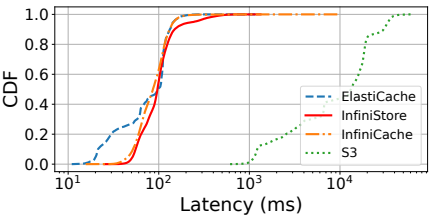
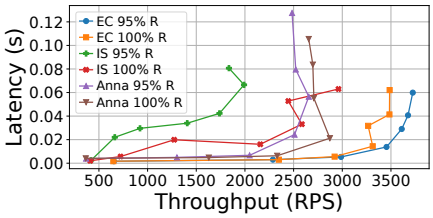
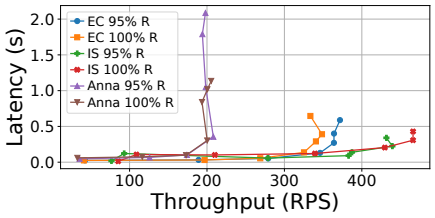


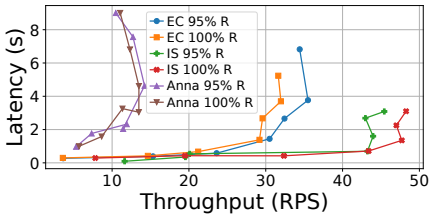
Figure 13: Latency comparison (>10 MB).



(a) 1 MB objects.



(b) 10 MB objects.



(c) 100 MB objects.

Figure 14: Throughput vs. the p90 read latency obtained under YCSB with various object sizes. IS: InfiniStore.

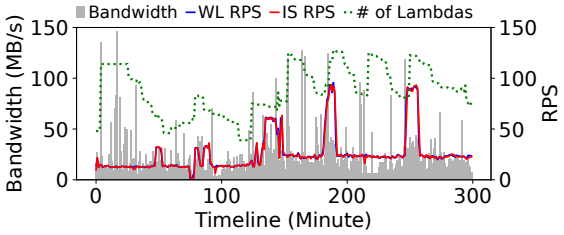


Figure 15: Timeline of the bandwidth and RPS (request per second) changes of the Azure Functions blob access workload. InfiniStore(IS)'s trace replayer faithfully replays the workload in real-time (`IS RPS`) with the same I/O concurrency as recorded in original traces (`WL RPS`).

applications and is highly-bursty (Figure 1(d)). We selected a 5-hour trace (hour 179–184), which contains the most bursty I/O pattern (with a mean $CoV > 1$) blob access information issued by a total of 37 function applications. We increased the blob size by $10,000\times$ to highlight how InfiniStore elastically scales in response to increasing bandwidth requirements. Figure 8 shows that 45% of the objects are larger than 10 MB. Since the Azure workload has much shorter object reuse intervals (see Figure 1(c)) than that of the container registry workload, we set the GC interval as 1 minute, and the number of active (M) and degraded (N) GC-buckets as 8 and 15, respectively. As shown in Figure 15, InfiniStore dynamically adjusts the number of Lambda functions based on the bandwidth requirement and the RPS (requests per second) changes. InfiniStore can effectively scale to the bandwidth requirement, thanks to InfiniStore's optimization for handling large objects (§5.3.4), which splits large objects into smaller pieces. This way, a large request gets converted into multiple smaller requests that can be quickly absorbed in parallel by scaled out Lambda function instances.

6.2 YCSB Microbenchmarking

Next, we stress-test the throughput-latency performance trade-off using the commonly-used YCSB [17] benchmark. We compare InfiniStore against two state-of-the-art cloud storage systems: (1) AWS ElastiCache for Redis, and (2) Anna [53], an auto-scaled, multi-tier cloud storage system used by Cloudburst [46] to support stateful serverless workloads. For both ElastiCache and Anna, we deploy a small serverful cluster consisting of three 8-vCPU, 50-GB-memory EC2 VMs with up to 30 Gbps aggregated, cluster-wide network bandwidth. The ElastiCache deployment and the Anna deployment use slightly different VM instance types due to limited options. Specifically, ElastiCache uses the `cache.r6g.2xlarge` cache node type, while Anna uses the `r4.2xlarge` EC2 instance type. The Anna deployment uses one additional `r4.2xlarge` EC2 instance as the routing server and one `m4.xlarge` EC2 instance for management and monitoring services. Both of the two deployments are configured with a replication factor of 1. YCSB runs on a `c5n.9xlarge` client VM with a fixed 50 Gbps network bandwidth to ensure that the client's network does not become a performance bottleneck. We configure Anna to use 8 worker threads for each of the three nodes in the storage cluster. InfiniStore uses a GC interval of 1 minute. For Anna, since there is no Anna-based YCSB binding available, we modified Anna's own benchmarking utility to make it generate YCSB-styled I/O tests. We use two read/update ratios: 95 : 5 and 100 : 0, and the Zipfian key popularity distribution with a Zipfian coefficient of 0.99. We run each YCSB test for 30 seconds for the following three object sizes: 1 MB, 10 MB, and 100 MB. For each test, the concurrency (i.e., number of YCSB threads) is configured to start from 1 and then gradually increase to 5, 10, 25, 50, 75, and finally 100. Figure 14 reports the performance results.

ElastiCache is highly optimized for AWS-hosted I/O workloads in that the YCSB benchmark can easily saturate the aggregated 30 Gbps cluster bandwidth for all three object sizes. ElastiCache also achieves the lowest p90 (90^{th}-percentile) latency for the 1-MB-object tests. Anna cannot fully utilize the cluster bandwidth. As the object size increases, Anna tends to have a reduced network bandwidth utilization (see Figure 14(b)-14(c)). To figure out the reason, we investigate Anna's implementation and observe that Anna incurs extra serialization/deserialization overhead when passing large messages through Anna's ZeroMQ service [56]. Such overhead increases significantly when object sizes increase. InfiniStore's elastic, scaled-out serverless design offers ample network bandwidth resources for applications. That is why (1) InfiniStore's

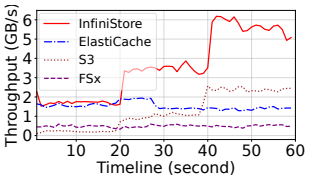


Figure 16: Throughput.

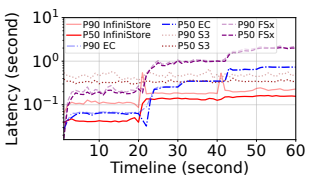


Figure 17: Latency (log).

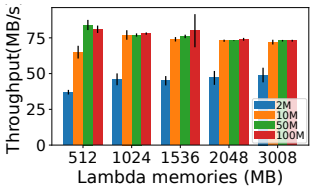


Figure 18: λ throughput.

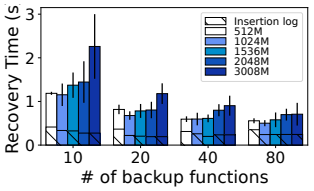


Figure 19: Recovery time.

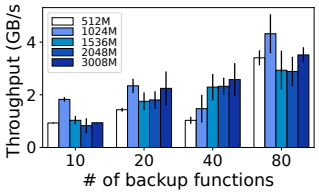


Figure 20: Recovery thpt.

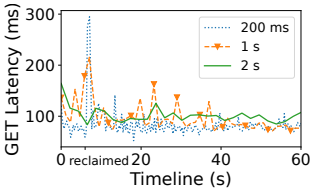


Figure 21: Impact on GETs.

throughput is only capped by the bandwidth limit of the client VM for both the 10 MB and 100 MB objects, and (2) InfiniStore achieves the lowest p90 latency compared to both ElastiCache and Anna. For the 100%-read, 1-MB-object workloads, InfiniStore achieves on-par throughput as Anna, with the highest throughput between 2,500-3,000 RPS and a slightly higher p90 latency (see Figure 14(a)). This is because InfiniStore fetches data from serverless functions but not a stable serverful cluster. The YCSB results show that InfiniStore is well suited for large-object-intensive workloads with object size larger than 10 MB.

6.3 Elasticity Microbenchmarking

In this section, we setup a multi-VM deployment using 10 `c5n.4xlarge` EC2 VMs to simulate a realistic use case in which a tenant has multiple microservices that concurrently issue `GET` requests for 10 MB objects. To evaluate InfiniStore's ability to scale out on demand, we vary the number of I/O threads on each client VM from 1 to 5 to 10. The load increases every 20 seconds. The WSS of the workload is 211 GB. We compare InfiniStore with three cloud storage services, an ElastiCache deployment of one 314.32 GB `cache.m5.24xlarge` instance (the smallest ElastiCache instance that provides 25 Gbps network bandwidth), a baseline FSx deployment of 1.2 TB, and S3.

Figure 16 shows that InfiniStore's throughput scales instantly as more clients are added. We can also see that S3 scales but with a much lower throughput. ElastiCache and FSx, on the other hand, hit a network bandwidth bottleneck with a capped throughput of 1.52 GB/s and 489 MB/s (FSx provides burst throughput), respectively. Both ElastiCache and FSx see a linear increase in latency (Figure 17), since they both need to be manually re-configured to scale out. At p90, we see a latency spike when the load increases. This is because more functions need to be invoked on demand to sustain the burst. As shown, InfiniStore quickly scales out to sustain the latency spike. InfiniStore achieves 46.72% and 81.84% lower latency at the 90^{th} percentile compared to ElastiCache and FSx, respectively.

6.4 Parallel Recovery

Next, we evaluate the performance and effectiveness of InfiniStore's parallel recovery scheme.

Lambda Throughput. To better understand the limitations of the parallel recovery performance, we examine the network throughput of the Lambda function instances. Each Lambda instance runs 10 threads that concurrently fetch data from S3 and we vary the Lambda function-memory size from 512 MB to 3008 MB while measuring the throughput of downloading objects with sizes ranging from 2 MB to 100 MB. Figure 18 reports the results. We observe that: (1) downloading objects of 2 MB cannot saturate a Lambda's network bandwidth, and (2) for all memory configurations, the sustained network throughput can reach up to around 75 MB/s, suggesting that a single recovery function instance is capable of recovering 100 MB of data in 1 or 2 seconds.

Performance of Parallel Recovery. To analyze InfiniStore's parallel recovery performance, we measure the time it takes to recover the data objects stored in a Lambda function instance with various memory configurations from 512 MB to 3, 008 MB. As shown in Figure 19, recovering a 3, 008 MB Lambda function instance takes, on average, 1.18 s when using 20 parallel recovery functions. The recovery time is reduced by only 39.9% if the number of recovery functions scales from 20 to 80. Insertion log downloading takes, on average, 271.4 ms of the average recovery time for each recovery function. To balance the gain and overhead, we choose to use 20 recovery functions in other experiments.

Accordingly, Figure 20 shows that the aggregate recovery throughput achieved to recover a 3, 008 MB function increases from 2.25 GB/s to 3.51 GB/s when scaling the recovery group from 20 to 80 functions. Similar trends can be observed for other Lambda configurations and recovery group sizes.

Impact of Recovery on Latency. We next evaluate the impact of parallel recovery on `GET` latency. In this test, we first load 2, 400 unique 10 MB objects into a InfiniStore deployment of one FG (Function Group) of 12 3, 008 MB Lambda instances with an EC code of $(10 + 2)$. We then run a client that issues random `GET` requests in a fixed interval. We choose three different intervals: 200 ms, 1 s, and 2 s. For each test run, we kill one Lambda function 5 s after the start of the `GET` workload to simulate a reclamation event performed by the provider. On detecting the reclamation of the Lambda instance, InfiniStore is configured to invoke a group of 20 independent recovery instances to perform the parallel recovery.

As shown in Figure 21, the client does not experience any service interruptions during the run with a 2 s request interval. Because InfiniStore finishes parallel recovery within 2 s, and the requests for the lost object chunks were seamlessly served by the 20 recovery instances while waiting for the storage instance to fully recover from S3. In the tests with a request interval of 200 ms or an interval of 1 s, we observe a latency increase of about 200 ms. This is because, although the build-in erasure coding feature can tolerate losing 2 out of $(10 + 2)$ chunks, the process of decoding and reconstructing the object has an impact on latency. However, after parallel recovery, the `GET` latency decreases, demonstrating the effectiveness of InfiniStore's parallel recovery scheme.

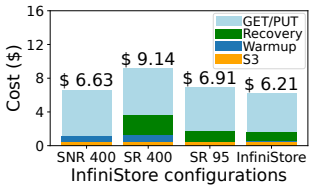


Figure 22: Cost of different configurations.

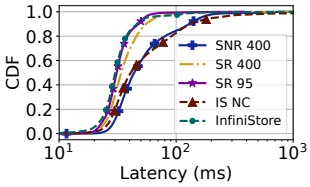


Figure 23: Latency of different configurations.

6.5 Factor Analysis

Finally, we measure how different design options affect InfiniStore's cost effectiveness and latency. We use the following InfiniStore configurations: (1) `SNR 400`: a static cluster of 400 Lambda functions with no parallel recovery; (2) `SR 400`: a static cluster of 400 Lambda functions with parallel recovery to emulate a static InfiniCache setup; (3) `SR 95`: a static cluster of 95 Lambda functions with parallel recovery; the rationale of using 95 functions is that we observe that InfiniStore uses an average of 95 functions per minute during the 24-hour workload, though with dynamic adaptations throughout; thus, for comparison, we also test the baseline `SR 95` to highlight the benefits brought by InfiniStore's sliding-window management; (4) `IS NC`: InfiniStore with no temporary cache functions for serving bursty request; (5) InfiniStore with all features enabled. We drive the tests using a 24-hour IBM container workload with all workload configurations the same as used in §6.1.1. Figure 22 and 23 report the results. `IS NC`'s cost is omitted in Figure 22 as it is almost the same as that of InfiniStore. Figure 23 omits the latency results of objects larger than 10 MB as they show negligible differences across all configurations.

Impact of Parallel Recovery. As shown in Figure 23, `SNR 400` observes increased latency as data objects lost due to function reclamation must be fetched from S3. Interestingly, `SNR 400` costs even more than InfiniStore despite the smaller cost overhead of warmup (Figure 22): `SNR 400` spends more money serving `GET/PUT` requests as significantly more SMS misses lead to prolonged Lambda execution time for fetching missed objects from S3.

Impact of Sliding-window Management and Cache Functions. InfiniStore dynamically adjusts SMS capacity based on the workload's WSS, while `SR 95` uses a fixed function instance pool whose capacity is on average slightly larger than the actual WSS: `SR 95` costs 11.3% more than InfiniStore (Figure 22). Recall that temporary cache functions are launched to serve request bursts (§5.3.1 and §5.3.3). `IS NC` has an object chunks miss rate of 18.3% at the SMS layer with cache functions disabled, compared to a much lower miss rate of 1.73% under InfiniStore. As a result, we see from Figure 23 that `IS NC` suffers a big latency penalty.

7 RELATED WORK

Serverless Data Services. Fully-managed cloud storage services [5, 9, 21, 33, 48] transparently manage storage resources for tenants. Researchers have also explored offloading data services to serverless functions. Starling [37] supports database query processing using serverless functions. Zion [43] extends object storage by offloading data processing to stateless serverless functions. Unlike InfiniStore, these systems do not directly store data in functions, which presents new challenges that InfiniStore addresses.

Caches/Storage for Serverless Applications. OFC [34] is an opportunistic memory cache that leverages overbooked host memory of serverless functions to accelerate function execution. Faa$T [40] is a distributed cache co-located with FaaS applications. Faa$T scales as the application scales. Pocket [28, 29] is an elastic ephemeral storage for serverless analytics [13, 14, 24, 38]. SONIC [32] optimizes data exchange among chained functions. AFT [45] is a fault-tolerant shim providing atomic transaction guarantees to serverless applications that store data in cloud storage. Shedder [58] moves function computation directly to cloud storage. Anna's lattice support can achieve coordination-free causal consistency [53]. Cloudburst [46] adds host-local cache atop Anna to support stateful function pipelines. FaaSNet [50] accelerates container provisioning for serverless functions using a P2P tree. In contrast, InfiniStore's SMS layer leverages serverless functions to achieve rapid and find-grained storage-level scaling and is designed to serve the I/Os of general cloud applications.

Cost-effective Cloud Storage. Prior works exploit the performance and cost heterogeneity of a combination of cloud storage services (e.g., VM-local block drives, memory caches, and object stores) offered by hybrid cloud providers in order to minimize the cost of data services [1, 3, 15, 16, 36, 39, 54, 55]. InfiniStore takes a new route—exploiting the pay-per-use pricing model of FaaS to achieve pay-per-access storage.

Garbage Collection in Storage Systems. LFS [41] designs a scavenging GC-style segment cleaner to defragment log segments in the file system. RAMCloud [42] inherits the idea and uses the cleaner to free the space in its log-structured memory. Jiffy [27] uses leases to identify live data and reclaims memory if the leases expire. Data repartitioning is needed for Jiffy to be elastic. InfiniStore is the first cloud memory storage that combines the traditional garbage collection technique and the FaaS properties to achieve fine-grained elasticity in disaggregated memory management.

8 CONCLUSION

In this paper, we rethink the fundamental design of cloud storage systems and propose ServerlessMemory, a new cloud storage service. ServerlessMemory fills a gap in cloud storage services by offering a truly elastic, highly performant, yet low-cost with a pay-per-access memory storage layer. By seamlessly combining ServerlessMemory with a cost-effective cloud object store, we present a generic and novel cloud storage system, InfiniStore, which supports uninterrupted data services by employing an automated, fast, and reliable parallel recovery scheme. Extensive evaluation via real-world production storage workloads and microbenchmarks demonstrates that InfiniStore outperforms existing systems while lowering costs with its pay-per-access pricing model. Furthermore, InfiniStore presents a new performance-$cost tradeoff in the cloud storage landscape. InfiniStore is open source and publicly available at: https://github.com/ds2-lab/infinistore.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their valuable feedback and comments. This work was sponsored in part by NSF grants: CNS-2045680 (an NSF CAREER Award), CAREER-2048044, an NSF CloudBank grant, CCF-1919075, CCF-1919113, OAC-2106446, and supported by an Adobe Research gift.

REFERENCES

[1] Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. 2010. RACS: A Case for Cloud Storage Diversity. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. ACM, New York, NY, USA, 229–240. https://doi.org/10.1145/1807128.1807165

[2] AlDanial. 2022. CLOC: Count Lines of Code. https://github.com/AlDanial/cloc/.

[3] Ali Anwar, Yue Cheng, Aayush Gupta, and Ali R. Butt. 2016. MOS: Workload-Aware Elasticity for Cloud Object Stores. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing* (Kyoto, Japan) *(HPDC '16)*. Association for Computing Machinery, New York, NY, USA, 177–188. https://doi.org/10.1145/2907294.2907304

[4] Ali Anwar, Mohamed Mohamed, Vasily Tarasov, Michael Littley, Lukas Rupprecht, Yue Cheng, Nannan Zhao, Dimitrios Skourtis, Amit S. Warke, Heiko Ludwig, Dean Hildebrand, and Ali R. Butt. 2018. Improving Docker Registry Design Based on Production Workload Analysis. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*. USENIX Association, Oakland, CA, 265–278. https://www.usenix.org/conference/fast18/presentation/anwar

[5] AWS. 2022. Amazon DynamoDB: Fast, flexible NoSQL database service for single-digit millisecond performance at any scale. https://aws.amazon.com/dynamodb/.

[6] AWS. 2022. AWS ElastiCache: Unlock microsecond latency and scale with in-memory caching. https://aws.amazon.com/elasticache/.

[7] AWS. 2022. AWS FSx: Launch and run feature-rich and highly-performant file systems with just a few clicks. https://aws.amazon.com/fsx/.

[8] AWS. 2022. AWS Lambda execution context. https://docs.aws.amazon.com/lambda/latest/dg/runtimes-context.html.

[9] AWS. 2022. AWS S3: Object storage built to retrieve any amount of data from anywhere. https://aws.amazon.com/s3/.

[10] AWS. 2023. AWS Lambda: Run code without thinking about servers or clusters. https://aws.amazon.com/lambda/.

[11] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. 2010. Finding a Needle in Haystack: Facebook's Photo Storage. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI'10)*. USENIX Association, Berkeley, CA, USA, 47–60. http://dl.acm.org/citation.cfm?id=1924943.1924947

[12] Alysson Bessani, Ricardo Mendes, Tiago Oliveira, and Nuno Neves. 2014. SCFS: A Shared Cloud-backed File System. *Proceedings of the 2014 USENIX Annual Technical Conference* (2014), 169–180. https://www.usenix.org/conference/atc14/technical-sessions/presentation/bessani

[13] Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, Panruo Wu, and Yue Cheng. 2020. Wukong: A Scalable and Locality-Enhanced Framework for Serverless Parallel Computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3419111.3421286

[14] Benjamin Carver, Jingyuan Zhang, Ao Wang, and Yue Cheng. 2019. In Search of a Fast and Efficient Serverless DAG Engine. In *4th International Parallel Data Systems Workshop (PDSW 2019)*.

[15] Yue Cheng, M. Safdar Iqbal, Aayush Gupta, and Ali R. Butt. 2015. CAST: Tiering Storage for Data Analytics in the Cloud. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing* (Portland, Oregon, USA) *(HPDC '15)*. ACM, New York, NY, USA, 45–56. https://doi.org/10.1145/2749246.2749252

[16] Yue Cheng, M. Safdar Iqbal, Aayush Gupta, and Ali R. Butt. 2015. Pricing Games for Hybrid Object Stores in the Cloud: Provider vs. Tenant. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*. USENIX Association, Santa Clara, CA. https://www.usenix.org/conference/hotcloud15/workshop-program/presentation/cheng

[17] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. ACM, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[18] Francisco Cruz, Francisco Maia, Miguel Matos, Rui Oliveira, João Paulo, José Pereira, and Ricardo Vilaça. 2013. MeT: Workload Aware Elasticity for NoSQL. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic) *(EuroSys '13)*. Association for Computing Machinery, New York, NY, USA, 183–196. https://doi.org/10.1145/2465351.2465370

[19] Docker. 2022. Docker Hub: Container Image Library. https://www.docker.com/products/docker-hub.

[20] Google. 2022. Google Cloud Functions. https://cloud.google.com/functions/.

[21] Google. 2022. Google Cloud Storage. https://cloud.google.com/storage.

[22] IETF HTTP Working Group. 2022. HTTP/2. https://http2.github.io/.

[23] IBM. 2022. IBM Cloud Functions. https://console.bluemix.net/openwhisk/.

[24] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the Cloud: Distributed Computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. ACM, New York, NY, USA, 445–451. https://doi.org/10.1145/3127479.3128601

[25] Richard Jones and Rafael. Lins. 1996. Garbage collection : algorithms for automatic dynamic memory management. (1996), 377.

[26] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. 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* (El Paso, Texas, USA) *(STOC '97)*. Association for Computing Machinery, New York, NY, USA, 654–663. https://doi.org/10.1145/258533.258660

[27] Anurag Khandelwal, Yupeng Tang, Rachit Agarwal, Aditya Akella, and Ion Stoica. 2022. Jiffy: Elastic Far-Memory for Stateful Serverless Analytics. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 697–713. https://doi.org/10.1145/3492321.3527539

[28] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. 2018. Understanding Ephemeral Storage for Serverless Analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 789–794. https://www.usenix.org/conference/atc18/presentation/klimovic-serverless

[29] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 427–444. https://www.usenix.org/conference/osdi18/presentation/klimovic

[30] Andrew W. Leung, Shankar Pasupathy, Garth Goodson, and Ethan L. Miller. 2008. Measurement and Analysis of Large-Scale Network File System Workloads. In *USENIX 2008 Annual Technical Conference* (Boston, Massachusetts) *(ATC'08)*. USENIX Association, USA, 213–226.

[31] M. Littley, A. Anwar, H. Fayyaz, Z. Fayyaz, V. Tarasov, L. Rupprecht, D. Skourtis, M. Mohamed, H. Ludwig, Y. Cheng, and A. R. Butt. 2019. Bolt: Towards a Scalable Docker Registry via Hyperconvergence. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. 358–366. https://doi.org/10.1109/CLOUD.2019.00065

[32] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2021. SONIC: Application-aware Data Passing for Chained Serverless Applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 285–301. https://www.usenix.org/conference/atc21/presentation/mahgoub

[33] Microsoft. 2022. Azure Blob Storage. https://azure.microsoft.com/en-us/services/storage/blobs/.

[34] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. 2021. OFC: An Opportunistic Caching System for FaaS Platforms. In *Proceedings of the Sixteenth European Conference on Computer Systems*, Vol. 21. ACM, New York, NY, USA, 17. https://doi.org/10.1145/3447786.3456239

[35] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. 2011. Fast Crash Recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 29–41. https://doi.org/10.1145/2043556.2043560

[36] T. G. Papaioannou, N. Bonvin, and K. Aberer. 2012. Scalia: An adaptive scheme for efficient multi-cloud storage. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–10. https://doi.org/10.1109/SC.2012.101

[37] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. 2020. Starling: A Scalable Query Engine on Cloud Functions. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 131–141. https://doi.org/10.1145/3318464.3380609

[38] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 193–206. https://www.usenix.org/conference/nsdi19/presentation/pu

[39] Krishna P.N. Puttaswamy, Thyaga Nandagopal, and Murali Kodialam. 2012. Frugal Storage for Cloud File Systems. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys '12)*. ACM, New York, NY, USA, 71–84. https://doi.org/10.1145/2168836.2168845

[40] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. 2021. *Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications*. Association for Computing Machinery, New York, NY, USA, 122–137. https://doi.org/10.1145/3472883.3486974

[41] Mendel Rosenblum and John K. Ousterhout. 1992. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)* 10 (2 1992), 26–52. Issue 1. https://doi.org/10.1145/146941.146943

[42] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. 2014. Log-structured Memory for DRAM-based Storage. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*. USENIX Association, Santa Clara, CA, 1–16. https://www.usenix.org/conference/fast14/technical-sessions/presentation/rumble

[43] Josep Sampé, Marc Sánchez-Artigas, Pedro García-López, and Gerard París. 2017. Data-Driven Serverless Functions for Object Storage. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference* (Las Vegas, Nevada) *(Middleware '17)*. Association for Computing Machinery, New York, NY, USA, 121–133. https://doi.org/10.1145/3135974.3135980

[44] Mikhail Shilkov. 2018. Serverless: Cold Start War. https://mikhail.io/2018/08/serverless-cold-start-war/.

[45] Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph E. Gonzalez, Joseph M. Hellerstein, and Jose M. Faleiro. 2020. A Fault-Tolerance Shim for Serverless Computing. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 15, 15 pages. https://doi.org/10.1145/3342195.3387535

[46] Vikram Sreekanti, Chenggang Wu, Charles Lin, Johann Schleier-Smith, Joseph E Gonzalez, Joseph M Hellerstein, Alexey Tumanov, U C Berkeley, Georgia Tech, and Jose M Faleiro. [n.d.]. Cloudburst: Stateful Functions-as-a-Service. ([n. d.]). https://doi.org/10.14778/3407790.3407836

[47] Huangshi Tian, Yunchuan Zheng, and Wei Wang. 2019. Characterizing and Synthesizing Task Dependencies of Data-Parallel Jobs in Alibaba Cloud. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 139–151. https://doi.org/10.1145/3357223.3362710

[48] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, James Corey, Kamal Gupta, Murali Brahmadesam, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvilli, and Xiaofeng Bao. 2018. Amazon Aurora: On Avoiding Distributed Consensus for I/Os, Commits, and Membership Changes. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 789–796. https://doi.org/10.1145/3183713.3196937

[49] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building An Elastic Query Engine on Disaggregated Storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 449–462. https://www.usenix.org/conference/nsdi20/presentation/vuppalapati

[50] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. 2021. FaaSNet: Scalable and Fast Provisioning of Custom Serverless Container Runtimes at Alibaba Cloud Function Compute. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 443–457. https://www.usenix.org/conference/atc21/presentation/wang-ao

[51] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. 2020. InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 267–281. https://www.usenix.org/conference/fast20/presentation/wang-ao

[52] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 133–146. https://www.usenix.org/conference/atc18/presentation/wang-liang

[53] Chenggang Wu, Jose M. Faleiro, Yihan Lin, and Joseph M. Hellerstein. 2021. Anna: A KVS for Any Scale. *IEEE Transactions on Knowledge and Data Engineering* 33 (2 2021), 344–358. Issue 2. https://doi.org/10.1109/TKDE.2019.2898401

[54] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V. Madhyastha. 2013. SPANStore: Cost-effective Geo-replicated Storage Spanning Multiple Cloud Services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. ACM, New York, NY, USA, 292–308. https://doi.org/10.1145/2517349.2522730

[55] Zhe Wu, Curtis Yu, and Harsha V. Madhyastha. 2015. CosTLO: Cost-Effective Redundancy for Lower Latency Variance on Cloud Storage Services. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 543–557. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/wu

[56] ZeroMQ. 2022. ZeroMQ. https://zeromq.org/.

[57] Jingyuan Zhang, Ao Wang, Xiaolong Ma, Benjamin Carver, Nicholas John Newman, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. 2022. InfiniStore: Elastic Serverless Cloud Storage. https://doi.org/10.48550/ARXIV.2209.01496

[58] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. 2019. Narrowing the Gap Between Serverless and its State with Storage Functions. In *Proceedings of the ACM Symposium on Cloud Computing - SoCC '19*. Association for Computing Machinery (ACM), New York, New York, USA, 1–12. https://doi.org/10.1145/3357223.3362723