InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache

Ao Wang and Jingyuan Zhang, *George Mason University;*
Xiaolong Ma, *University of Nevada, Reno;* Ali Anwar, Lukas Rupprecht,
Dimitrios Skourtis, and Vasily Tarasov, *IBM Research–Almaden;*
Feng Yan, *University of Nevada, Reno;* Yue Cheng, *George Mason University*

https://www.usenix.org/conference/fast20/presentation/wang-ao

This paper is included in the Proceedings of the
18th USENIX Conference on File and
Storage Technologies (FAST '20)
February 25–27, 2020 • Santa Clara, CA, USA

978-1-939133-12-0

InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache

Ao Wang[1]*, Jingyuan Zhang[1]*, Xiaolong Ma[2], Ali Anwar[3], Lukas Rupprecht[3], Dimitrios Skourtis[3], Vasily Tarasov[3], Feng Yan[2], Yue Cheng[1]

[1]*George Mason University* [2]*University of Neveda, Reno* [3]*IBM Research–Almaden*

Abstract

Internet-scale web applications are becoming increasingly storage-intensive and rely heavily on in-memory object caching to attain required I/O performance. We argue that the emerging serverless computing paradigm provides a well-suited, cost-effective platform for object caching. We present InfiniCache, a *first-of-its-kind* in-memory object caching system that is completely built and deployed atop ephemeral serverless functions. InfiniCache exploits and orchestrates serverless functions' memory resources to enable elastic pay-per-use caching. InfiniCache's design combines erasure coding, intelligent billed duration control, and an efficient data backup mechanism to maximize data availability and cost effectiveness while balancing the risk of losing cached state and performance. We implement InfiniCache on AWS Lambda and show that it: (1) achieves 31 – 96× tenant-side cost savings compared to AWS ElastiCache for a large-object-only production workload, (2) can effectively provide 95.4% data availability for each one hour window, and (3) enables comparative performance seen in a typical in-memory cache.

1 Introduction

Internet-scale web applications are becoming increasingly important as they offer many useful services to the end users. Examples range from social networks [22] that serve billions of photo and video files every day to hosted container image repositories such as Docker Hub [5]. These web applications typically require a large storage capacity for the massive amount of data they must store. For instance, Docker Hub hosts over 2.6 million container images, and Facebook generates 4 PB of data daily [6].

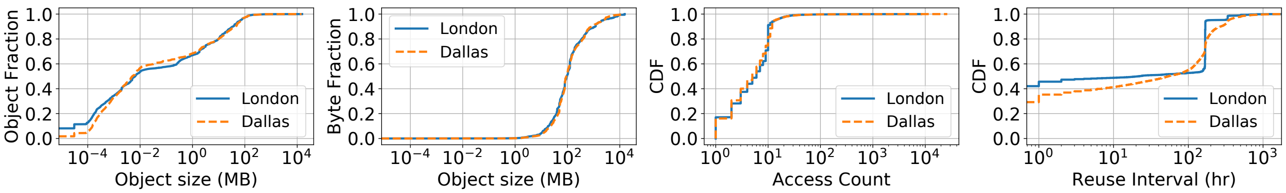
Cloud object stores (e.g., Amazon S3, Google Cloud Storage, OpenStack Swift, etc.) have become the first choice for serving the simple object GET/PUT requests of these storage-intensive web applications. To improve request latencies for better user experience, cloud object stores are typically being used in combination with networked, lookaside In-Memory Object Caches (IMOCs) such as Redis [10] and Memcached [9]. Serving requests from an IMOC is much faster than serving them directly from a backing object store. However, due to the high cost of main memory, IMOCs are largely used only as a small cache for buffering small-sized objects that range in size from a few bytes to a few KBs [19]. Caching large objects (i.e., objects with sizes of MBs–GBs) is believed to be relatively inefficient in an IMOC as large objects consume significant memory capacity and network bandwidth. This either causes cache churn with evictions of many small objects that would be reused soon if the cache is too small, or incurs high cost for larger cache sizes.

Large object caching has been demonstrated to be effective and beneficial in cluster computing [16, 38, 47, 58]. To verify that these benefits also apply to web applications, we analyzed production traces from an IBM Docker registry [17] and identified two key properties for large objects: (1) large objects are heavily reused with strong data locality and are accessed less frequently than small ones, and (2) achieving a fast access speed for large objects is critical for system performance though it does not require as stringent a service level objective (SLO) as that for small objects, the latter of which demands sub-millisecond latencies. These properties suggest that web applications can benefit from large object caching, which state-of-the-art IMOCs currently do not provide.

The emerging serverless computing paradigm (cloud function services, or Function-as-a-Service (FaaS)) [36] introduces a new way of building and deploying applications, in which the service providers take care of resource scaling and management. Developers can thus focus on developing the function logic without managing servers. Popular uses of serverless computing today are event-driven and stateless applications such as web/API serving and batch ETL (extract, transform, and load) [1]. However, we find that serverless computing can also provide a potential cost-effective solution for resolving the tension between small and large objects in memory caching.

We demonstrate how to build an IMOC as a serverless application. A serverless application is structured as a collection of cloud functions. A function has memory that can be used to store objects that are needed during its execution. We use this memory to store cached objects. Functions are executed on demand. In our serverless IMOC, the functions are invoked by the tenant to access the cached objects. FaaS providers cache invoked functions and their state so in-memory objects are retained between function invocations. This provides a sufficient lifetime for cached objects. Providers only charge tenants when a function is invoked, in our case, when a cached object is accessed. Thus the memory capacity used to cache an object is billed only when there is a request hitting that object. *Our serverless IMOC reduces the tenants' monetary cost*

[*] These authors contributed equally to this work.



(a) Object size. (b) Object footprint. (c) Access count for obj. > 10 MB. (d) Reuse interval for obj. > 10 MB.

Figure 1: Characteristics of object sizes and access patterns in the IBM Docker registry production traces.

of memory capacity compared to other IMOCs that charge for memory capacity on an hourly basis whether the cached objects are accessed or not.

Utilizing the memory of cloud functions for object caching introduces non-trivial challenges due to the limitations and constraints of serverless computing platforms: Cloud functions have limited resource capacity (e.g., 1 CPU, up to several GB memory, and limited network bandwidth) with strict network communication constraints (e.g., no inbound TCP connection); providers may reclaim a function and its memory at any time, creating a risk of loss of the cached data.

We present INFINICACHE, a cost-effective in-memory object cache that exploits and orchestrates serverless cloud functions. INFINICACHE synthesizes a series of techniques into a holistic design to overcome the aforementioned challenges and to achieve high performance, cost effectiveness, scalability, and fault tolerance. INFINICACHE leverages erasure coding to: (1) provide fault tolerance against data loss due to function reclamation by the service provider; (2) improve performance by utilizing the aggregated network bandwidth of multiple cloud functions in parallel; and (3) use redundancy to handle tail latencies caused by straggling functions. INFINICACHE implements function orchestration policies that improve reliability while lowering cost. Specifically, INFINICACHE implements a lightweight data backup mechanism in which a cloud function periodically performs delta synchronization (delta-sync) with a *clone* of itself so as to minimize the chances that a reclaimed function causes a data loss.

In summary, this paper makes the following contributions:

- Identify the opportunities and challenges of serverless function-based object caching by performing a long-term analysis of the internal mechanisms of a popular serverless computing platform (AWS Lambda [2]).
- Design and implement INFINICACHE, the very first in-memory object caching system powered by ephemeral and “stateless” cloud functions.
- Provide an analytical model of INFINICACHE’s fault tolerance mechanism built using erasure coding and periodic delta-sync techniques.
- Perform an extensive evaluation using both microbenchmark and production workloads. Experimental results show that INFINICACHE achieves performance comparable to ElastiCache for large objects and improves the cost effectiveness of cloud IMOCs by 31 – 96×.

2 Background and Motivation

Large-scale web applications have increasingly complex storage workload characteristics. Many modern web applications utilize a microservice architecture, which consists of hundreds to thousands of microservice modules [33]. Different modules exhibit different object size distributions and request patterns. For example, a Docker image registry service uses Redis to store small-sized container metadata (i.e., manifests), and an object store to store large-sized container images [17, 40]. While in-memory caching has been extensively studied in the context of large-scale web applications focusing on small objects, cloud cache management for large objects remains poorly explored and poses further challenges.

2.1 Large Object Caching

To obtain a better understanding of large object caching, we analyze production traces from an IBM Docker registry collected in 2017 from two datacenters (one in London, UK, and the other in Dallas, US) [17]. The goal is to reveal patterns that enable us to make realistic assumptions for the design of INFINICACHE.

Extreme Variability in Object Size. We first analyze the object size distributions. As shown in Figure 1(a), we find that object sizes span over nine orders of magnitude, and that more than 20% of objects are larger than 10 MB in size. This observation highlights the extreme variability and heterogeneity of real-world object store workloads, which further increases the complexity of cloud IMOC management.

Tension between Small and Large Objects. Efficiently managing both small and large objects in an IMOC is challenging due to two performance-cost tradeoffs. First, with limited cache capacity, large objects occupy a large amount of memory and would cause evictions of many small objects that might be reused in the near future, thus hurting performance. This is evidenced by Figure 1(b), where large objects (with size larger than 10 MB) occupy more than 95% of the total storage footprint. Second, large object requests typically consume significant network bandwidth resources, which may inevitably affect the latencies of small objects.

On one end, to prevent large objects from consuming too much memory and starving small object requests, an object size threshold is defined to not admit objects larger than the threshold [13, 23]. On the other end, system administrators can simply provision more memory (and thus more servers) to increase the capacity of the cache. However, this would increase the total cost of ownership (TCO) with reduced resource utilization. In fact, according to our analysis of the production Docker registry workloads, for the busiest deployment among seven datacenters, the average throughput of requests with object sizes greater than 10MB is below $3,500$ `GETs` per hour.

Caching Large Objects Matters. While large object caching is challenging, it can provide significant benefit as large object workloads exhibit strong data locality. Figure 1(c) plots the access frequency distribution for all objects larger than 10 MB. About 30% of large objects are accessed at least 10 times, and the object popularity shows a long-tail distribution, with the most popular objects absorbing more than 10^4 accesses. Figure 1(d) shows the temporal reuse patterns of the large object workloads. Around 37%–46% large objects are reused within 1 hour since the last time they were accessed. The strong temporal locality patterns underscore the benefit for caching large objects for web applications.

2.2 Building a Memory Cache on Cloud Functions: Opportunities and Challenges

The above observations lead to an important question to the storage system designers and cluster administrators: *can we build a new cloud caching model that relieves the tension between performance and cost while serving large objects in a cost-effective manner?* We argue that what is missing is a truly elastic cloud storage service model that charges tenants in a request driven mode instead of capacity usage, which the emerging serverless computing naturally enables, with the following desirable properties:

Pay-Per-Use Pricing: FaaS providers (including AWS Lambda [2], Google Cloud Functions [7], Microsoft Azure Functions [4], and IBM Cloud Functions [8]) charge users at a fine granularity – for example, AWS Lambda bills on a per-invocation basis ($0.02 per 1 million invocations) and charges (CPU and memory bundle) resource usage by rounding up the function's execution time to the nearest 100 milliseconds with a rate of $0.0000166667 per second for each GB of RAM. Note the function startup cost is not billed, and does not count for its execution time. Large object IMOC workloads can take advantage of this fine-grained pay-as-you-go pricing model to keep the tenant's monetary costs low.

Short-Term Caching: More importantly, FaaS providers keep functions "warm" by caching their state in memory for a short period of time to mitigate the "cold-start" penalty[1] [15, 43, 54]. Functions that are not invoked for a while can be reclaimed by the provider, and the state stored in the functions is lost. The duration of the "warm" period may vary (ranging from tens of minutes to longer than 6 hours as observed in §4.1) for AWS Lambda, and largely depends on how frequently the Lambda function gets invoked.

Ideally, a cloud tenant can leverage the above properties naturally enabled by a FaaS provider to build an opportunistic IMOC on a serverless platform. As such, a naive design would simply invoke a cloud function and store objects into the function's memory until the function is reclaimed by the provider, and then re-insert the objects into a new function.

This approach is appealing for several reasons. First and foremost, it inherently redefines the pay-as-you-go pricing model in the context of storage (in our case memory cache storage) by realizing a new form of memory elasticity — the memory capacity used to cache an object is billed only when there is a request hitting that object. This significantly differentiates the proposed cache model against conventional cloud storage or cache services, which start charging tenants for capacity usage whenever the capacity has been committed in use. Second, it offers a virtually infinite (yet cheap) short-term capacity, which is advantageous for large object caching, since the tenants can invoke many cloud functions but have the provider pay the cost of function caching[2].

However, FaaS providers place limits on the use of cloud resources to simplify resource management, which introduces challenges in building a stateful cache service atop stateless cloud functions. Take AWS Lambda for example — each Lambda function comes with a limited CPU and memory capacity; tenants can choose a memory amount between 128MB and 3008MB in 64MB increments. Lambda allocates CPU power linearly in proportion to the amount of memory configured, capped by 1.7 cores. Each Lambda function can run at most 900 seconds (15 minutes) and will be forcibly returned when the function times out. In addition, Lambda only allows outbound TCP network connections and bans inbound connections and UDP traffic, meaning a Lambda function cannot be used to implement a server, which is necessary for stateful applications such as IMOC. However, once an outbound TCP connection is established, it can be used to issue (multiple) requests to the function. Another limitation that plagues the performance of serverless applications is the lack of quality-of-service (QoS) control. As a result, functions suffer from straggler issues [45]. *Therefore, an ideal IMOC built atop cloud functions must provide effective workaround solutions to all the above challenges.*

3 INFINICACHE Design

INFINICACHE has three components: an INFINICACHE client library, a proxy, and a Lambda function runtime used to implement cache nodes[3]. As shown in Figure 2, an INFINICACHE deployment consists of a cluster of Lambda cache nodes, which are logically partitioned and managed by multiple proxies. Each proxy orchestrates a *Lambda cache pool*. Applications interact with INFINICACHE via a client library

[1] "Cold start" refers to the first-ever invocation of a function instance.

[2] FaaS providers essentially pay for the cost of storing the objects, while the tenants pay for the function invocations and function duration.

[3] We use Lambda cache node and Lambda function (runtime) interchangeably in different contexts.

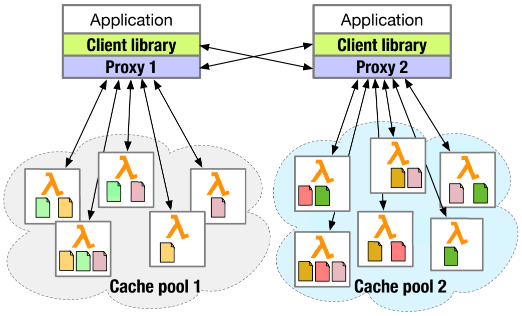


Figure 2: INFINICACHE architecture overview. Icon 🗎 denotes EC-encoded object chunks. Chunks with same color belong to the same object.

that is responsible for cache invalidation upon an overwrite and cache insertion upon a read miss assuming a read-only, write-through cache; the client library encodes and decodes the objects using erasure coding (EC) and interfaces with a proxy serving as a rendezvous that streams the EC-encoded object chunks between a client library and the Lambda nodes.

INFINICACHE introduces a proxy primarily because a Lambda node cannot run in server mode due to banned inbound connections. Thus a client library has to rely on an intermediate server (the proxy) for accepting connection requests from Lambda nodes. In INFINICACHE, the client library and proxy are logically separated as they have clearly partitioned functionality, but in deployment they can be physically co-located on the same machine. To enable data sharing across different Lambda cache pools, a client can communicate with any proxy (see Figure 2).

3.1 Client Library

INFINICACHE's client library exposes to the application a clean set of `GET(key)` and `PUT(key, value)` APIs (see Figure 3). The client library is responsible for: (1) transparently handling object encoding/decoding using an embedded EC module, (2) load balancing the requests across a distributed set of proxies, and (3) determining where EC-encoded chunks are placed on a cluster of Lambda nodes.

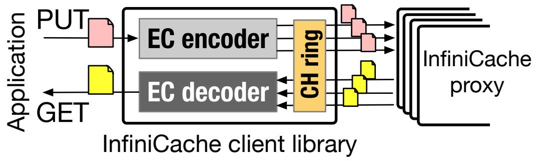


Figure 3: INFINICACHE client library (CH: consistent hashing).

Erasure Coding Processing. In our initial design, we observed that adding EC processing to the proxy would stall the chunk streaming pipeline (§3.2) and significantly impact the overall data transfer performance. Hence we made a design choice to move the computation-heavy EC part from the proxy to the client library.

The `PUT` Path. Assume that we have a multi-proxy deployment in which each proxy manages a separate Lambda node pool with shared access among clients. For a `PUT` request, INFINICACHE's client library first determines the destination proxy (and therefore its backing Lambda pool) by using a consistent hashing-based load balancing approach. The client library then encodes the object with a pre-configured EC code ($(d+p)$ using a Reed-Solomon (RS) code) and produces a number of object chunks, each with a unique identifier ID_{obj_chunk} (computed as a concatenation of the object key and the chunk's sequence number). To handle extremely large objects, INFINICACHE can encode them with more aggressive EC code (e.g., $(20+4)$). Next, the client decides which Lambda nodes to store the chunks on by randomly generating a vector of non-repetitive ID_{λ}. Each encoded chunk with its piggybacked $<ID_{obj\_chunk}, ID_{\lambda}>$ is sent to the destination proxy, which streams the data to the destination Lambda nodes and remembers the locations in the Lambda pool where the chunks are cached.

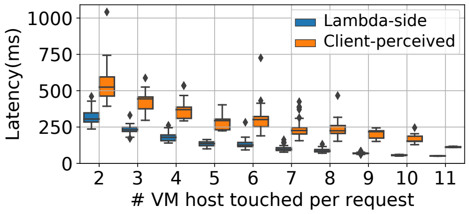


Figure 4: The box-and-whisker plot of latencies as a function of the number of VM hosts touched per request.

The `GET` Path. A `GET` request is first sent to the proxy by using consistent hashing; the proxy then consults its mapping table, which records the chunk to Lambda node association and fetches the object chunks from the associated Lambda nodes (see §3.2). Once the chunks arrive at the client, the client library decodes the chunks, reconstructs the original object, and returns the object to the application.

Eliminating Lambda Contention. Lambda functions are hosted by EC2 Virtual Machines (VMs). A single VM can host one or more functions. AWS seems to provision Lambda functions on the smallest possible number of VMs using a greedy binpacking heuristic [54]. This could cause severe network bandwidth contention if multiple network-intensive Lambda functions get allocated on the same host VM.

We conduct an empirical study to verify this. In our study setup, each Lambda function has 256 MB memory. We use an RS code of $(10+1)$ to split a 100 MB object into 10 data chunks and 1 parity chunk, and place each chunk on a Lambda node randomly selected from a fixed sized Lambda node pool. We measure the latency of `GET` requests by scaling-up the pool from 20 to 200 Lambda nodes. As a result, the number of host VMs that the 11-chunk object spans varies proportionally as the Lambda node pool scales up and down[4]. Figure 4 shows the latency distribution as a function of the number of underlying host VM touched per request. With a larger Lambda node pool (where the request is more likely to be spread across more host VMs), we observe a decreasing

[4] We run command `uname` in Lambda to get the underlying host VM's IP.

trend in the latency on the Lambda-side (the time that each Lambda node spends serving the chunk request) as well as the client-perceived (end-to-end) latencies.

These results stress the need to minimize resource contention among multiple Lambda functions sharing the same VM host. While over-provisioning a large Lambda node pool with many small Lambda functions would help to statistically reduce the chances of Lambda co-location, we find that using relatively bigger Lambda functions largely eliminates Lambda co-location. Lambda's VM hosts have approximately 3 GB memory. As such, if we use Lambda functions with ≥ 1.5 GB memory, every VM host is occupied exclusively by a single Lambda function, assuming InfiniCache's cache pool consists of Lambda functions with the same configuration[5].

3.2 Proxy

Each InfiniCache proxy (Figure 5) is responsible for: (1) managing a pool of Lambda nodes, and (2) streaming data between clients and the Lambda nodes. Each Lambda node proactively establishes a persistent TCP connection with its managing proxy.

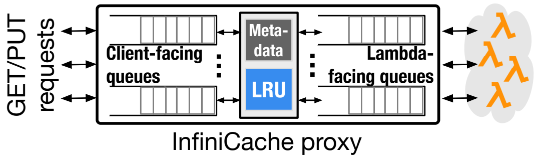


Figure 5: InfiniCache proxy.

Pool Management. Each proxy manages a pool of Lambda nodes, and also maintains the metadata to record the mapping between object chunks and Lambda nodes. To achieve fault tolerance, the proxy also serves as a coordinator to coordinate data migration and delta sync (see detail in §4). Each proxy tracks the memory usage of every Lambda node in the pool. The proxy starts to evict objects as long as there is not enough free memory in the Lambda pool using a CLOCK based [30] LRU policy. The LRU module operates at the object granularity at the proxy. After the eviction process, the proxy updates the mapping metadata, and inserts the new data.

First-d based Parallel I/O. The proxy sends and receives object chunks in parallel by utilizing I/O parallelism to maximize network bandwidth utilization. To mitigate the Lambda straggler problem, the proxy directly streams the first d out of $(d+p)$ encoded object chunks to the client. Though accepting the first-d arrived chunks may likely result in an EC decoding process at the client library, as we show in §5.1, the performance benefit of the optimization outweights the EC decoding overhead with reduced tail latency for `GET` requests.

3.3 Lambda Function Runtime

The Lambda function runtime executes inside each Lambda instance and is designed to manage the cached object chunks in the function's memory. Our Lambda runtime uses several techniques to work around the inherent limitations of AWS Lambda. These techniques, as described below, ensure that caching is robust and cost-effective with negligible overhead.

Memory and Connection Management. The Lambda runtime tracks cached key-value pairs that are sorted with a CLOCK-based priority queue[6] for facilitating the ordered chunk backup process described in §4.2. Since AWS Lambda does not allow inbound TCP or UDP connections, each Lambda runtime establishes a TCP connection with its designated proxy server, the first time it is invoked. A Lambda node gets its proxy's connection information via its invocation parameters. The Lambda runtime then keeps the TCP connection established until reclaimed by the provider.

Anticipatory Billed Duration Control. AWS charges Lambda usage per 100 ms (which we call a *billing cycle*). To maximize the use of each billing cycle and to avoid the overhead of restarting Lambdas, InfiniCache's Lambda runtime uses a timeout scheme to control how long a Lambda function runs. When a Lambda node is invoked by a chunk request, a timer is triggered to limit the function's execution time. The timeout is initially set to expire within the first billing cycle. The runtime employs a simple heuristic to decide whether to extend the timeout window. If no further chunk request arrives within the first billing cycle, the timer expires and returns 2–10 ms (a short time buffer) before the 100 ms window ends. This avoids accidentally executing into the next billing cycle. The buffer time is configurable, and is empirically decided based on the Lambda function's memory capacity. If more than one request can be served within the current billing cycle, the heuristic extends the timeout by one more billing cycle, anticipating more incoming requests.

Preflight Message. While the proxy knows whether a Lambda node is running or has already returned, it does not know when a Lambda node will expire and return. Because of the billed duration control design that was just described, a Lambda node may return at any time. For example, right after the proxy has sent a request but before the request arrives at the Lambda, the Lambda function may expire, resulting in a denial of the request. The proxy could maintain global knowledge about the Lambda node's real-time states by periodically polling the Lambda node. However, this is costly especially if the Lambda pool size scales up to several thousand nodes.

To eliminate such overhead, the proxy issues a preflight message (`PING`) each time a chunk request is forwarded to the Lambda node. Upon receiving the preflight message, the Lambda runtime responds with a `PONG` message, delays the timeout (by extending the timer long enough to serve the incoming request), and when the request has been served, adjusts the timer to align it with the ending of the current billing cycle. To further reduce overhead, the proxy can attach the `PING` message as a parameter of a Lambda function

[5] AWS does not allow sharing Lambda-hosting VMs across tenants [20].

[6] Note that CLOCK is being leveraged for two unrelated purposes: per-proxy for object eviction (§3.2), and per-node for chunk backup ordering.

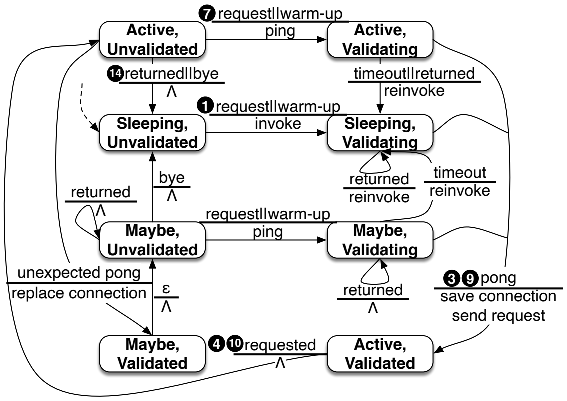


Figure 6: Lambda connection validation process in a proxy.

invocation request, if the Lambda node is in sleep mode (i.e., not running but cached by AWS). Once awoken, the Lambda runtime sends a `PONG` response back to the proxy.

3.4 Reliable Lambda Connections

To maintain reliable network connections between Lambda nodes and their proxy, each proxy lazily validates the status of a Lambda node every time there is a request to send. A proxy maintains three states for each Lambda connection: 1) A `Sleeping` state—a Lambda node that is not actively running; 2) An `Active` state—an actively running Lambda node; 3) A `Maybe` state—during data backup (§4.2) the original Lambda connection might have been temporarily replaced with a new connection connecting the proxy to the destination Lambda node. Figure 6 and Figure 7 depict the state transition graphs for the proxy and the Lambda function runtime, respectively. Note the step numbers show the interactions between a proxy (Figure 6) and a Lambda function (Figure 7).

Connection Lifecycle. Initially, no Lambda node is connected to the proxy. The connection is `(Sleeping,Unvalidated)`. ① When a request comes, or if a pre-warm-up is necessary, ② the proxy invokes a Lambda node. ③ Once the Lambda node is actively running and has successfully connected to its proxy, the Lambda runtime sends a `PONG` message to proxy. Now the connection's state becomes `(Active,Validated)`, and the proxy can start issuing chunk requests. ④ After the proxy sends a chunk request, the connection transits to the state `(Active,Unvalidated)`. Having served the request (⑤ transits from `Active,Idling` to `Active,Serving` while ⑥ transits back), if the proxy forwards the next request continuously, a re-validation of the connection is necessary. ⑦ A `PING` message is sent. ⑧ This time, the Lambda node replies with a `PONG` directly, which ⑨ makes the connection `(Active,Validated)` again, and ⑩ the proxy continues to issue the next chunk request. Note that the Lambda node may return anytime, or a message may timeout. In this case, the proxy re-invokes the Lambda node while marking the connection as `(Sleeping,Validating)`. Having served the request (⑪ transits from `Active,Idling` to

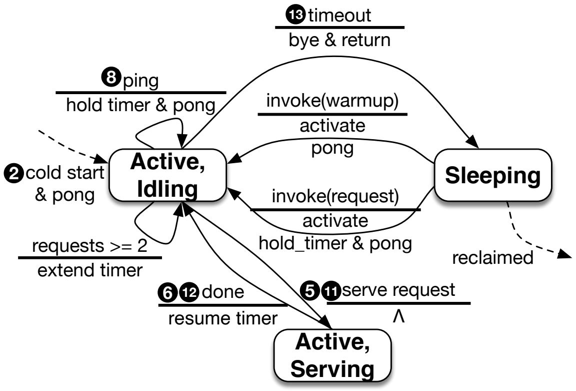


Figure 7: State transitions of a Lambda function runtime.

`Active,Serving` while ⑫ transits back), if no request arrives, the Lambda node ⑬ sends `BYE` to the proxy and returns, and then the proxy ⑭ transits the connection state back to `(Sleeping,Unvalidated)`.

When a connection is in the `Maybe` state, it behaves like an `Active` connection except that the proxy ignores the "return" of the source Lambda node. This does not cause a correctness issue since the source has already been replaced by a new one (i.e., the destination). The connection is marked as `(Sleeping,Unvalidated)` if a `BYE` message is received via the connection.

4 Data Availability and Fault Tolerance

In this section, we conduct a case study with AWS Lambda and describe the approaches INFINICACHE employs for maintaining practical data availability and fault tolerance over a fleet of ephemeral cloud functions with a high churn rate.

4.1 AWS Lambda Properties

While AWS allows function caching to mitigate "cold start" overhead, it does not provide any availability guarantees for the cached function and can reclaim it anytime. Hence, INFINICACHE needs to be robust against frequent failures of cache nodes. To better understand the stateless property of AWS Lambda and its implications on short-term data availability, we conduct an extensive black-box analysis. We analyze reclamation behaviors by quantifying the number of reclaimed Lambda functions in a 24-hour period under different warm-up strategies.

According to a recent study [54], a Lambda function that finishes execution is kept by AWS for at most 27 minutes if that function is not invoked again. A function's lifespan can be extended to hours if that function instance is invoked periodically (i.e., by so-called warm-up operations). The lifespan extension varies according to the warm-up strategy as well as AWS' internal resource management policy.

We deploy a pool of 300–400 Lambda functions with the same memory configuration, and re-invoke each one of them every *N* minute(s). Each function simply returns an ID value that the function computed when it was invoked the first time. If AWS reclaims an already invoked, cached function, a new

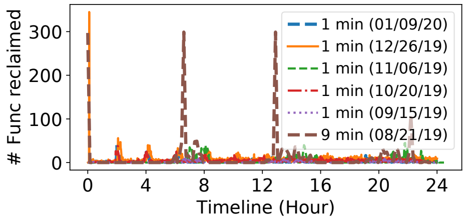


Figure 8: Number of functions being reclaimed over time under various warm-up strategies.

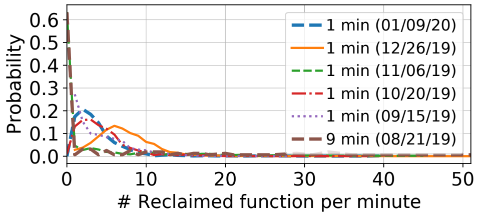


Figure 9: Probability distribution of the number of functions reclaimed per minute on the sampled days.

function instance will be instantiated at the next invocation request and the ID of this function will change. We keep track of the ID to detect whether a function has been reclaimed or not. We evaluate two warm-up strategies: a low warm-up frequency (every 9 minutes) and a relatively high frequency (every 1 minute). We ran each strategy during a span of 6 months (from August 2019 to January 2020), and recorded the number of function reclaiming events.

As shown in Figure 8, for `9 min (08/21/19)`, we observe a large number of function reclaiming events clustered around hour 6, hour 12, and hour 20–22. The number of reclaimed functions spiked roughly every 6 hours and almost all the functions get reclaimed. For `1 min (09/15/19)`, the situation got much better; the peak number of reclaiming events gets reduced to 22, 21, and 16 at hour 6, respectively. Similar trends appeared in November, but got substantially changed in December and January – for example for `1 min (12/26/19)`, instead of spiking every 6 hours, AWS continuously reclaimed Lambda functions with an hourly reclaiming rate of 36. This is possibly due to AWS Lambda's internal policy changes after AWS announced the launch of provisioned concurrency [3] for Lambda on December 03, 2019.

Figure 9 shows the function reclaiming events roughly follow a Zipf distribution for August, September, and November (with different s values), and a Poisson distribution for October, December, and January (with different λ values). With that, we can calculate an approximate range of probabilities of r functions being reclaimed simultaneously in a user-defined interval (§4.3). Motivated by these observations, we argue that with careful design, we can improve the data availability for INFINICACHE.

4.2 Maximizing Data Availability

INFINICACHE adopts three techniques for maximizing data availability: (1) EC is used to enable data recovery for up to p object chunk losses given an RS code $(d+p)$. In the case that there are more than p chunks lost, tenants need to retrieve the data from the backing object store; (2) each Lambda runtime is warmed up after every T_{warm} interval of time; we use a T_{warm} value of of 1 minute as motivated by our observations in §4.1; (3) to further enhance availability, a delta-sync based data backup scheme provides incremental backups every T_{bak} interval. The selection of T_{bak} is a trade-off between availability, runtime overhead, and cost effectiveness: a shorter interval lowers the data loss rate while a longer interval incurs less backup overhead and less cost. In the following, we explain the backup scheme in more detail.

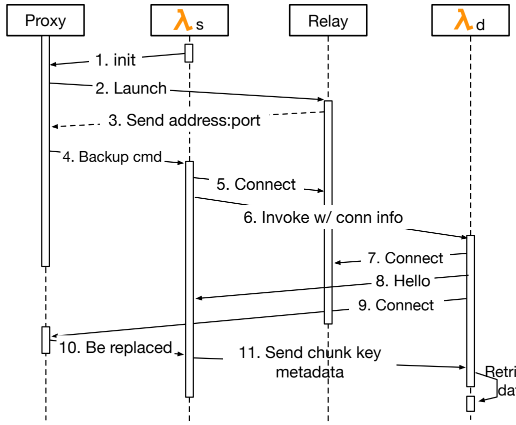


Figure 10: INFINICACHE's backup protocol.

Backup Protocol. INFINICACHE performs periodic delta-sync backups between two peer replicas[7] of the same Lambda function. We choose peer replicas instead of distinct Lambdas to be able to seamlessly failover to one of them in case the other gets reclaimed.

In light of the observations in Figure 8, we design a backup scheme that preserves the following properties: (1) autonomicity—a Lambda node should backup itself with minimum help from the proxy to keep proxy logic simple; (2) high availability—the service provided by the Lambda node should not be interrupted; and (3) low network overhead—large object workloads are network bandwidth sensitive so backups should cause low or no extra network overhead. To this end, we adopt an efficient, Lambda-aware mechanism that performs delta-sync between two peer replicas of the same Lambda function.

The protocol sequence graph is depicted in Figure 10. In Step 1, a Lambda node λ_s, serving as the source cache node, sends an `init-backup` message to its proxy to initialize a backup process every T_{bak}. Acknowledging this message, in Step 2, the proxy launches a new process called *relay* (co-allocated with proxy), which serves to forward TCP packets between λ_s and a destination Lambda node λ_d, i.e., the Lambda node that receives the backup. In Step 3, the relay process sends its own network information (address:port) to the proxy, which issues a `backup` command in Step 4 to λ_s, piggybacked with the relay's connection information.

[7] Concurrent invocations to the same function produce multiple concurrent Lambda instances – this process is called auto-scaling. Here we call each instance of the same function a *peer replica*.

In Step 5, λ_s establishes a TCP connection with the relay and in Step 6 invokes a peer replica instance of the λ_s function, which serves as λ_d; at the same time, λ_s passes the connection information of both the relay and proxy to λ_d as the Lambda invocation parameters. In Step 7, λ_d establishes a TCP connection with the relay. If connected successfully, an indirect network channel is bridged through a relay between λ_s and λ_d. Then λ_d sends a `hello` message to λ_s in Step 8 and connects to the proxy in Step 9.

Upon establishing the connection with λ_d, in Step 10, the proxy disconnects from λ_s, which makes λ_d the only active connection to the data of λ_s. Hence, the proxy forwards all requests to λ_d while λ_d forwards requests to λ_s, if it has not yet received the requested data. To receive data, λ_d sends a `hello` to λ_s in Step 11 and λ_s starts sending metadata (stored chunk keys) in an order from MRU to LRU. Once λ_d has received all the keys, it starts the data migration by retrieving the data associated with the keys from λ_s.

If λ_d receives a `PUT` request during data retrieval and the key is not found, it inserts the new data in its cache and then forwards it to λ_s. If a `GET` request is received for a key that has been retrieved already from λ_s, λ_d directly responds with the requested chunk. Otherwise, λ_d forwards the request to λ_s, responds to the proxy, and then caches the key and the corresponding chunk.

After data retrieval completes, λ_d returns and the connection to the proxy becomes inactive. Hence, the next time the proxy invokes this Lambda function, AWS would launch one of the two, λ_s or λ_d, if they have not been reclaimed yet. As they are now in sync, they can both serve the data. After another interval T_{bak}, the whole backup procedure repeats. λ_d only retrieves the "delta" part of data to reduce overhead.

4.3 Data Availability and Cost Analysis

Availability Analysis. To better understand the data availability of INFINICACHE, we build an analytical model. Assume N_λ is the total number of Lambda nodes. At time T_r, a number r of nodes are found reclaimed. m is the minimum number of chunks that leads to an object loss and n is the number of EC chunks of a object. An object is considered not available if there are at least m chunks lost due to function reclaiming. The probability $P(r)$ that an object is not available (i.e., lost) is formalized as: $P(r) = \sum_{i=m}^{n} p_i$, where:

$$p_i = \frac{C(r,i)C(N_\lambda - r, n-i)}{C(N_\lambda, n)}. \tag{1}$$

Here $C(r,i)$ is the combinations in which r reclaimed Lambda nodes happens to hold i chunks belonging to the same object. $C(N_\lambda - r, n-i)$ is the combinations in which the rest chunks of that object are held in Lambda nodes that have not been reclaimed. $C(N_\lambda, n)$ is the combinations in which all Lambda nodes hold all chunks of an object.

Assuming $p_d(r)$ is the probability distribution of reclaiming r Lambda nodes at T_r, the probability of losing an object P_l is the sum of the probabilities of losing one object when at least m Lambda nodes are reclaimed:

$$P_l = \sum_{r=m}^{N_\lambda} P(r) p_d(r) = \sum_{r=m}^{N_\lambda} \sum_{i=m}^{n} \frac{C(r,i)C(N_\lambda - r, n-i)}{C(N_\lambda, n)} p_d(r). \tag{2}$$

One observation is that $\frac{p_m}{p_{m+1}}$ can be larger than 10. E.g., for a 400-Lambda nodes deployment with $N_\lambda = 400$, an RS code of $(10+2)$, and a warm-up interval of 1 minute, if 12 nodes get reclaimed simultaneously at time T_r, we have $p_3/p_4 = 18.8$ for $r = 12$, and $P(r)$ is only about 5% larger than p_3. So we can simplify the formulation as $P(r) \approx p_m$, thus P_l can be simplified as:

$$P_l \approx \sum_{r=m}^{N_\lambda} \frac{C(r,m)C(N_\lambda - r, n-m)}{C(N_\lambda, n)} p_d(r). \tag{3}$$

In our case study, $N_\lambda = 400$, $n = 12$, $m = 3$, and $T_{warm} = 1$ *min*. With Equation 3 we get $P_l = 0.0039\% \sim 0.11\%$ or an availability $P_a = 99.89\% \sim 99.9961\%$ for 1 minute, and $93.36 \sim 99.76\%$ for 1 hour based on the variable probability distribution of Lambda reclaiming policies we observed over a six-month period (§4.1).

Cost Analysis. To maintain high availability, INFINICACHE employs EC, warm-up, and delta-sync backup, which all incur extra cost. For a better understanding of how these techniques impact total cost, we build an analytical cost model. To simplify our presentation, we do not explicitly express the EC configuration using an RS code $(d+p)$, but rather reflect it in the total number of instances N_λ. The total cost per hour C is therefore composed of (1) serving chunk requests (C_{ser}), (2) warming-up functions (C_w), and (3) backing up data, (C_{bak}). Thus, $C = C_{ser} + C_w + C_{bak}$ Next, we introduce each term respectively.

- *Serving cost C_{ser}.* AWS charges function invocations and function duration. We denote the price per invocation as c_{req} and the duration price of per GB-second as c_d. The function duration is rounded up to the nearest 100 ms, we define a round-up operation $ceil_{100}(.)$. Assume Lambda's memory is M GB, the average hourly request rate is n_{ser}, and the duration of each invocation is t_{ser} ms, we have:

$$C_{ser} = n_{ser} * c_{req} + n_{ser} * ceil_{100}(t_{ser})/1000 * M * c_d. \tag{4}$$

- *Warm-up cost C_w.* The backup frequency $f_w = 60/T_{warm}$. The warm-up duration t_w is typically in the range of a few ms and therefore we have $ceil_{100}(t_w) = 100$ ms. Thus we have:

$$C_w = N_\lambda * f_w * c_{req} + N_\lambda * f_w * 0.1 * M * c_d. \tag{5}$$

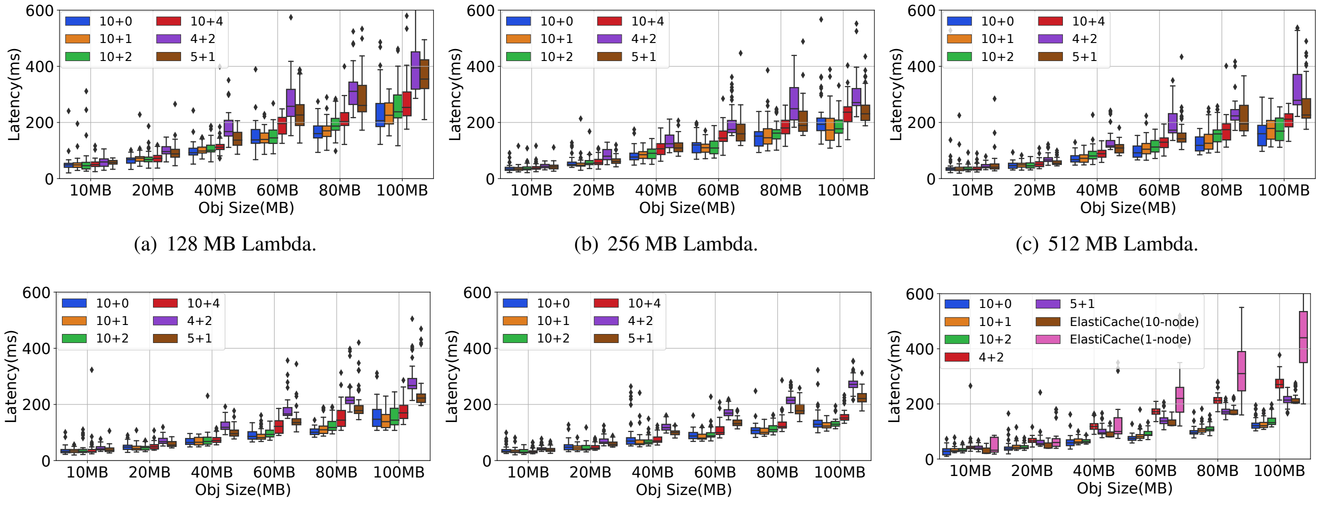
- *Backup cost C_{bak}.* The backup frequency is denoted as $f_{bak} = 60/T_{bak}$. We have:

$$C_{bak} = N_\lambda * f_{bak} * c_{req} + N_\lambda * f_{bak} * t_{bak} * M * c_d. \tag{6}$$

As shown in §5.2, the backup cost is a dominating factor whose proportion increases as more data are being cached.

5 Evaluation

In this section, we evaluate INFINICACHE on AWS Lambda using microbenchmarks and a production workload from the IBM Docker registry [17].



(a) 128 MB Lambda.

(b) 256 MB Lambda.

(c) 512 MB Lambda.

(d) 1024 MB Lambda.

(e) 2048 MB Lambda.

(f) 3008 MB Lambda.

Figure 11: Microbenchmark performance.

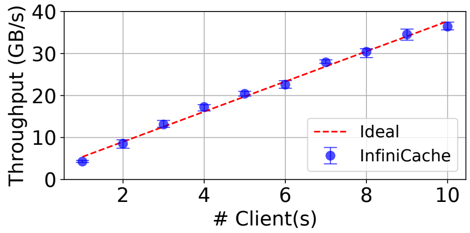


Figure 12: Scalability of INFINICACHE.

Implementation. We have implemented a prototype of INFINICACHE using 5,340 lines of Go (460 LoC for the client library, 3,447 for the proxy, and 1,433 for the Lambda runtime). The EC module of the client library is implemented using the Golang reedsolomon lib [11], which uses Intel's AVX-512 for accelerating EC computation.

Setup. Our experiments use AWS Lambda functions with various configurations. Unless otherwise specified, we deploy the client (with INFINICACHE's client library) and proxy on `c5n.4xlarge` EC2 VM instances. The Lambda functions are in the same Amazon Virtual Private Cloud (VPC) as the EC2 instances and are equipped with a 10 Gbps network connection. The Lambda functions' network bandwidth increases with its memory amount; we observed a throughput of 50–160 MBps (from the smallest memory amount of 128 MB to the largest memory amount of 3008 MB) between a `c5n.4xlarge` EC2 instance and a Lambda function using `iperf3`.

5.1 Microbenchmark Performance

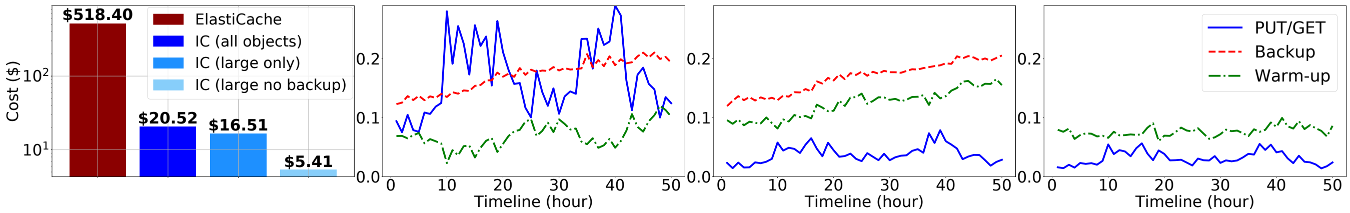
We first evaluate the performance of INFINICACHE under synthetic `GET`-only workloads generated using a simple benchmark tool. With the microbenchmarking tests, we seek to understand how different configuration knobs impact INFINICACHE's performance. The evaluated configuration knobs include: EC RS code (we compare $(10+1)$, $(10+2)$, $(4+2)$, $(5+1)$, with a $(10+0)$ baseline, which directly splits an object into 10 chunks without EC encoding/decoding), object sizes (10–100 MB), and the Lambda function's resource configurations (128–3008 MB).

Figure 11 shows the distributions of end-to-end request latencies seen under different configuration settings. Invoking a warm Lambda function takes about 13 ms on average (with the Go AWS SDK API), which is included in the end-to-end latency results. We observe that the $(10+1)$ code performs best compared to other RS code configurations. This is due to two reasons. First, $(10+1)$ results in a maximum I/O parallelism factor of 10 (first-k parallel I/O is described in §3.2), and second, it keeps the EC decoding overhead at a minimum (the higher the number of parity chunks, the longer it takes for RS to decode). The caveat of using $(10+1)$ is that it trades off fault tolerance for better performance.

Another observation is that the $(10+0)$ case does not seem to lead to a better performance than that of $(10+1)$ and in several cases even sees higher tail latencies. This is due to the fact that $(10+0)$ suffers from Lambda straggler issues, which outweighs the performance gained by fully eliminating the EC decoding overhead. In contrast, $(10+1)$'s first-d approach adds redundancy and this request-level redundancy helps mitigate the impact of stragglers.

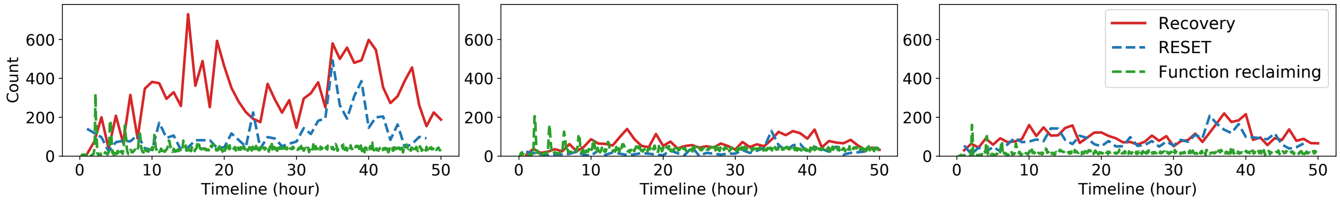
A Lambda function's resource configuration has a great impact on INFINICACHE's latency. For example, $(10+1)$ achieves latencies in the range of 110–290 ms (Figure 11(c)) with 512 MB Lambda functions for objects of 100 MB, whereas with 2048 MB Lambda functions, latencies improve to 100–160 ms (Figure 11(e)). In addition, latency improvement hits a plateau for Lambda functions equipped with more than 1024 MB memory because larger Lambda functions eliminate the network bottleneck for large chunk transfers.

To compare INFINICACHE with an existing solution, we choose ElastiCache (Redis) and deploy it in two modes, a `1-node` deployment using a `cache.r5.8xlarge` instance, and a scale-out `10-node` deployment using `cache.r5.xlarge` in-



(a) Total cost (b) All objects (c) Large obj only (d) Large obj only w/o backup

Figure 13: Total $ cost (a) for ElastiCache and INFINICACHE (IC), and INFINICACHE's hourly cost breakdown under various settings (b)-(e).



(a) All objects (b) Large obj only (c) Large obj only w/o backup

Figure 14: Timeline of INFINICACHE's fault tolerance activities under various workload settings.

Workload	WSS	Thpt	EC	IC	IC w/o backup
All objects	1,169 GB	3,654	67.9%	64.7%	-
Large obj. only	1,036 GB	750	65.9%	63.6%	56.1%

Table 1: Workloads' working set sizes (WSS), throughput (average GETs per hour), and the cache hit ratio achieved by ElastiCache (EC) and INFINICACHE (IC).

stances. As shown in Figure 11(f), INFINICACHE outperforms the 1-node ElastiCache for all object sizes, as Redis is single-threaded and cannot handle concurrent large I/Os as efficiently. For larger object sizes, INFINICACHE with $(10+1)$ and $(10+2)$ consistently achieves lower latencies compared to the 10-node ElastiCache, thanks to INFINICACHE's first-d based data streaming optimization. These results show that INFINICACHE's performance is competitive as an IMOC.

Scalability. In this test, we setup a multi-client deployment to simulate a realistic use case in which a tenant has multiple microservices that concurrently read from and write to INFINICACHE. To do so, we vary the number of clients from 1 to 10. We also deploy a 5-proxy cluster where each proxy manages a 50-node Lambda pool (and each Lambda function has 1024 MB memory). Each client uses consistent hashing to talk to different proxies for shared data access (see Figure 2).

Figure 12 shows the throughput in terms of GB/s. We observe that INFINICACHE's throughput scales linearly as the number of clients increases. Ideally, INFINICACHE can scale linearly as long as more Lambda nodes are available for serving GET requests.

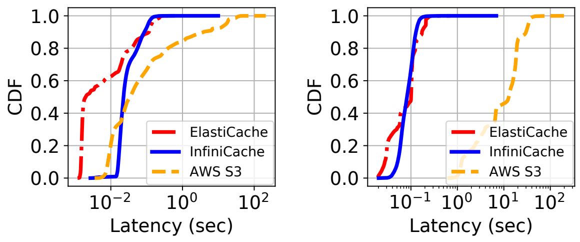
5.2 Production Workload

In this section, we evaluate INFINICACHE using the IBM Docker registry production workload (detailed in §2). The original workload contains a 75-day request trace spanning 7 geographically distributed datacenters. Out of the 7 datacenters, we select Dallas, which features the highest load. We parse the Dallas trace for GET requests that read a blob (i.e., a Docker image layer). We test two workload settings: 1) all objects (including both small and large, with a working set size (WSS) of 1,169 GB as shown in Table 1), and 2) large object only (only including objects larger than 10 MB, with a WSS of 1,036 GB).

We replay the first 50 hours of the Dallas trace in *real time* and skip the largest object which was 8 GB (there was only one object). A GET upon a miss results in a PUT that inserts the object into the cache. INFINICACHE is configured with a pool consisting of 400 1.5 GB Lambda functions, which are managed by one proxy co-located with our trace replayer as the client. We use an EC RS configuration of $(10+2)$ to balance performance with fault tolerance. We select a warm-up interval T_{warm} as 1 minute (due to our study in Figure 8) and a backup interval T_{bak} as 5 minutes (to balance the cost-availability tradeoff). For the large object only workload, we test two INFINICACHE configurations: the default case with backup enabled, and a case with backup disabled (without backup).

Cost Savings. Figure 13(a) shows the accumulated monetary cost of INFINICACHE in comparison with an ElastiCache setup of one cache.r5.24xlarge Redis instance with 635.61 GB memory. By the end of hour 50, ElastiCache costs $518.4, while INFINICACHE with all objects costs $20.52. Caching only large objects bigger than 10 MB leads to a cost of $16.51 for INFINICACHE. INFINICACHE's pay-per-use serverless substrate effectively brings down the total cost by 96.8% with a cost effectiveness improvement of 31×. By disabling the backup option, INFINICACHE further lowers down the cost to $5.41, which is 96× cheaper than ElastiCache. However, the low monetary cost for tenants comes at a price of impacted availability and hit ratio – INFINICACHE without backup sees a lower hit ratio of 56.1% (Table 1) – thus presenting a reasonable tradeoff for tenants to choose.

INFINICACHE's monetary cost is composed of three parts: (1) serving GETs/PUTs, (2) warming-up Lambda functions, and (3) backing up data. Figure 13(b)-(d) details the cost



(a) All objects. (b) Objects > 10 MB.

Figure 15: InfiniCache latencies vs. AWS S3 and ElastiCache.

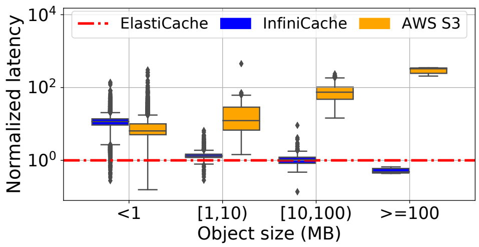


Figure 16: Normalized latencies grouped by object sizes. Each is normalized to that of ElastiCache.

breakdown, further explaining the cost variations of different combinations of workload and InfiniCache settings. In Figure 13(b), we see that about 41% of the total cost is spent on serving data requests under the workload with `all objects`; this is because a significant portion of requests are for small objects. In contrast, for the `large object only` workload shown in Figure 13(c), the backup and warmup cost dominates, occupying around 88.3% of the overall cost. This is because the hourly request rate for `large object only` is significantly lower than that for the `all object` workload. Furthermore, disabling backup leads to a dramatic cost-effectiveness improvement (see Figure 13(d)). The warm-up cost is different between Figure 13(c) and Figure 13(d), because with the backup option enabled, a warm-up invocation may trigger a backup, and thus increase the warm-up duration.

Fault Tolerance. Figure 14 shows InfiniCache's fault tolerance activities for different cases. An object loss (losing all the replicas of more than p chunks) results in a cache miss which triggers a `RESET`; the `RESET` fetches the lost object from a backing store and reinserts it into InfiniCache. We observe that EC-based recovery activities and `RESET`s mostly coincide with the occurrence of request spikes at hour 15–20 and hour 34–42. Under the workload of `all objects` (Figure 14(a)), we see a total of 5,720 `RESET` events. This number is reduced to 1,085 for the `large object only` workload (Figure 14(b)), leading to an availability of 95.4%; as shown in Figure 14(c). InfiniCache `without backup` sees 3,912 `RESET`s, which is 18.6% of 21,022 read hits in total. `RESET`s also result in a lower cache hit ratio for InfiniCache, compared to ElastiCache, as shown in Table 1.

Performance Benefit. We replay the first 50 hours of the Dallas trace against AWS S3 to simulate a deployed Docker registry service using S3 as a backing store. We compare InfiniCache's performance against AWS ElastiCache and S3 seen under the same workload (`all objects`). Figure 15 shows the overall trend of latency distribution, and Figure 16 shows the distribution of the normalized latencies as a function of the object sizes.

We make the following three observations. (1) In Figure 15(b), we see that, compared to S3, InfiniCache achieves superior performance improvement for large objects. For about 60% of all large requests, InfiniCache is able to achieve an improvement of at least 100×. This trend demonstrates the efficacy of InfiniCache in serving as an IMOC in front of a cloud object store. (2) InfiniCache is particularly good at optimizing latencies for large objects. This is evidenced by two facts: i) InfiniCache achieves almost identical performance as ElastiCache for objects sizing from 1–100 MB; and ii) InfiniCache achieves consistently lower latencies than ElastiCache for objects larger than 100 MB (see Figure 16), due to InfiniCache's I/O parallellism. (3) InfiniCache incurs significant overhead for objects smaller than 1 MB (Figure 16), since fetching an object from InfiniCache typically requires to invoke Lambda functions, which takes on average 13 ms and is much slower than directly fetching a small object from ElastiCache.

6 Discussion

In this section, we discuss the limitations and possible future directions of InfiniCache.

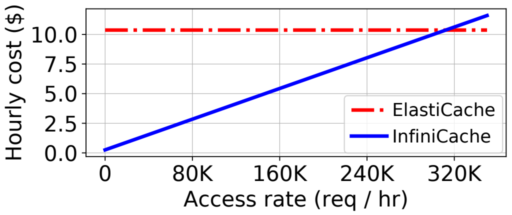


Figure 17: Hourly $ cost (Y-axis) of InfiniCache with 400 1.5 GB Lambdas vs. one `cache.r5.24xlarge` ElastiCache instance, as a function of access rate (X-axis).

Small Object Caching. Small object-intensive memory caching workloads have a high traffic rate, typically ranging from thousands to hundreds of thousands of requests per second [19]. Serverless computing platforms are not cost-effective under such workloads, because the high data traffic will significantly increase the per-invocation cost and completely outweigh the pay-per-use benefit. Figure 17 compares the hourly cost of InfiniCache with ElastiCache, assuming the cost models in §4.3 and configurations in §5.2. The hourly cost increases monotonically with the access rate, and eventually overshoots ElastiCache when the access rate exceeds 312 K requests per hour (86 requests per second).

Porting InfiniCache to Other FaaS Providers. To the best of our knowledge, major serverless computing service providers such as Google Cloud Functions and Microsoft Azure Functions all provide function caching with various lifespans to mitigate the cost of cold startups [54]. Google

Cloud Functions imposes similar constraints on tenants: e.g., banned in-bound TCP connections and limited function CPU/memory resources. The design of INFINICACHE should be portable to other major serverless computing platforms such as Google Cloud Functions, with minor source code modifications to work with Google Cloud's APIs.

Service Provider's Policy Changes. Service providers may change their internal implementations and policies in response to systems like INFINICACHE. On the one hand, *statefulness* is urgently demanded by today's FaaS tenants – providing durable state caching is critical to support a broader range of complex stateful applications [12, 21, 52] such as data analytics [45] and parallel & scientific computing [25, 49]. On the other hand, to strike a balance, providers could introduce new pricing models for *stateful* FaaS applications – tenants can get stateful Lambda functions by paying slightly more than that is charged by a completely stateless one. The new feature recently launched by AWS Lambda, provisioned concurrency [3], pins warm Lambda functions in memory but without any availability guarantee (provisioned Lambdas may get reclaimed, and re-initialized periodically. But the reclamation frequency is low compared to non-provisioned Lambdas), and charges tenants hourly ($0.015 per GB per hour, no matter whether the provisioned functions get invoked), which is similar to EC2 VMs' pricing model. Nonetheless, it opens up research opportunities for new serverless-oriented cloud economics. We leave developing durable storage atop INFINICACHE in support of new stateful serverless applications as considerations for our future work.

Using INFINICACHE as a White-Box Approach. INFINICACHE presents a practical yet effective solution that exploits AWS Lambda as a black-box to achieve cost effectiveness, availability, and performance for cloud tenants. Our findings also imply that modern datacenter management systems could potentially leverage such techniques to provide short-term (e.g., intermediate data) caching for data-intensive applications such as big data analytics. Serving as a white-box solution, datacenter operators can use global knowledge to optimize data availability and locality. We hope future work will build on ours to develop new storage frameworks that can more efficiently utilize ephemeral datacenter resources.

7 Related Work

Cost-Effective Cloud Storage. Considerable prior work [14, 44, 46, 56, 57] has examined ways to minimize the usage cost of cloud storage. SPANStore [56] adopts a hybrid cloud approach by spreading data across multiple cloud service providers and exploits pricing discrepancies across providers. By contrast, INFINICACHE focuses on exploiting stateless cloud function services to achieve pay-per-use storage elasticity with dramatically reduced cost.

Exploiting Spot Cloud Resources. Researchers have explored spot and burstable cloud resources to improve the cost effectiveness of applications such as memory caching [53], IaaS services [50], and batch computing [51]. INFINICACHE differs from them in several aspects: (1) ephemeral cloud functions exhibit significantly higher churn than the more stable spot instances; (2) cloud functions are inherently "serverless" and cannot directly host serverful long-running applications which accept inbound network connections; and (3) spot instances are not automatically cached by providers unlike cloud functions.

In-Memory Key-Value Stores. A large body of research [26, 27, 28, 29, 37, 39, 41, 42, 48, 55] focuses on improving the performance of in-memory key-value stores for small-object intensive workloads. INFINICACHE is specifically designed and optimized for large objects with sizes ranging from MBs to GBs. EC-Cache [47] and SP-Cache [58] are in-memory caches built atop Alluxio [38] to provide large object caching for data-intensive cluster computing workloads. They split the large objects into smaller chunks (EC-Cache leverages erasure coding while SP-Cache directly partitions objects) and perform curated chunk placement to achieve load balancing. The role of erasure coding in INFINICACHE is multi-fold: similar to EC-Cache [47], INFINICACHE leverages erasure coding to mitigate the cloud functions' straggler issue; erasure coding also provides space-efficient fault tolerance against potential loss of cloud functions.

New Applications of Serverless Computing. Researchers have identified new applications for serverless computing in data analytics [25, 35], video processing [18, 32], linear algebra [49], machine learning [24, 34], and software compilation [31]. However, these applications exploit the computing power of serverless platforms to parallelize and accelerate compute-intensive jobs, whereas INFINICACHE presents a completely new use case of cloud function services—implementing a stateful storage service atop stateless cloud functions by exploiting transparent function caching.

8 Conclusion

With web applications becoming increasingly storage-intensive, it is imperative to revisit the design of in-memory object caching in order to efficiently deal with both small and large objects. We have presented a novel in-memory object caching solution that achieves high cost effectiveness and good availability for large object caching by building INFINICACHE on top of a popular serverless computing platform (AWS Lambda). For the first time in the literature, INFINICACHE enables request-driven pay-per-use elasticity at the cloud storage level with a serverless architecture. INFINICACHE does this by synthesizing a series of techniques including erasure coding and a delta-sync-based data backup scheme. Being serverless-aware, INFINICACHE intelligently orchestrates ephemeral cloud functions and improves cost effectiveness by 31× compared to ElastiCache, while maintaining 95.4% availability for each hour time window.

INFINICACHE's source code is available at: https://github.com/mason-leap-lab/InfiniCache.

Acknowledgments

We are grateful to our shepherd, Carl Waldspurger, as well as the anonymous reviewers, for their valuable comments and suggestions that significantly improved the paper. We would also like to thank Benjamin Carver and Richard Carver for their careful proofreading. This work is sponsored in part by NSF under CCF-1919075, CCF-1756013, IIS-1838024, and AWS Cloud Research Grants.

References

[1] 2018 Serverless Community Survey: huge growth in serverless usage. https://serverless.com/blog/2018-serverless-community-survey-huge-growth-usage/.

[2] AWS Lambda. https://aws.amazon.com/lambda/.

[3] AWS Lambda announces Provisioned Concurrency (Posted on: Dec 3, 2019). https://aws.amazon.com/about-aws/whats-new/2019/12/aws-lambda-announces-provisioned-concurrency/.

[4] Azure Functions. https://azure.microsoft.com/en-us/services/functions/.

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

[6] Facebook's Top Open Data Problems. https://research.fb.com/blog/2014/10/facebook-s-top-open-data-problems/.

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

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

[9] Memcached. https://memcached.org/.

[10] Redis. https://redis.io/.

[11] Reed-Solomon Erasure Coding in Go. https://github.com/klauspost/reedsolomon.

[12] The Serverless Supercomputer: Harnessing the power of cloud functions to build a new breed of distributed systems. https://read.acloud.guru/https-medium-com-timawagner-the-serverless-supercomputer-555e93bbfa08.

[13] Varnish HTTP Cache. https://varnish-cache.org/.

[14] Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. Racs: A case for cloud storage diversity. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 229–240, New York, NY, USA, 2010. ACM.

[15] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards high-performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, Boston, MA, 2018. USENIX Association.

[16] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Warfield, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. Pacman: Coordinated memory caching for parallel jobs. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 267–280, San Jose, CA, 2012. USENIX.

[17] 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. Improving docker registry design based on production workload analysis. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 265–278, Oakland, CA, 2018. USENIX Association.

[18] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 263–274, New York, NY, USA, 2018. ACM.

[19] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM.

[20] AWS. Whitepaper: Security overview of AWS Lambda Security and compliance best practices. March 2019.

[21] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. On the faas track: Building stateful distributed applications with serverless architectures. In *Proceedings of the 20th International Middleware Conference*, Middleware '19, pages 41–54, New York, NY, USA, 2019. ACM.

[22] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a needle in haystack: Facebook's photo storage. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 47–60, Berkeley, CA, USA, 2010. USENIX Association.

[23] Daniel S. Berger, Ramesh K. Sitaraman, and Mor Harchol-Balter. Adaptsize: Orchestrating the hot object memory cache in a content delivery network. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 483–498, Boston, MA, 2017. USENIX Association.

[24] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew

Zhang, and Randy Katz. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, pages 13–24, New York, NY, USA, 2019. ACM.

[25] Benjamin Carver, Jingyuan Zhang, Ao Wang, and Yue Cheng. In search of a fast and efficient serverless dag engine. In *4th International Parallel Data Systems Workshop (PDSW 2019)*, 2019.

[26] Yue Cheng, Aayush Gupta, and Ali R. Butt. An in-memory object caching framework with adaptive load balancing. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 4:1–4:16, New York, NY, USA, 2015. ACM.

[27] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 379–392, Santa Clara, CA, March 2016. USENIX Association.

[28] Diego Didona and Willy Zwaenepoel. Size-aware sharding for improving tail latencies in in-memory key-value stores. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 79–94, Boston, MA, February 2019. USENIX Association.

[29] Bin Fan, David G. Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 371–384, Lombard, IL, 2013. USENIX.

[30] Fernando J Corbato. A paging experiment with the multics system. Technical Report.

[31] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 475–488, Renton, WA, July 2019. USENIX Association.

[32] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, 2017. USENIX Association.

[33] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 3–18, New York, NY, USA, 2019. ACM.

[34] V. Ishakian, V. Muthusamy, and A. Slominski. Serving deep learning models in a serverless platform. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 257–262, April 2018.

[35] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 445–451, New York, NY, USA, 2017. ACM.

[36] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. Technical Report UCB/EECS-2019-3, EECS Department, University of California, Berkeley, Feb 2019.

[37] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 137–152, New York, NY, USA, 2017. ACM.

[38] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 6:1–6:15, New York, NY, USA, 2014. ACM.

[39] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, 2014. USENIX Association.

[40] M. Littley, A. Anwar, H. Fayyaz, Z. Fayyaz, V. Tarasov, L. Rupprecht, D. Skourtis, M. Mohamed, H. Ludwig, Y. Cheng, and A. R. Butt. Bolt: Towards a scalable docker registry via hyperconvergence. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 358–366, July 2019.

[41] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li,

Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 143–157, Boston, MA, February 2019. USENIX Association.

[42] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 183–196, New York, NY, USA, 2012. ACM.

[43] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, Boston, MA, 2018. USENIX Association.

[44] T. G. Papaioannou, N. Bonvin, and K. Aberer. 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*, pages 1–10, Nov 2012.

[45] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 193–206, Boston, MA, 2019. USENIX Association.

[46] Krishna P.N. Puttaswamy, Thyaga Nandagopal, and Murali Kodialam. Frugal storage for cloud file systems. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 71–84, New York, NY, USA, 2012. ACM.

[47] K. V. Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 401–417, Savannah, GA, November 2016. USENIX Association.

[48] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. Log-structured memory for dram-based storage. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 1–16, Santa Clara, CA, February 2014. USENIX Association.

[49] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. numpywren: serverless linear algebra. *arXiv preprint arXiv:1810.09679*, 2018.

[50] Prateek Sharma, Stephen Lee, Tian Guo, David Irwin, and Prashant Shenoy. Spotcheck: Designing a derivative iaas cloud on the spot market. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 16:1–16:15, New York, NY, USA, 2015. ACM.

[51] Supreeth Subramanya, Tian Guo, Prateek Sharma, David Irwin, and Prashant Shenoy. Spoton: A batch computing service for the spot market. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 329–341, New York, NY, USA, 2015. ACM.

[52] Tim Wagner. Serverless State: What comes next for serverless. In *ServerlessConf NYC'19*.

[53] Cheng Wang, Bhuvan Urgaonkar, Aayush Gupta, George Kesidis, and Qianlin Liang. Exploiting spot and burstable instances for improving the cost-efficacy of in-memory caches on the public cloud. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 620–634, New York, NY, USA, 2017. ACM.

[54] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, Boston, MA, 2018. USENIX Association.

[55] Xingbo Wu, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, and Song Jiang. Zexpander: A key-value cache with both high performance and fewer misses. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[56] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V. Madhyastha. Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 292–308, New York, NY, USA, 2013. ACM.

[57] Zhe Wu, Curtis Yu, and Harsha V. Madhyastha. Costlo: Cost-effective redundancy for lower latency variance on cloud storage services. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 543–557, Oakland, CA, May 2015. USENIX Association.

[58] Yinghao Yu, Renfei Huang, Wei Wang, Jun Zhang, and Khaled Ben Letaief. Sp-cache: Load-balanced, redundancy-free cluster caching with selective partition. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18. IEEE Press, 2018.