Glider: Serverless Ephemeral Stateful Near-Data Computation

Daniel Barcelona-Pons
daniel.barcelona@urv.cat
Universitat Rovira i Virgili
Tarragona, Spain

Pedro García-López
pedro.garcia@urv.cat
Universitat Rovira i Virgili
Tarragona, Spain

Bernard Metzler
bmt@zurich.ibm.com
IBM Research Europe
Zurich, Switzerland

ABSTRACT

Serverless data analytics generate a large amount of intermediate data during computation stages. However, serverless functions, which are short-lived and lack direct communication, face significant challenges in managing this data effectively. The traditional approach of using object storage to carry the data proves to be slow and costly, as it involves constant movement of data back and forth. Although specialized ephemeral storage solutions have been developed to address this issue, they fail to tackle the fundamental challenge of minimizing data movements. This work focuses on incorporating near-data computation into an ephemeral storage system to reduce the volume of transferred data in serverless analytics. We present Glider with the aim to enhance communication between serverless compute stages, allowing data to smoothly "glide" through the processing pipeline instead of bouncing between different services. Glider achieves this by leveraging stateful near-data execution of complex data-bound operations and an efficient I/O streaming interface. Under evaluation, it reduces data transfers by up to 99.7%, improves storage utilization by up to 99.8%, and enhances performance by up to 2.7×. In sum, Glider improves serverless data analytics by optimizing data movement, streamlining processing, and avoiding redundant transfers.

KEYWORDS

Serverless, cloud, ephemeral, near-data, stateful, intermediate data

ACM Reference Format:
Daniel Barcelona-Pons, Pedro García-López, and Bernard Metzler. 2023. Glider: Serverless Ephemeral Stateful Near-Data Computation. In *24th International Middleware Conference (Middleware '23), December 11–15, 2023, Bologna, Italy*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3590140.3629119

1 INTRODUCTION

Serverless analytics rely on fleets of functions (FaaS) to handle data processing workloads [16, 17, 22]. However, these functions are stateless, short-lived, and cannot communicate directly with each other. To coordinate multi-stage jobs, FaaS must resort to external solutions to handle large volumes of *intermediate data*. This is a well-known issue [20, 27, 33, 37] product of a data-shipping architecture, which causes heavy data traffic, strains the network, and often becomes a bottleneck, especially on the limited bandwidth of FaaS [26, 44]. Moreover, due to the limited resources of functions, jobs may require further partitioning and additional stages, increasing intermediate data and challenging its effective management.

A solution to the problem of serverless data-shipping is still not available on cloud platforms. Although optimizations of the storage technology [27, 33, 37] (such as faster in-memory stores, smart partitioning, or specialized ephemeral stores) have been explored, they fail to address the fundamental issue, namely the long-distance transfers of large data. Some approaches have explored data locality [2, 40] and various forms of caching data within the FaaS platform [31, 35, 41, 47]. However, these approaches require modifications to the platform (which hinders the qualities of FaaS) and are limited to handling small amounts of data.

Near-data processing (NDP) is a common solution for reducing data movement by executing logic on hardware accelerators embedded in disks [1] or NICs [13, 14]. On a systems level, active storage addresses data-shipping in cluster computing by performing operations on object storage nodes that intercept data accesses [30], achieving impressive reductions in data ingestion [19]. However, this approach is not feasible in the cloud due to resource contention and isolation challenges in multi-tenant setups [11, 36].

To overcome the challenges of inter-stage communication in serverless analytics, we introduce Glider, a novel ephemeral storage service for the cloud that incorporates *serverless ephemeral near-data computation*. Glider is designed as a companion to FaaS, enabling complex data connections between functions engaged in big data analytics. To communicate intermediate (ephemeral) data, it makes sense to also employ ephemeral operations that transform it along the way. To the best of our knowledge, Glider is the first solution to integrate ephemeral computation into ephemeral storage. To achieve this, Glider introduces *storage actions*, which encapsulate stateful computation as full-fledge storage elements (i.e., at the level of files in a file system). This enables them to handle complex data-bound operations like aggregates or shuffles. Additionally, actions provide an I/O streaming interface crucial for data streamlining and efficient resource utilization in serverless functions. In essence, Glider not only enables data streaming between function groups or stages but also allows for in-line data transformation with complex arbitrary patterns, eliminating the need for redundant data transfers. We evaluate Glider on various applications, including a large-scale genomics job with over 700 serverless functions. In them, Glider reduces intermediate data transfers by up to 99.7%, storage accesses by up to 50%, and overall storage utilization by up to 99.8%. Glider is open-source and available online [18].

In summary, this work makes the following contributions:

- We present Glider, the first system to combine ephemeral computation and ephemeral storage for serverless analytics.
- We propose storage actions, an encapsulation for near-data ephemeral stateful computation.

Middleware '23, December 11–15, 2023, Bologna, Italy

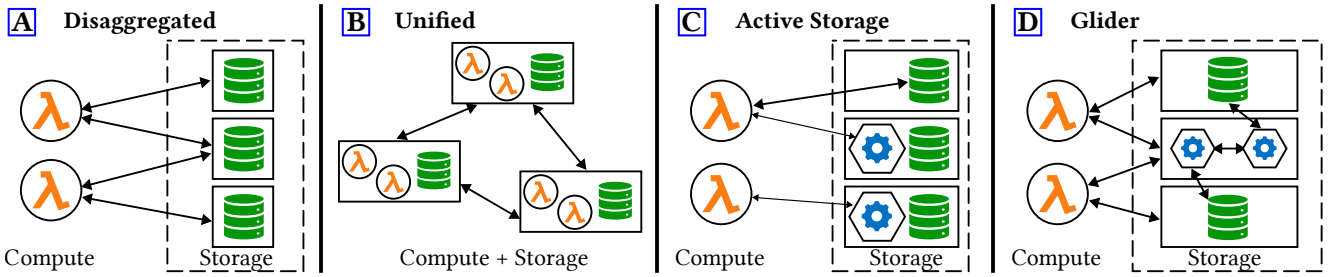

Figure 1: Architectural approaches to data-shipping. See §8 for an extended discussion.

- We implement Glider and show reductions in intermediate data transfers (99.7%), storage accesses (50%) and utilization (99.8%) for serverless data processing jobs.

2 BACKGROUND AND MOTIVATION

2.1 Serverless and temporary data

Most data transferred in analytics workloads is temporary. That is, the intra-job data created, handled, or consumed during processing are, thus, not relevant after the job completes. In other words, temporary data is short-lived and easy to regenerate. Efficiently handling temporary data requires a dedicated solution that correctly handles its peculiarities. For instance, the variability in size and access patterns need a versatile system that embraces it. This kind of specialized stores exist in cluster computing [42].

In serverless analytics, handling temporary data is challenging. Cluster solutions may keep intermediate data in memory across compute stages or directly transfer it between workers. Unfortunately, FaaS functions preclude these methods due to their transient nature and inability to communicate directly. Instead, functions are forced to ship data through remote storage.

The common choice to relay this data is object storage, which is too slow for some types of temporary data [23, 42]. Faster storage solutions are not available as cloud-managed serverless services. In fact, achieving serverless properties for high-performance (in-memory) stores is not trivial. One recurrent problem in this endeavor is fine-grained elasticity of in-memory stateful elements; an open research topic [25, 27, 41]. Nonetheless, serverless data processing poses important challenges that cannot be solved with just faster storage. The fundamental issue is the constant movement of temporary data between FaaS and the storage service, which is a by-product of a data-shipping architecture.

2.2 Data-shipping in serverless

Data-shipping (repetitive transfers between compute and storage) has presence in the literature due to the pressure it exerts on the network. The peculiarities of FaaS (limited memory and compute power, short lifespan, small bandwidth, etc.) increase data transfers and their cost [20], resulting in sub-par performance for applications. Figure 1 illustrates different approaches to this problem.

First, several works [22, 33, 37] optimize data dependencies with faster stores and intelligent data partitioning models to achieve the best cost to performance tradeoff (A in Figure 1). Still, they fail to reduce the amount of transfers and thus suffer the performance and financial costs of shipping lots of data.

In light of this, other projects decide to fully combine a FaaS platform and a store (B in Figure 1). Works range from functions sharing memory [2, 40] to FaaS platforms implemented on top of cache stores [31, 35, 41, 47]. Unfortunately, this model does not work well for data processing jobs. Literature shows that compute and storage tiers must be disaggregated and isolated to correctly scale them to their distinct demands and avoid contention [11, 36].

A solution to data-shipping is to ship code to data instead: offload tasks to the storage system to minimize the amount of data transferred between clusters (C in Figure 1). Active storage [19, 30, 36] exhibits impressive improvements for cluster data ingestion, but it only supports object stores, which are unfit for ephemeral data (§2.1). Also, it is not applied in multi-tenant settings due to the difficulty to scale compute and durable storage jointly and avoid performance interference [11, 36]. Therefore, the solutions are limited to a layer of stateless data access interception. Ephemeral data stores, in contrast, are more elastic since they avoid data redistribution when scaling the system thanks to its transient nature [42]. This enables currently unexplored ways to integrate computation within a storage system that are more akin to serverless, such as having both elements independently grow or shrink to the demand of multiple jobs or tenants and linking them in a single storage namespace (beyond interception). Further, it creates a unique potential for new types of near-data computation (such as stateful operations) and new challenges that we discuss shortly (§2.4).

2.3 An overview of Glider

By combining past knowledge on temporary data, its presence in serverless analytics, and the problem of data-shipping, this paper presents Glider. Glider explores a new approach to face data-shipping of serverless intermediate data. We observe that the FaaS platform (where the heavy computation occurs) must remain unaltered to conserve its properties (e.g., serverless functions must be able to spawn and expire without depending on stateful components) [22, 23]. Hence, to connect compute stages requires specialized remote storage. Our key principle is that, consequently, to reduce data movement, we must ship code to storage instead.

Glider defines a novel service model for ephemeral computational storage. Understanding that temporary data needs a specialized ephemeral store [27], Glider expands this idea by including ephemeral computation within it (Figure 1 D). Data and compute populate the same system and may enjoy better links or co-location,

but they remain isolated and are managed independently. Since ephemeral data and computation are short-lived and easily regenerated, it is easy to allocate and remove them from the system, which allows system elasticity by the join and leave of resources without redistribution [27]. Near-data computation allows to run data-bound operations with low access overhead and to minimize both, the necessary connections from serverless functions to storage and the amount of data they transmit. Thanks to this, our solution lets the data "glide" through the computation pipeline, meaning that data is transformed (advances processing) with every transfer, instead of jumping back and forth between services.

2.4 Challenges

We study the following challenges in facing data-shipping of serverless intermediate data with an ephemeral computational storage.

Synergize compute and storage. Running computation within the storage tier is tricky, especially in a multi-tenant service. If the compute operations grow uncontrolled on storage resources, they create contention and highly impact performance for basic storage operations and other users [11, 36]. A closed library of storage operations may enable such control, but pushing meaningful operations to storage requires allowing users to define them in arbitrary code. Additionally, in a serverless system, the expected fine-grained elasticity restricts how resources may be managed.

Stateful computation. Serverless analytics stages communicate in complex patterns that suppose huge data transfers with current solutions. Active storage intercepts data accesses with processors that are anonymous and stateless. This is not enough. First, multiple accesses to the same data trigger duplicate computations and waste resources. But more importantly, stateless processors fail to reduce data transfer in complex patterns such as aggregations or shuffles. A simple word count requires multiple tasks to send partial counts to storage and another stage that reads them all to complete the counting. All data is transferred twice without alteration. Stateful processors solve these patterns easily by allowing to connect to the same processor multiple times. For instance, a processor may hold a counter that aggregates the partial word counts of multiple tasks and thus provide the final counting with a single data transfer.

Large intermediate data. Intermediate data in data processing workloads may be very large, such as when shuffling in MapReduce jobs, and thus very hard to handle correctly. For instance, traditional in-memory caches are not prepared to support large amounts of data. For this reason, there exist specialized systems to support temporary data efficiently [42]. Again, this is especially relevant for serverless computing. On one hand, resource limitations prohibit workers to load big files entirely. On the other hand, their often-limited network burdens them with long transmission times. Similarly, ephemeral near-data computation must also handle large data effectively to remain lightweight and easy to manage.

Relevant considerations. For temporary data, usual storage features such as durability and fault-tolerance are not a priority [42]. Durability is mostly irrelevant for short-lived data. Fault-tolerance is often implemented at the level of the compute framework (e.g., a function orchestrator in serverless). Thus, although generally useful for storage, its management hindrances and overheads are hard to justify for ephemeral data.

Two projects in the literature propose serverless ephemeral storage systems. Pocket [27] builds automatic scaling and multi-tenant management into an ephemeral storage solution with serverless in mind. Its evaluation shows huge improvements for storing intermediate data against existing storage solutions in the cloud. Jiffy [25] designs a scalable remote memory for serverless functions to use as a shared space that grows and shrinks on demand. Glider is orthogonal to these projects and it may benefit from their contributions on dynamic elasticity and multi-tenancy. However, they do not address the problems derived from data-shipping. Therefore, here we focus on the novel challenges presented before.

3 GLIDER

We present Glider, a novel serverless service model for ephemeral computational storage. In essence, Glider is a storage system for large temporary data with capacity for ephemeral computation within it. The goal is to exploit the synergy of both elements together to counteract the issues of data-shipping in serverless data processing workloads. To that end, Glider allows to reduce (i) the amount of data transferred (bytes through network), (ii) the number of transfers (storage accesses), and (iii) storage utilization (stored data). To wit, Glider is conceived as a companion to serverless computing (FaaS) services that allows data to smoothly "glide" through the different computation stages. Specifically, its near-data computation allows to streamline data processing between the compute and storage systems and, thus, avoid transferring back and forth the same data multiple times.

As described previously, the challenges of Glider include an effective integration of compute and storage, the necessity of ephemeral computation to be flexible and stateful, and the imperative of a practical interface to handle large data. Therefore, Glider defines **storage actions** to encapsulate computation with three key properties: (i) they are integrated as addressable storage elements in the namespace; (ii) they are defined as arbitrary objects with stateful logic; and (iii) they offer a common streaming I/O interface.

Near-data computation. Glider organizes the ephemeral storage with namespaces, a logical structure of storage elements (e.g., files or directories). Storage actions are integrated as a type of storage element. In particular, actions have a name or identifier, and applications use it to directly read or write on an action, or to organize them in the namespace. Actions are automatically managed and distributed by the system in the same way as other elements. This simplifies its usage, interaction, and management.

Glider solves resource interference with *storage spaces.* A storage space is a set of isolated resources that contributes a certain amount and type of storage capacity to a specific namespace. Some storage spaces contribute data capacity, while others compute capacity. This brings two important benefits. First, the capacity of each namespace is dynamic and adapts to demand through the joining and leaving of storage spaces, providing fine-grained resource scaling. Second, compute and data elements coexist for improved synergy while isolated to improve management and avoid contention.

Arbitrary stateful code. Actions trigger a computation whenever they are accessed. This computation is defined by arbitrary, user-provided code that processes the data in and out. Thanks to being part of the storage namespace, actions are stateful and may be directly addressed multiple times. In consequence, an operation on an action may depend on the results of a previous one, which makes them great for aggregating or caching data, among many other uses. Like any other element in Glider, actions are ephemeral, as they represent intermediate data connections, and should not hold long term data.

I/O streams. Consuming and producing data in small chunks is key to process large data in small-sized workers like serverless functions. Therefore, Glider defines a stream-based I/O interface for all storage elements, including actions, that workers can handle with a small memory footprint. For actions, this adds extra benefits. A worker-action stream allows parallel processing at both ends in a simple and straightforward way to improve the overall performance and storage utilization (e.g., with filters or aggregations).[1]

3.1 What code should we ship?

When offloading computation to storage, which tasks are "appropriate"? Essentially, the objective is to reduce the amount of network data transfers. Also, we may argue that offloaded tasks should be lightweight and transient to ease integration into storage. With this in mind, we identify two types of computation: compute-bound and data-bound tasks. Compute-bound tasks truly shine in the dedicated compute tier, where they may be freely scaled, and would suffer from being attached to storage. Examples of these tasks are those that perform number crunching and heavy mathematical computations like matrix operations or ML training. In contrast, data-bound tasks are the ones that benefit most from near data computation and have a direct effect on data transfers. These are data management tasks. Therefore, it makes sense to offload data-bound computations to storage, while we should never do so with compute-bound tasks. In the case of data-bound tasks, traditional active storage only supports stateless tasks such as data filtering, transformation, or simple queries. Storage actions are stateful and may host complex data-bound operations such as aggregations, data shuffling, indexing, or interactive queries.

3.2 Using storage actions

The Glider model[2] is, in essence, that of a common cloud-managed storage service. Users manage their storage namespace, where they add or remove elements in a structure. Applications only interact with storage elements, which have typical access operations for reading, writing, or removing them (e.g., CRUD). Storage spaces are managed by the service, never by users. However, a cloud vendor must put some limitations to efficiently manage them. To this end, a service may allow users to configure some parameters (e.g., size, compute power, timeout) to adjust spaces and the service behavior for a specific namespace.

To use storage actions, programmers should first code its logic by following an interface. The interface defines a set of functions that the developer may implement as desired. Each of these code elements will run for different operations performed on the action. The main operations are reading and writing, for which the application utilizes I/O streams. Action definitions (the code) must be deployed into the service. This procedure is similar to deploying a function in a FaaS platform. Users upload a package and register each action so that they may reference them later for instantiation. The service may also allow certain action configuration parameters. Actions are instantiated as any other storage element. Applications must provide an identifier within the storage namespace and a reference to the action definition to instantiate. Likewise, they may get references to existing actions through their identifier. The reference may be used to delete the action or perform data operations by obtaining an I/O stream.

4 SYSTEM DESIGN

To evaluate the Glider model, we first define a concrete system design by extending a multi-tiered, high-performance ephemeral storage architecture: the NodeKernel [42]. We choose NodeKernel because it is extensible and specialized for ephemeral data. Its design may easily be expanded for managing multi-tenant deployments and allows to grow and handle resources elastically at fine granularity [27]. This makes it a great substrate for a serverless service.

With Glider, we integrate into NodeKernel the concept of storage actions (i.e., ephemeral near-data computation, arbitrary stateful logic, and I/O streams), which add multiple desireable benefits for serverless data processing applications. Near-data computation reduces the problems derived from a data-shipping architecture. Its statefulness allows to redistribute computation for more efficient pipelines. And the I/O interface allows to process large data with modest-sized workers and parallelize execution.

4.1 NodeKernel in brief

NodeKernel [42] is a state-of-the-art storage architecture specialized on temporary data in data processing workloads. We summarize its key components as a baseline for Glider's contributions:

Storage semantics. The high-level storage semantics are provided as data "nodes" and organized in a hierarchical namespace. Nodes are defined as specialized data types implementing a common interface with basic operations to handle data (e.g., `read` and `write`) or structure (e.g., `getPath` and `addChild`). Each node may hold data of arbitrary size. The general organization is managed by a shared storage kernel, which is responsible for allocating storage resources for nodes, handling the hierarchical namespace, and implementing data access operations. Applications connect to the kernel to create, look up, remove, and attach or fetch data to/from nodes. To identify nodes within the storage hierarchy, they are given path names similarly to a file system. Data nodes are extensible and may provide different specialized data access semantics.[3]

System architecture. NodeKernel manages data in a set of metadata and storage servers. Internally, data is handled in blocks. A block is a fixed sequence of bytes residing in a storage server. The

[1] The cloud has adopted this idea recently with a similar objective [4, 5].
[2] Here we provide an overview of how to interact with the Glider service model. Details of our concrete implementation are provided later (§6).
[3] NodeKernel defines five custom node types that semantically represent data (`File` and `KeyValue`), containers (`Directory` and `Table`), or specialized structures (`Bag`).

nodes in the storage hierarchy present their data as a byte stream that abstracts a sequence of blocks. The metadata servers[4] administer the hierarchical namespace and the fleet of blocks. The storage servers allocate storage blocks and register them on a metadata server. The metadata servers maintain a list of free and used blocks (and their mapping to storage servers) and assign them to nodes as needed. This way, data is distributed across the cluster. To perform data operations, clients first contact a metadata server, and it replies with the location of the storage block(s) affected by the operation. The client then uses this information to perform the operation on the appropriate storage server(s). Structure operations are directly executed at the metadata server.

To accommodate for different types and sizes of data, NodeKernel supports a tiered storage design. Each storage server implements a type of storage (such as DRAM, NVMe, or HDD) that it uses to allocate its blocks. Storage servers are deployed as logical entities encapsulating a set of resources, which allows them to exploit the different hardware in the same physical or virtual machine but manage them separately. When a storage server joins the system, it is registered into one, and only one, storage class. Storage classes allow to group storage servers and create relations between them as users find appropriate. Typically, a storage class could represent a concrete technology, so that we may have a preferred DRAM tier that falls back to an NVMe tier when full. This freedom is key for temporary data that may have disperse requirements.

4.2 Storage actions

Glider adds *storage actions* to NodeKernel to achieve ephemeral near-data computation. A storage action is a new storage element that encapsulates a user-provided stateful computation with a stream I/O interface. Towards this integration, we first identify storage spaces within the NodeKernel architecture. Then, we add two components to support actions: a new node type and a new storage server type. This also requires new logic in the metadata server and the necessary client tools to create and operate actions.

Near-data computation. Glider leverages storage servers to create storage spaces, our abstraction for a particular set of contributing storage or compute resources. We run storage servers within containers to provide the necessary isolation and elasticity to harbor compute and storage elements in harmony within the same system. With this, each element enjoys isolated resources to avoid contention and performance degradation. But at the same time, they are managed equally within the system for improved synergy. The isolation and transient model of storage servers has already been exploited as a substrate for a multi-tenant storage solution and to elastically scale storage resources on demand in a serverless flavor [25, 27] (orthogonal to Glider). Like these projects, Glider inherits from NodeKernel the view that transient data can be easily regenerated in case of a server failure (§2.4). The same applies to actions and their state. If needed, failure handling and consistency mechanisms may be applied orthogonally (beyond this paper), such as action checkpointing. If a stateful action interacts with other data, checkpointing should also consider this dependency to correctly restore them (e.g., synchronizing persistence of both elements). Users may develop their actions with such mechanisms as required by their applications in expense of performance.

[4]Metadata servers may distribute their work by partitioning the namespaces, allowing to scale the system if needed.

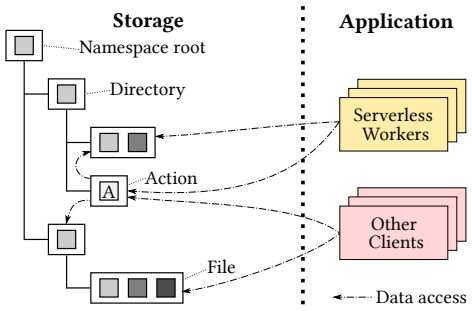

Figure 2: Glider's storage semantics.

Our design focuses on the integration of compute elements and omits storage space management. We leave this topic for future research. Ephemeral storage and compute elements have requirements that vary between applications. Also, a commercial service must put limits to resource occupancy to effectively manage the pool of resources. The service should thus determine the granularity of storage spaces and the mechanisms to scale them through evaluating a tradeoff between versatility and resource utilization. Some decisions include configurable capacity, CPU time limitation, or timeouts for storage spaces.

The action node type. From a high-level, logical view, actions are a new node type in the storage semantics, implementing the common interface. As such, they may be organized in the hierarchical namespace and access operations are done by obtaining I/O streams to send and retrieve data. Figure 2 shows a view of Glider's storage semantics and this integration. Action nodes do not simply store and then return the data like the other node types, but they house an in-memory object (OOP) that may process the data with arbitrary stateful logic. Each operation on an action node triggers the execution of one of the object methods. E.g., an aggregation during a write or an infinite data generation in a read.

Storage blocks for actions. For other node types, the metadata server assigns them storage blocks in a chain as their attached data grows. Actions, in contrast, represent computational entities. As such, they operate on all their data in the same place and may not be split into multiple storage blocks. Therefore, actions are allocated in a single block.

The active storage server. Glider adds a new type of storage server: the active storage server. Figure 3 draws Glider's data management architecture, including active servers. Like the others, active servers are encapsulations of storage space. They register themselves with the metadata servers and contribute blocks for a namespace. But there are two key differences: (i) they are grouped into an active storage class, and (ii) their storage blocks are, in fact, action slots. The dedicated storage class allows the storage kernel to allocate action nodes only on active servers. Action slots facilitate managing the size of actions in terms of resources. Similar to other types of storage servers that encapsulate a space for data (a range of bytes in memory or disk), active servers encapsulate a dedicated set of

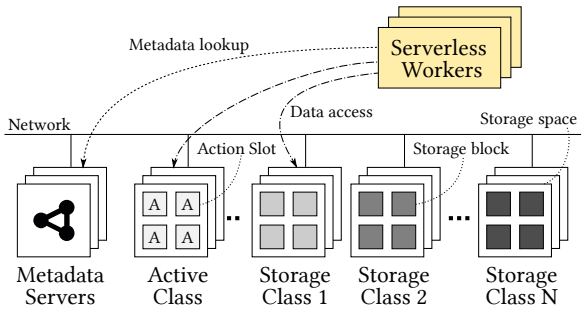

Figure 3: Glider's storage data management architecture.

resources (memory and compute power) for running actions. Thus, the size of an active server and the number of slots it registers determine the capacity and performance of its actions. Since the performance requirements of actions depend on the application, it is up to the developer to configure these properties; similar to configuring function memory size in a FaaS platform.

Distributing actions. Action distribution across storage servers is an interesting topic that opens lines of research on its own. Our model does not define a concrete mechanism as they can be applied to Glider orthogonally. The chosen method may depend on the service resource management, or be open to the requirements of a specific application. A service may put effort in co-locating actions and other storage elements to exploit locality. But others may invest in connecting remote storage spaces with a high-performance network [24] to enable great performance even without co-location and increase elasticity. There is extensive work in the literature about sophisticated scheduling mechanisms for storage and compute elements with dependencies, e.g., gang scheduling [21], or affinity languages [38], to name some. Following NodeKernel [42], we (§5) distribute data and actions uniformly across available servers. Glider embraces the same policy as Pocket [27] to avoid redistribution when scaling the system. Since ephemeral data and actions in Glider are short-lived, migrating them has high overhead. Instead, thanks to their isolation, new storage spaces may be added freely and easily removed when their ephemeral contents expire.

Accessing actions. Actions in Glider are storage elements at the level of, e.g., files. This is very different from other computational storage solutions [13, 30, 36] whose computations only intercept accesses to other elements. From the outside, actions contain data that may be accessed through I/O streams. Each stream opened to an action, however, triggers the execution of one of its methods on the server. The method handles the other end of the stream, so that client and action may pass a flow of data.

Like with other node types, clients first contact a metadata server to obtain the location of the action. Then, clients communicate directly with the corresponding storage server to perform operations. Since actions only occupy a single block (an action slot), each client only needs to contact the metadata server once.

The action I/O streams work similar to reading or writing data in the other types. Long transfers are split into multiple chunks of data, and each chunk is sent in a basic remote data operation. Glider allows to perform these operations asynchronously for concurrent processing and better network utilization.

In contrast with other storage servers, active servers feed the data into the action object, instead of simply storing it. The goal is that action methods also process an I/O stream. To achieve this, actions run in execution threads decoupled from the network workers and communicate with them through task queues. When a request (basic operation) reaches the server, a network worker identifies its type and destination and queues it as a data task for a specific action. Each I/O stream has its own task queue that collects the multiple data tasks performed on it. Action threads consume task queues by running the appropriate action method. While the code of the action method consumes or populates its stream, internally, data tasks are processed and completed. The client decides when to finalize the transfer by closing the stream on its side. This sends a final request that closes the stream at the server side and ends the method execution.

Actions and concurrency. As remote elements, multiple clients may access actions concurrently. Since actions are stateful elements, concurrent execution of their logic may result in unexpected behavior due to uncontrolled modification of the action state. Therefore, we need to define a concurrency model for actions that allows programmers to easily reason about their execution.

Glider executes each action as if it were run by a single thread. This means that, at any time, there is only one method being run on each action. Multiple actions may freely execute concurrently. This effectively eliminates unexpected state modifications. Furthermore, it simplifies action development to avoid complex concurrency issues. To achieve this, action threads obtain exclusive control of an action object while running one of its methods.

There are special cases where an application may benefit from multiple I/O operations running concurrently on the same action. For instance, if we want multiple clients to read data from an action at the same time, the default concurrency model will serialize the operations from each client, holding the read from one of them until the other one completes. To allow multiple clients to access actions concurrently, Glider supports action interleaving. Interleaving may be configured when creating the action. The concept is applied similar to Orleans [10]. When activated, the execution of an action method may yield control while it is waiting for more I/O tasks. In such event, another action method may take control. Execution is still guaranteed to be single threaded, but methods may take turns in execution before their completion.

5 IMPLEMENTATION

Glider is implemented in about 3K SLOC in the Java language by extending Apache Crail's base implementation of the NodeKernel architecture. It inherits Crail's metadata plane, server management, and basic node types. On top, Glider integrates storage actions and all their features to resolve its targeted unique challenges. We implement the new action node type and the new active server type, modify the metadata servers to support the new elements, and provide new client abstractions to access them.

Metadata servers in Glider include new structures to organize action nodes in the storage hierarchy and the action slot management logic, including the new active class. Action nodes are distributed

across the active storage servers that joined that class. Like Crail, we uniformly distribute actions and objects in the system with a round-robin mechanism. Sophisticated action distribution and resource management are beyond the scope of the paper (§4.2).

Our prototype active server implementation is based on the DRAM-backed storage server in Crail. Instead of plainly storing data in memory, it runs an action manager object that handles the creation, execution, and deletion of action objects. The action manager allocates slots for actions depending on configuration and available resources, and it registers them on the metadata servers.

Active servers employ a pool of network threads that respond to client calls. Create and delete requests do so on the corresponding action object in memory. Data access methods are executed by a separated pool of action threads. This allows to decouple the execution of action methods from the individual data operations that enable the stream I/O interface. Each method execution is assigned an id and sequence number that are used to create a task queue. Action threads consume these queues to run action methods. The single-thread-like execution of action methods is achieved with locks. Action threads take the action lock while running one of its methods. For actions with interleaving, the lock is released when the method is waiting for more data in its queue.

Table 1: Glider's application programming interface.

Object	Methods
Store Client	**create**<T extends Node>(*path*, *sc*) → future(*T*) creates a new data node of type *T* **lookup**<T extends Node>(*path*) → future(*T*) finds an existing node at the given *path* **delete**(*path*) → future(*boolean*) deletes an existing node at the given *path*
Action Node	**create**<T extends Action>(*il*) → future(*boolean*) creates an action object of type *T* in this node **delete**() → future(*boolean*) deletes the action object in this node **getInputStream**() → *InputStream* **getOutputStream**() → *OutputStream* obtains a stream to read or write to this action
Action Object	abstract **onRead**(*outputStream*) → void abstract **onWrite**(*inputStream*) → void abstract **onCreate**() → void abstract **onDelete**() → void

6 USING GLIDER

The interface of Glider provides simple abstractions with two goals: 1) actions are managed like other storage elements, and 2) they are coded and deployed like functions in FaaS.

6.1 Application interface

Glider's application interface, summarized in Table 1, extends the one in Crail with new components to manage storage actions. The top-level object in the interface is the `StoreClient`, that connects to a specific namespace. Its methods allow to `create`, `lookup`, and `delete` data nodes in the storage hierarchy. The identifier for nodes is their path in the namespace, like a file system. When creating a new node, a parameter (`sc`) allows to specify a preferred storage class. Action nodes are always stored in an active class. Applications receive proxy references to nodes to interact with them.

The action node proxy data type implements four basic primitives. The `create` method instantiates an action object into the node. This method requires a concrete action type definition (action logic; §6.2). An optional parameter (`il`) toggles interleaving for that action. The `delete` method removes the action object in the node. This allows action finalization logic, to change the action definition on an existing node, or to simply recreate it to clear its state. The other two primitives allow to obtain I/O streams to transfer data.

It should be noted that all remote operations are asynchronous and clients handle execution through future objects. This common pattern integrates with modern software interfaces and allows to efficiently handle call results or failures. It also allows the construction of buffered I/O streams, keep a data operation always in flight, and not block the application on network access. Another implementation of direct streams gives the user full control of operations.

6.2 Developing actions

To define the logic of actions, developers specialize the `Action` data type (Action Object in Table 1). This interface defines four methods; all of them optional and empty by default. `onCreate` and `onDelete` run when the action node instantiates or removes the action object, following the analogous method calls to the action node proxy. They have no parameters and may be used to initialize and finalize the action object. `onRead` and `onWrite` run for each I/O stream that connects to the action through the dedicated proxy methods. Both receive one parameter representing the corresponding I/O stream. The `onRead` method receives a writeable stream that it should populate with data as desired. The `onWrite` method receives a readable stream from which it may consume the data that is being written to the action. Applications may use the object fields as desired to keep a modest action state (e.g., a counter, a small key-value table, a custom data type, or references to other storage nodes). Conveniently, action objects get a store client, by default, to access other storage nodes, including other actions, and construct data processing patterns within the ephemeral store.

Programmers should make their action definitions available to Glider before creating actions. To this end, programmers upload a package containing their definitions, which is then provided to active storage servers. Each action definition is registered with a name. Applications may use this name when instantiating action objects into storage nodes, as detailed in §6.1. This process resembles function deployment in FaaS platforms.

6.3 Application example

Aggregations are one of the main use cases for storage actions. They exploit the statefulness of actions to receive data from multiple workers in a single computational element and reduce network transfers. Moreover, thanks to the I/O streams, applications may

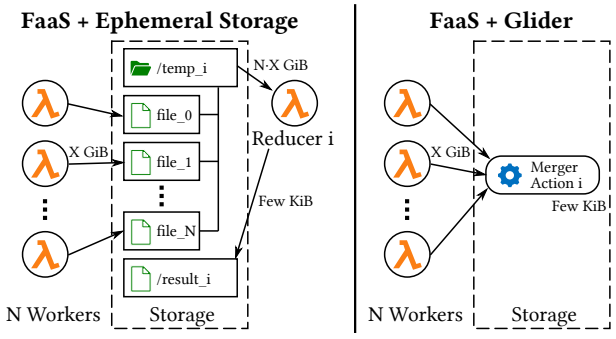

Figure 4: Diagram of a data processing aggregation with and without actions. For reducer *i* in a group of them.

store only the merged data (instead of intermediate results in full) to keep storage utilization low.

We illustrate these cases with a word counting job. Figure 4 depicts a diagram of this workload on Glider and actions (right) against a solution with FaaS and traditional storage (left). Workers generate the counting for their part of the input text and split it across the multiple reducers. The reducers then aggregate the counts into a dictionary each, which may be used in a future computation phase. For simplicity, the figure only shows one reducer of the group. In the base solution (left), workers write the counts in storage files. Each worker writes a file for each reducer. Each reducer then reads their files, aggregates data, and writes a new file for the next phase. With Glider (right), this lightweight aggregation is done in storage actions. Workers write their counts directly to actions. Each action merges them as they arrive and only stores the aggregated data. A next computation stage may read this result directly from the action. If the application requires a single dictionary, the results may be further combined in a reduction tree. This is easy through concatenating actions, instead of requiring additional workers and temporary files. In seldom cases that still require resiliency (§4.2), the user may implement checkpointing to recover failed actions or idempotence mechanisms to allow worker retries. More complex mechanisms to maintain consistency across multiple actions (such as transactions) could also be considered.

The definition of the merger action is shown in Listing 1 with simplified Java code. This action contains an object field, a dictionary, to save the aggregated data, which is initialized in the creation method. The `onWrite` method processes the key-count pairs that workers write into the action, combining them appropriately into the local dictionary. Note that our application interface allows to wrap the input stream with specialized readers, such as to obtain a stream of lines. This code will run until the stream reaches an end, meaning that the client has closed the operation. The `onRead` method allows to read the aggregation result from the action. For that, it serializes the local dictionary into the output stream. Note that closing the stream finishes the operation and notifies the client.

This action benefits from interleaving to allow several workers to write to the same action concurrently. In this case, an operation waiting for more text at line 10 may yield control to another operation. This effectively optimizes network utilization.

```
1  public class MergeAction extends Action {
2    private Map<Integer, Long> result;
3
4    public void onCreate() {
5      result = new HashMap<>();
6    }
7
8    public void onWrite(InputStream input) {
9      Stream<String> lines = input.lines();
10     lines.forEach(line -> {
11       result.merge(
12           Integer.parseInt(line.split(",")[0]),
13           Long.parseLong(line.split(",")[1]),
14           (val, acc) -> val + acc);
15     });
16   }
17
18   public void onRead(OutputStream output) {
19     output.writeObject(result);
20     output.close();
21   }
22 }
```

Listing 1: Action definition to perform an aggregation.

7 EVALUATION

Goals and scope. The objective is to understand the benefits of Glider for serverless data processing workloads. To this end, we use the following key indicators: (i) the amount of data transferred between compute (FaaS) and storage systems, (ii) the number of data transfers between the systems, (iii) the temporary storage utilization, and (iv) the overall application performance. Glider enjoys storage features from Apache Crail and is compatible with Pocket's automatic scaling. These are orthogonal to our goals and we do not evaluate them. Instead, we focus on Glider's new contributions to face data-shipping issues. Our evaluation campaign starts by exploring the benefits of Glider in the above indicators for different common patterns in serverless analytics (§7.1). Then, we characterize actions with micro-benchmarks (§7.2) and finish with two real-world workloads: a distributed sort (§7.3) and a genomics variant calling pipeline (§7.4). Our baseline for comparison is the state-of-the-art approach for serverless analytics, i.e., serverless workers generate intermediate data that write and read from remote storage throughout different computation stages. We follow the descriptions of PyWren [22], the AWS Lambda MapReduce reference architecture [3], and Pocket [27].

Setup. We use a cluster of servers driving two Intel® Xeon® CPU E5-2690 @ 2.90GHz (16 physical cores) and 96 GiB of memory. The network link is a 100 Gbps Mellanox Technologies ConnectX-5 MT27800. We run each storage server alone in a machine, meaning that all communication between actions and other elements is remote. All active servers have enough CPU to dedicate one core to each action. Data servers are DRAM-backed. All experiments require a single metadata server. We simulate serverless workers (FaaS) as processes in another machine in the same cluster. The genomics variant calling pipeline is fully evaluated on AWS Lambda and up to 25 EC2 machines.

7.1 Benefits

Impact of actions on data movement. We evaluate this effect in a common data processing pipeline where distributed workers

Table 2: Data processing pipeline on 10 GiB with 10 workers.

	Ingested	Time (s)	Throughput
Data-shipping	10 GiB	28.866	2.98 Gbps
Glider	25.7 MiB	10.813	7.94 Gbps
Glider (RDMA)	25.7 MiB	9.182	9.36 Gbps

(FaaS) need to ingest data from storage, but data needs to be parsed, arranged, or pre-processed before the main computation.

In this situation, Glider's opportunity for improvement consists in offloading pre-processing to actions. Instead of reading the full files, workers read from actions that act as proxies. With this approach, the communication between workers and storage is reduced to already prepared data, while actions achieve faster data access thanks to near-data execution within the storage system.

We consider an example of word counting where text files need to be filtered on a per line condition first. We run Glider's approach against the baseline with around 10 GiB of data (Wikipedia backup files [45]) and 10 workers (1 GiB each). This experiment uses one active server, and one data server (for files).

Table 2 summarizes the results. With actions, data transfer between workers and storage is reduced by 99.75%. This reduction is key when the workers have limited network bandwidth (like in FaaS) and data transfer contributes importantly to execution cost.

Impact of actions on storage accesses. We study this impact in a situation where data generated by a set of workers (FaaS) must be aggregated. The baseline approach uses another worker to perform the reduction. This means that intermediate data must be stored in full and then read back by the reduce worker.

Here, Glider's opportunity for improvement consists in pushing the reduction actions. This is possible since actions are stateful, and it eliminates a stage in the compute tier with its extra storage connections. Action will receive data concurrently (with interleaving) from multiple workers while they perform the aggregation.

We illustrate this situation with a synthetic example. Workers generate random numeric pairs (key, value) that they emit as strings. The reducer adds the values for each key in an aggregated dictionary. The generated keys are 1024 distinct integers, and the values comprise the full range of a Java Long. Each worker generates 50M pairs: just over 1 GiB of data when sent through network. This experiment uses one active and one data servers.

Figure 5 presents the results for different numbers of workers. Glider cuts storage accesses by half (50%). Indeed, instead of (1) sending the intermediate data, (2) reading it in full again, to then (3) write the result so it is (4) available for the next stage, actions require a single connection to (1) send the data and the result becomes (2) available for the next stage. Thanks to that, data movement is also halved in this case (see Figure 5 right).

Impact of actions on storage utilization. The previous aggregation also evidences the advantage of Glider in storage utilization. While the baseline needs to save all generated data in storage (about 11 GiB with 10 workers), actions process the input stream as it is generated, storing only the resulting dictionary (≈24 KiB). In this case, storage utilization is reduced by approximately 99.8%.

Figure 5: Reduce operation with Glider against a data-shipping model. Left shows total time elapsed. Right shows data transferred between application workers and storage.

Impact of actions on performance. The first example shows a 2.7× performance boost and, in the second, Glider reduces execution time by up to 27% with 5 workers. Several factors contribute to this matter. First, less data ingestion directly affects total application run time. We simulate them on a fast network, but serverless functions have limited bandwidth and benefit even more from this matter. Second, the elimination of computation stages (offloading them to actions) reduces storage utilization and simplifies connections. This also removes the need to transfer the full data back to the compute tier and, hence, lowers data movement again with the above benefits. Lastly, an important performance booster is the ability to stream data between workers and actions. This type of pattern is not possible between serverless functions, but it allows actions to work in parallel with workers to speed up applications.

To illustrate the first point, consider the ingestion example (Table 2). Since it runs in a setup where workers and storage are in the same network, there is no difference to read files from workers and actions. The boost seen is possible thanks to the parallelism explained above: actions filter data at the same time that workers count words. However, action integration within the storage system allows them to exploit locality or use advanced technologies. We demonstrate this with an RDMA-enabled high-performance network. Actions exploit this technology, unavailable for serverless workers, to speed up performance to 3.14×.

7.2 Micro-benchmarks

Action bandwidth. The objective is to assess the bandwidth to an action against a base storage file. The extra logic necessary to run arbitrary code on actions suggests a small penalty to be expected.

The experiment writes and reads 10 GiB to/from each data type for varying operation sizes (buffer size). We use direct streams to take full control of operations. To maximize network utilization, asynchronous operations are done in batches to always keep data transfers in flight without collapsing the network. We adjust the batch size to achieve best performance on each configuration. Note that operations with more than 1 MiB surpass block size and would be split into smaller operations. Actions run empty methods.

The top of Figure 6 shows the average results. Actions do not add overhead with respect to files. Read operations achieve at most 11% less bandwidth, while writes reach up to 12% higher bandwidth since they do not require allocating new blocks as the data grows and skip communication with the metadata server.

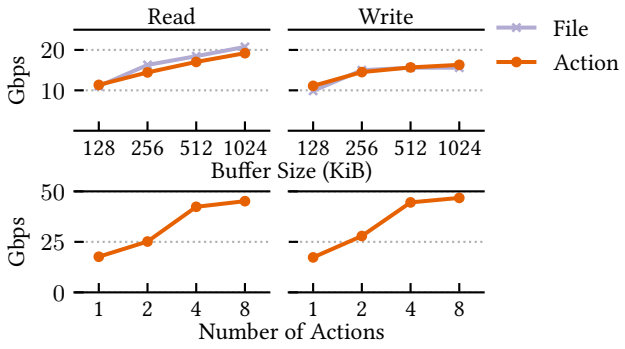

Figure 6: Average access bandwidth to files and actions for different buffer sizes and number of concurrent actions.

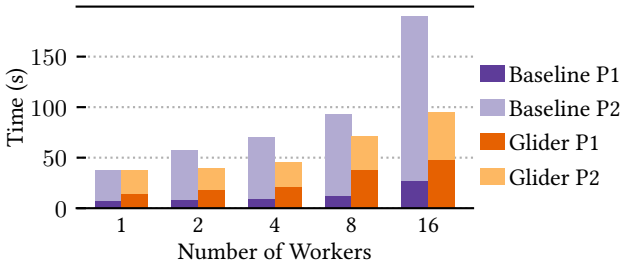

Figure 7: Sort execution time with Glider against a data-shipping approach. Each worker processes 1 GiB.

Action scale. We evaluate the capacity of actions to leverage the full compute and network resources in their storage space. We use the same setup of the previous experiment with 1 MiB operations and replicate it up to 8 parallel actions. Each action still transfers 10 GiB and is accessed by a dedicated client. The active storage space runs 8 network threads to enable this parallelism and bandwidth is computed globally for the aggregated result.

Figure 6 bottom shows the results. Running parallel actions improves bandwidth but plateaus around 45 Gbps (which we identified as the limit for TCP operations in the cluster). Similar results are drawn from the same experiment assessing files. We conclude that actions scale to the resources of their storage space.

7.3 Distributed sort

In this section, we evaluate Glider's benefits in a real world application. In particular, we study a distributed sort of data. Serverless shuffle operations generate a lot of intermediate data and, consequently, large data transfers (§2). Sorting is a severe example of this, because the temporary data generated contains the full input dataset [33, 37]. Since functions are stateless, each stage needs to read and write everything, and with the resource limitations of functions, this process becomes slow and expensive.

Our baseline is an implementation following the common approach [3, 22, 27]. To perform a sort, a set of workers compute in two phases (map (P1) and reduce (P2)). The input dataset, the intermediate data, and the resulting sorted data are saved in the storage system as files (in data servers). The first stage reads the input dataset and, using a sorting key, distributes the text between the reducers. Each worker generates a file for each reducer (intermediate data). In the second stage, reducers read back these files, sort their content, and write the result again.

Glider improves this situation by pushing the reduce to storage. This mechanism is similar to a scaled version of Figure 4. It presents three clear advantages. First, an entire stage of workers is no longer needed, which *reduces the number of storage accesses*. Second, with less storage accesses, *less data must be transferred* in and out of the storage system. And third, thanks to actions and the I/O streams, part of the sorting is done in *parallel* to the first stage, without waiting for the shuffle to finish. In detail, first-stage workers do not write into files, but send the classified data to the actions. Note that the process of writing is unchanged since actions and files share the same interface. As actions receive data through a stream, they parse and keep it in memory (P1). When all workers have finished, the actions sort the data and write the result as new files (P2).

To explain these benefits, let us recount and compare the amount of *data movement* in each approach. The baseline implementation fully reads and writes the entire dataset to storage twice; accounting for a data transfer of four times the size of the data. Glider only reads and writes once, since it only employs one stage of workers outside storage. In the second phase, actions do not read the data (which is streamed to them by the workers) and write the result from within the storage cluster. Therefore, data transfer is cut to just twice the dataset size, for a 50% *reduction in data movement*.

For the experimental comparison, we use a randomly generated dataset with 1 GiB partitions. We evaluate with up to 16 workers, each of them reading a full partition (i.e., 16 workers sort 16 GiB). Both phases use the same number of workers and actions. This experiment uses two active and one data servers with uniform action distribution.

Figure 7 presents the results. The solution with actions is always faster than the baseline approach. In particular, Glider reduces run time by 49.8% with 16 workers. The baseline keeps the map phase time (P1) constant, but the reduction (P2) is slow due to shipping intermediate data back from far storage. On the contrary, Glider is slower during the first phase (P1) because it includes actions parsing the data. However, the second phase (P2) is up to 71% faster since actions avoid the extra data transfer.

7.4 Genomics variant calling

To complete the evaluation, we consider Glider's targeted environment: a cloud deployment in collaboration with serverless functions. We assess a distributed variant calling pipeline. Variant calling is a genomics process to identify genetic variants through the alignment of sequencing data to a reference genome. This allows to identify where sequence reads differ from the reference and has many important uses in health research. Reference genomes are stored in the FASTA format and are sized from some MiB to over 100 GiB. Sequencing reads are collected in the FASTQ format and can grow up to several TiB. These sizes require a huge amount of data transfers during computation, which rapidly become a problem in traditional cluster computing and present important challenges in serverless settings. It clearly manifests the issues of data-shipping.

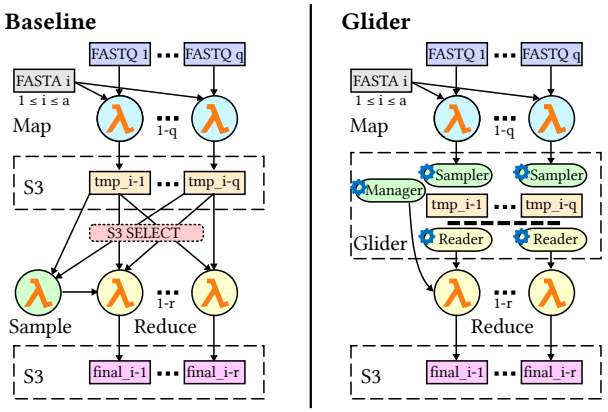

Figure 8: Diagram of the variant calling pipeline for a single FASTA chunk i with serverless functions and Glider.

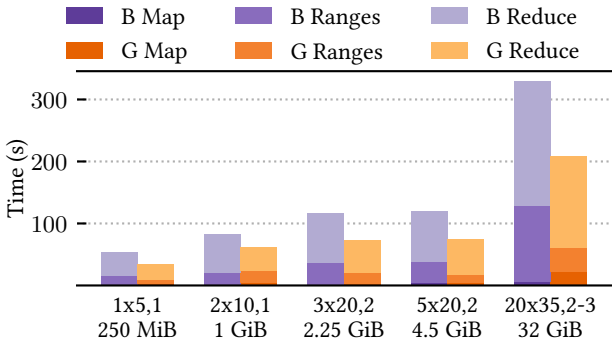

Figure 9: Serverless genomics variant calling. Execution time with Glider (G) against the baseline (B). Labels are $a \times q, r$ showing the number of chunks and reducers per FASTA chunk with an approximate size of the temporary files.

We study a baseline solution following the Map-Reduce model atop serverless functions and cloud object storage. The process compares a FASTQ sequencing file against a reference FASTA file. To scale, the FASTA and FASTQ files are split into a and q chunks respectively. The process requires matching all FASTQ chunks against all FASTA chunks, creating $m = aq$ map tasks, each generating a temporary file with the aligned reads. Due to chunking, these files contain partial results that need aggregation. The aggregation sorts and combines all intermediate files from a FASTA chunk into a file with the variants called. This task may be scaled to r reducers per FASTA chunk. Hence, shuffling data is required. The resulting files may be appended in order into a single final file.

The specific implementation runs on AWS. The left side of Figure 8 shows the process for one FASTA chunk. Mapper tasks are modelled as AWS Lambda functions with input and output to Amazon S3. We simplify the map computation (to only produce the intermediate data) to focus on the data shuffle. Shuffling with serverless functions is tricky [33, 37]. To tailor the number of reducers (r) and their data ranges to the size of the intermediate data, the baseline uses S3 SELECT to first sample the files. This allows to explore the tradeoff between parallelism and function memory utilization. The reducer functions use S3 SELECT again to download only the parts of the temporary files they need, achieving the data shuffle. This avoids reducers ingesting data outside their range.

Glider presents an opportunity to improve the reduction stage. The right side of Figure 8 shows the process for one FASTA chunk. Instead of writing temporary files to S3 and requiring S3 SELECT to read them multiple times for shuffling, mappers directly write their output to Glider actions (Sampler Action). This first set of actions sample the data as they receive it and store it on ephemeral files. When the mapper functions finish, these actions quickly interact with a manager action that computes the number of reducers and their data ranges. Reduce functions connect with another set of actions (Reader Action). These exploit near-data computation to solve shuffling and provide reducers a single stream with the ranges of data they need from the multiple temporary files.

With this approach, similar to S3 SELECT before, reducers only ingest their range of data. However, it also adds the following benefits: (1) intermediate data is sent to a specialized ephemeral store, faster than object storage; (2) intermediate files and actions are in the same system for improved data access; (3) a single storage request is enough to access multiple temporary files; (4) stateful computation enables more flexible data processing than S3 SELECT (limited to simple SQL SELECT queries on specific data formats); and (5) computations are streamlined with data transfers thanks to I/O streams, allowing parallelism between mappers and data sampling and reducers and data shuffling. As an example, the last two points allow to provide each reduce worker with a single sorted data stream. On the contrary, the baseline solution needs to open requests to multiple storage objects and sort their content in full as part of the aggregation process.

We run this comparison with the Human Genome FASTA file (3 GiB) and the SRR15068323 FASTQ file (5.25 GiB) [32]. These are split into $a = 20$ and $q = 35$ chunks. Hence, the full experiment runs $m = 700$ mappers. The intermediate files (about 32 GiB) are shuffled to $r = 2$ or 3 reducers per FASTA chunk (total of 47). We run partial executions with a subset of the chunks to evaluate scale. Map and reduce Lambda functions are configured with 2 GiB and 8 GiB respectively. Glider is deployed with one metadata server (t3.xlarge), up to 4 data servers (r6i.large), and up to 20 active servers (c5.12xlarge). Data and actions are uniformly distributed across nodes in their tiers.

Figure 9 draws the results. Compared with the baseline, Glider allows to avoid a full read of the intermediate data to perform sampling. This minimizes storage accesses and *eliminates* 1/3 *of the intermediate data transfer.* The baseline transfers the intermediate data 3 times (mappers write, samplers read, and reducers read), while Glider only twice (mappers and reducers). Consequently, our solution with actions is always faster than the baseline. The plot shows how this behavior is kept with scale. In particular, *Glider reduces execution time by* 36% with the full data. The map stage is slower with Glider since it includes data sampling at the actions. However, this allows a much faster range distribution. Finally, the reduce is also faster due to the benefits above, including lower data ingestion to workers, a richer data processing on actions, and streaming data transfer.

8 RELATED WORK

Data-shipping is a well-known problem in the serverless computing model that creates heavy data transit, strains the network, and often becomes a bottleneck [20, 23, 26, 41]. We highlight three main approaches to tackle it (left to right in Figure 1): (A) optimized disaggregation of compute and storage, (B) introduction of storage within a FaaS platform or vice-versa, and (C) disaggregation with task offloading. The three currently fall short to effectively solve it.

Optimized disaggregation. The first approach (A in Figure 1) exploits the fast elasticity of FaaS to obtain massive parallelism [3, 22]. However, huge intermediate data must be transferred between functions through disaggregated storage. Commonly, cloud object storage is chosen due to its high bandwidth. Nonetheless, it quickly becomes a bottleneck in large data processing workloads [8, 20, 23]. Concerned by this, some works explore configuration optimizations based on the amount of data [37] or combine multiple storage solutions [33]. Others search for ways to achieve function to function communication [12, 17], or share application state between functions in remote memory [9, 25]. Pocket [27] focuses on improving file-based access for serverless workers in an elastic multi-tenant store. All these examples evidence the data-shipping issue and the struggle of insufficient tools in the cloud. Despite the efforts, none of these works confront the fundamental challenges of data-shipping: *reduce the amount of data being transferred during computation.*

Unified systems. Several projects implement FaaS atop a storage system or vice-versa (B in Figure 1). Their goal is to co-locate computation and its data (caching it). One line of research opts to modify the FaaS platform to co-locate related functions and allow shared memory between them [2, 40]. Differently, some projects implement a FaaS platform on top of a storage system [41, 47]. Another trend is to exploit existing serverless platforms to build cache stores on the function resources themselves [31, 35, 43, 46]. All these projects fully couple storage and computation in shared resources, which has been discouraged in the past [11, 36]. In particular, this approach creates interferences between the storage and compute features. Managing the scale of both components jointly is usually inefficient for one, or both, of them. Consequently, computation cannot scale freely like it usually does in dedicated FaaS platforms and their storage capacity is very limited, which is unfitting for large intermediate data.

Computation-enabled storage. For distributed storage systems, close to the data computation has been studied as active storage [34]. Active storage [7, 19, 30] uses computing resources in the storage system to enable analytics frameworks (e.g., Apache Spark) to offload operations on them (C in Figure 1). Accessing storage elements triggers the execution of simple stateless interceptions that may transform data in-line. These works demonstrate huge data transfer savings between compute and storage tiers and an effective way to counteract data-shipping in cluster computing. In the cloud, a multi-tenant setting, data access and management follows different methods than in clusters and resource contention becomes an important issue for active storage [11, 36]. Amazon S3 SELECT [6] controls this with predefined operations (simple SQL SELECT queries), but it becomes too limited in versatility for many applications (§7.4). As a solution, Zion [36] proposes an architecture with an active storage layer correctly isolated from the storage resources that intercepts data accesses with user-provided functions. Yet, it is limited to stateless interception of the data path. Similarly, S3 Object Lambda [4] creates an alternative S3 endpoint that intercepts all object accesses with AWS Lambda functions. However, it cannot be considered near-data computation since it runs on the usual Lambda resources and not within the storage service. It, thus, follows approach A and suffers from far data transfers [39].

Actions and actors. Glider actions resemble actors [10, 28] in their execution and concurrency models, but with significant differences. Actors are typically durable, event/message-based entities, while actions are designed to handle large temporary data transfers with I/O streams. Azure Durable Entities [29] offers actors (with limitations) as a serverless service. Flink StateFun [15] similarly creates stateful entities. These solutions still require large data movement between compute and storage since they persist actor state in a remote store that is fed to them between activations.

9 CONCLUSION

We introduce Glider, a cloud storage service model addressing the data-shipping problem in serverless analytics. Its primary contribution lies in the integration of ephemeral near-data computation within a storage service specifically designed to collaborate with existing FaaS platforms. As far as our knowledge goes, Glider is the first solution to incorporate stateful ephemeral computation into ephemeral storage, tailored for intermediate data. The main objective is to minimize the amount of data transferred between compute and storage systems, thereby optimizing the connections between various stages of serverless functions. To achieve this, Glider introduces storage actions as named storage elements within the storage namespace. They do not only encapsulate stateful computation but also provide I/O streams to efficiently handle large volumes of data.

Our evaluation demonstrates the benefits of Glider. Specifically, we showcase substantial reductions in data transfers between the compute and storage tiers, reaching up to 99.75%. Moreover, Glider promotes efficient data flow, resulting in decreased intermediate data volume and storage space utilization up to 99.8% in certain scenarios. Collectively, these improvements effectively enhance application performance. For example, a distributed sort operation on 16 GiB becomes 49.8% faster, and an intensive serverless variant calling process with over 700 serverless functions improves execution time by up to 40%.

Therefore, serverless stateful near-data computation proves to be an asset in enhancing the programmability and performance of cloud applications, benefiting both cloud platforms and their users.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd Haris Volos for their valuable feedback. Daniel Barcelona-Pons was with IBM Research Europe, Zurich while performing this work; he thanks all the colleagues at the Cloud Data Platforms group. Thanks to Xavier Roca for his help. This work is partially funded by the Horizon Europe programme under grant agreements No. 101092644 (Neardata), 101092646 (CloudSkin), and 101093110 (Extract).

REFERENCES

[1] Anurag Acharya, Mustafa Uysal, and Joel Saltz. 1998. Active Disks: Programming Model, Algorithms and Evaluation. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS VIII)*. Association for Computing Machinery, New York, NY, USA, 81–91. https://doi.org/10.1145/291069.291026

[2] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 923–935. https://www.usenix.org/conference/atc18/presentation/akkus

[3] Amazon Web Services. 2019. Serverless Reference Architecture: MapReduce. Retrieved September 15, 2023 from https://github.com/awslabs/lambda-refarch-mapreduce

[4] Amazon Web Services. 2021. S3 Object Lambda. Retrieved September 15, 2023 from https://aws.amazon.com/s3/features/object-lambda/

[5] Amazon Web Services. 2023. Introducing AWS Lambda response streaming. Retrieved September 15, 2023 from https://aws.amazon.com/blogs/compute/introducing-aws-lambda-response-streaming/

[6] Amazon Web Services. 2023. S3 Select. Retrieved September 20, 2023 from https://docs.aws.amazon.com/AmazonS3/latest/userguide/selecting-content-from-objects.html

[7] Alex Barcelo, Anna Queralt, and Toni Cortes. 2022. Revisiting active object stores: Bringing data locality to the limit with NVM. *Future Generation Computer Systems* 129 (2022), 425–439. https://doi.org/10.1016/j.future.2021.10.025

[8] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. 2019. On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures. In *Proceedings of the 20th International Middleware Conference* (Davis, CA, USA) *(Middleware '19)*. Association for Computing Machinery, New York, NY, USA, 41–54. https://doi.org/10.1145/3361525.3361535

[9] Daniel Barcelona-Pons, Pierre Sutra, Marc Sánchez-Artigas, Gerard París, and Pedro García-López. 2022. Stateful Serverless Computing with Crucial. *ACM Trans. Softw. Eng. Methodol.* 31, 3, Article 39 (mar 2022), 38 pages. https://doi.org/10.1145/3490386

[10] Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. 2014. *Orleans: Distributed Virtual Actors for Programmability and Scalability.* Technical Report MSR-TR-2014-41. Microsoft. https://www.microsoft.com/en-us/research/publication/orleans-distributed-virtual-actors-for-programmability-and-scalability/

[11] Chao Chen, Yong Chen, and Philip C. Roth. 2012. DOSAS: Mitigating the Resource Contention in Active Storage Systems. In *2012 IEEE International Conference on Cluster Computing*. IEEE, New York, NY, USA, 164–172. https://doi.org/10.1109/CLUSTER.2012.66

[12] Marcin Copik, Roman Böhringer, Alexandru Calotoiu, and Torsten Hoefler. 2023. FMI: Fast and Cheap Message Passing for Serverless Functions. In *Proceedings of the 37th International Conference on Supercomputing* (Orlando, FL, USA) *(ICS '23)*. Association for Computing Machinery, New York, NY, USA, 373–385. https://doi.org/10.1145/3577193.3593718

[13] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. 2019. NICA: An Infrastructure for Inline Acceleration of Network Applications. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 345–362. https://www.usenix.org/conference/atc19/presentation/eran

[14] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 51–66. https://www.usenix.org/conference/nsdi18/presentation/firestone

[15] Apache Flink. 2022. Stateful Functions. Retrieved September 15, 2023 from https://nightlies.apache.org/flink/flink-statefun-docs-master/

[16] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 475–488. http://www.usenix.org/conference/atc19/presentation/fouladi

[17] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. 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)*. USENIX Association, Boston, MA, 363–376. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi

[18] Glider. 2023. GitHub. Retrieved September 15, 2023 from https://github.com/CLOUDLAB-URV/glider-store

[19] Raúl Gracia-Tinedo, Marc Sanchez-Artigas, Pedro Garcia-Lopez, Yosef Moatti, and Filip Gluszak. 2019. Lamda-Flow: Automatic Pushdown of Dataflow Operators Close to the Data. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, New York, NY, USA, 112–121. https://doi.org/10.1109/CCGRID.2019.00022

[20] Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2018. Serverless Computing: One Step Forward, Two Steps Back. https://doi.org/10.48550/ARXIV.1812.03651

[21] K. R. Jayaram, Vinod Muthusamy, Parijat Dube, Vatche Ishakian, Chen Wang, Benjamin Herta, Scott Boag, Diana Arroyo, Asser Tantawi, Archit Verma, Falk Pollok, and Rania Khalaf. 2019. FfDL: A Flexible Multi-Tenant Deep Learning Platform. In *Proceedings of the 20th International Middleware Conference* (Davis, CA, USA) *(Middleware '19)*. Association for Computing Machinery, New York, NY, USA, 82–95. https://doi.org/10.1145/3361525.3361538

[22] 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)*. Association for Computing Machinery, New York, NY, USA, 445–451. https://doi.org/10.1145/3127479.3128601

[23] 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. 2019. *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. Technical Report UCB/EECS-2019-3. EECS Department, University of California, Berkeley.

[24] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 1–16. https://www.usenix.org/conference/nsdi19/presentation/kalia

[25] 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

[26] 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

[27] 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

[28] Lightbend. 2023. Akka. Retrieved September 15, 2023 from https://akka.io/

[29] Microsoft Azure. 2023. Entity functions. Retrieved September 15, 2023 from https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-entities

[30] Yosef Moatti, Eran Rom, Raul Gracia-Tinedo, Dalit Naor, Doron Chen, Josep Sampe, Marc Sanchez-Artigas, Pedro Garcıa-Lopez, Filip Gluszak, Eric Deschdt, Francesco Pace, Daniele Venzano, and Pietro Michiardi. 2017. Too Big to Eat: Boosting Analytics Data Ingestion from Object Stores with Scoop. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, New York, NY, USA, 309–320. https://doi.org/10.1109/ICDE.2017.243

[31] 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* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 228–244. https://doi.org/10.1145/3447786.3456239

[32] National Library of Medicine. 2021. WGS of tumor sample from patient P94 (SRR15068323). Retrieved September 15, 2023 from https://trace.ncbi.nlm.nih.gov/Traces/?view=run_browser&acc=SRR15068323

[33] 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

[34] Erik Riedel, Garth A. Gibson, and Christos Faloutsos. 1998. Active Storage for Large-Scale Data Mining and Multimedia. In *Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB '98)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 62–73.

[35] 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. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 122–137. https://doi.org/10.1145/3472883.3486974

[36] 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

[37] Marc Sánchez-Artigas, Germán T. Eizaguirre, Gil Vernik, Lachlan Stuart, and Pedro García-López. 2020. Primula: A Practical Shuffle/Sort Operator for Serverless Computing. In *Proceedings of the 21st International Middleware Conference Industrial Track* (Delft, Netherlands) *(Middleware '20)*. Association for Computing Machinery, New York, NY, USA, 31–37. https://doi.org/10.1145/3429357.3430522

[38] Bo Sang, Pierre-Louis Roman, Patrick Eugster, Hui Lu, Srivatsan Ravi, and Gustavo Petri. 2020. PLASMA: Programmable Elasticity for Stateful Cloud Computing Applications. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 42, 15 pages. https://doi.org/10.1145/3342195.3387553

[39] Pablo Gimeno Sarroca and Marc Sánchez-Artigas. 2023. On Data Processing through the Lenses of S3 Object Lambda. In *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications*. IEEE, New York, NY, USA, 1–10. https://doi.org/10.1109/INFOCOM53939.2023.10228890

[40] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Berkeley, CA, USA, 419–433. https://www.usenix.org/conference/atc20/presentation/shillaker

[41] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful Functions-as-a-Service. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2438–2452. https://doi.org/10.14778/3407790.3407836

[42] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Ana Klimovic, Adrian Schuepbach, and Bernard Metzler. 2019. Unification of Temporary Storage in the NodeKernel Architecture. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 767–782. https://www.usenix.org/conference/atc19/presentation/stuedi

[43] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. 2020. InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache. In *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

[44] 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

[45] Wikimedia. 2023. Wikimedia downloads. Retrieved September 15, 2023 from https://dumps.wikimedia.org/

[46] 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. Sion: Elastic Serverless Cloud Storage. https://doi.org/10.48550/ARXIV.2209.01496

[47] 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* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3357223.3362723