FuncStore: Resource Efficient Ephemeral Storage for Serverless Data Sharing

Yijie Liu[1], Zhuo Huang[1], Jianhui Yue[2], Hanxiang Huang[1], Song Wu[1], and Hai Jin[1]
[1] *National Engineering Research Center for Big Data Technology and System*
Services Computing Technology and System Lab, Cluster and Grid Computing Lab
School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China
[2] *Department of Computer Science, Michigan Technological University, Houghton, Michigan, 49931, USA*
{*lyjhust, huangzhuo, handsonhuang, wusong, hjin*}*@hust.edu.cn, jyue@mtu.edu*

***Abstract*—Serverless architecture decomposes applications into numerous fine-grained functions, typically executed within individual containers. These functions share data with remote storage to achieve high flexibility. However, existing cloud storage services adopt either application-level or function-level resource allocation strategies. In the application-level approach, users request all necessary resources at the start of application execution and release them upon completion. On the contrary, in the function-level approach, resources are allocated at the start of each function and reclaimed if they remain idle for a specified duration. Regrettably, both methods tend to yield suboptimal utilization of storage resources, as data often lingers beyond its relevant lifecycle.**

To this end, we introduce FuncStore, an innovative serverless ephemeral storage system designed to optimize resource utilization. FuncStore adopts a resource allocation strategy that provides resources precisely when data is generated and releases those resources when the data reaches the end of its lifecycle. Specifically, FuncStore determines the anticipated number of reads for each data object based on the application's execution logic. During runtime, the read count is decreased with each data access, deleting the data once the final read operation is executed. This approach ensures that storage space is not occupied unnecessarily. To reduce fragmentation of storage space, FuncStore strategically groups data objects with similar expiration times within the same storage unit, facilitating simultaneous data deletion when the time comes. Compared to state-of-the-art ephemeral systems, Pocket and Jiffy, FuncStore can reduce the platform's resource usage by 86.6% and 81.7% respectively, while guaranteeing request latency and throughput.

I. Introduction

Traditional cloud computing abstracts computing resources in data centers to users in the form of virtual servers, greatly streamlining the resource management overhead for developers. However, when a business faces burst loads, the virtual server-centric approach imposes a significant management burden on developers. They must estimate the required number of servers and configurations to handle the increased business load. In contrast, serverless computing has emerged as a novel cloud computing paradigm in recent years, offering a *Function as a Service* (FaaS) abstraction that further simplifies resource deployment and management for developers [1, 2]. In serverless computing, developers only need to upload program code as functions to the cloud platform and pre-define the execution logic of these functions, creating serverless workflows. The cloud platform dynamically starts containers on demand to execute the corresponding program code according to defined function workflows, thus providing the required services [3, 4]. Serverless workflows commonly incorporate parallel branches and data dependencies, and they are typically organized in a logical manner using a *direct acyclic graph* (DAG) representation [5] containing multiple stages while each stage consists of a set of functions with similar or complementary purposes.

Serverless applications lack awareness of instance scheduling and placement, making it challenging to facilitate direct data sharing between functions. A common approach to achieve data sharing is to store the data to be shared in remote storage systems, such as *Amazon Simple Storage Service* (S3) [6], *Amazon DynamoDB* (DynamoDB) [7], and *Amazon ElastiCache* (ElastiCache) [8]. This enables data sharing across various function instances deployed on different nodes within the serverless environment. This separation of storage and computation in serverless computing contributes to exceptional application resilience. Prior research works have focused primarily on aspects such as request latency, operation throughput, storage access interfaces, and user costs associated with storage systems, all from the user's perspective [9–13]. For example, Pocket [11] can automatically configure multiple storage media based on application characteristics such as latency requirements, data volume, maximum concurrency, and maximum bandwidth. The goal is to establish a highly flexible serverless storage system that strikes a balance between I/O operation latency and storage expenses. Locus [14] introduces a machine learning model designed to recommend optimal storage configuration parameters for applications to reduce storage costs while meeting application requirements. Faa$T [15] concentrates on caching strategies for storage systems to efficiently enhance data access speed.

In addition to the factors mentioned above, efficient utilization of storage space is of paramount importance. From the platform's perspective, high utilization implies that valuable storage resources are maximally employed, enabling the platform to accommodate more applications within the same resource constraints. For users, increased utilization of storage resources results in lower unit storage costs, ultimately reducing the overall cost of storage. For this problem, the current state-of-the-art solution is Jiffy [16]. Jiffy recognizes

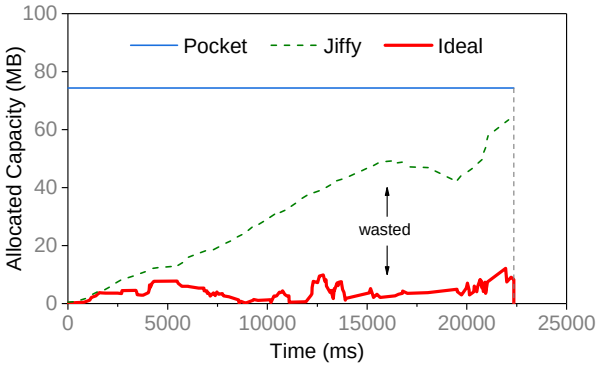


Fig. 1: The size of required memory for compiling *mosh*

that the existing practice of allocating data at the granularity of an entire application leads to significant resource waste. This inefficiency arises because an application's storage requirements can vary widely, sometimes spanning up to two orders of magnitude. Allocating storage based on the average demand adversely impacts application performance, while allocating based on the highest demand leads to underutilization of resources. To remedy this, Jiffy proposes a novel approach to allocate data at the granularity of individual functions. Specifically, allocation of the corresponding storage space is completed when the function performs write operations. Moreover, when writing data, Jiffy employs a time-lease mechanism. If data is accessed before the lease expires, the system extends the lease; otherwise, if the data remains untouched beyond the lease duration, Jiffy deletes the data and reclaims the memory occupied by the data. This approach significantly reduces storage resource idle time because storage space allocation is deferred until the moment when data is produced, and data removal occurs promptly after its utility expires.

Although Jiffy's allocation strategy based on the granularity of functions improves the resource utilization of the storage system to some extent, there remains a notable gap between Jiffy's optimization and the ideal scenario. The ideal scenario means that if the data is no longer being used, then it is deleted immediately to release storage space. As shown in Figure 1, a substantial amount of expired data still occupies a significant portion of the storage capacity. To this end, we introduce FuncStore, a data-centric serverless storage system designed to achieve high resource utilization. The core concept behind FuncStore is the timely deletion of data when its lifecycle ends to prevent unnecessary storage space consumption. Specifically, FuncStore incorporates a *Consistent Monitoring* mechanism, which identifies the number of data accesses based on workflow execution logic and accurately monitors each access operation of the data to effectively be aware of the data lifecycle. To mitigate the issue of fragmented free storage space resulting from data deletions, FuncStore proposes a *Lifetime-aware Memory Management* mechanism, which groups data with similar lifetimes into the same block, facilitating simultaneous reclamation of space within that block.

Our contributions can be summarized as follows:

- We conduct a deep study highlighting the distinction between data lifecycle and storage time. This analysis reveals the suboptimal resource utilization observed in existing serverless intermediate data storage systems.
- We propose FuncStore, a novel system that uses precise data lifecycle monitoring and strategic data placement to significantly improve storage resource utilization.
- We implement and evaluate FuncStore. Compared to state-of-the-art works, FuncStore outperforms in terms of storage resource utilization without increasing data access latency, reducing memory resource usage by 86.6% and 81.7%, respectively. The code of FuncStore is available at: https://github.com/CGCL-codes/FuncStore.

II. Background and Motivation

A. Serverless Computing & Intermediate Data Sharing

Serverless computing has gradually become a new paradigm for cloud computing development due to the ease of development and deployment, extreme application elasticity, and pay-as-you-go characteristics. In a serverless environment, the application is split into multiple stateless functions that are assembled in the form of a workflow to implement complex application logic. During the development phase, developers write function code and use the *workflow description language* (WDL) provided by serverless platforms (e.g., AWS Step Functions [17], Google Workflows [18]) to write the *workflow description script* (WDS) to describe how to compose functions into workflows. WDL supports sequential and parallel execution logic to implement complex application services. A simple WDS example using JSON-based WDL is shown in Figure 2. When deploying an application, the user provides the serverless platform with the written function code and WDS.

Data transfer between functions within a workflow is essential for implementing the internal execution logic of a serverless application. Unlike the input data and output results of an application, this data is generated by specific functions and used by others during the execution of the application. Consequently, it is often referred to as "intermediate data", and the process of transferring data between functions is termed "intermediate data sharing". As shown in Figure 3, these various patterns of data transfer, include the following:

- **Pass**: In this pattern, data objects are passed exclusively between two functions.
- **Broadcast**: This pattern involves a function passing all its data objects to multiple functions.
- **Scatter**: In this pattern, a function passes different data objects to different functions.
- **Aggregate**: The pattern occurs when multiple functions collectively pass data objects to a single function.

The current serverless research tends to represent the workflow described by WDS as a DAG to describe the relationship between functions. In this case, the DAG representations corresponding to the above data transfer modes are shown in Figure 4.

Since serverless applications are not aware of function scheduling and placement, it is difficult to address functions

```
{
    "app": "demo",
    "workflow_id": 1704038400,
    "category": "image-process",
    "spec": {
        "split_stage": {
            "stage_id": 1,
            "functions": [
                {
                    "start": true,
                    "name": "split_function",
                    "input": ["raw_object"],
                    "output": ["intermediate_object_00", "
                        intermediate_object_01"],
                    "next": ["process_function_00", "
                        process_function_01"]
                }
            ]
        },
        "process_stage": {
            "stage_id": 2,
            "functions": [
                {
                    "start": false,
                    "name": "process_function_00",
                    "input": ["intermediate_object_00"],
                    "output": ["result_object_00"],
                    "next": []
                },
                {
                    "start": false,
                    "name": "process_function_01",
                    "input": ["intermediate_object_01"],
                    "output": ["result_object_01"],
                    "next": []
                }
            ]
        }
    }
}
```

Fig. 2: An example of JSON-based workflow description script

directly, which presents a significant challenge to transferring data directly between functions. In addition, even if *network address translation* (NAT) or other techniques can be used to achieve direct data transfer between functions, the transfer process requires that the lifecycle of the sending and receiving functions overlap, thus bringing limitations to the flexibility of function execution. Consequently, one way to accomplish intermediate data transfer is by utilizing the remote storage system. Specifically, the data-sending function writes the data into the remote storage system, and the data-receiving function reads the corresponding data from it. Note that the size of intermediate data can vary widely, from a few bytes to several hundred megabytes [11]. Moreover, Microsoft Azure's publicly available dataset reveals that a significant majority, specifically 80%, of the data objects are smaller than 12KB in size [15].

B. State-of-the-art Works

Traditional cloud storage services are widely employed in today's serverless environments to facilitate the sharing of intermediate data among functions. Examples of such services include S3 [6] and DynamoDB [7]. However, these services have some notable limitations. In the case of S3, the read latency for 1KB of data is approximately 12ms, posing a substantial overhead for serverless applications. DynamoDB

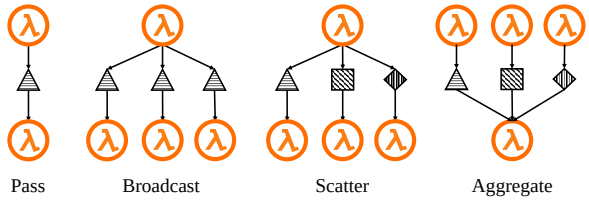


Fig. 3: Data passing patterns in serverless workflow

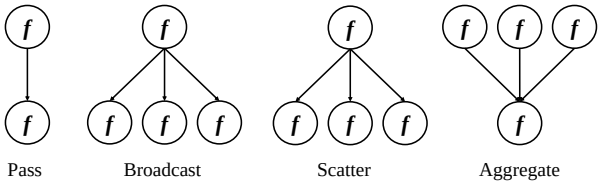


Fig. 4: Data passing patterns represented by DAG

imposes a size limit of 400KB for a single item, while the size of serverless intermediate data can reach several hundred megabytes. This restriction significantly narrows the applicability of DynamoDB in various scenarios. Another category of commonly used ephemeral storage solutions is in-memory storage service, exemplified by ElastiCache and Redis [19]. These services harness the speed of memory to store intermediate data, meeting the demanding read and write performance requirements. However, this approach comes with higher storage costs than the previous two options. Furthermore, ElastiCache and Redis require users to explicitly configure storage nodes and manually select resources when scaling up, substantially increasing the operational cost for users.

Various research efforts have focused on optimizing these systems from different angles to tackle the challenges associated with existing cloud storage solutions in serverless computing. Pocket [11] is a distributed storage system that takes advantage of multiple storage media and can automatically scale the cluster capacity up and down based on resource usage. Pocket intelligently selects the appropriate storage media, such as DRAM, Flash, or HDD, to store intermediate data, aligning the storage system's performance with the data storage cost. For individual applications, Pocket incorporates a *hint* mechanism for rightsizing resource allocation. This mechanism utilizes user-provided application information, including latency requirements, data volume, maximum concurrency, peak bandwidth, and more, to estimate the required storage capacity across different storage media when the application is submitted. During application execution, when a function needs to write data, Pocket allocates storage space within the reserved storage resources. At the end of the application's execution, Pocket releases the allocated storage space. Furthermore, Pocket improves the *put()* and *get()* interfaces by introducing a flag parameter. This parameter allows users to specify the behavior of data read and write operations, enabling actions such as deleting specific data at the end of the function.

Although Pocket effectively combines multiple storage me-

dia to balance storage performance and cost, and implements auto-scaling of storage resources, it employs application-level resource requests, which can lead to reduced resource utilization. Specifically, in this application-level resource allocation method, if memory is allocated according to the average demand of the application, data exceeding the allocated memory amount will be spilled onto the SSD or HDD when the application is running, resulting in higher latency and lower bandwidth, adversely affecting application performance. In contrast, if memory is allocated based on the peak demand of the application, memory utilization may remain suboptimal. To address this challenge, Jiffy introduces a function-level resource allocation policy. In this approach, Jiffy organizes the memory of storage nodes into fixed-size memory blocks and dynamically establishes mapping relationships with different functions to enable efficient reuse of memory resources. Furthermore, Jiffy implements a lease mechanism to quickly reclaim the storage space for data. In Jiffy, the system assigns a lease to a data block when it establishes a mapping relationship with a function, and then initiates a timer. If the data in the block is accessed again before the lease expires, the system updates the lease for that block. However, if the data remains untouched beyond the duration of the lease, the system deletes the data and reclaims the corresponding memory block. This mechanism ensures the efficient utilization of storage resources while accommodating varying data access patterns.

C. Motivation

The fundamental challenge facing current serverless ephemeral storage systems lies in their low resource utilization. This issue stems from the disparity between the expiration time of intermediate data and the end time of the application. Deleting data after the application is completed leads to a situation where the storage duration of the data exceeds its actual effective period. Existing cloud storage services (e.g., S3 [6], DynamoDB [7], ElastiCache [8], and Redis [19]) are not explicitly tailored for serverless computing and do not align with the serverless pay-as-you-go model. These services necessitate users to pre-request all resources and incur charges based on their initial requests rather than their actual usage. This approach can significantly reduce the efficient utilization of storage resources, particularly given the variable resource consumption associated with serverless workloads.

In existing work, Pocket [11] not only handles the deletion of intermediate data at the end of the application, but also offers a *get()* interface that allows users to delete data after it has been read, thereby freeing up storage space. However, this *"delete after read"* approach is limited to data that is read only once. For data that undergoes multiple reads within the application, it becomes impractical for the user to determine which function should implement the *"delete after read"* operation, rendering it non-universal. This contradicts the serverless paradigm, where users typically focus solely on the code logic without delving into storage management intricacies. In Jiffy [16], the responsibility of releasing expired data storage space is handled by the lease mechanism. Although this mechanism autonomously reclaims resources for all data without necessitating explicit user intervention, it may not be pinpoint accurate in determining the exact expiration time of data. Since Jiffy operates with a global lease time, a trade-off arises. To support a diverse range of serverless applications, the lease time must be set long enough to prevent an application's data from expiring prematurely before it can be accessed. However, a longer lease time means that, for short-lifecycle data, it may still occupy memory for a period after it has expired.

To this end, a straightforward idea is to design a data-centric serverless storage system, which should allocate storage space when data is generated and reclaim the corresponding storage resources at the end of data lifecycle, to achieve high storage resource utilization. Additionally, from the perspective of application performance, low-latency DRAM is suitable as the medium for ephemeral storage systems, which can minimize the data transfer overhead of serverless applications, and has been selected by Jiffy.

Key challenges. Through an in-depth analysis of existing storage systems, we identify two key challenges to the achievement of a highly resource-efficient serverless ephemeral storage system:

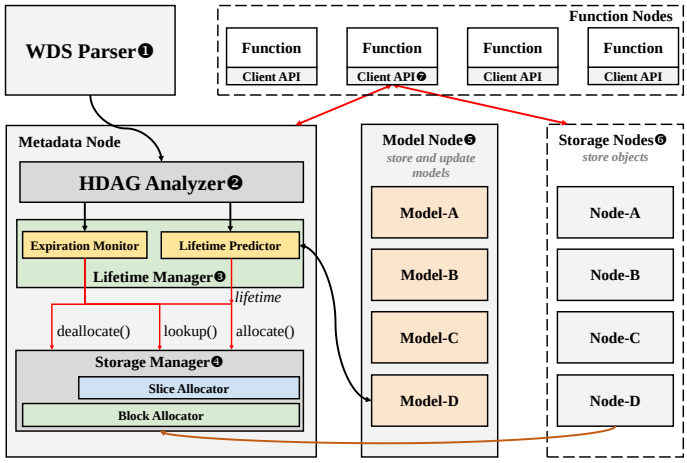
- **How to determine the expiration time of data objects so that the storage system can reclaim them in time to free up storage space?** Existing storage systems usually release data objects until applications are finished, which fails to free unused data objects in time and wastes storage space. As for indicating when the data object should be deleted, it is difficult for users, which mainly focus on the code logic.
- **How to place data objects so that there are as few idle fragment spaces as possible?** If data objects with different lifecycles are stored in the same block, there will be free fragment spaces in the block when the data objects are deleted one by one. These free fragments are difficult to be reused due to their unmatched size.

III. Design

FuncStore is designed as an ephemeral storage system with high resource utilization for efficient sharing of intermediate data in serverless applications. FuncStore improves resource utilization by introducing two optimizations: 1) Firstly, FuncStore accurately monitors the expiration time of data and promptly releases the memory space of expired data; 2) Secondly, it places small data objects with similar lifetimes in the same memory block, reducing intra-block fragmentation.

A. System Architecture

We show the FuncStore system architecture in Figure 5. FuncStore is divided into two parts: the front-end and the back-end. The front-end includes *workflow description script parser* (WDS Parser) and *Client Library*, and the back-end includes *Metadata Server*, *Model Server*, and *Storage Server*. When users deploy serverless applications, they provide WDS



❶**WDS Parser**	• Parse a WDS into HDAG • Upload HDAG to Metadata Server
❷**HDAG Analyzer**	• Extract lifetime features and read count for data objects and pass them to Lifetime Manager❸
❸**Lifetime Manager**	• Monitor data expirations based on read count • Predict data objects' lifetime by communicating with Model Server❺ and store them • Collect data objects' real lifetime and report to Model Server for model tuning
❹**Storage Manager**	• Manage memory registered from Storage Servers❻
❺**Model Server**	• Store models for different application categories and provide lifetime prediction service • Tune models continuously based on metrics from Lifetime Manager❸
❻**Storage Servers**	• Register/deregister memory space to/from Storage Manager❹ • Store data objects and serve read/write requests from Client Library❼
❼**Client Library**	• Access Metadata Server and Storage Servers to put/get data objects

Fig. 5: Architecture overview of FuncStore

to WDS Parser. WDS Parser parses WDS into a generic intermediate representation named *heterogeneous directed acyclic graph* (HDAG) and uploads HDAG to *Metadata Server*. *Metadata Server* internally implements HDAG *Analyzer*, *Expiration Monitor*, *Lifetime Predictor*, and *Storage Manager*. Among them, HDAG *Analyzer* analyzes each HDAG uploaded to *Metadata Server*, extracts information on intermediate data involved in HDAG, and provides the information to the *Expiration Monitor* and *Lifetime Predictor*. The former monitors the expiration time of intermediate data during application execution, the latter and *Model Server* work together to predict the lifetime of intermediate data to optimize memory allocation. The *Model Server* stores prediction models for various application categories, receives running metrics from the *Metadata Server*, and continuously adjusts the corresponding models. The *Storage Server* stores the relevant data based on the address provided by the *Storage Manager*. When the serverless application is running, its functions access the *Metadata Server* and *Storage Server* by using the API provided by the *Client Library*, read and write intermediate data objects to achieve data sharing.

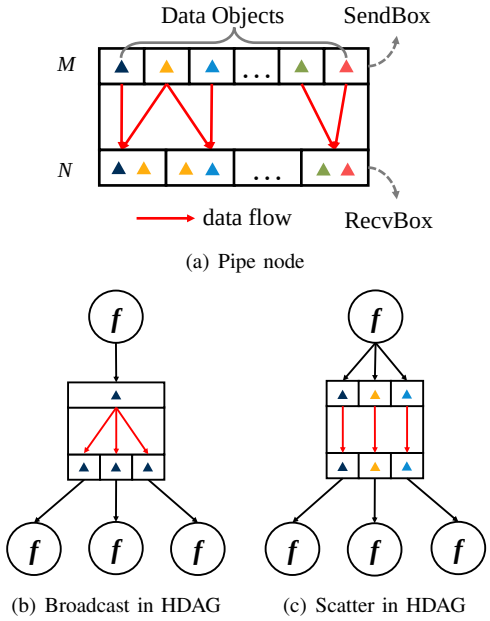
B. Consistent Monitoring

We design a method that effectively tracks the expiration time of the intermediate data, enabling the prompt release of the memory occupied by the data objects. One possible approach is to determine the access count of a data object by analyzing the workflow, and then create a counter for the object with an initial value equal to its access count. The counter is decreased by one each time the data object is accessed, until it reaches zero, indicating that the data object has expired. However, the current widely used intermediate representation of workflows, which is function DAG, fails to clearly depict the data transfer patterns between functions, leading to the storage system's inability to accurately retrieve the read times of data objects under different patterns. As illustrated in Figure 3, the **Broadcast** mode refers to an upstream function passing a data object to three downstream functions, resulting in a read count of 3 for the data object. On the other hand, the **Scatter** mode involves an upstream function passing three data objects to three downstream functions, resulting in a read count of 1 for each data object. Both transfer patterns have the same representation in the function DAG as shown in Figure 4. Due to the limited expressiveness of the function DAG in Figure 4, determining the read times of data objects passed between functions is challenging. Furthermore, the basic counting-based approach is susceptible to the impact of function crashes and retries, potentially resulting in counting errors. Detailed explanations regarding this matter will be provided in the following sections.

To address the aforementioned issues, we propose a mechanism called *Consistent Monitoring* to accurately monitor the expiration time of data objects. *Consistent Monitoring* comprises a novel intermediate representation of the workflow named *heterogeneous directed acyclic graph* and a new counter-update strategy named *Deferred Commit*. The former effectively distinguishes between different data transfer patterns, ensuring the correct capture of data object read times. The latter ensures the correctness of counter updates.

*1) **Heterogeneous Directed Acyclic Graph**:* To effectively express the differences between different modes, FuncStore proposes a new workflow intermediate representation, *heterogeneous directed acyclic graph* (HDAG), which clearly expresses the data transfer modes between functions. Specifically, in addition to function nodes, HDAG introduces a pipe node that connects two function nodes to represent the data transfer relationship between functions. As shown in Figure 6(a), each pipe node contains three parts: 1) the data object, which represents the different data generated by the function node; 2) the data send box (*SendBox*), which corresponds one-to-one to the data objects; 3) the data receive box (*RecvBox*), which corresponds one-to-one to the function nodes that need to read data. *SendBox* and *RecvBox* are connected by arrows to indicate the flow of data. Assume that a function node will write `M` data, and there are `N` function nodes that will read these data, then `M` *SendBox*es and `N` *RecvBox*es will appear in the

corresponding pipe node. Secondly, based on the description of data transfer in the workflow, the mapping relationships between *SendBox*es and *RecvBox*es are established. By adding pipe nodes, HDAG can clearly express the data transfer situations between different function nodes. The **Broadcast** and **Scatter** modes are represented in HDAG as Figure 6(b) and Figure 6(c), respectively. In the figure, the difference in data transfer between the two modes is clearly distinguished.



(a) Pipe node

(b) Broadcast in HDAG (c) Scatter in HDAG

Fig. 6: Overview of HDAG

2) ***Deferred Commit****:* In the counting-based data expiration time monitoring mechanism, counters of data objects are stored on the metadata node. Each time a data object is read, its count number is reduced by one. This is reasonable for applications where every function executes normally. However, the actual execution of serverless applications can be more complex.

Serverless functions tend to run in separate containers and may be scheduled on arbitrary compute nodes. Due to network failure, memory overselling, system cleanup, and other factors on the compute node where the function resides, there is a risk that the function may crash during execution. To avoid application execution failure caused by function crashes [20–23], current serverless platforms often provide fault tolerance against function execution crashes through the function re-execute mechanism. Specifically, when a running function crashes due to the above reasons, the workflow platform restarts a new container and executes the function code. However, in this case, the crashed function may have read partially dependent intermediate data, which causes the counters of these data objects to be updated. At this point, re-executing the function can lead to counting errors and premature deletion of data objects, which in turn can cause the application to fail. The root cause of these problems is that the storage system is not aware of the execution state of the function and therefore treats all read operations as valid. Thus, once the expected number of reads is reached, the data is deleted.

To this end, FuncStore proposes a *Deferred Commit* mechanism. Specifically, in this mechanism, the update of the count value is divided into two phases: the update phase and the commit phase. When a function reads intermediate data from FuncStore, the count value of the data object is not updated immediately, but enters the update phase. In the update phase, FuncStore adds an operation record, which is in the format *<object name>*, to the *Commit Buffer* allocated for each function. When adding an operation record to the *Commit Buffer*, FuncStore first determines whether the same operation record already exists in the *Commit Buffer*, and adds it only if the same operation record does not exist in the *Commit Buffer*, otherwise it does nothing. Since the operation records in the *Commit Buffer* of a function are not duplicated, it is guaranteed that reads of data by repeatedly executed functions will only be recorded once.

Since the function follows the execution logic of read dependencies, data processing, and results writing back [10, 24–26], when FuncStore receives a write request from the function, it means that the function has been successfully executed, and the update of the count value enters the commit phase. In the commit phase, FuncStore traverses the operation records in the *Commit Buffer* of the function, and decreases the count value of involved data objects by one, to complete the update of the count value corresponding to all read operations of the function. When the function has more than one output result, FuncStore gets the object name of the last data from the WDS, and determines whether it is the last write by matching the name of the object when function performs write operations, and only the write request for the last result will cause the count update commit.

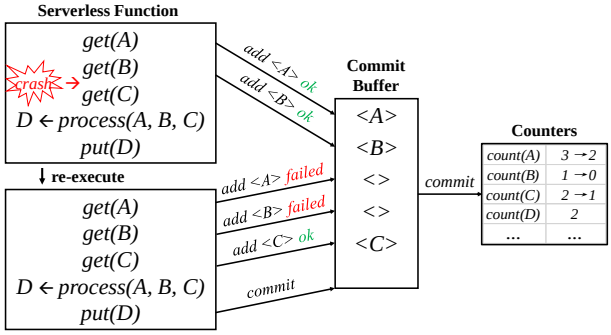


Fig. 7: An example of deferred commit

For example, Figure 7 shows that the function first reads the data objects A, B, and C, processes the data, and then writes back the result D. The function first reads the dependent data A and B, and while reading the data it successfully adds two operation records to the *Commit Buffer*. If the function crashes at this point, the workflow platform will restart a container and execute the function again. In this run, the function tries to add two operation records to the *Commit Buffer* while reading data A and B. The addition fails because the operation records for

A and B already exist in the *Commit Buffer*. After that, the function follows the above logic to read the data object C, process data, and generate the result D. Finally, the function writes D back to FuncStore, at which point it traverses the operation records in the *Commit Buffer* and reduces the count value of the data objects A, B, and C.

C. Lifetime-aware Memory Management

To address the problem of low resource utilization caused by block-granular memory allocation in small data-intensive scenarios, our basic idea is to provide more fine-grained memory management for small data based on block-granular memory allocation. Considering that the lifetimes of intermediate data generated by the applications vary greatly, FuncStore proposes a *Lifetime-aware Memory Management* mechanism. The key idea of this mechanism is to predict the lifetime of intermediate data and store data with similar lifetimes in the same memory block. The memory block can be released after all the data in the block expires.

1) ***Lifetime-based Memory Allocator****:* As shown in Figure 8, we design a *Lifetime-based Memory Allocator* that manages the storage of data objects. Initially, the allocator partitions the memory across all storage nodes into fixed-size memory blocks, such as 1MB, 16MB. When the size of the data object is smaller than the size of a memory block, the *Slice Allocator* selects a memory block and allocates a continuous memory space with the requested size from the memory block. This allocated continuous memory space for a data object is a memory slice.

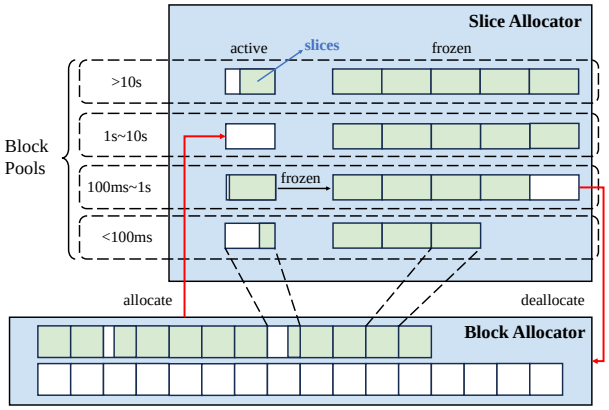


Fig. 8: Lifetime-based memory allocator

In the *Slice Allocator*, a *Block Pool* is comprised of multiple memory blocks, and different *Block Pool*s serve data objects with different lifetimes. To allocate memory for a data object, the *Slice Allocator* allocates a slice from the *Block Pool* that contains data objects with similar lifespans, which are predicted by the *Lifetime Predictor*. A *Block Pool* includes an active block, multiple frozen blocks, and some free blocks. An active block is the memory block that *Block Pool* is serving slice allocation requests. To allocate a slice, the system allocates a continuous memory space from the active block. When the remaining space in the active block can not meet an allocation request, the system changes the active block to a frozen block, adds the frozen block to the frozen block queue, and changes a free block to the active block. When all data objects in a frozen block expire, this frozen memory block is changed to become a free block.

Specifically, each block has two attributes: *offset* and *fragment size*. The *offset* refers to the currently assigned offset, while the *fragment size* refers to the space in the block that is taken up by expired data objects. When a slice is allocated, the *offset* of the block is added by the size corresponding to the slice. When the count of a data object is reduced to zero through the *Consistent Monitoring* mechanism, it means that the data object has expired and its corresponding slice can be released. However, FuncStore does not immediately release the slice, but only increases the *fragment size* of the block by the slice size. When the *fragment size* and *offset* of the block are equal, all data in the block have expired and FuncStore releases the entire memory block.

When the size of the data object exceeds the size of the memory block, FuncStore directly allocates an appropriate number of free memory blocks from the registered memory blocks through the *Block Allocator*. During allocation, it will try to allocate memory blocks on different nodes to facilitate the use of parallel acceleration.

2) ***Data Lifetime Prediction****:* To reduce the fragmentation caused by the expiration of some data in the block and further improve the resource utilization in the block, it is critical to accurately predict the lifetime of the data and select the appropriate block for the data. In traditional applications, it is difficult to analyze the lifetime of the storage space requested by the application due to the lack of understanding of information related to the stored data. However, in serverless applications, the dependencies between intermediate data and functions are described in the workflow provided by the user, allowing us to use this information to predict the lifetime of the data. In addition, the feature of deploying once and calling multiple times enables us to collect historical information about data access and to apply data-driven methods to predict data lifetime.

In a serverless workflow, many factors affect the lifetime of intermediate data. Intuitively, the lifetime of a data object is related to the stage in the workflow of its writing function, the number of times the data object is read, and the total number of data objects on which the data object reading functions depend. We represent these factors with F_{stage}, T_{read}, and $\mathrm{N}_{functions}$, respectively. In addition, we also consider the paths between the function to write a data object and the function to read this data object, as these paths affect the lifetime of the data object. Specifically, we use two indicators, D_{stage} and M_{path}, to describe the impact of these paths, respectively. D_{stage} represents the maximum value of the stage interval difference between the stage when the function writes the data and the stages when the functions read the data, while M_{path} represents the maximum number of function nodes on all potential impact paths. The potential impact paths refer to paths from functions that are in the same stage as the data

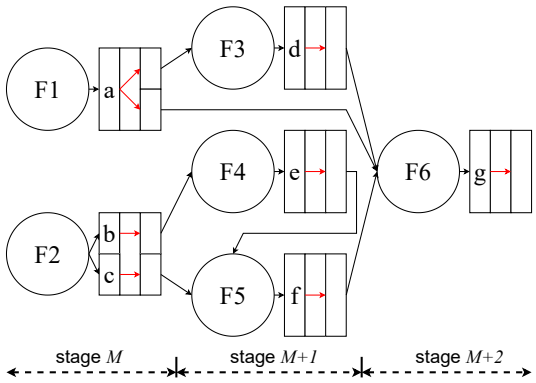


Fig. 9: HDAG example for feature extract

writing function to all functions that read the data object.

Figure 9 demonstrates the process of extracting features for the data object a in the given HDAG. The feature F_{stage} of a is M since the function F1 that writes a is at stage M. As functions F3 and F6 read a, the feature T_{read} of a is two. Function F3 depends on one data object, while function F6 depends on three data objects, resulting in a feature $N_{functions}$ of four for a. Function F1 writes a at stage M, while functions F3 and F6 read a at stages $M+1$ and $M+2$, respectively. Hence, the feature D_{stage} of a is two. The potential impact paths for a include *<F1, F3>*, *<F1, F6>*, *<F1, F3, F6>*, *<F2, F5, F6>*, and *<F2, F4, F5, F6>*, with a maximum path length of four. Consequently, the feature M_{path} of a is four.

We employ the feature extraction algorithm, as delineated in Algorithm 1, to extract features of data objects from the HDAG associated with the serverless workflow. Algorithm 1 initially traverses the linked list of pipe nodes in the HDAG, locates the pipe node and *SendBox* where the data object of interest is located (lines 1-3), and reads the stage F_{stage} and the number of read functions T_{read} (lines 4-5). It then iterates through all *RecvBox*es and function nodes corresponding to the *SendBox*, calculating features $N_{functions}$ and D_{stage} (lines 7-10), and utilizes Algorithm 2 to identify all potential impact paths and obtain the maximum path length (lines 12-13). Algorithm 2 recursively searches for potential impact paths from the read functions of the data object to the stage of the write function in a depth-first manner. For each current node, the parent node is initially added to the current path (lines 1-3). If the parent node is in the same stage as the write function, the current path is added to the potential impact paths (lines 4-5); otherwise, the parent node is considered the new current node, and the search continues (lines 6-7).

Algorithm 1: Extract Features

Input: $fList$ - function node list in hdag ;
$pList$ - pipe node list in hdag ;
$current$ - object for extracting features ;
Output: $features$ - extracted features of $current$;

1 **for** $pNode$ *in* $pList$ **do**
2 **for** $SendBox$ *in* $pNode.SendBoxes$ **do**
3 **if** $SendBox.object == current$ **then**
4 $F_{stage} := pNode.parent\_fNode.stage$
5 $T_{read} := SendBox.RecvBoxes.size$
6 $N_{functions} := 0$, $D_{stage} := 0$, $M_{path} := 0$
7 **for** $RecvBox$ *in* $SendBox.RecvBoxes$ **do**
8 $f := RecvBox.child\_fNode$
9 $N_{functions} += f.obj\_dependencies\_num$
10 $D_{stage} := max(D_{stage}, f.stage - F_{stage})$
11 $paths := \{\}, cur\_path := \{\}$
12 $FindPath(F_{stage}, f, cur\_path, paths)$
13 $M_{path} := max(M_{path}, paths.longest.len)$
14 $return(F_{stage}, T_{read}, N_{functions}, D_{stage}, M_{path})$

Algorithm 2: Find Path

Input: $stage\_num$ - stage number of object writer function;
$fNode$ - object reader function node;
$currentPath$ - current searching path;
$Paths$ - paths between stage $stage\_num$ and $fNode$;
Output: $currentPath$ - current searching path;
$Paths$ - paths between stage $stage\_num$ and $fNode$;

1 **for** $pNode$ *in* $fNode.parent\_pNodes$ **do**
2 $parent := pNode.parent\_fNode$
3 $currentPath.add(parent)$
4 **if** $parent.stage$ == $stage\_num$ **then**
5 $Paths.add(currentPath)$
6 **else if** $parent.stage > stage\_num$ **then**
7 $FindPath(stage\_num, parent, currentPath, Paths)$
8 $currentPath.remove(parent)$

In each category of applications evaluated, we select 20 applications from the category and run them 50 times. The recorded lifetimes of the generated intermediate data serve as the *lifetime_ms* of data objects. For each data object, we generate 50 items in the format *<*F_{stage}*,* T_{read}*,* $N_{functions}$*,* D_{stage}*,* M_{path}*, lifetime_ms>*, constituting the dataset for each application category. We allocate 80%, 10%, and 10% of the dataset for training, cross-validation, and testing, respectively.

We adopt a fully connected neural network as a proof-of-concept. Note that the exact choice of method or model is not our concern, we currently use neural network as a reference solution because it is widely used in serverless environments to optimize different system metrics [27–30]. Our model comprises three hidden layers, each with 64 neurons, using ReLU as the activation function. The entire neural network outputs an integer value representing the lifetime in milliseconds. For each data object, we calculate the mean and standard deviation of the actual lifetime more than 50 times. Using a normal distribution, we calculate the 95% confidence interval, considering predictions within this interval to be accurate. All models for the evaluated application categories achieve an accuracy greater than 90% in the test set. The trained models are deployed on the *Model Node* for lifetime prediction.

IV. Implementation

We have implemented the FuncStore prototype, which comprises a total of 7006 lines of code. This includes 6399 lines of C++ code dedicated to the implementation of key components

such as the WDS Parser, *Client Library*, *Metadata Server*, and *Storage Server*. Additionally, we have incorporated 607 lines of Python code to construct the *Model Server*. Our implementation is based on the Apache Thrift framework [31] to facilitate communication between these aforementioned modules. Within the *Metadata Server*, we employ in-memory hash tables to store counters for data objects, utilize an in-memory linked list as the *Commit Buffer* for each function, and facilitate rapid communication between different components through direct function calls. To ensure high performance and concurrent operation support across different modules, we opt to employ libcuckoo [32] as a concurrent hash table and lock-free concurrent queue [33] as a concurrent queue. This decision is motivated by the inherent limitations of unordered_map and queue in the C++ *Standard Template Library* (STL), which does not offer built-in support for thread-safe concurrent operations. Furthermore, in pursuit of enhanced support for concurrent client sessions, we use asynchronous framed I/O techniques within the *Storage Server*. This approach enables us to efficiently handle a larger number of concurrent clients, enhancing the overall performance of the system.

V. Evaluation

A. Experimental Setup

For the evaluation, all ephemeral storage systems are deployed on *Amazon Elastic Compute Cloud* (Amazon EC2). We utilize two types of virtual servers, namely *c5.xlarge* and *r5.xlarge*, for the *Metadata Node* and *Storage Node* of the storage system, respectively. Additionally, a *c5.xlarge* machine is selected as the *Model Node* in FuncStore. Note that all these machines are deployed within the same *Virtual Private Cloud* (VPC) to enable seamless interconnection. AWS Lambda serves as the FaaS platform for executing serverless functions, which are also deployed within the same VPC.

Concerning the configuration of the storage system, the block size plays a crucial role in systems like Pocket, Jiffy, and FuncStore. In essence, the block size significantly impacts storage efficiency. Larger block sizes result in increased space wastage due to the block-based granularity memory management policy, especially for data smaller than the block size. This inefficiency arises because a block can store only one data object, and larger blocks lead to more wasted space within the block for smaller data. Conversely, smaller block sizes result in greater metadata overhead, consuming more memory in metadata nodes. Meanwhile, smaller block sizes also involve more network transfer round trips. This, in turn, hampers the performance of reading and writing data due to the increased software stack processing overhead. To determine an optimal block size, we conduct performance evaluations on ephemeral storage systems using varying block sizes. Figure 10 illustrates the read and write latency of data objects across different block sizes. Notably, the access latency decreases as the block size increases, with a minimal latency drop observed when the block size surpasses 1MB. To strike a balance between ensuring efficient read and write latency for large data objects and minimizing intra-block space wastage, we select 1MB as the default block size for subsequent experiments. This choice guarantees optimal read and write performance while mitigating space wastage associated with block granularity allocation.

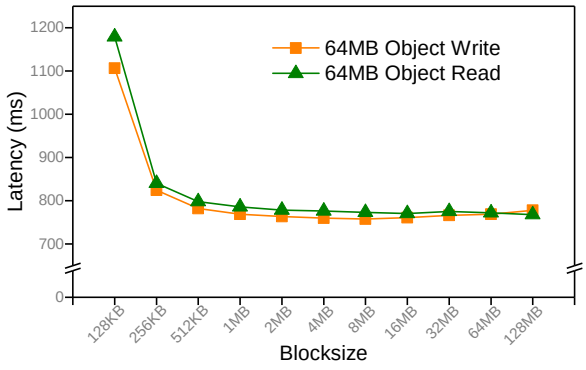


Fig. 10: Latency of read/write for 64MB object

Moreover, in the case of Jiffy, determining the appropriate lease time is also important. A short lease time might result in premature data deletion before it becomes invalid, while an excessively long lease time can lead to extended storage durations, reducing the storage system's resource utilization. To address this challenge, we set the lease time as the maximum value derived from all data object lifecycles in all experiments. To achieve this, we proactively run the application and record all data lifecycles. Subsequently, the longest recorded lifecycle among all data objects is selected as the lease time. This approach ensures that none of the data is evicted prematurely, guaranteeing optimal data retention without unnecessary storage prolongation.

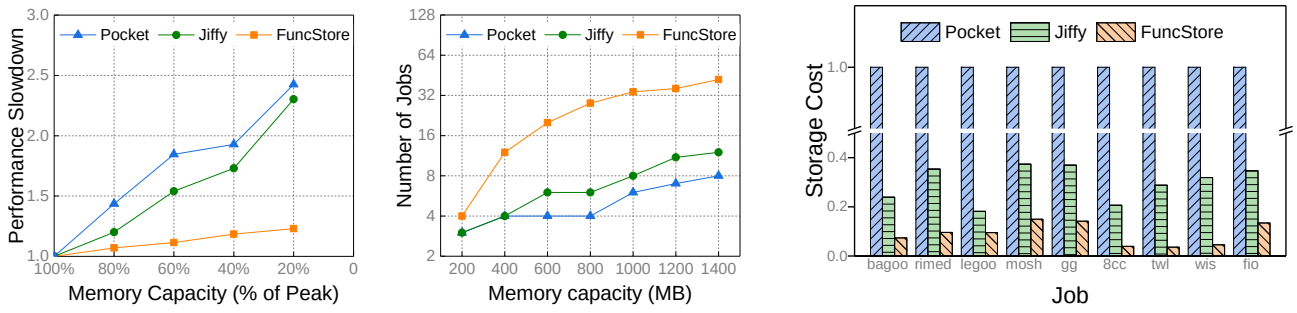
TABLE I: Applications from different categories as workloads

Category	Job	Passing Pattern
Video Encode	April_09_brush_hair_u_nm _np1_ba_goo_0 (bagoo)	Pass, Broadcast, Scatter, Aggregate
	Aussie_Brunette_Brushing_Hair _II_brush_hair_u_nm _np1_ri_med_3 (rimed)	Pass, Broadcast, Scatter, Aggregate
	Aussie_Brunette_Brushing_Hair _II_brush_hair_u_nm _np2_le_goo_1 (legoo)	Pass, Broadcast, Scatter, Aggregate
Compile	mosh	Pass, Aggregate
	gg	Pass, Aggregate
	8cc	Pass, Aggregate
Word Count	thewasteland (twl)	Pass, Scatter, Aggregate
	whatisserverless (wis)	Pass, Scatter, Aggregate

As for workloads, we choose eight widely-used applications from three distinct categories common to serverless applications [25, 34, 35], as outlined in Table I.

B. Benefits of FuncStore

In this section, we empirically demonstrate the advantages of FuncStore in job execution performance, job throughput, and storage cost when compared to Pocket and Jiffy.



(a) Job Performance (b) Job Throughput (c) Job Storage Cost

Fig. 11: Benefits of FuncStore

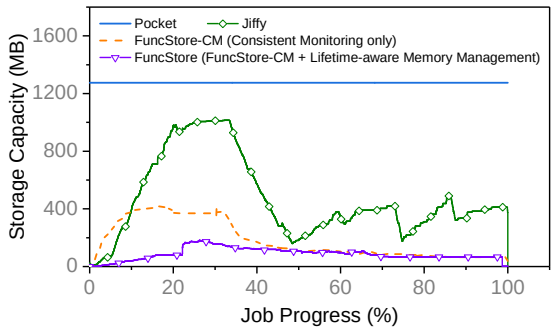
Initially, we evaluate the runtime performance of the workload under varying memory capacities. Specifically, we configure the available memory resources of the storage system to various fractions of the peak memory requirement necessary for executing all the jobs listed in Table I. Subsequently, we measure the completion time of the workload. Note that if there were no available memory resources during workload execution, the storage system stored subsequent intermediate data on the slower SSD. The experimental results are shown in Figure 11(a).

As the available memory diminishes, the completion time of the workload increases across all three storage systems. However, when the available memory of the storage system decreases, FuncStore exhibits a lower growth in workload completion time compared to Pocket and Jiffy. For instance, when the available memory is set to 60% of the workload's peak memory usage, the workload completion time with Pocket and Jiffy increases to 1.85 and 1.54 times the completion time under peak memory, respectively. In contrast, with FuncStore, the workload completion time only reaches 1.12 times. If the memory availability is further reduced, the performance gap among the three storage systems becomes more evident. When the memory availability is only 20% of the peak, FuncStore's workload completion time is only 1.23 times the peak, while Pocket and Jiffy's workload completion time is 1.97 and 1.87 times that of FuncStore, respectively. This highlights FuncStore's ability to promptly release memory occupied by expired intermediate data, enabling more efficient reuse of limited memory. In contrast, the other two storage systems are compelled to store a substantial amount of data on the slower SSD, consequently prolonging the workload completion time.

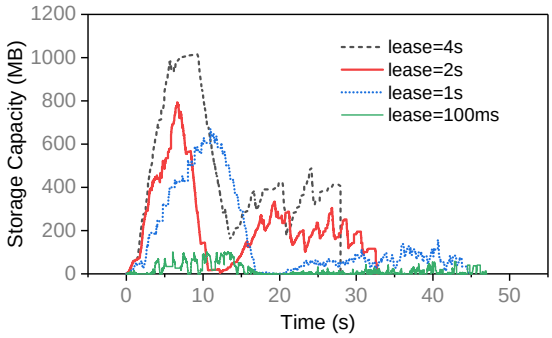
Additionally, we conduct tests to determine the maximum number of jobs that can be executed concurrently on different storage systems given a specific amount of available memory. We incrementally add jobs to the test workload in the order that they are listed in Table I until the execution of the test workload encounters a failure. Note that if the cumulative memory usage of the test workload exceeds the available memory, the workload execution is considered a failure, signifying the need for more memory to support the simultaneous execution of these jobs. The results of this test are illustrated in Figure 11(b). When the available memory on the system is 200MB, FuncStore can support the concurrent execution of 4 jobs, whereas Pocket and Jiffy can only support 3 jobs concurrently. If the available memory is increased to 400MB, FuncStore can support the concurrent execution of 12 jobs, while Pocket and Jiffy can only support 4 jobs. In essence, FuncStore, as an ephemeral storage solution, reduces the memory usage of each job, enabling the support of more jobs compared to Pocket and Jiffy when the same amount of memory is utilized.

Lastly, we assess the storage cost of running the jobs detailed in Table I. Specifically, we execute each job without limiting memory availability and record the real-time memory usage with a granularity of 1 ms. We additionally test an I/O-intensive application, *lambda-fio*, to illustrate that FuncStore's optimization of intermediate data storage is not affected by I/O during the function execution phase. *Lambda-fio* uses *fio* to perform a large number of I/O operations to test the performance of functions' local storage, and processes the test results. It includes two data transfer modes: **Pass** and **Aggregate**. Since the application cost in absolute dollars is affected by the amount of data in the application itself and the frequency of application calls, which makes the benefits expressed in dollars vary greatly between applications. At the same time, current cloud storage does not support serverless' pay-as-you-go billing philosophy very well. ElastiCache, for example, either charges for resources on-premises or supports coarse-grained billing on a per GB-hour basis. We compare the relative costs of different storage systems using the product of the amount of memory used by the application and the time at millisecond granularity, which provides a more focused and accurate representation of the differences between different storage systems. The comparisons of storage costs for different applications using the three storage systems are depicted in Figure 11(c). Compared to Pocket, Jiffy significantly reduces the storage cost for workload runs, with FuncStore further reducing the storage cost. When using Jiffy and FuncStore as ephemeral storage, the average storage cost is 29.7% and 9.1% of using Pocket, respectively.

To give a sense of the magnitude of the benefits, we use *gg* as an example to illustrate. The *gg* has a medium-sized intermediate data volume, which is a few hundred megabytes. We assume that the frequency of *gg* calls is 1000 times per day,



(a) Systems comparison: Pocket, Jiffy, and FuncStore



(b) Jiffy with different leases

Fig. 12: Understanding FuncStore benefits

which happens moderately in cloud vendor workloads [36]. We opt for ElastiCache in the *ap-east-1* region, priced at $0.166 per GB-hour, which we convert to a per MB-second rate. Utilizing the product of the capacity of intermediate data generated during *gg*'s running and the execution time, coupled with the per MB-second price and a daily invocation frequency of 1000 times, we calculate the costs of employing Pocket, Jiffy, and FuncStore for this application to be $2.06/day, $0.76/day, and $0.29/day, respectively.

C. Understanding FuncStore Benefits

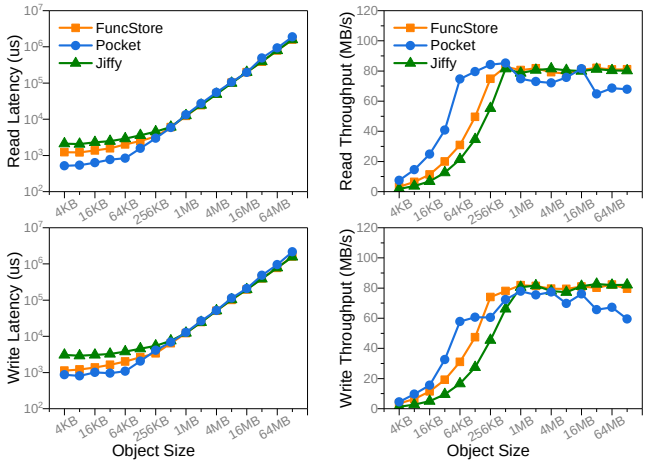
FuncStore enhances resource utilization through a *Consistent Monitoring* mechanism and a *Lifetime-aware Memory Management* policy. In this subsection, we will illustrate resource changes during application execution through experiments. We still choose the applications in Table I as the test workload. Subsequently, we execute these applications concurrently, and record the storage system's resource consumption throughout the execution process. Pocket, Jiffy, and FuncStore serve as the storage systems supporting data storage during the execution of the workload. Note that the amount of resources requested in advance by Pocket is the peak value during the running process of the workload, and Jiffy's lease duration is configured to 4 seconds, aligning with the maximum intermediate data lifecycle of the applications.

Figure 12(a) illustrates the variation in storage resources at different stages of the workload when utilizing Pocket, Jiffy, and FuncStore as storage systems. In the figure, both Jiffy and FuncStore significantly optimize memory resource utilization during runtime in comparison to Pocket, which pre-allocates fixed storage resources. Precisely, Jiffy and FuncStore reduce resource usage by 21.4% and 86.6%, respectively. Notably, FuncStore demonstrates a more pronounced effect compared to Jiffy. Through the implementation of the *Consistent Monitoring* mechanism alone, FuncStore reduces memory resource usage by 67.3%, which is 3.14 times more efficient than Jiffy. When incorporating the *Lifetime-aware Memory Management* policy, FuncStore's efficiency surpasses Jiffy's by 4.04 times.

In the case of Jiffy, different lease durations influence how frequently its data is reclaimed from memory. To investigate this impact, we conduct tests with varying lease durations: 4s, 2s, 1s, and 100ms. The changes in storage space are shown in Figure 12(b). Since Jiffy stores deleted data on SSD and loads it back into memory upon reuse, the reduction of the lease duration accelerates the release of memory resources. However, this can also lead to a significant amount of data being read from SSD, substantially prolonging the runtime of the workload. Specifically, when the lease duration is reduced from a 4s configuration to 2s, 1s, and 100ms, memory resource usage declines by 21.9%, 34.0%, and 90.1%, respectively. Concurrently, there is an increase in workload execution time by factors of 1.2x, 1.6x, and 1.7x, respectively.

D. Performance Benchmarks

In addition to resource utilization, the performance of data read and write operations is a fundamental metric for evaluating a storage system. Hence, we conduct tests to assess the data read/write performance of Pocket, Jiffy, and FuncStore. Specifically, we execute 1000 read/write requests for data of various sizes to calculate the average read/write latency and throughput of the three storage systems. The test results are displayed in Figure 13. For 4KB size data, FuncStore exhibits read/write latencies of 1225us and 1126us, respectively. Pocket's read/write latencies are 846us and 865us, while Jiffy's read/write latencies measure 2107us and 3078us, respectively.



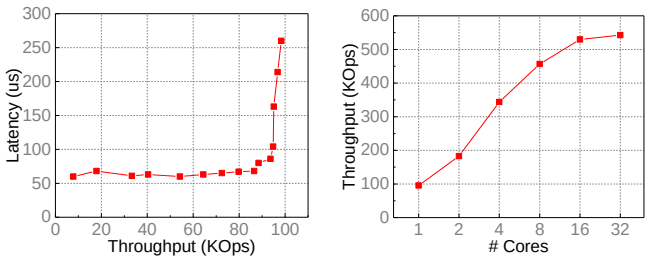
(a) Latency (b) Throughput

Fig. 13: Performance comparison with data object size from 4KB to 128MB

In summary, FuncStore attains superior resource utilization without compromising on performance, making it comparable to state-of-the-art ephemeral storage systems. Note that the throughput of all three systems, Pocket, Jiffy, and FuncStore, is capped at approximately 80MB/s. This limitation stems from AWS Lambda, which restricts the network bandwidth available to each function instance to a maximum of 640Mbps [2, 11].

E. Metadata Server Analysis

As a key component of the system, the performance of the *Metadata Server* is also a point of interest. In this section, we evaluate the performance of the *Metadata Server*. We first test the throughput and latency of the *Metadata Server* on a single CPU core, followed by testing its scalability on multiple cores. Specifically, we choose a machine of type *c5.9xlarge* to run the *Metadata Server*, using *taskset* to limit the number of CPU cores that it can utilize. We use a client to continuously send write requests to the *Metadata Server*, increasing the number of concurrent clients to enhance throughput until it becomes congested, with the test results shown in Figure 14(a). Initially, as the number of concurrent clients increases, the throughput of the *Metadata Server* also increases, with the latency of each request remaining relatively stable. When the throughput reaches about 95KOps, increasing the number of clients almost does not increase the throughput of the *Metadata Server*, while the processing latency of requests significantly increases, meaning it has reached a point of congestion.



(a) Single core (b) Multiple cores

Fig. 14: Metadata server performance

Subsequently, we increase the number of CPU cores that the *Metadata Server* can utilize, testing the maximum throughput that can be achieved under different CPU core counts, with the test results shown in Figure 14(b). When the CPU core counts are 2, 4, 8, 16, and 32, the maximum throughputs are respectively 182KOps, 343KOps, 456KOps, 529KOps, and 543KOps. Notably, when the throughput demand for the *Metadata Server* exceeds 543KOps, we can enhance the throughput through horizontal scaling. Specifically, we can increase the number of metadata nodes, while having each metadata node manage some storage nodes and serve a portion of workflow requests, to ensure that the *Metadata Server* does not become a bottleneck in the system.

F. System Overhead

In this subsection, we discuss the additional overheads in the design of FuncStore. These overheads encompass increased space occupancy, aiming at enhancing resource utilization, as well as the time overhead associated with reading and writing data.

TABLE II: Spatial overhead

counter	lifetime	op record	offset	size	Total
1 byte	4 bytes	several bytes	4 bytes	4 bytes	∼20 bytes

*1) **Spatial Overhead**:* Unlike Pocket and Jiffy, FuncStore introduces storage overhead for the initial count value and predicted lifetime of each data object. The initial counter and the predicted lifetime require one byte and four bytes, respectively. The four bytes can represent lifetimes up to 2^{32} ms, which significantly surpasses the lifetime of all the intermediate data encountered.

During application execution, operation records in the *Commit Buffer* also consume storage space. For each intermediate data, the operation record size is determined by the data object's name, roughly amounting to a few bytes. Additionally, while Pocket and Jiffy only need to record the mapping of the data object to the storage block, FuncStore must also record the data object's start offset and size when allocating a slice, requiring an additional 8 bytes.

In summary, each data object stored in FuncStore incurs an additional metadata overhead of approximately 20 bytes, compared with Pocket. The specific overhead items are detailed in Table II. Besides the metadata storage overhead mentioned above, the prediction models also consume storage space. In our implementation, the size of each model is approximately 34KB. However, the storage overhead of the prediction models is negligible since one model can be shared across all applications within the same category.

*2) **Temporal Overhead**:* For each read and write request, FuncStore performs additional operations on the *Metadata Server* to update the data object's maintenance information. These operations involve adding operation records to the *Commit Buffer* every time the data is read, committing all operation records in the *Commit Buffer* and updating the count value when the function writes the last data object. We separately measure the overhead of these operations. Specifically, it takes approximately 1us to add an operation record to the *Commit Buffer* and about 15us to commit a *Commit Buffer* containing 10 operation records and update the corresponding count values. These additional time overheads are acceptable for individual read/write operations.

VI. Related Work

Remote Storage. In addition to Pocket [11] and Jiffy [16], there are several other storage systems employed to facilitate the sharing of intermediate data in serverless computing. For example, Crail [37] leverages diverse storage media and RDMA, catering to the storage requirements of data in various sizes. Another solution is Anna [38, 39], a high-performance, auto-scaling, and multi-tier storage system designed for cloud environments. It focuses on how to automatically scale nodes and improve read/write performance. In contrast to these storage systems, which primarily focus on data placement across

various storage media and the dynamic scaling of cluster size, FuncStore allocates and releases storage space by monitoring the lifecycle of the data to maximize the utilization of limited storage resources. SDCM [40] proposes a mathematical model to select the cost-optimal storage strategy by comparing the cost of data storage with the re-create cost. It is orthogonal to FuncStore. SDCM can use FuncStore's techniques to reduce storage costs regardless of the durability level of the storage. In contrast, FuncStore currently chooses memory as the storage medium and simply relies on re-creating data to cope with storage failures. It can be further optimized through SDCM to make a better choice between re-creating data and using storage with higher durability.

Caching Strategies. To enhance data transfer performance, some research works build a cache to accelerate the function's access to intermediate data. OFC [26] constructs a RAMCloud [41] cluster directly on the compute nodes. Using machine learning techniques, it can predict the amount of idle memory available for a given function. This idle memory is then allocated to the RAMCloud cluster to serve as a cache for the intermediate data accessed during the execution of the function. Similarly, Faa$T [15] employs a cache layer located between the function and remote storage. Unlike OFC, which offers a shared cache cluster for all applications, Faa$T takes a different approach. It provides a dedicated cache for each serverless application and unloads cached data when the application is inactive. This strategy helps minimize memory usage. Duo [42] takes a unique approach by analyzing the access patterns of the intermediate data. Based on these patterns, it designs a cache algorithm that prioritizes data reads. This strategy improves the overall performance of data access by giving high priority to frequently read data. However, since the cache is still built on top of the existing remote storage, it does not improve the resource utilization of the storage system. Furthermore, there remains a notable overhead when accessing remote storage in the event of cache misses.

Alternative Data Sharing Methods. Apart from utilizing remote storage for data transfer, various alternative methods are available. Some research works [1, 5, 43–45] try to run functions of the same application on the same physical node, which in turn reduces network access overhead by sharing memory on the host. This approach increases the scheduling constraints of the functions. When the functions of a single application cannot run on a node at the same time, it still needs to pass the data through remote storage. Additionally, approaches such as Boxer [46] and FMI [47] enable direct communication between AWS Lambda functions through NAT hole punching. However, implementing NAT hole punching requires users to manually deploy a hole punching server, which increases the cost of use.

Garbage Collection. A lot of high-level programming languages (e.g., Java, Go, and Python) use *garbage collection* (GC) mechanisms to implement automatic memory management to reduce the user's burden on memory management. This is similar to our idea of releasing memory in a timely manner without the user being aware of it. Tracing garbage collection [48, 49] is a commonly used method. It maintains the reference relationship between data objects, and determines whether a data object can be recycled by analyzing the reachability of the reference relationship path from the data object to some starting points (GC roots). Reference counting [50–52] is another common garbage collection method. It determines whether the object's space can be reclaimed by recording the number of times each data object is referenced. The method of FuncStore's *Consistent Monitoring* to determine the expiration of intermediate data is similar to reference counting, while it is used in distributed memory management. Furthermore, we propose a fault-tolerant mechanism for function retry on this basis.

VII. Conclusion

Existing serverless ephemeral storage systems suffer from low storage resource utilization. The reason is that it is difficult for the system to accurately sense the lifecycle of immediate data, leading to a large amount of storage resources being occupied and not released promptly. In response to this issue, we introduce FuncStore, which accurately senses the lifecycle of data according to the number of times it has been read, and deletes the data at the end of the lifecycle instantly to avoid the occupation of storage space. Furthermore, FuncStore places data objects with similar expiration times in the same storage unit to reduce storage space fragmentation. Compared to state-of-the-art works, FuncStore reduces the storage resource usage by 86.6% and 81.7% respectively, while guaranteeing request latency and throughput.

VIII. Acknowledgements

We thank our shepherd, Jay Lofstead, and the anonymous reviewers for their insightful feedback. This work was supported in part by the National Key Research and Development Program of China under grant No.2022YFB4500704 and in part by the National Natural Science Foundation of China under grant No.62032008. Zhuo Huang (huangzhuo@hust.edu.cn) and Jianhui Yue (jyue@mtu.edu) are the corresponding authors.

References

[1] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "SAND: Towards High-Performance Serverless Computing," in *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC)*, 2018, pp. 923–935.

[2] A. Klimovic, Y. Wang, C. E. Kozyrakis, P. Stuedi, J. Pfefferle, and A. K. Trivedi, "Understanding Ephemeral Storage for Serverless Analytics," in *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC)*, 2018, pp. 789–794.

[3] Z. Huang, S. Wu, S. Jiang, and H. Jin, "FastBuild: Accelerating Docker Image Building for Efficient Development and Deployment of Container," in *Proceedings of the 35th IEEE International Conference on Massive Storage Systems and Technology (MSST)*, 2019, pp. 28–37.

[4] Z. Huang, Q. Zhang, H. Fan, S. Wu, C. Yu, H. Jin, J. Deng, J. Gu, and Z. Tang, "Multi-grained Trace Collection, Analysis, and Management of Diverse Container Images," *IEEE Transactions on Computers*, pp. 1–12, 2024.

[5] Z. Li, Y. Liu, L. Guo, Q. Chen, J. Cheng, W. Zheng, and M. Guo, "FaaSFlow: Enable Efficient Workflow Execution for Function-as-a-Service," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022, pp. 782–796.

[6] Amazon Simple Storage Service. [Online]. Available: https://aws.amazon.com/s3/

[7] Amazon DynamoDB. [Online]. Available: https://aws.amazon.com/dynamodb/

[8] Amazon ElastiCache. [Online]. Available: https://aws.amazon.com/elasticache/

[9] A. Wang, J. Zhang, X. Ma, A. Anwar, L. Rupprecht, D. Skourtis, V. Tarasov, F. Yan, and Y. Cheng, "INFINICACHE: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache," in *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST)*, 2020, pp. 267–281.

[10] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "Encoding, Fast and Slow: Low-latency Video Processing Using Thousands of Tiny Threads," in *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017, pp. 363–376.

[11] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic Ephemeral Storage for Serverless Analytics," in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018, pp. 427–444.

[12] A. Bhardwaj, C. Kulkarni, and R. Stutsman, "Adaptive Placement for In-memory Storage Functions," in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC)*, 2020, pp. 127–141.

[13] A. Merenstein, V. Tarasov, A. Anwar, S. Guthridge, and E. Zadok, "F3: Serving Files Efficiently in Serverless Computing," in *Proceedings of the 16th ACM International Systems and Storage Conference (SYSTOR)*, 2023, pp. 8–21.

[14] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure," in *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019, pp. 193–206.

[15] F. Romero, G. I. Chaudhry, I. Goiri, P. Gopa, P. Batum, N. J. Yadwadkar, R. Fonseca, C. Kozyrakis, and R. Bianchini, "Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications," in *Proceedings of the 12th ACM Symposium on Cloud Computing (SoCC)*, 2021, pp. 122–137.

[16] A. Khandelwal, Y. Tang, R. Agarwal, A. Akella, and I. Stoica, "Jiffy: Elastic Far-Memory for Stateful Serverless Analytics," in *Proceedings of the 17th ACM European Conference on Computer Systems (EuroSys)*, 2022, pp. 697–713.

[17] AWS Step Functions. [Online]. Available: https://aws.amazon.com/step-functions/

[18] Google Cloud Workflows. [Online]. Available: https://cloud.google.com/workflows/

[19] Redis. [Online]. Available: https://redis.io/

[20] V. Sreekanti, C. Wu, S. Chhatrapati, J. E. Gonzalez, J. M. Hellerstein, and J. M. Faleiro, "A Fault-Tolerance Shim for Serverless Computing," in *Proceedings of the 15th ACM European Conference on Computer Systems (EuroSys)*, 2020, pp. 1–15.

[21] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu, "Fault-tolerant and Transactional Stateful Serverless Workflows," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020, pp. 1187–1204.

[22] Z. Jia and E. Witchel, "Boki: Stateful Serverless Computing with Shared Logs," in *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, 2021, pp. 691–707.

[23] S. Qi, X. Liu, and X. Jin, "Halfmoon: Log-Optimal Fault-Tolerant Stateful Serverless Computing," in *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2023, pp. 314–330.

[24] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, "Sprocket: A Serverless Video Processing Framework," in *Proceedings of the 9th ACM Symposium on Cloud Computing (SoCC)*, 2018, pp. 263–274.

[25] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein, "From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers," in *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC)*, 2019, pp. 475–488.

[26] D. Mvondo, M. Bacou, K. Nguetchouang, L. Ngale, S. Pouget, J. Kouam, R. Lachaize, J. Hwang, T. Wood, D. Hagimont, N. D. Palma, B. Batchakui, and A. Tchana, "OFC: An Opportunistic Caching System for FaaS Platforms," in *Proceedings of the 16th ACM European Conference on Computer Systems (EuroSys)*, 2021, pp. 228–244.

[27] H. Yu, A. A. Irissappane, H. Wang, and W. J. Lloyd, "FaaSRank: Learning to Schedule Functions in Serverless Platforms," in *Proceedings of the 2nd IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, 2021, pp. 31–40.

[28] A. Mampage, S. Karunasekera, and R. Buyya, "Deep Reinforcement Learning for Application Scheduling in Resource-constrained, Multi-tenant Serverless Computing Environments," *Future Generation Computer Systems*, vol. 143, pp. 277–292, 2023.

[29] S. Eismann, J. Grohmann, E. Van Eyk, N. Herbst, and S. Kounev, "Predicting the Costs of Serverless Work-

flows," in *Proceedings of the 11th ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2020, pp. 265–276.

[30] S. Eismann, L. Bui, J. Grohmann, C. Abad, N. Herbst, and S. Kounev, "Sizeless: Predicting the Optimal Size of Serverless Functions," in *Proceedings of the 22nd ACM/IFIP International Middleware Conference (Middleware)*, 2021, pp. 248–259.

[31] Apache Thrift. [Online]. Available: https://thrift.apache.org/

[32] Libcuckoo. [Online]. Available: https://github.com/efficient/libcuckoo/

[33] ConcurrentQueue. [Online]. Available: https://github.com/cameron314/concurrentqueue/

[34] The Mobile Shell. [Online]. Available: https://github.com/mobile-shell/mosh/

[35] HMDB. [Online]. Available: https://serre-lab.clps.brown.edu/resource/hmdb-a-large-human-motion-database/

[36] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider," in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC)*, 2020, pp. 205–218.

[37] P. Stuedi, A. Trivedi, J. Pfefferle, A. Klimovic, A. Schuepbach, and B. Metzler, "Unification of Temporary Storage in the NodeKernel Architecture," in *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC)*, 2019, pp. 767–781.

[38] C. Wu, J. M. Faleiro, Y. Lin, and J. M. Hellerstein, "Anna: A KVS for Any Scale," *IEEE Transactions on Knowledge and Data Engineering*, vol. 33, no. 2, pp. 344–358, 2019.

[39] C. Wu, V. Sreekanti, and J. M. Hellerstein, "Autoscaling Tiered Cloud Storage in Anna," *Proceedings of the VLDB Endowment*, vol. 12, no. 6, pp. 624–638, 2019.

[40] A. Merenstein, X. Wang, V. Tarasov, P. Agarwal, S. Guthridge, K. Thakkar, K. Wu, A. Anwar, and E. Zadok, "Balancing Costs and Durability for Serverless Data," in *Proceedings of the 38th IEEE International Conference on Massive Storage Systems and Technology (MSST)*, 2024.

[41] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang, "The RAMCloud Storage System," *ACM Transactions on Computer Systems*, vol. 33, no. 3, pp. 1–55, 2015.

[42] Z. Huang, H. Fan, C. Cheng, S. Wu, and H. Jin, "Duo: Improving Data Sharing of Stateful Serverless Applications by Efficiently Caching Multi-Read Data," in *Proceedings of the 37th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2023, pp. 875–885.

[43] S. Shillaker and P. Pietzuch, "FAASM: Lightweight Isolation for Efficient Stateful Serverless Computing," in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC)*, 2020, pp. 419–433.

[44] S. Kotni, A. Nayak, V. Ganapathy, and A. Basu, "Faastlane: Accelerating Function-as-a-Service Workflows," in *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC)*, 2021, pp. 957–971.

[45] Z. Jia and E. Witchel, "Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021, pp. 152–166.

[46] M. Wawrzoniak, I. Müller, R. Bruno, and G. Alonso, "Boxer: Data Analytics on Network-enabled Serverless Platforms," in *Proceedings of the 11th International Conference on Innovative Data Systems Research (CIDR)*, 2021, pp. 1–11.

[47] M. Copik, R. Böhringer, A. Calotoiu, and T. Hoefler, "FMI: Fast and Cheap Message Passing for Serverless Functions," in *Proceedings of the 37th ACM International Conference on Supercomputing (ICS)*, 2023, pp. 373–385.

[48] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens, "On-the-Fly Garbage Collection: An Exercise in Cooperation," *Communications of the ACM*, vol. 21, no. 11, pp. 966–975, 1978.

[49] M. Maas, K. Asanović, and J. Kubiatowicz, "A Hardware Accelerator for Tracing Garbage Collection," in *Proceedings of the 45th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2018, pp. 138–151.

[50] D. Anderson, G. E. Blelloch, and Y. Wei, "Concurrent Deferred Reference Counting with Constant-Time Overhead," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, 2021, pp. 526–541.

[51] W. Zhao, S. M. Blackburn, and K. S. McKinley, "Low-Latency, High-Throughput Garbage Collection," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, 2022, pp. 76–91.

[52] D. Anderson, G. E. Blelloch, and Y. Wei, "Turning Manual Concurrent Memory Reclamation into Automatic Reference Counting," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, 2022, pp. 61–75.