Ephemera: Accelerating I/O-Intensive Serverless Workloads with a Harvested In-Memory File System

LINGXIAO JIN, School of Computer Science, Shanghai Jiao Tong University, Shanghai, China
ZINUO CAI, School of Computer Science, Shanghai Jiao Tong University, Shanghai, China
HAOXIN WANG, School of Computer Science, Shanghai Jiao Tong University, Shanghai, China
ZONGPU ZHANG, Shanghai Jiao Tong University, Shanghai, China
RUHUI MA, School of Computer Science, Shanghai Jiao Tong University, Shanghai, China
HAIBING GUAN, School of Computer Science, Shanghai Jiao Tong University, Shanghai, China
YUAN LIU, School of Artificial intelligence and Computer Science, Jiangnan University, Wuxi, China
BUYYA RAJKUMAR, The University of Melbourne, Melbourne, Australia

Serverless computing has gained popularity for its ability to shift the burden of server management from developers to cloud providers, which allows providers to exercise greater control over resource management, optimizing configurations to enhance efficiency and performance. The diversity of serverless computing tasks, from short-lived, event-driven tasks to more complex workloads, highlights the growing importance of efficient file I/O performance for I/O-intensive workloads, yet effectively handling ephemeral storage for I/O-intensive tasks remains a challenge. Traditional file system approaches often introduce substantial latency and fail to fully leverage available memory resources within the execution environment, limiting performance and efficiency. Our work stems from the observation of the under-utilization of memory resources in serverless computing platforms and the potential efficiency improvement of I/O operations using an in-memory file system. Based on this observation, we propose Ephemera, a system designed to enhance ephemeral storage efficiency and memory utilization. Ephemera satisfies three design goals: *transparent memory I/O integration*, *heterogeneous tasks resource synergy*, and *harmonized cluster workload orchestration*. Ephemera integrates three components: the Runtime Daemon, responsible for managing a container's in-memory file system; the Tenant Manager, facilitating memory configuration sharing across containers; and the Cluster Controller, optimizing workload balancing. Our experiments demonstrate that Ephemera significantly improves performance for I/O-intensive tasks compared to traditional file systems. Specifically, Ephemera decreases I/O processing time by 50% on average and reduces latency by up to 95.73% in certain scenarios with negligible overhead.

Lingxiao Jin and Zinuo Cai equally contributed to this work.

This work was supported by Shanghai Key Laboratory of Scalable Computing and Systems, the National Natural Science Foundation of China under the contract No.62472200 and Project supported by Shanghai Municipal Science and Technology Major Project.

Authors' Contact Information: Lingxiao Jin, School of Computer Science, Shanghai Jiao Tong University, Shanghai, China; e-mail: jinlingxiao1122@sjtu.edu.cn; Zinuo Cai, School of Computer Science, Shanghai Jiao Tong University, Shanghai, China; e-mail: kingczn1314@sjtu.edu.cn; Haoxin Wang, School of Computer Science, Shanghai Jiao Tong University, Shanghai, China; e-mail: wanghaoxin@sjtu.edu.cn; Zongpu Zhang, Shanghai Jiao Tong University, Shanghai, China; e-mail: zhangz-z-p@sjtu.edu.cn; Ruhui Ma (corresponding author), School of Computer Science, Shanghai Jiao Tong University, Shanghai, China; e-mail: ruhuima@sjtu.edu.cn; Haibing Guan, School of Computer Science, Shanghai Jiao Tong University, Shanghai, China; e-mail: hbguan@sjtu.edu.cn; Yuan Liu, School of Artificial intelligence and Computer Science, Jiangnan University, Wuxi, China; e-mail: lyuan1800@jiangnan.edu.cn; Buyya Rajkumar, The University of Melbourne, Melbourne, Victoria, Australia; e-mail: rbuyya@unimelb.edu.au.

ACM 1544-3973/2025/09-ART102
https://doi.org/10.1145/3747846

CCS Concepts: • **Computer systems organization → Cloud computing**; • **Information systems →** *Hierarchical storage management;*

Additional Key Words and Phrases: serverless computing, file system, resource harvesting

ACM Reference Format:
Lingxiao Jin, Zinuo Cai, haoxin Wang, Zongpu Zhang, Ruhui Ma, Haibing Guan, Yuan Liu, and Buyya Rajkumar. 2025. Ephemera: Accelerating I/O-Intensive Serverless Workloads with a Harvested In-Memory File System. *ACM Trans. Arch. Code Optim.* 22, 3, Article 102 (September 2025), 24 pages. https://doi.org/10.1145/3747846

1 Introduction

Since the release of AWS Lambda,[1] serverless computing [17, 24, 31] has transformed from a novel cloud computing concept into a widely recognized cloud computing paradigm by academic and industrial communities. Serverless computing does not imply the absence of servers; instead, it abstracts cloud computing into **Functions as a Service** (**FaaS**) and **Backend as a Service** (**BaaS**) [14]. Cloud tenants only need to encode their business logic as functions and organize them into applications. In contrast, the cloud service providers are responsible for resource management, load balancing, and other backend services. Compared to existing computing paradigms [25], serverless computing relieves cloud tenants of underlying infrastructure management while still obtaining highly reliable services. A recent advance in serverless computing has continuously addressed various bottlenecks, including cold starts [9, 20, 28], resource management [1, 24, 41], heterogeneous hardware support [4, 5, 7], and workflow optimization [13, 18, 29, 33].

Although serverless computing was initially designed for short-lived, event-driven tasks, the types of functions supported by serverless computing services have become more diverse, including big data analytics, machine learning model training, and inference. Compared to short-lived jobs, these more complex job types involve common compute operations and require I/O operations related to persistent storage. For example, serverless MapReduce [6] requires storing intermediate results from the map phase for the reduce phase. During the training process of machine learning models [12], the best-performance models need to be saved, while the initial stage of model inference [35] requires loading pre-trained models from disk into memory. The diversification of job types presents new requirements for serverless computing platforms' file system management solutions.

This article proposes an in-memory file system solution to utilize over-provisioned memory resources to optimize I/O efficiency during function execution. Our idea stems from the observation that users on serverless computing platforms often resort to resource over-provisioning to improve the computational efficiency of functions. Specifically, AWS Lambda and OpenWhisk[2] employ a memory-centric resource allocation approach where the platform automatically allocates CPU resources based on the amount of memory specified by the user. Therefore, users resort to memory over-provisioning to access more computational resources. Although Aliyun Function Compute[3] proposes resource decoupling to enhance flexibility in resource allocation, its constrained vCPU-to-memory ratios still lead to memory over-provisioning. In addition, over-provisioning resources can reduce operational complexity for less experienced users and effectively handle input-sensitive functions.

However, it is not trivial to implement an in-memory file system for a serverless platform. Our design needs to satisfy the following three design goals to meet the requirements of complex

[1]https://aws.amazon.com/lambda/
[2]https://github.com/apache/openwhisk
[3]https://www.aliyun.com/product/fc

serverless computing workflows. Firstly, we aim to achieve Transparent Memory I/O Integration, with the challenge of seamlessly integrating this feature without requiring users to modify their existing codebase. This intricate process entails intercepting I/O APIs and transforming disk-based file access into memory-based operations while preserving backward compatibility with diverse function execution environments. Secondly, we focus on Heterogeneous Tasks Resource Synergy, which presents challenges due to the diverse memory requirements of different instances. Enabling efficient memory sharing between heterogeneous instances like CPU-intensive or I/O-intensive tasks is crucial to optimize resource utilization and enhance overall performance. Lastly, we emphasize Harmonized Cluster Workload Orchestration, which addresses the challenge of maintaining the in-memory file system's efficacy at the cluster level. Balancing workload distribution across cluster nodes is essential to prevent resource contention and ensure optimal performance of the in-memory file system by avoiding assigning the same type of workload to a single node.

Therefore, we design `Ephemera`, a memory-optimized framework that leverages allocated but unused memory for serverless computing functions to enhance I/O efficiency. To meet the aforementioned design goals, `Ephemera` consists of three components: a *function-level* Daemon, a *tenant-level* Manager, and a *cluster-level* Controller. Firstly, the Runtime Daemon resides in each running function instance and achieves the conversion from the native file system to the in-memory file system by intercepting I/O operations in the user's source code, which is transparent to cloud tenants. Secondly, to facilitate resource sharing among heterogeneous function instances, we introduce the Tenant Manager, which allows dynamic memory sharing among different function instances of the same tenant during execution. Besides, Managers for different tenants are isolated from each other, ensuring data security between tenants. Lastly, to address the workload balancing issue across different nodes in the cluster, we design the Cluster Controller, which dynamically adjusts the selection of nodes and resource allocation during function runtime.

We implement a prototype of `Ephemera` in C and Python with 3000+ lines of code. We evaluate each component by a self-built serverless cluster and analyze the application latency, I/O bandwidth, and memory utilization. The results indicate that our system achieves a 50% reduction in file operation latency compared to the traditional file systems, reaching a latency reduction of up to 95.73%. In addition, our system can enhance memory utilization.

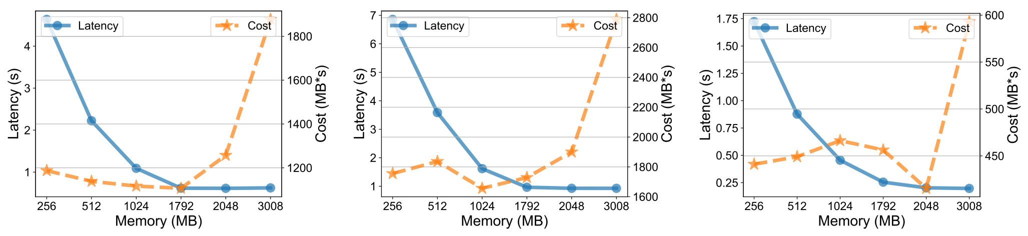
In summary, our contributions are highlighted as follows:

- We first discover an optimization opportunity for a memory-optimized file system on a serverless computing platform. Our observation stems from the low utilization of memory in serverless computing platforms and the potential improvement in file access efficiency with a memory-based file system.
- We design `Ephemera`, a file system that leverages allocated but idle memory space to optimize file access efficiency in FaaS platforms. `Ephemera` consists of three components: a *cluster-level* Controller, a *tenant-level* Manager, and a *function-level* Daemon, forming a scalable computing framework.
- We implement a prototype of `Ephemera` and conduct extensive experiments on it. Experiments show that our approach reduces file operation latency by 50% on average, with latency reductions up to 95.73%.

2 Background and Motivation

2.1 Background: File System for Serverless Computing

Serverless computing represents an emerging paradigm within cloud applications, wherein tenants (those who upload and pay for the function execution) employ high-level languages such as Python, Go, or JavaScript to write functions that execute specific application logic [17]. This paradigm



(a) Gzip Compression (b) Pyaes (c) Linpack

Fig. 1. Evaluation on AWS Lambda.

operates on a “pay-as-you-go” billing approach, whereby tenants incur costs solely for the compute time and resources allocated during function execution. Compared to traditional cloud computing services, such as PaaS or IaaS, FaaS paradigm offered by serverless computing allows for more flexible service provision, effectively meeting users’ demands.

Existing works [10, 16, 26, 30] have already considered how to design and optimize file systems for serverless computing, providing support for I/O-intensive function types. The mainstream solution is to offer configurable disk space for users to choose from, which is common in commercial serverless computing platforms like AWS Lambda and Aliyun Function Compute, as well as open-source serverless computing platforms. For example, AWS Lambda provides 512 MB of free temporary and configurable storage of up to 10 GB. Users can perform disk operations for I/O-intensive functions by reading from and writing to the “/tmp” directory. Aliyun Function Compute has a similar file system support solution. Improving file I/O performance is crucial for serverless computing, as it enables more efficient execution of I/O-intensive workloads, leading to faster function invocations and reduced overall costs for serverless applications.

2.2 Observation: Under-Utilization of Memory Resources

To validate the under-utilization of memory resources during the execution of functions on serverless computing platforms, we select three representative functions from FunctionBench [15] and evaluate them on AWS Lambda and Aliyun Function Compute under different resource configurations. The selected functions include *gzip compression*—evaluates file compression performance using the `gzip` library, *pyaes*—measures the performance of AES encryption and decryption using the `pyaes` library, and *linpack*—uses `numpy` to solve linear equations. These three functions require a minimum memory of 77 MB, 40 MB, and 93 MB for execution, respectively. The first function is I/O-intensive, while the latter two are CPU-intensive tasks.

AWS Lambda provides memory-centric resource configuration, allowing developers to set memory quotas, with vCPU allocated in proportion to the memory allocation. We sample memory configurations ranging from 256 MB to 3008 MB, including allocating a complete vCPU when the memory is set to 1792 MB. Figure 1 illustrates the execution time and cost of the three applications under different resource configurations on AWS Lambda. The price is calculated as the product of execution time and memory. From the perspective of execution time, as the CPU capacity on AWS Lambda is directly proportional to the allocated memory, the execution time decreases as the memory increases. However, the rate of decrease diminishes after reaching 1792 MB. From the cost perspective, the three applications exhibit the lowest cost at 1792 MB, 1024 MB, and 2048 MB, respectively. Therefore, regardless of whether the user’s execution goal is latency-optimal or cost-optimal, the memory allocated by AWS Lambda to the functions is significantly higher than the minimum required memory for the applications, which results in under-utilization of memory resources.

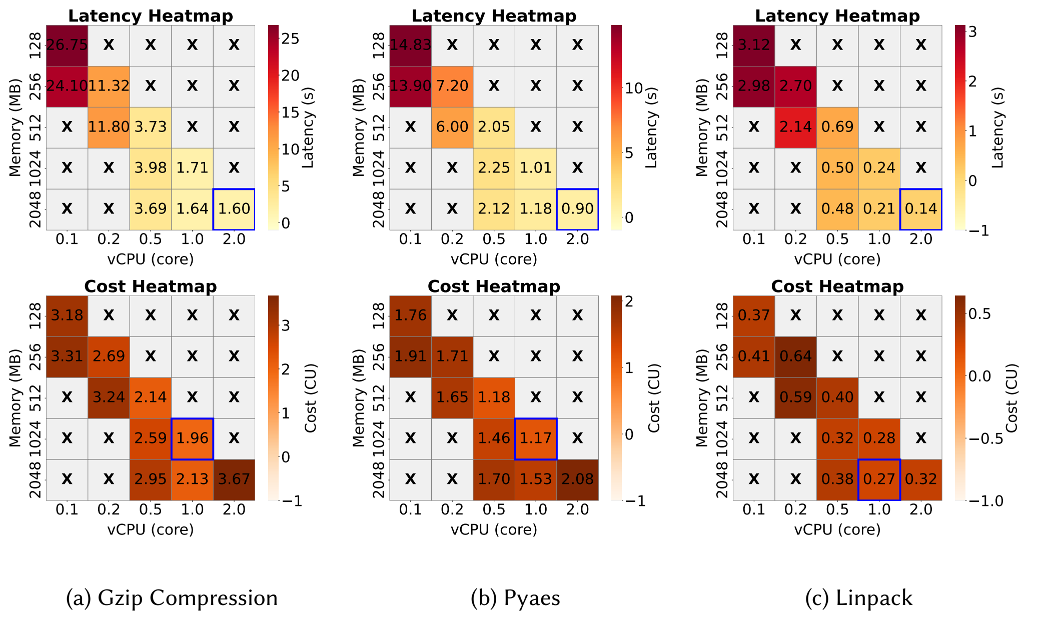


Fig. 2. Evaluation on Aliyun Function Compute.

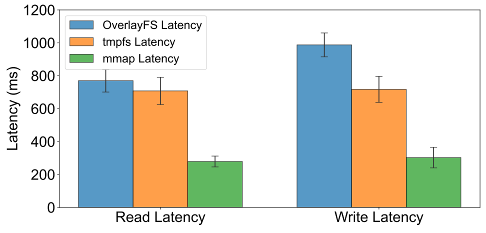
Unlike AWS Lambda, Aliyun Function Compute decouples the allocations of memory capacity and CPU with the ratio of vCPU size to memory size (GB) must be between 1:1 and 1:4. We sample memory configurations ranging from 128 MB to 2048 MB and vCPU configurations ranging from 0.1 to 2. Figure 2 illustrates the execution time and cost of three applications under different resource configurations on Aliyun Function Compute. Due to the constrained vCPU-to-memory ratios in cloud instances, certain configurations are unavailable and are denoted by `X` in the figure. The configurations yielding minimum latency and minimum cost are highlighted with blue boxes. Our analysis reveals that regardless of whether users optimize for latency or cost, the resulting configurations lead to memory over-provisioning. Moreover, the cloud provider's practice of offering multiple memory options for the same vCPU configuration may inadvertently encourage tenants to allocate excessive memory resources.

2.3 Opportunity: In-Memory Operations to Accelerate I/Os

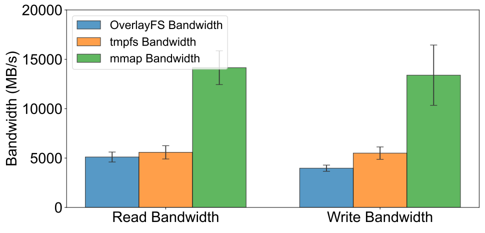
To assess the impact of memory on file operation performance, we evaluate the effects of `mmap`, `tmpfs`, and OverlayFS[4]on random disk I/O performance. OverlayFS is used as a baseline, which overlaps multiple directory layers to form a unified file system view. Like other traditional filesystems, OverlayFS utilizes the kernel's page cache mechanism to buffer file data in memory, improving read/write performance by reducing disk I/O operations. The `mmap` technique maps file contents to memory address space, allowing applications to access file data as if they were memory operations directly. On the other hand, `tmpfs`, as an in-memory file system, stores file data directly in memory.

The experiment uses Docker containers as the experimental environment to evaluate the performance of random disk I/O operations, including latency and bandwidth. The results are shown in Figure 3. Due to the characteristics of the in-memory file system, `tmpfs` demonstrates advantages in low

[4]https://docs.kernel.org/filesystems/overlayfs.html



(a) Latency



(b) Bandwidth

Fig. 3. Comparison of memory-based file systems.

latency and high bandwidth during random I/O operations. Compared to OverlayFS, `tmpfs` showed an 8% improvement in read performance and a significant 28% improvement in write performance. Different from traditional file operations, `mmap` leverages memory mapping to map file contents directly into user space, allowing applications to read and write data without copying between user and kernel space. Hence, `mmap` reduces the overhead of data movement between the kernel and user space, significantly improving the performance of I/O operations. Compared to OverlayFS, `mmap` showed a 63% improvement in read performance and a 69% improvement in write performance.

Therefore, while traditional filesystems already benefit from page cache, leveraging memory more aggressively through memory-based solutions can further improve file system performance. Whether by directly storing file data through a memory file system or indirectly utilizing memory resources through memory mapping techniques, both can effectively reduce file operation latency and increase data processing speed.

3 Design

3.1 Overview

We design `Ephemera` to enhance the efficiency of ephemeral storage access on serverless computing platforms by leveraging each function runtime's allocated but idle memory space. As depicted in Figure 4, `Ephemera` consists of three components from top to bottom: a *cluster-level* Controller, a *tenant-level* Manager, and a *function-level* Daemon. **The Cluster Controller** resides at the core of the cluster and handles function deployment and invocation requests. During the deployment phase, an integrated **profiler** within the Controller conducts trials to analyze function execution patterns, including memory usage and file I/O. Conversely, upon receiving function invocation requests, the Controller's internal **scheduler** becomes active, distributing requests based on node workload and the functions' previously profiled patterns to achieve workload balance. **The Tenant Manager**, allocated for each tenant on each node in the cluster, implements and controls a tenant-level memory-sharing mechanism that enables dynamic sharing of memory resources among different function instances of the same tenant to optimize resource utilization. It maintains a memory resource pool and categorizes the execution instances on each node into two groups: the harvest pool, which reclaims excess memory, and the allocation pool, which holds instances waiting for available memory. **The Runtime Daemon** is integrated into each function runtime and implements an in-memory file system by utilizing the idle memory within the runtime. The Daemon dynamically adjusts the size of the in-memory file system based on the actual resource usage of the runtime, thereby improving resource utilization efficiency.

Figure 4 illustrates the overall workflow of `Ephemera`, including the function deployment and invocation stages. During the function deployment stage, when receiving the tenant's uploaded

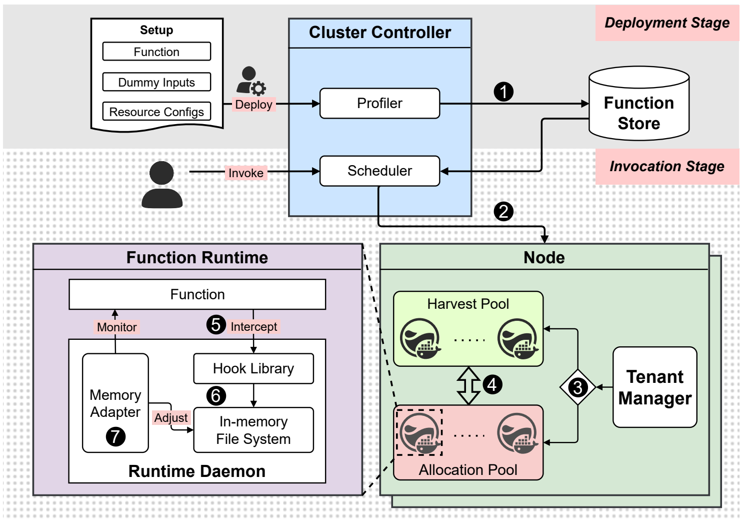


Fig. 4. Ephemera architecture and workflow.

source code or function image, resource configuration (including CPU and memory limits), and provided dummy inputs, the profiler in the Controller ❶ executes the function with the dummy inputs. During this execution, the profiler evaluates the execution patterns of the function, such as memory usage and file I/O behavior, and these profile data are then stored. During the function execution stage, when receiving an invocation request, the scheduler in the Controller ❷ assigns the request to a node based on the nodes' current workload conditions and the function's previously profiled pattern, to balance memory requirements. After a function execution instance is launched on the selected node, the Tenant Manager on that node ❸ places the instance into the harvest pool or the allocation pool. This management by the Manager ❹ facilitates dynamic sharing of memory resources among different instances belonging to the same tenant through these pools. Simultaneously, within the function runtime environment of an instance, the Runtime Daemon plays a role in handling function I/O and managing memory usage. The hook library, a component of the Daemon, ❺ intercepts the function's file I/O requests during execution, diverting them from the standard system. These intercepted requests are then ❻ forwarded to the in-memory file system managed by the Daemon. Furthermore, the memory adapter component within the Daemon ❼ continuously monitors the function's runtime resource usage and dynamically adjusts the size of the in-memory file system as needed.

3.2 Runtime Daemon

FaaS platform allocates a separate runtime for each function invocation request, and we embed a Daemon within each runtime to manage the function's I/O requests during execution. Figure 5 illustrates the architecture of the Daemon. The Daemon consists of three modules: a hook library to intercept I/O requests for delegation, an in-memory file system to enable file operations in memory, and a memory adapter to monitor memory usage and dynamically adjust the size of the in-memory file system.

3.2.1 Hook Library. The hook library identifies and intercepts operations related to the file system. It registers itself when the serverless function is first executed in the container. During the interception phase, the hook library validates the command type, verifying whether it is file-related.

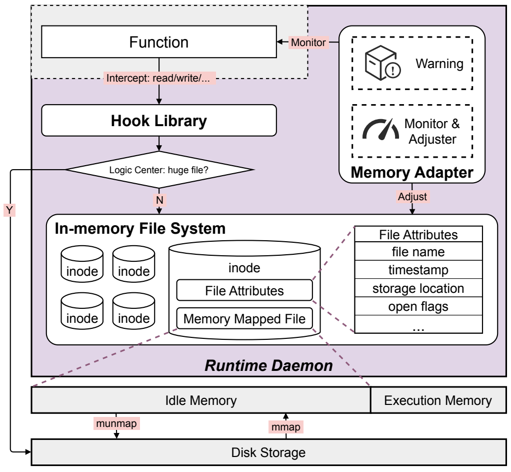


Fig. 5. Runtime Daemon overview.

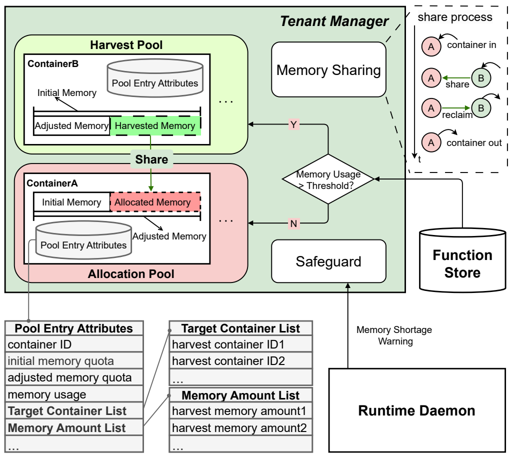


Fig. 6. Tenant Manager overview.

Table 1. Intercepted APIs

API Name	Description	API Name	Description
`mkdir`	make directories	`read`	read from a file descriptor
`open`	open and possibly create a file	`write`	write to a file descriptor
`close`	close a file descriptor	`lseek`	reposition read or write file offset

Table 1 shows all intercepted APIs, and the hook library passes the parameters of file operations to the in-memory file system after an interception.

3.2.2 In-Memory File System. We design an in-memory file system to facilitate access to ephemeral files on serverless platforms by leveraging each function instance's allocated but idle memory. It stores the memory limit for the container. Aside from the memory used for program execution, the rest can be allocated for file storage to accelerate access. Similar to traditional file systems, the in-memory file system maintains inode structure to record information for each directory or file. File information is stored in inodes, which include the file's name, `open flags`, file offset, read-write locks, access timestamp, and the current storage location of the file, whether on disk or in memory. With `open flags` in the inode, the in-memory file system reinforces protection for read and write permissions.

A key feature of our in-memory file system is its reliance on `mmap` to manage file data at a file granularity. The `mmap` operation maps a file into the process's virtual address space, allowing direct memory operations on file contents. This mechanism potentially reduces the overhead associated with traditional read and write system calls, which must transition data between user space and kernel space. In the in-memory file system, the inode of each file also stores a block of memory space used to hold files mapped by `mmap`. The allocated space can be directly manipulated for read and write operations in subsequent processes.

When a file is opened via `open` API, it is prioritized for storage in memory using `mmap`. When using `mmap` to load a file into memory, it automatically retains parts of the file in memory based on actual usage. Despite the performance benefits of memory mapping, it is not always appropriate to store an entire file in memory—particularly if the file size is large or if the memory resource is constrained. In such cases, the system can decide to place the file on disk to avoid the overhead of excessive page swapping. The inode then marks the file's location as being on disk, which informs the core logic that subsequent read or write requests should be handled through conventional disk I/O.

Based on the file's storage location—whether in memory or on the disk—the logic center in the in-memory file system invokes different processing logic. For files stored in memory, the logic center manipulates the inode's associated memory space, allocated via `mmap`, to perform file operations directly within memory. This process may include modifying the inode's metadata to reflect changes in file size, offset, or other attributes due to these operations. Conversely, the system relies on original API calls for disk operations for files stored on disk. Similarly, it updates the inode to record any changes made to the file on disk. Ultimately, the in-memory file system returns the processed results to the main program, completing the interception and processing of file system operations.

3.2.3 Memory Adapter. Due to the memory limit set by the FaaS platform for each function runtime, excessive memory allocation to the in-memory file system can potentially result in **out-of-memory** (**OOM**) failures. Therefore, we supplement a memory adapter module in the Daemon to facilitate dynamic adjustment of memory resources within the runtime and resource adjustment with the Manager for scaling instances. The memory adapter has three main functions. Firstly, it monitors the total memory usage of the native serverless function and the in-memory file system. Secondly, when it detects insufficient available memory, the memory adapter communicates with the Manager to request additional sharable memory space or reclaim allocated memory. Finally, when the memory limit available to the instance is determined, the memory adapter invokes a file-swapping mechanism to ensure the smooth functioning of file operations.

To prevent memory overflow caused by introducing a memory file system, the memory adapter incorporates a mechanism for swapping memory files with disk files, thus maintaining system stability. The adapter continuously monitors the access status of files and adjusts their storage location in memory based on their access timestamp. When a file is closed, the scheduler does not immediately unload it to the disk. Instead, it remains in memory, with only the inode's flag set to zero to indicate its closure state. However, in cases of insufficient memory, the file with the smallest timestamp, corresponding to the **Least Recently Used** (**LRU**) file, is removed from memory using the `munmap` function. By setting the timestamp of closed files to zero, closed files are prioritized for eviction. Crucially, when a file is currently being operated on, ensuring consistency between the application's operations and the underlying memory management is paramount. A separate, explicit memory scheduling mechanism, even if designed with fine granularity, struggles to completely avoid conflicts with application-level concurrent operations on file memory pages, potentially introducing complex consistency issues. Introducing mutex locks to guarantee consistency would, in turn, incur performance overhead. Therefore, we rely on the operating system's automatic mmap-based memory management when the file is currently being operated on.

3.3 Tenant Manager

Figure 6 illustrates the architecture of the Manager, which is implemented for each tenant in each node. Within the Manager, we design a tenant-level memory-sharing mechanism. To facilitate efficient sharing, we introduce container pools that classify running instances based on their operational patterns, thereby enabling effective memory sharing. Furthermore, we propose a safeguard mechanism to mitigate potential memory shortages due to excess memory sharing proactively.

3.3.1 Memory Sharing Mechanism. Sharing granularity is essential when considering the mechanism of memory multiplexing, which ranges from fine-grained to coarse-grained levels. Existing works have considered multiplexing mechanisms like single-function and single-workflow sharing. The single-function sharing mechanism [32] allows memory sharing between instances of the same function. However, since these instances typically have similar memory needs, they often concurrently experience excess or insufficient memory, making sharing ineffective. The single-workflow sharing mechanism [22, 23] enables memory sharing among instances in the same workflow but fails to cover cases outside workflows. Due to a specific execution order, instances can't share memory concurrently within a workflow. Here, a workflow refers to a set of functions that a tenant has explicitly defined to execute in a specific order or dependency graph as a single application task, typically managed by a serverless orchestration service.

Instead, we adopt single-tenant sharing, which allows for memory sharing among all instances of functions within the same tenant. A tenant represents a user or organization account on the serverless platform, and all functions deployed and owned by this account are considered part of the same tenant. This approach provides a wide range of sharable objects, enables memory sharing between instances of different patterns, and accommodates instances running concurrently. Within each node, the system adopts an isolation strategy to manage the resources of multiple tenants. The system assigns a dedicated Manager for each tenant per node. The Manager independently manages all function instances for that tenant. Such tenant-level isolation prevents potential conflicts or security vulnerabilities related to data and applications, thus enhancing the system's overall security. Moreover, memory sharing follows a priority principle. Function instances with more available memory gain higher priority during memory sharing. Harvesting memory from these instances can avoid performance degradation by frequently retrieving memory from many minor function instances. Conversely, when the system needs to allocate memory to existing function instances, those requiring less memory are prioritized. This approach can ensure that as many function instances as possible run efficiently, thereby maximizing user satisfaction.

3.3.2 Container Pool. The container pool is responsible for managing container memory allocation and sharing. It categorizes containers based on memory requirements and usage patterns to optimize memory utilization. Based on the profiles from the Controller, if a container's file size and memory usage exceed the threshold of the memory limit, it is placed in the allocation pool, from which additional memory allocations can be obtained from containers in the harvest pool. Conversely, if a container's file size and memory usage are below the threshold of the memory limit, it is placed in the harvest pool, and its extra memory can be allocated to containers in the allocation pool. Otherwise, the container is considered self-sufficient.

Each entry in the container pool records relevant vital attributes and two associated lists to track memory allocations between containers in subsequent memory-sharing mechanisms. Formally, each entry for a container C can be represented as a tuple of attributes:

$$\text{Entry}(C) = (C_{ID}, T_{lim}, M_{peak\_native}, F_{max\_size\_runtime}, T_{current\_lim}, L_{peers}, L_{deltas})$$

where:

- C_{ID}: The unique identifier for the container.
- T_{lim}: The initial memory limit assigned to the container by the scheduler.
- M_{peak_native}: The peak native memory usage observed for the function executing in this container during its lifecycle.
- $F_{max\_size\_runtime}$: The maximum size of files that has been processed by the In-memory File System within this container.
- $T_{current\_lim}$: The current effective memory limit of the container, which can be adjusted dynamically by the Manager. Initially, $T_{current\_lim} = T_{lim}$.

- $L_{peers} = [P_1, P_2, \dots, P_k]$: A list of k peer container IDs involved in memory sharing with container C.
- $L_{deltas} = [\Delta M_1, \Delta M_2, \dots, \Delta M_k]$: A parallel list of k memory amounts, where ΔM_i is the sharing amount associated with peer P_i.

The interpretation of L_{peers} and L_{deltas} depends on the pool to which container C belongs:

- If C is in the harvest pool: L_{peers} lists the containers that have *received* memory from C. ΔM_i is the amount of memory *harvested from C* and allocated to P_i. The current limit is calculated as the initial limit minus the total harvested amount:

$$T_{current\_lim} = T_{lim} - \sum_{i=1}^{k} \Delta M_i$$

- If C is in the allocation pool: L_{peers} lists the containers from which C has *received* memory. ΔM_i is the amount of memory *allocated to C* and harvested from P_i. The current limit is calculated as the initial limit plus the total allocated amount:

$$T_{current\_lim} = T_{lim} + \sum_{i=1}^{k} \Delta M_i$$

When $k = 0$, the container is not actively sharing memory with peers, and $T_{current\_lim} = T_{lim}$.

When a request triggers an invocation of a specific container, the Manager identifies its state to determine the appropriate sharing strategy. Containers in the harvest pool donate their memory to active functions, while containers in the allocation pool obtain additional memory from the currently executing functions. Conversely, self-sufficient containers can start without the need for external memory intervention. After the function execution, the Manager reevaluates the containers' states to ensure proper memory management. For containers in the harvest pool, the system proactively reclaims memory previously allocated to other containers. Conversely, containers in the allocation pool are instructed to return any excess memory received. Meanwhile, self-sufficient containers undergo standard termination. Therefore, the latency of the Tenant Manager depends on the number of containers in the Container Pool and the pattern of containers. As the number of containers in the Container Pool grows, the Tenant Manager spends more time calculating how to share memory. But when the Tenant Manager has already allocated or reclaimed enough memory for the containers, it will stop looking for other containers that can participate in memory sharing.

For instance, container A with insufficient memory and container B with excess memory arrive sequentially. First, A is allocated to the allocation pool and executes with its initial memory quota. Subsequently, container B arrives and is allocated to the harvest pool. Since container A requires more memory, container B shares a part of its initial memory quota, denoted by the harvested memory amount, with container A. Following this, the system updates container A's target container list with container B's ID and the memory amount list with the allocated memory amount, i.e., the memory container B has shared with container A. Thus, both containers execute based on their adjusted memory quotas. If container B finishes early, it will reclaim the memory allocated to container A and update the relevant lists.

3.3.3 Safeguard. Due to variations in input size, node workload, and other factors during the actual execution, memory usage may deviate from initial expectations. We propose a safeguard mechanism to prevent programs from stopping due to OOM failures. When the Manager receives information about a memory shortfall detected by a Daemon, it forcibly reclaims allocated memory from the container pool. Concurrently, the container is marked as non-reclaimable to ensure that no additional memory allocations are taken from this container when other containers are initiated.

ALGORITHM 1: Workload Balance Allocation

```
Input: Task: memory limit $T_{lim}$, max memory usage $T_{mem}$; Node: node memory limit $N_{lim}$, memory
       allocation $N_{alloc}$, and memory requirement $N_{req}$
Output: Optimal Node $optimalNode$
Function SelectOptimalNode(task, nodes):
    candidates ← {}
    for node in nodes do
        if $node.N_{lim} \leq node.N_{alloc} + task.T_{lim}$ then
            candidates.add(node)
        end
    end
    if candidates is not empty then
        optimalNode ← null
        optimalScore ← $-\infty$
        for node in candidates do
            score ← $-|node.N_{req} + task.T_{mem} - node.N_{alloc} - task.T_{lim}|$
            if score > optimalScore then
                optimalScore ← score
                optimalNode ← node
            end
        end
        optimalNode.allocate(task)
        return optimalNode
    else
        return null
    end
End Function
```

3.4 Cluster Controller

Ephemera allocates a unique Controller to each tenant, facilitating a single-tenant shared memory mechanism. In the deployment stage, the profiler in the Controller analyzes the function patterns. During the invocation stage, the scheduler in the Controller dispatches the request to the optimal node based on the function pattern and nodes' workload.

3.4.1 Profiler. Through analysis, the profiler determines each function's memory limit, memory usage, and file usage. Recognizing that a function's resource utilization may be related to the input scale, the profiler calculates the upper limit of resource consumption during actual use to prevent resource shortages. Based on several sets of dummy inputs prepared by the user, the profiler executes functions in parallel, thereafter monitoring the memory usage and the size of accessed disk files. The largest observed resource consumption scenario is selected and preserved as the basis for subsequent workload balancing and the Manager's division of container pools. The sum of the memory usage and the size of the files accessed is the maximum upper limit of memory a function may use since files can be stored in an in-memory file system to speed up access.

3.4.2 Scheduler. The scheduler leverages data derived from profiling and the current workloads across various nodes to distribute function-packaged containers among these nodes using Algorithm 1. Upon receiving a request, the scheduler first retrieves data previously analyzed for the task, including the memory limit T_{lim} and the maximum memory usage T_{mem}. It then assesses the current workload on each node, which includes the node memory limit N_{lim}, total memory allocation N_{alloc}, and total memory requirement N_{req} within each node. Specifically, N_{alloc} is the sum of T_{lim} for all tasks currently allocated to the node, and N_{req} is the sum of T_{mem} for these tasks. Using this information, the scheduler identifies candidate nodes that can accommodate the new task without exceeding their memory limits. Once it has filtered out nodes that cannot accommodate the new task, the scheduler then seeks to minimize the absolute difference between the node's allocation and requirement, i.e., $\left|(N_{req} + T_{mem}) - (N_{alloc} + T_{lim})\right|$, thereby balancing the supply and demand of memory as effectively as possible. The primary objective of this approach is to minimize the absolute difference between memory allocation and memory requirement without exceeding the node memory limit, thereby ensuring an efficient and balanced distribution of workloads across nodes. With a time complexity of O(N), where N is the number of nodes, the algorithm's running time scales linearly with the number of nodes.

4 Implementation

We implement the prototype of Ephemera efficiently (with around 3000+ lines of code). Specifically, the Daemon is implemented in C because C makes it easy to manage memory manually, and the Manager and Controller are implemented in Python due to Python's simplicity and extensive library support. Our system uses Docker[5] as the application sandbox.

Function Runtime. Similar to OpenWhisk, we set up a proxy to receive request information, which is then passed on to the launcher for the execution of the actual function. The launcher initializes and executes user-defined functions in an isolated environment, managing input/output and handling errors. The proxy communicates with the Manager through two pipes: one for receiving requests or updates on memory limits, and another for returning results. The proxy handles memory limit updates, modifying the pointer variable in the in-memory file system that stores the memory limit. Similarly, the launcher also communicates with the proxy through two pipes: one to receive requests and another to return results. Our setup ensures that the system blocks when there are no requests in the pipe, thereby not consuming extra CPU resources.

In-memory File System. We utilize the `mmap` and `munmap` system calls to achieve access files in memory. `mmap` enables the mapping of a file on disk into the process's address space, allowing the contents of the file to be read and written directly as if they were in memory. `munmap` can revoke this mapping, ensuring the resources are correctly released.

Instruction Interception. We compile the hook library into a **dynamic link library** (**DLL**) and register it when the launcher starts. By employing the method of function overriding, we replace the standard C library's file operation APIs, thereby achieving the capability to intercept these operations.

Memory Monitoring. In the Daemon, the memory adapter polls `/proc/[pid]/statm` to monitor memory usage. The `/proc/[pid]/statm` file in Linux systems provides detailed memory usage information for a specific process, where `[pid]` represents the process ID. This file offers key metrics, including total virtual memory, resident memory, shared memory, and code memory, measured in pages. We use resident memory as the metric, which reflects the number of pages currently in physical memory, including those mapped via mmap.

Memory Sharing. We employ the docker-update API from the Docker library to facilitate the memory-sharing operation. The API enables the real-time update of memory configuration for

[5]https://www.docker.com/

Table 2. Experimental Testbed Configuration

Component	Specification	Component	Specification
CPU device	Intel Xeon Gold 6248R	Number of sockets	4
Processor BaseFreq	3.00 GHz	Threads	192 (96 physical cores)
Memory Capacity	512GB	SSD Capacity	11TB
Operating System	Ubuntu 22.04 LTS	Docker version	24.0.7

containers. We wrap the docker-update using Python code, allowing the Manager to invoke it asynchronously.

Cluster Controller. During the deployment phase, the profiler launches functions with dummy inputs, simultaneously utilizing a profiling mode where the in-memory file system exclusively logs file operations instead of retaining file contents in memory. The profiler actively monitors peak conventional memory usage utilizing `/proc/[pid]/statm`, concurrently employing the in-memory file system to quantify the size of opened files. Transitioning to the invocation phase, the scheduler leverages these determined function metrics, along with the nodes' workload memory demands and their maximum memory capacities, to determine scheduling assignments.

5 Performance Evaluation

5.1 Experiment Setup

Environment. We evaluate `Ephemera` with practical workloads in the context of serverless computing. Table 2 summarizes the configurations of the computing infrastructure. In our prototype system, function instances are assigned a Docker container as their runtime environment. The hardware and software configuration provides a comprehensive environment for scrutinizing the performance attributes of `Ephemera` in real-world scenarios.

Metrics. The performance metrics considered in this evaluation encompass application execution latency, I/O bandwidth, I/O intensity, and memory utilization. Specially, I/O intensity refers to the degree of I/O operations relative to the total execution time of an application. It can be measured by combining the use of `strace` and `/usr/bin/time`. First, `strace` is used to record the proportion of system calls related to I/O operations, denoted as p. Then, `time` is used to capture the time spent in kernel mode, t_k, and user mode, t_u. The I/O intensity is calculated as $\frac{t_k \times p}{t_k + t_u}$.

Workloads. The serverless workloads we use for evaluation are listed in Table 3. In our micro-benchmarks experiments, we design I/O-intensive functions to evaluate the effects of memory limitations, file size usage, and file operation frequency on the in-memory file system. We then employ three functions from FunctionBench [15] and four realistic workloads to further illustrate our system's performance. To evaluate the memory sharing mechanism, we treat the micro-benchmarks functions as I/O-intensive tasks and use large integer factorial calculations as CPU-intensive tasks. Finally, we similarly use these I/O-intensive and CPU-intensive tasks to evaluate the scheduler's performance as workloads.

Baseline. This evaluation entails a comparative analysis, pitting `Ephemera` against the Docker with OverlayFS and the Docker with tmpfs. OverlayFS is a union file system commonly used in containerization. It overlays a read-only base image (lower layer) with a writable layer (upper layer) for runtime modifications. Reads check the upper layer first, then the lower. Read data is cached for efficiency while writes are confined to the upper layer. Tmpfs is a temporary file storage

Table 3. Workloads and Their I/O Intensity

Workload Name	**I/O Intensity**	**Description**
Micro Benchmark	Dynamic	Performs multiple file operations on two files of the same size.
Sequential Disk I/O	96.34%	Conducts multiple sequential read and write operations on a 32MB file, emphasizing continuous data access.
Random Disk I/O	94.23%	Executes multiple random read and write operations on a 32MB file, highlighting non-sequential data access.
Gzip Compression	Dynamic	Compresses and decompresses a file using gzip, involving multiple read and write operations.
Image Processing	64.17%	Reads an image, processes it to grayscale, and writes the processed image back to disk.
MapReduce	87.84%	Utilizes 2 Mappers to read files and count word occurrences, writing results to disk, followed by 1 Reducer to aggregate Mapper outputs.
ML Inference	19.93%	Uses a DNN model to recognize handwritten characters, focusing on model inference operations.
Video Processing	62.12%	Reads a video file, processes it to grayscale, and writes the processed video back to disk.
Large Integer Factorial	0%	Calculates the factorial of a number without file operations.

filesystem that keeps data in memory rather than on disk. When we start a Docker instance, we use the `--mount type=tmpfs` command to mount a tmpfs. The memory space it can use is limited by the memory configuration allocated to the Docker instance.

5.2 Micro-Benchmarks

We conduct performance evaluations by subjecting the system to diverse conditions, altering memory constraints, file usage sizes, and the frequency of file operations. The Daemon starts as part of the Docker runtime and subsequently runs along with the function instances in the container.

5.2.1 Impact of Memory Limit. We evaluate the impact of memory limit on the latency of `read` and `write` system calls. We run experiments of the memory limit from 32 MB to 256 MB on `Ephemera` and compare with the same workload on the original OverlayFS and mounted tmpfs. Figure 7 illustrates the impact of memory limit on system performance. When the memory limit is 128 MB, which is sufficient to accommodate all files, the latency reduction of our system relative to OverlayFS reaches 54.73%. Notably, when the memory limit is 64 MB, which is sufficient to store only a portion of the files, but not all, our system achieves a latency reduction of 92.67%. Under conditions of severe memory scarcity, the performance of our system approximates that of OverlayFS. This similarity arises because our scheduling mechanism, aiming to avoid the performance degradation caused by frequent scheduling of large files between memory and disk, refrains from storing files in memory and instead relies on traditional file system read-write operations.

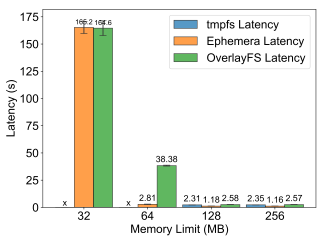


Fig. 7. Impact of memory limit.

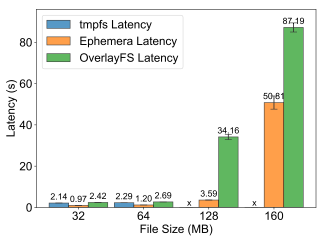


Fig. 8. Impact of file usage size.

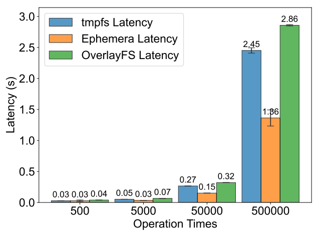


Fig. 9. Impact of file operation times.

5.2.2 Impact of File Usage Size. We evaluate the impact of actual file usage size on the latency of `read` and `write` system calls. We run experiments of file usage size from 32 MB to 160 MB on `Ephemera` with a memory limit of 150 MB and compare with the same workload on OverlayFS. Figure 8 demonstrates that the system's performance varies with the size of the files in practice. When the memory capacity is adequate to accommodate all files, there is a significant improvement in system performance. Particularly, when the memory can store some but not all files, the latency reduction reaches 89.5%. Even when the memory can only hold a majority of a single file, and scheduling operations are required, there is still a 41.72% latency reduction compared to OverlayFS.

5.2.3 Impact of File Operation Times. We evaluate the impact of file operation times on the latency of `read` and `write` system calls. We run experiments of file operation times from 500 to 500000 on `Ephemera` and compare them with the same workload on an OverlayFS on a local disk. Besides, each file operation is 8192 B, the same as the default buffer size. Figure 9 elucidates that the system's performance advantage becomes more pronounced with an increasing number of file operations. When the frequency of file operations is low, there is a modest 28.03% reduction in latency. However, as the number of file operations escalates, the latency reduction can reach up to 53.04%. This phenomenon occurs because establishing file mappings incurs a certain time overhead, and the benefits of operating files in memory become more evident as the number of operations increases.

5.2.4 Scalability. In Figure 7, when the memory is set to 32MB and 64MB, Docker with tmpfs mounted fails due to OOM, indicated by an `X` in the figure. Similarly, in Figure 8, when the file size increases to 128 MB and 160 MB, the approach using tmpfs cannot execute normally. Compared to tmpfs, `Ephemera` can dynamically adjust the in-memory file size based on memory limits and the size of the file being operated on, thereby avoiding OOM and achieving better scalability.

5.3 Benchmarks

We use FunctionBench[15] to evaluate `Ephemera`. FunctionBench is a popular testing repository for serverless computing functions, capable of evaluating CPU, memory, disk, and network performance. In FunctionBench, we extract three disk performance-related functions to demonstrate the advantages of our system: the compression performance evaluation, the sequential read or write performance evaluation, and the random read or write performance evaluation.

5.3.1 Random and Sequential Disk I/O Performance. In the random disk I/O performance evaluation, the function operates on a 32 MB file. It starts by writing 8192 B of content, then randomly adjusts the offset using `lseek`, and repeats this writing process 500000 times. Afterward, it reads 8192 B of content, again using `lseek` to adjust the offset randomly, and repeats this reading process 500000 times. In the sequential disk I/O performance evaluation, unlike the random evaluation, the

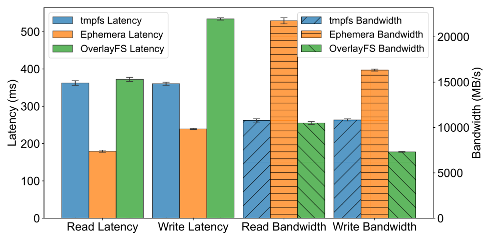


Fig. 10. Sequential disk I/O performance.

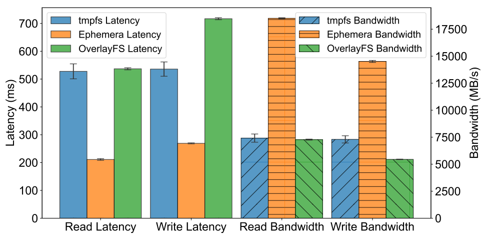


Fig. 11. Random disk I/O performance.

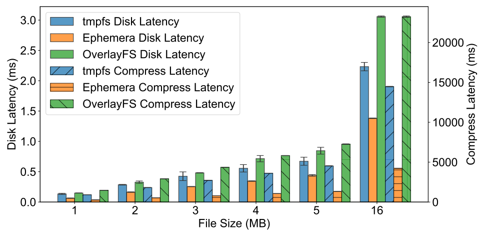


Fig. 12. Compression performance.

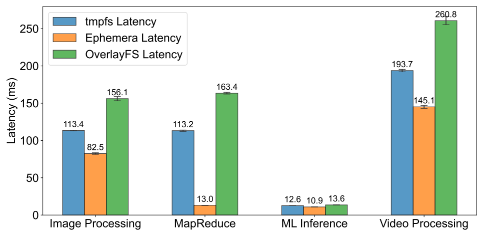


Fig. 13. Workloads performance.

file is read and written continuously until the end before `lseek` is used to move the offset back to the start of the file. We set a memory limit of 150 MB to ensure enough space to store all files. It tests the latency of writing to disk (disk latency) as well as the latency during the compression phase (compress latency). A comparison of Figure 10 and Figure 11 shows that the performance of random read-write operations is inferior to that of sequential read-write operations. Due to the prefetching mechanism of the CPU cache, sequential read-write operations perform better than random ones, even when files are operated on in memory. Although the latency for random read-write operations increases, the relative performance improvement of the in-memory file system compared to the original OverlayFS also rises, from an initial 55% to 62%. Hence, random read-write operations more effectively highlight the advantages of the in-memory file system.

5.3.2 Compress Performance Evaluation. This function begins by writing to a file from one MB to 16 MB, followed by frequent file reading using a compression algorithm, simultaneously compressing and writing the compressed content to an archive. We set a memory limit of 64 MB to ensure enough space to store all files. It tests the latency of writing to disk (disk latency) as well as the latency during the compression phase (compress latency). Figure 12 reflects that disk latency can achieve an improvement of approximately 50%, while compress latency can reach about 81.8%. This is because the implementation of the compression algorithm involves frequent file access, where direct memory operations have an advantage.

5.3.3 Realistic Workloads. In the workloads of Image Processing, MapReduce, ML Inference, and Video Processing, the performance of `Ephemera`, OverlayFS, and tmpfs is shown in Figure 13. Compared to OverlayFS, `Ephemera` achieves latency reductions of 47.2%, 92.0%, 19.9%, and 44.3%, respectively. This is attributed to the acceleration of file operations by memory. Additionally, when compared to tmpfs, `Ephemera` shows latency reductions of 27.3%, 88.5%, 13.9%, and 25.1%, respectively. This improvement is mainly due to the reduced overhead of context switching between user space and kernel space achieved by mmap.

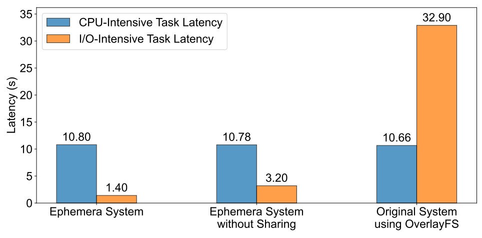


Fig. 14. Efficacy of memory sharing mechanism.

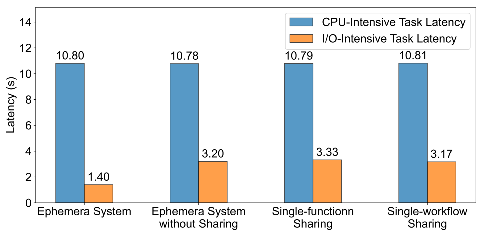


Fig. 15. Sharing mechanisms comparison.

5.3.4 Impact of I/O Intensity. Comparing Figure 13 and Table 3, a general trend emerges: the higher the I/O intensity, the more significant the latency reduction achieved by `Ephemera`. This aligns with `Ephemera`'s optimization for I/O operations; since it is not optimized for computation, it exhibits more pronounced latency reductions in scenarios with higher I/O intensity. However, it's worth noting that despite sequential disk I/O tasks having an I/O intensity 2.11% higher than random disk I/O tasks, the latency reduction shown in Figure 10 reaches 55%, which is lower than the 62% shown in Figure 11. This indicates that while latency reduction is generally positively correlated with I/O intensity, it is also influenced by the specific type of disk operation.

5.4 Effectiveness of Memory Sharing Mechanism

The Manager allocates tasks to either the harvest pool or the allocation pool based on the distinct characteristics of the tasks and the memory configuration. It adjusts the memory limit upon function initiation to facilitate memory sharing.

This section examines a CPU-intensive task that is allocated excess memory in pursuit of higher computational power alongside an I/O-intensive task that needs more memory. The I/O-intensive tasks are the same as the functions in Section 5.2. The CPU-intensive tasks are large integer factorial calculations. The evaluation assesses the performance of the in-memory file system, both with and without the activation of the sharing mechanism, and compares them to the original OverlayFS. Figure 14 demonstrates that enabling memory sharing has a negligible impact on CPU-intensive tasks but significantly benefits I/O-intensive tasks. Without the memory sharing mechanism, I/O-intensive tasks utilizing the in-memory file system can achieve a 90.27% latency reduction over OverlayFS. However, when the memory sharing mechanism is employed, I/O-intensive tasks can harvest more memory from CPU-intensive tasks, leading to a latency reduction of 95.73%.

5.5 Different Memory-Sharing Mechanisms

We design experiments to demonstrate that the single-function sharing mechanism and the workflow sharing mechanism are not effective in sharing memory among different function instances. For the single-function sharing mechanism, the container pool managed by the Manager only holds different instances of a single function. We conduct experiments by running two I/O-intensive tasks and two CPU-intensive tasks simultaneously. For the single-workflow sharing mechanism, we design workflows that first execute the CPU-intensive task and then the I/O-intensive task. In this case, the container pool managed by the Manager contains different function instances from the same workflow. We sequentially execute the CPU-intensive task and the I/O-intensive task to simulate the workflow for the experiment. Figure 15 illustrates the latency of CPU-intensive and I/O-intensive tasks for scenarios using the three sharing mechanisms or the scenario without the sharing mechanism. The latency of CPU-intensive tasks is almost the same across all scenarios,

Table 4. Node Workload Condition

	Node1			Node2		
	Mem Req	Mem Alloc	Mem Usage	Mem Req	Mem Alloc	Mem Usage
Ephemera	778 MB	750 MB	748 MB	262 MB	450 MB	262 MB
Baseline	1032 MB	600 MB	600 MB	8 MB	600 MB	8 MB

while the latency of I/O-intensive tasks shows performance improvement only in the single-tenant sharing mechanism compared to the scenario without any sharing mechanism. Because the single-function sharing mechanism only shares memory among different instances of the same function. Instances of CPU-intensive tasks generally have surplus memory while I/O-intensive tasks suffer from insufficient memory, there is no memory sharing occurring between instances. In the single-workflow sharing mechanism, CPU-intensive tasks in the harvest pool can share memory with function instances running concurrently in the allocation pool. However, by the time I/O-intensive tasks enter the allocation pool, the CPU-intensive tasks have already ended, thus preventing memory sharing at that point.

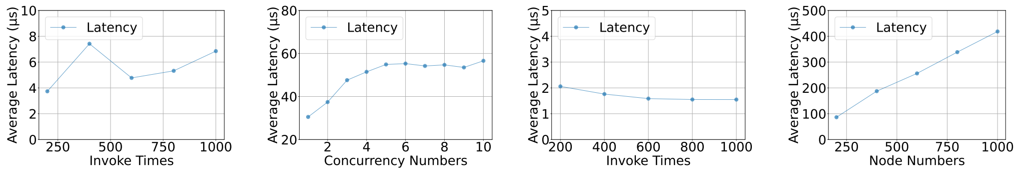
5.6 Efficacy of Ephemera's Scheduling

The scheduler in the Controller determines the allocation of the task to nodes based on the nodes' current workload, the task's memory requirement, and the task's memory configurations to achieve optimal overall performance. For comparison, a workload-balancing algorithm that prioritizes nodes with the most available memory as the optimal choice serves as the baseline. This section employs six tasks arriving every second to simulate a real-world serverless computing request environment, observing the workload distribution on nodes and the response latency of each task. The task set consists of four I/O-intensive tasks and two CPU-intensive tasks from the same tenant, distributed across two nodes, each with a total memory of 1024 MB. We launch two Managers on a single machine to simulate two nodes.

Table 4 indicates that our workload-balancing algorithm more effectively balances the allocation and usage of memory, enhancing the performance of I/O-intensive tasks without diminishing the performance of CPU-intensive tasks. In the baseline configuration, Node One is exclusively occupied by I/O-intensive tasks, each suffering from insufficient memory allocation; Node Two is solely filled with CPU-intensive tasks, leading to significant memory wastage. Through our workload-balancing method, it becomes possible for each I/O-intensive task to receive additional memory allocations from CPU-intensive tasks. This results in a minimal memory waste of two MB on Node one, because while the CPU-intensive tasks only consume four MB of memory, Docker imposes a minimum memory requirement of six MB. Implementing this workload-balancing approach facilitates an average latency reduction of 44.92%. Our methods, utilizing a greedy strategy, lack foresight regarding future events, thereby preventing some I/O-intensive tasks from receiving sufficient memory, which could otherwise lead to further performance improvements.

5.7 Overhead of Ephemera

The overhead of the Runtime Daemon comes from the cost of monitoring memory. Using Ephemera with CPU-intensive tasks incurs additional overhead without benefiting from the performance optimizations provided by the in-memory file system. As shown in Figure 14, when using Ephemera, the latency of the CPU-intensive task increases from 10.66 seconds to 10.80 seconds, with a performance degradation of only 1.29%.



(a) Invoke Times Impact on Tenant Manager (b) #Concurrency Impact on Tenant Manager (c) Invoke Times Impact on Cluster Controller (d) #Node Impact on Cluster Controller

Fig. 16. Overhead of Ephemera.

The overhead of the Tenant Manager comes from calculating the memory resource usage in the container pool and determining whether to allocate the remaining memory or reclaim the required memory. It also involves calculating the number of container instances allocated or reclaimed. We define the overhead of the Tenant Manager as the average latency from when the Manager receives an invocation request to when the Manager sends it to the container. We deploy a CPU-intensive factorial computation task and an I/O-intensive random read-write task on a node to evaluate the impact of different request frequencies on the overhead. As shown in Figure 16(a), when requests are evenly distributed over one second, and the number of requests increases from 200 to 1000, there is no significant change in the average latency, which remains at around six microseconds, which is negligible. Furthermore, we investigate the impact of the number of concurrent instances on the overhead of the Tenant Manager. Figure 16(b) shows that as the concurrency numbers increase from 1 to 10, the average latency first increases and then stabilizes with an overall latency below 60 microseconds. This is because higher concurrency results in more container instances in the Container Pool that can be allocated or harvested, thus causing the average latency to initially increase. The average latency of the Tenant Manager tends toward stability as the number of concurrent instances continues to increase because the Tenant Manager stops calculating once a container has been allocated or harvested sufficient memory.

The overhead of the Cluster Controller comes from the scheduler's analysis of workloads across different nodes. We define the overhead of the Cluster Controller as the average latency from when the scheduler receives an invocation request to when the scheduler dispatches it to a specific node. We evaluate variations in overhead by adjusting the number of nodes and the frequency of requests. Figure 16(c) shows that when the number of nodes is two and requests arrive uniformly within one second, the average latency remains at two microseconds without significant changes as the number of requests increases from 200 to 1000. Figure 16(d) shows that when the number of requests is 200, average latency increases as the number of nodes scales from 200 to 1000. When the number of nodes reaches 1000, the average latency is approximately 418 microseconds, which is still negligible compared to the function execution time. The overhead scales linearly with the number of nodes because of the need to check the current workload distribution across nodes. More nodes result in higher overhead.

Considering the memory overhead of Ephemera, we will discuss the sources of memory consumption for each component separately. The main memory overhead of the Runtime Daemon, which primarily stems from storing file information, is less than one MB in our experiments and is negligible. The Tenant Manager, which maintains the Container Pool and stores information for each container in memory, consumes less than one MB in experiments and is negligible. The Cluster Controller, which needs to temporarily store information about each node, consumes less than one MB in experiments and is also negligible.

6 Discussion

Instruction Interception Capability of the In-Memory File System. As a prototype system, the current in-memory file system can perform primary I/O operations, including file creation, reading, and writing. Future research needs to focus on expanding the range of intercepted instructions to support broader application needs to deploy EPHEMERA in practice. Additionally, the Daemon now serves as a wrapper for function instances, providing interception interfaces for C language I/O operations. This architecture allows for future expansion to accommodate various programming languages.

Profiling for Input-sensitive Tasks. In the Serverless platform, several existing methods [3, 27, 37] predict the actual execution effects of functions based on the input data, and these methods are orthogonal to our approach. We can integrate these techniques to predict memory and file size consumption based on real-time input data, thereby enabling a more precise and efficient memory-sharing mechanism for Serverless functions.

Expansion of Memory-Sharing Mechanisms. The current memory-sharing mechanism of the Tenant Manager is limited to single-node, which restricts the system's scalability and flexibility. To implement a more robust memory-sharing mechanism, future research could consider introducing designs for multi-node systems, such as enabling cross-node memory access through **remote direct memory access** (**RDMA**) technology [2, 11, 21]. Introducing this technology would allow the in-memory file system to share and manage memory resources more effectively in a distributed environment, thereby enhancing the overall performance and scalability of the system.

7 Related Work

Serverless File System. To support generic tasks, serverless platforms need to allow connections to file-based storage systems. Merenstein et al. [26] design the stackable file system F3, which features locality-aware data scheduling, can distinguish between ephemeral data and data requiring high durability, and transparently directs ephemeral data to node-local disks. Schleier-Smith et al. [30] propose FaaSFS, a shared file system, which optimistically handles POSIX calls, utilizing locally cached state and encapsulating cloud function file system interactions within a transaction mechanism to restore consistency in the event of conflicts. RunD [16], a lightweight secure container runtime, uses virtio-fs to support the read-only part of rootfs for sharing page cache between host and guests and uses virtio-blk to support the writeable part of rootfs for high I/O performance. Hattori et al. [10] introduce a runtime called Sentinel to mitigate the cold start latency and memory usage with a read-only mount to the specified file system. Ephemera is one that not only accelerates file access in single-containers but also enables multi-container memory sharing, which is orthogonal to previous designs.

Tiered File System. Tiered file systems manage a hierarchy of heterogeneous storage devices, placing data in storage devices that match the data's performance requirements and the application's future access patterns. AutoTiering [36] manages the allocation and migration of virtual machine disk files (VMDK) in all-flash multi-tier data centers. With the rise of **Non-Volatile Main Memory** (**NVMM**), Zheng et al. [42] introduce a tiered file system named Ziggurat, which utilizes an efficient migration mechanism, leveraging the characteristics of different storage devices to achieve high migration efficiency. Zheng et al. [43] introduce TPFS, a tiered file system that integrates byte-addressable **persistent memory** (**PM**) and slow disks to establish a storage system with performance close to the PM and substantial capacity. These efforts skillfully combine the characteristics of individual storage media, such as the high performance of memory and the high capacity of disks. On top of that, they also explore how to allocate content to the corresponding storage media more rationally. These works can be used as underlying optimizations in conjunction with Ephemera, and to some extent share similar characteristics with this article's use of free memory to optimize user memory usage in serverless computing platforms.

Resource Harvesting. The inherent pre-allocation of resources in serverless computing results in underutilization during idle periods, and resource harvesting is a prominent research direction to tackle this inefficiency [8, 34, 38]. Yu et al. [37] propose Libra, a comprehensive provider-side solution that safely and timely harvests idle resources to accelerate large-scale serverless function invocations with varying inputs. However, Libra's resource harvesting and acceleration rely on machine learning models for estimating resource demands and timeliness. Zhang et al. [40] characterize the serverless workloads and Harvest VMs on Microsoft Azure, and devise a serverless load balancer capable of discerning evictions and resource variations within Harvest VMs. Freyr$^+$ [39] is a novel resource manager designed for serverless platforms. It conducts real-time monitoring of each function's resource utilization, dynamically harvesting idle resources from over-provisioned functions, and accelerates under-provisioned functions by supplementing them with additional resources. Different from harvesting idle resources in serverless systems, Liu et al. [19] design SMore to reclaim available GPU resources by collocating serverless functions with existing cloud workloads. Ephemera changes the pattern of memory allocation from single-instance to single-tenant, harvesting memory resources wisely to accelerate file access.

8 Conclusion and Future Work

In this article, we propose Ephemera, a framework based on serverless computing to optimize the efficiency of ephemeral storage access and memory utilization. Ephemera consists of three components top-down: a *cluster-level* Controller, a *tenant-level* Manager, and a *function-level* Daemon. The Runtime Daemon, using an in-memory file system, transforms file operations to memory-based for I/O optimization. The Tenant Manager enables dynamic memory sharing with security. The Cluster Controller dynamically adjusts workload and resources for optimal performance. Through prototype testing, Ephemera demonstrates an average 50% reduction in file access latency and even 95.73% reduction in specific conditions. As part of our future work, we plan to enhance the system's functionality and scalability by expanding instruction interception, integrating input-sensitive profiling, and enabling distributed memory sharing.

References

[1] Mohammad Sadegh Aslanpour, Adel N. Toosi, Muhammad Aamir Cheema, and Mohan Baruwal Chhetri. 2024. Faashouse: Sustainable serverless edge computing through energy-aware resource scheduling. *IEEE Transactions on Services Computing* 17, 4 (2024), 1533–1547. DOI: https://doi.org/10.1109/TSC.2024.3354296

[2] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, et al. 2023. Empowering azure storage with {RDMA}. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*. 49–67.

[3] Vivek M. Bhasi, Jashwant Raj Gunasekaran, Aakash Sharma, Mahmut Taylan Kandemir, and Chita Das. 2022. Cypress: Input size-sensitive container provisioning and request scheduling for serverless platforms. In *Proceedings of the 13th Symposium on Cloud Computing*. 257–272.

[4] Vivek M. Bhasi, Aakash Sharma, Shruti Mohanty, Mahmut Taylan Kandemir, and Chita R. Das. 2024. Paldia: Enabling SLO-compliant and cost-effective serverless computing on heterogeneous hardware. In *Proceedings of the 2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 100–113.

[5] Zinuo Cai, Zebin Chen, Ruhui Ma, and Haibing Guan. 2024. SMSS: Stateful model serving in metaverse with serverless computing and GPU sharing. *IEEE Journal on Selected Areas in Communications* 42, 3 (2024), 799–811. DOI: https://doi.org/10.1109/JSAC.2023.3345401

[6] Marcin Copik, Roman Böhringer, Alexandru Calotoiu, and Torsten Hoefler. 2023. Fmi: Fast and cheap message passing for serverless functions. In *Proceedings of the International Conference on Supercomputing*. 373–385.

[7] Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. 2022. Serverless computing on heterogeneous computers. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems.*

[8] Alexander Fuerst, Stanko Novaković, Íñigo Goiri, Gohar Irfan Chaudhry, Prateek Sharma, Kapil Arya, Kevin Broas, Eugene Bak, Mehmet Iyigun, and Ricardo Bianchini. 2022. Memory-harvesting VMs in cloud platforms. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems.* 583–594.

[9] Muhammed Golec, Guneet Kaur Walia, Mohit Kumar, Felix Cuadrado, Sukhpal Singh Gill, and Steve Uhlig. 2024. Cold start latency in serverless computing: A systematic review, taxonomy, and future directions. *ACM Computing Surveys* 57, 3 (2024), 1–36.

[10] Joe Hattori and Shinpei Kato. 2022. Sentinel: A fast and memory-efficient serverless architecture for lightweight applications. In *Proceedings of the 8th International Workshop on Serverless Computing.* 13–18.

[11] Jialiang Huang, MingXing Zhang, Teng Ma, Zheng Liu, Sixing Lin, Kang Chen, Jinlei Jiang, Xia Liao, Yingdi Shan, Ning Zhang, et al. 2024. TrEnv: Transparently share serverless execution environments across different functions and nodes. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles.* 421–437.

[12] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. 2021. Towards demystifying serverless machine learning training. In *Special Interest Group on Management of Data.*

[13] Lingxiao Jin, Zinuo Cai, Zebin Chen, Hongyu Zhao, and Ruhui Ma. 2025. AARC: Automated affinity-aware resource configuration for serverless workflows. In *Proceedings of the Design Automation Conference.*

[14] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. Retrieved from https://arxiv.org/abs/1902.03383

[15] Jeongchul Kim and Kyungyong Lee. 2019. Functionbench: A suite of workloads for serverless cloud function service. In *Proceedings of the IEEE International Conference on Cloud Computing.*

[16] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. 2022. RunD: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing. In *Proceedings of the Usenix Annual Technical Conference.*

[17] Zijun Li, Linsong Guo, Jiagan Cheng, Quan Chen, BingSheng He, and Minyi Guo. 2022. The serverless computing survey: A technical primer for design architecture. *ACM Computing Surveys (CSUR)* 54, 10s (2022), 1–34.

[18] Zijun Li, Chuhao Xu, Quan Chen, Jieru Zhao, Chen Chen, and Minyi Guo. 2023. DataFlower: Exploiting the data-flow paradigm for serverless workflow orchestration. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4.* 57–72.

[19] Junhan Liu, Zinuo Cai, Yumou Liu, Hao Li, Zongpu Zhang, Ruhui Ma, and Rajkumar Buyya. 2025. SMore: Enhancing GPU utilization in deep learning clusters by serverless-based co-location scheduling. *IEEE Transactions on Parallel and Distributed Systems* 36, 5 (2025), 903–917.

[20] Xuanzhe Liu, Jinfeng Wen, Zhenpeng Chen, Ding Li, Junkai Chen, Yi Liu, Haoyu Wang, and Xin Jin. 2023. Faaslight: General application-level cold-start latency optimization for function-as-a-service in serverless computing. *ACM Transactions on Software Engineering and Methodology* 32, 5 (2023), 1–29.

[21] Fangming Lu, Xingda Wei, Zhuobin Huang, Rong Chen, Minyu Wu, and Haibo Chen. 2024. Serialization/deserialization-free state transfer in serverless workflows. In *Proceedings of the 19th European Conference on Computer Systems.* 132–147.

[22] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. 2022. ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation.*

[23] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Eshaan Minocha, Sameh Elnikety, Saurabh Bagchi, and Somali Chaterji. 2022. Wisefuse: Workload characterization and DAG transformation for serverless workflows. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6, 2 (2022), 1–28.

[24] Anupama Mampage, Shanika Karunasekera, and Rajkumar Buyya. 2022. A holistic view on resource management in serverless computing environments: Taxonomy and future directions. *ACM Computing Surveys (CSUR)* 54, 11s (2022), 1–36.

[25] Sunilkumar S. Manvi and Gopal Krishna Shyam. 2014. Resource management for Infrastructure as a Service (IaaS) in cloud computing: A survey. *Journal of Network and Computer Applications* 41 (2014), 424–440. DOI: https://doi.org/10.1016/j.jnca.2013.10.004

[26] Alex Merenstein, Vasily Tarasov, Ali Anwar, Scott Guthridge, and Erez Zadok. 2023. F3: Serving files efficiently in serverless computing. In *Proceedings of the ACM International Conference on Systems and Storage.*

[27] Arshia Moghimi, Joe Hattori, Alexander Li, Mehdi Ben Chikha, and Mohammad Shahrad. 2023. Parrotfish: Parametric regression for optimizing serverless functions. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*. 177–192.

[28] Shanxing Pan, Hongyu Zhao, Zinuo Cai, Dongmei Li, Ruhui Ma, and Haibing Guan. 2024. Sustainable serverless computing with cold-start optimization and automatic workflow resource scheduling. *IEEE Transactions on Sustainable Computing* 9, 3 (2024), 329–340. DOI: https://doi.org/10.1109/TSUSC.2023.3311197

[29] Ali Raza, Nabeel Akhtar, Vatche Isahagian, Ibrahim Matta, and Lei Huang. 2023. Configuration and placement of serverless applications using statistical learning. *IEEE Transactions on Network and Service Management* 20, 2 (2023), 1065–1077.

[30] Johann Schleier-Smith, Leonhard Holz, Nathan Pemberton, and Joseph M. Hellerstein. 2020. A FaaS File System for Serverless Computing. Retrieved from https://arxiv.org/abs/2009.09845

[31] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. 2022. Serverless computing: A survey of opportunities, challenges, and applications. *ACM Computing Surveys (CSUR)* 54, 11s (2022), 1–32.

[32] Jovan Stojkovic, Tianyin Xu, Hubertus Franke, and Josep Torrellas. 2023. Mxfaas: Resource sharing in serverless environments for parallelism and efficiency. In *Proceedings of the International Symposium on Computer Architecture*.

[33] Weiguo Wang, Quanwang Wu, Zhiyong Zhang, Jie Zeng, Xiang Zhang, and Mingqiang Zhou. 2024. A probabilistic modeling and evolutionary optimization approach for serverless workflow configuration. *Software: Practice and Experience* 54, 9 (2024), 1697–1713.

[34] Yawen Wang, Kapil Arya, Marios Kogias, Manohar Vanga, Aditya Bhandari, Neeraja J. Yadwadkar, Siddhartha Sen, Sameh Elnikety, Christos Kozyrakis, and Ricardo Bianchini. 2021. Smartharvest: Harvesting idle CPUs safely and efficiently in the cloud. In *Proceedings of the 16th European Conference on Computer Systems*. 1–16.

[35] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. 2022. INFless: A native serverless system for low-latency, high-throughput inference. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.

[36] Zhengyu Yang, Morteza Hoseinzadeh, Allen Andrews, Clay Mayers, David Thomas Evans, Rory Thomas Bolt, Janki Bhimani, Ningfang Mi, and Steven Swanson. 2017. AutoTiering: Automatic data placement manager in multi-tier all-flash datacenter. In *Proceedings of the IEEE International Performance Computing and Communications Conference*.

[37] Hanfei Yu, Christian Fontenot, Hao Wang, Jian Li, Xu Yuan, and Seung-Jong Park. 2023. Libra: Harvesting idle resources safely and timely in serverless clusters. In *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing*.

[38] Hanfei Yu, Hao Wang, Jian Li, Xu Yuan, and Seung-Jong Park. 2022. Accelerating serverless computing by harvesting idle resources. In *Proceedings of the ACM Web Conference*.

[39] Hanfei Yu, Hao Wang, Jian Li, Xu Yuan, and Seung-Jong Park. 2024. Freyr$^+$: Harvesting idle resources in serverless computing via deep reinforcement learning. *IEEE Transactions on Parallel and Distributed Systems* 35, 11 (2024), 2254–2269. DOI: https://doi.org/10.1109/TPDS.2024.3462294

[40] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. 2021. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM Symposium on Operating Systems Principles*.

[41] Hongyu Zhao, Shanxing Pan, Zinuo Cai, Xinglei Chen, Lingxiao Jin, Honghao Gao, Shaohua Wan, Ruhui Ma, and Haibing Guan. 2024. faaShark: An end-to-end network traffic analysis system atop serverless computing. *IEEE Transactions on Network Science and Engineering* 11, 3 (2024), 2473–2484. DOI: https://doi.org/10.1109/TNSE.2023.3294406

[42] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. 2019. Ziggurat: A tiered file system for non-volatile main memories and disks. In *Proceedings of the USENIX Conference on File and Storage Technologies*.

[43] Shengan Zheng, Morteza Hoseinzadeh, Steven Swanson, and Linpeng Huang. 2023. TPFS: A high-performance tiered file system for persistent memories and disks. *ACM Transactions on Storage* 19, 2 (2023), 1–28.

Received 21 January 2025; revised 24 May 2025; accepted 27 June 2025