Serialization/Deserialization-free State Transfer in Serverless Workflows

Fangming Lu[1] Xingda Wei[1,2] Zhuobin Huang[†3] Rong Chen[1,2] Minyu Wu[1,2] Haibo Chen[1]

[1]Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University
[2]Shanghai Artificial Intelligence Laboratory [3] University of Electronic Science and Technology of China

Abstract

Serialization and deserialization play a dominant role in the state transfer time of serverless workflows, leading to substantial performance penalties during workflow execution. We identify the key reason as a lack of ability to efficiently access the (remote) memory of another function. We propose RMMAP, an OS primitive for remote memory map. It allows a serverless function to directly access the memory of another function, even if it is located remotely. RMMAP is the first to completely eliminates serialization and deserialization when transferring states between any pairs of functions in (unmodified) serverless workflows. To make remote memory map efficient and feasible, we co-design it with fast networking (RDMA), OS, language runtime, and serverless platform. Evaluations using real-world serverless workloads show that integrating RMMAP with Knative reduces the serverless workflow execution time on Knative by up to 2.6× and improves resource utilizations by 86.3%.

ACM Reference Format:
Fangming Lu, Xingda Wei, Zhuobin Huang, Rong Chen, Minyu Wu, and Haibo Chen. 2024. Serialization/Deserialization-free State Transfer in Serverless Workflows. In *European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3627703.3629568

1 Introduction

Serverless computing is an emerging cloud computing paradigm widely adopted by major vendors including AWS Lambda [24], Microsoft Azure Functions [61], Alibaba Serverless Application Engine [29] and Huawei Cloud Functions [40]. The basic unit of serverless computing is the *function*, which is user code that can be executed elastically by the platform. These functions can be combined into a *workflow* to simplify the development of complex applications. The workflow is usually represented by a directed acyclic graph (DAG) that specifies the execution dependencies between functions.

To improve resource isolation, management, and deployment, each function is executed in separate containers, which unfortunately introduces performance penalties for state transfer in serverless workflows. State transfer involves passing the computed state of one function (producer) to the next function (consumer) in the workflow. Because functions are isolated in containers, they cannot directly transfer state using traditional shared memory, as in a normal program. Instead, they must rely on message passing (*Messaging*) or distributed storage systems (*Storage*), which have been shown to account for up to 95% of the workflow execution [38, 50].

Reducing the cost of messaging or storage is challenging due to the involvement of serialization and deserialization—(de)serialization.[1] Specifically, serialization will transform the state (e.g., a Python object) at the producer into a byte array that can be transferred through the network (or stored in a storage). After the byte array is received by the consumer, deserialization will reconstruct the object from it. Our analysis (§2.3) revealed that both processes have non-trivial computation and memory copy overheads: they occupy 42–98% and 17–97% of the time during state transfer in real-world serverless workflows (§2.3). The high cost can be attributed to two reasons. First, serverless workflows are typically written in high-level languages like Python [31], which means they often transfer complex runtime objects. Unfortunately, complex objects have a high (de)serialization cost. For example, serializing a 3.2 MB Python pandas dataframe object needs transform 401,839 sub-objects to generate the byte array, which takes 10 ms. Second, functions are typically ephemeral. Lambda@Eedge has reported that 67% of its functions run in less than 20 ms [31]. Thus, the overhead of (de)serialization cannot be amortized by long-running functions.

Previous work that focuses on optimizing state transfer has largely ignored the cost of (de)serialization [22, 49, 50, 56, 67, 79]. , which is non-optimal. A few works eliminate (de)serialization in a restricted setting, i.e., when producer and consumer run on the same machine [50, 67]. Such a constraint is not suitable for serverless computing because first, it restricts the maximum number of concurrently running functions on a single machine, since a single machine has limited resources. Second, it may result in resource underutilization, because we have to allocate functions from the same workflow to a single machine, even if that machine is overloaded while others remain idle. Unfortunately, relaxing such a constraint is non-trivial. When running producer and consumer functions on the same machine, they are only isolated by the OS, so we can leverage existing OS primitives (e.g., shared memory) to share state. However, when running them on different machines, current OS lacks the ability to directly share memory between machines.

[†] Work done while intern at Institute of Parallel and Distributed Systems.

EuroSys '24, April 22–25, 2024, Athens, Greece

ACM ISBN 979-8-4007-0437-6/24/04…$15.00
https://doi.org/10.1145/3627703.3629568

[1]We use the term "(de)serialization" to refer to both serialization and deserialization.

In this paper, we present a systematic approach that eliminates (de)serialization when transferring states between any pairs of functions. The key observation is that as long as the consumer can directly access the memory of the producer, we no longer need (de)serialization to transform and reconstruct the object. The producer only needs to send a pointer of the state to the consumer, which is extremely efficient. Based on this observation, we propose RMMAP: an OS primitive that can map the memory of the remote producer container into the address space of the consumer container. If the consumer accesses the pointer of the producer's state (including the pointers pointed by the state), the OS will transparently read the mapped physical page from the producer machine.

Although the insights behind RMMAP are simple, we need to address the following challenges to make it feasible for serverless workflows (§3.1). First, how to efficiently read remote pages on the OS? Transferring a 4 KB page with messaging is slow. Meanwhile, if the state is scattered over the memory space of the producer, we need multiple network roundtrips to read it, causing network amplification. To this end, we adopt a hardware-software co-designed approach for better performance. First, we leverage RDMA (§4.1), a modern data center networking feature that is capable of offloading remote page reads to the hardware with high throughput and low latency. Our co-design fully utilizes RDMA's one-sided primitive for high-performance remote paging. Second, we design a semantic-aware prefetch scheme (§4.4), which precisely prefetches all the pages of the state in one batched network request without network amplification.

Second, the remote memory space of the producer may conflict with the local space of the consumer, particularly when the consumer needs to read states from multiple producers. A simple solution is to map the remote memory to a new address space. This is incorrect because the pointers in the remote memory space may have self-references. RMMAP plans the address spaces of function containers to prevent conflicts (§4.2). This is based on the observation that the serverless platform is aware of the execution dependencies of containers based on the dependencies of functions.

Finally, serverless functions are commonly written in high-level languages such as Python or Java [31]. However, existing language runtimes are unaware of RMMAP, which lacks a mechanism to manage cross-address space objects. We propose an efficient remote object management scheme for using RMMAP in serverless functions (§4.3).

We have implemented RMMAP in Linux and extended Python and Java—the dominating languages for serverless computing [31] to incorporate it. To demonstrate the effectiveness, we deploy our augmented OS and runtime on Knative, and further extend it to support RMMAP-aware workflow execution. On Knative, RMMAP improves the performance of representative serverless workflows including financial regulation and machine learning tasks by 1.4–2.6×. Under the same client request rate, it further reduces the Knative resources to run workflows by up to 86.3%.

Efficiently eliminating (de)serialization in serverless computing is not an easy task. Our journey towards developing RMMAP reveals that a co-design of RDMA, OS, language runtime, and serverless platform is necessary. This is because traditional software stacks are not designed for (de)serialization-free execution, and traditional networking primitives are not efficient enough for remote memory accesses. One might wonder whether such a co-design is suitable for serverless workflows, i.e., how easily they can benefit from our approach. We believe that RMMAP is feasible because unmodified serverless applications can seamlessly enjoy the performance benefits provided by RMMAP, resulting in significant reductions in execution time.

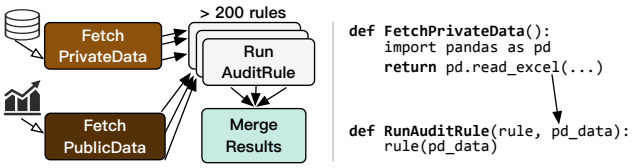

Figure 1. ***An illustration of a real-world serverless workflow [2].***

Contributions. Our highlighted contributions are:

- **Problem**: A detailed analysis of the serialization and deserialization costs in serverless workflows (§2).
- **RMMAP**: An efficient OS primitive that can map the memory of a remote container to a local one with RDMA, plus a scheme to incorporate this primitive into serverless platforms and language runtimes to eliminate the need of (de)serialization in serverless workflows (§4).
- **Demonstration**: An implementation on Linux, Python, Java and Knative with extensive evaluations on real-world serverless workflows that confirm RMMAP's efficacy (§5).

The source code of RMMAP is available at https://github.com/ProjectMitosisOS/dmerge-eurosys24-ae.

2 Background and Motivation

2.1 Serverless computing and serverless workflow

Serverless function. Serverless computing simplifies building and deploying cloud applications: Developers only need to write applications as *functions* in a high-level programming language (e.g., Python [31], Java [64]), upload these functions to the cloud vendors (via containers), and specify how to invoke them. Upon invocations, the cloud platform automatically manages the deployment of each function, including resource allocation scaling, and load balancing. Thus, serverless computing relieves developers from deploying and managing containers to run functions, and scaling these resources in case of workload increases.

Serverless workflow. A single function typically cannot encapsulate a complex application. Thus, serverless frameworks further allow developers to compose multiple functions into a *workflow* (e.g. AWS Step Functions [8]), where the workflow is abstracted as a directed acyclic graph (DAG). In the DAG, each node corresponds to one serverless function, while edges represent how states are transferred between functions. For example, $A \rightarrow B$ signifies that the return value of function A is the input to B[2]. Without losing generality, we term the function that produces the state *producer* (e.g., A) and the function that consumes the state *consumer* (e.g., B).

To coordinate the execution of different functions, i.e., function B must be called after function A returns. The coordination is typically

[2]We may use the function name to refer to its hosting container in this paper.

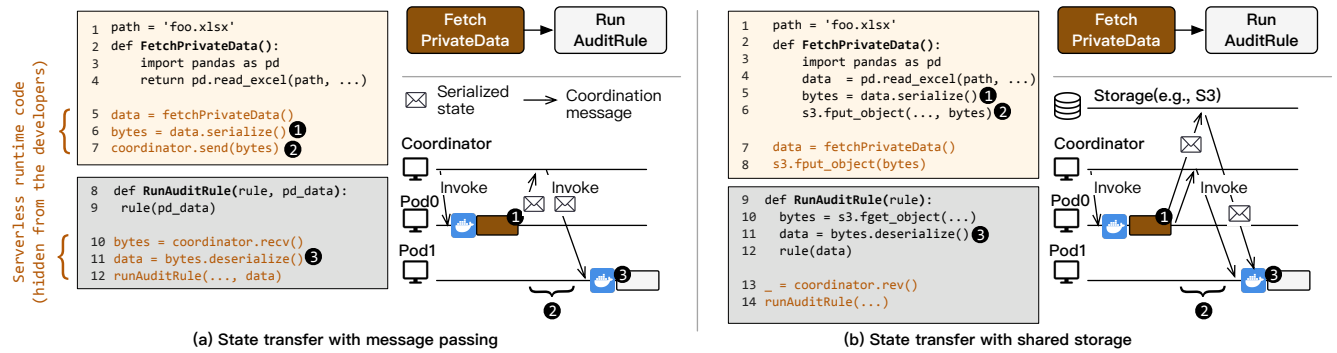

Figure 2. ***An overview of state transfer between two functions using (a) messaging and (b) shared storage. Functions are executed in containers at each machine (pod). The serverless runtime code will be called after receiving the coordinator message.***

done by a central component we term *coordinator* (e.g., broker in Knative [15]).

Motivating example: FINRA. Figure 1 shows a real-world serverless workflow from AWS Lambda [2]. It is a financial application that validates trades based on different sources of data. The workflow has four types of functions: two functions to prepare the input data (`FetchPrivateData` and `FetchPublicData`), one function to validate the data (`RunAuditRule`), and one to collect the validation results (`MergeResults`). Note that a single function type can be concurrently invoked, e.g., FINRA will concurrently invoke more than 200 `RunAuditRules` to validate different rules concurrently [12]. Different types of functions may have dependencies. For example, `FetchPrivateData` will prepare the fetched data to a Python pandas dataframe [17] to simplify `RunAuditRule` processing [50].

2.2 State transfer in serverless workflows

As we have mentioned in §1, existing systems have to deploy heavyweight mechanisms, namely, *message passing* and *shared storage*, to transfer states between functions that run in different containers.

State sharing in serverless workflows. More specifically, Figure 2 illustrates how existing serverless functions share states with message passing and shared storage. Without losing generality, we only consider two instances of function that need state transfer, i.e., one `FetchPrivateData` (producer) and one `RunAuditRule` (consumer) described in our motivating example. Note that how the function is invoked with the help of the coordinator is typically hidden from the developers through serverless framework code (e.g., Line 5–7 in Figure 2 (a)).

- *Message passing (Messaging).* As shown in Figure 2 (a), different functions share states by piggybacking states in the messages exchanged with the coordinator. The framework first uses a serialization tool (e.g., pickle [18] in Python) to generate the message from the object (❶). After receiving the message, the framework will further deserialize the message (❸) with the same tool to reconstruct the object. An important feature of messaging is that the message exchanged has a long execution path (❷): it will pass multiple components (e.g., gateways) before reaching the coordinator (and vice verse). As a result, sending a large message will significantly slowdown the workflow execution.

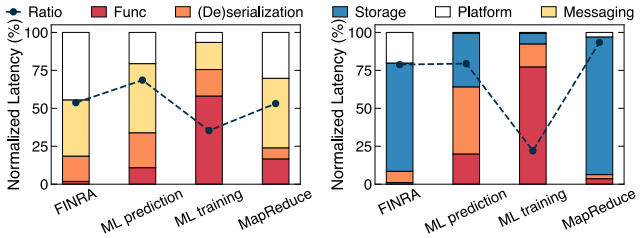

Figure 3. ***Analysis of the state transfer costs in representative serverless workflows: (a) using messaging and (b) shared storage. Func: function execution time, Ratio: the ratio of time of state transfer in the end-to-end time.***

- *Shared storage (Storage).* Since sending large data with messaging is slow, the platform typically limits the maximum payload, e.g., 256 KB in AWS [25]. For large states, functions need to transfer them with shared storage (e.g., S3 [3]). As shown in Figure 2 (b), the `FetchPrivateData` will first serialize the data into a file (❶) and upload it to the storage (❷). On the `RunAuditRule` side, it will read the file, deserialize the state from the file(❸) and finally execute the function. Note that, unlike messaging, transferring state with storage typically requires additional coding and configuration, as the storage is not free.

2.3 Analysis of the costs in serverless workflow

Figure 3 shows the breakdown of the state transfer costs to the end-to-end latency of representative realistic serverless workflows.[3] We deploy workflows on Knative [15], a state-of-the-art open-source serverless platform. For messaging, functions use cloudevents [10]—the default messaging primitive of Knative. For shared storage, we deploy Pocket [49], a state-of-the-art storage system for serverless computing. We pre-warm the evaluated functions to rule out interference from coldstart. We can see that the state transfer takes 42–98% and 17–97% of the workflow execution time for message passing and shared storage, respectively. We have identified the following major sources of costs:

Source #1. Platform overhead. The time required for the coordinator to invoke functions, as well as the time needed to schedule the functions for execution, accounts for 0.6–44% of the overall

[3] Detailed setups are listed in §5.1.

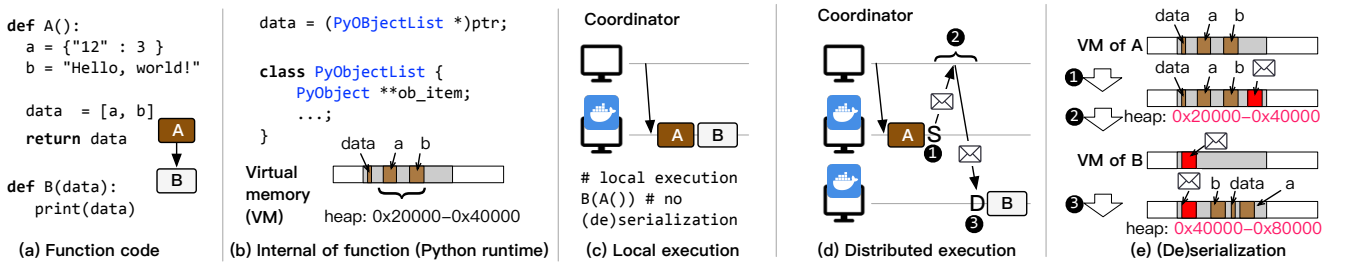

Figure 4. *An overview of serialization (S) and deserialization (D) for functions that share state across machines.*

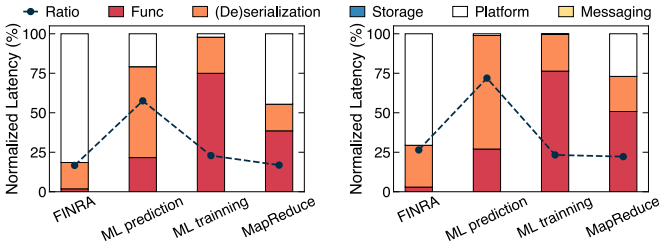

Figure 5. *End to end analysis of (de)serialization cost in serverless workflows excluding the messaging (a) and storage overhead (b).*

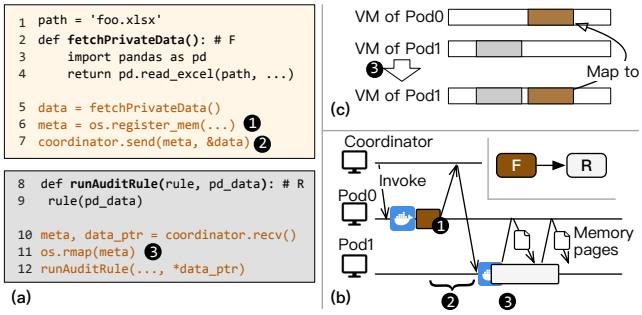

Figure 6. *An overview of serverless workflow with* RMMAP. *(a) How serverless framework leverages* RMMAP *to transfer state. (b) The detailed execution flow of state transfer and (c) How pod's virtual memory space changes with* RMMAP.

workflow execution time. These overheads are inherent to the platform [68] and are orthogonal to our work.

Source #2. Messaging and storage software overhead. Messaging and storage take 35–69% and 22–93% of the workflow execution time for various workflows, respectively. For messaging, the time is spent on sending/receiving messages with the coordinator, which is non-trivial since the message passes many components in Knative. For storage, the time is dominated by storage protocol overheads.

Reducing state transfer cost is an active research topic [22, 49, 50, 56, 67, 79, 91]. Most of them focus on reducing the aforementioned software costs. Nevertheless, they are far from optimal because they cannot eliminate the (de)serialization costs described below.

Source #3. (De)serialization overhead. To highlight the limitation of solely optimizing messaging or storage, Figure 5 shows an emulated setup where the software overhead of messaging or storage overhead is made zero. Even under such a setup, (de)serialization take 17–58% and 22–72% of the workflow execution time. We do the emulation by sending a zero byte message and not reading/write the storage. (De)serialization involves complex calculations on data types and extensive memory copies to compact a data type into a continuous by binary, whose overhead is more challenging to reduce than messaging or storage.

2.4 The necessities and costs of (de)serialization

(De)serialization is necessary because the producer's memory cannot be accessed by the consumer. Figure 4 (a) shows an example where function *A* shares a python list (*data*) with *B*. For functions written in python, the *data* is internally represented as multiple *PyOBject* on the runtime's heap (Figure 4 (b)). If *A* and *B* are executed on a single machine within a container (see Figure 4 (c)), *B* can directly access the pointer of *data* since they share the same address space. However, if we run these functions on different machines (Figure 4 (d)), *B* cannot access the pointer of *data* due to physically isolated address spaces. Thus, as shown in Figure 4 (e), we have to first serialize the data into a byte array (❶) that can be transferred through the network (❷). After receiving the array, *B* can reconstruct the list in its address space with deserialization (❸).

The performance penalties of (de)serialization are twofolds. First, it has computation costs for transforming/reconstructing an object to/from a byte array. Specifically, serialization will traverse all the reachable objects pointed by the root object of the state. In our FINRA example, serializing a 3.2 MB dataframe needs traversing 401,839 sub-objects. Moreover, copying objects to a continuous buffer has huge memory copy overhead. For reference, copying a 4 MB object takes 2.5 ms.[4]

3 Overview of RMMAP

Our approach: remote memory map. To eliminate (de)serialization, we propose a new OS primitive—RMMAP—an extension to local `mmap` that can map a remote range of memory into a local process, inspired by recent abstraction for using RDMA [21]. The goal is to bridge the address spaces of containers (run as processes) together. RMMAP has two core system calls (§4.1). The producer can call `register_mem` to register its virtual memory space on the caller machine. After knowing the producer's space information, a consumer can call `rmap` to map the producer space into its own. Finally, the consumer can read and write the state in the remote memory in a shared memory paradigm without (de)serialization.

[4]Note that since the copies in serialization are typically performed by a single thread with cache misses, the achieved bandwidth is much smaller than the DRAM bandwidth.

Execution flow with RMMAP. Figure 6 (a) illustrates how functions leverage RMMAP to transfer state and (b) shows the detailed execution flow, where the producer (`FetchPrivateData`) and consumer (`RunAuditRule`) run on two machines (pod0 and pod1), respectively. To share *data*, pod0 first calls `register_mem` (Line 6) to expose its memory to future consumers (❶). Afterward, it sends the memory information (*meta*)—including the starting address and size, together with the pointer of the state to the coordinator (❷). Next, the coordinator sends these information to pod1 for the consumer function. Finally, the consumer calls `rmap` (❸) to map the producer's memory to consumer's address space (see Figure 6 (c)). With the mapped remote memory, the consumer can directly access state (*data_ptr*) like a local object: if it touches a virtual memory belonging to the producer, the pod1's kernel will use a network request to read the physical page from the producer's machine.

3.1 Challenges and solutions

How to reduce the costs of remote page fetch? In RMMAP, the consumer can issue multiple network requests to fetch the remote pages associated with a state. Following the traditional networking primitive—sending a message to the producer to read the page is slow, since messaging involves extensive memory copies and huge CPU occupations. Meanwhile, it introduces network amplification, i.e., using multiple network requests to read a single state. To make RMMAP practical, we need to implement a fast remote paging mechanism and minimize network amplification.

Solution: RDMA (§4.1) with semantic-aware prefetch (§4.4). First, we co-design RMMAP with RDMA, a networking feature that enables zero-copy and CPU-bypassing remote read with extremely high performance. Second, we adopt a semantic-aware prefetching method. This method is based on the observation that, with the help of language runtime, we can discover most pages that are related to the state. Thus, we can use one RDMA request to read these pages [45, 89].

How to prevent map address conflicts? Unlike local `mmap`, we can only map the remote memory to the original remote address, since the state in the memory may points to other objects in the same address range. Thus, RMMAP alone is insufficient for serverless workflows.

Solution: co-designed address planning (§4.2). We co-design RMMAP with the serverless framework to pre-plan the address space for each workflow function, thereby preventing address conflicts. Specifically, each function is assigned a unique address space that doesn't overlap with others. Address space planning is always possible because first, serverless workflow graph is known beforehand. Second, serverless function must run with fixed memory budget [9, 11].

How to support high-level languages? RMMAP changes the scope of object management in high-level languages' runtime (e.g., Python): the managed objects may span on multiple heaps instead of the local heap. Executing the object management procedures (e.g., garbage collection, GC) naively on remote objects is inefficient because it involves extra remote memory reads [88].

Solution: a hybrid GC (§4.3). We adopt a hybrid GC that incorporates a corse-grained GC to manage the remote heap. The idea is to leverage the semantic of serverless workflow to reduce GC overhead: in a workflow, we only need to trace the lifecycle of the shared state.

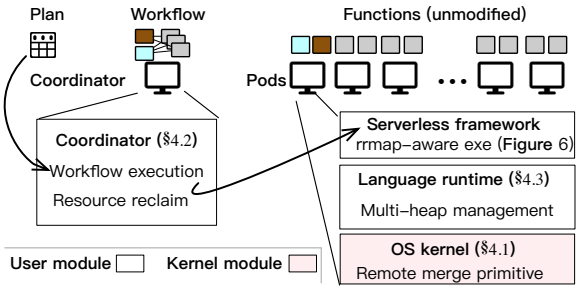

Figure 7. *System architecture and key roles of each module.*

Thus, we can manage the remote heap by only focusing on the root object of the state, which is simple and efficient.

3.2 System architecture

Figure 7 gives an overview of the system components of RMMAP. Following a common setup [15, 53, 57], each workflow is executed by a coordinator, invoking functions on multiple worker nodes (i.e., pods).

When a developer uploads a workflow to the platform, we first generate a plan (§4.2) assigning non-conflicting address range to each function in the workflow. This plan is enforced by the container that execute the function such that the coordinator can map the memory of one function to another during its execution. The coordinator also monitors the lifecycle of the mapped memory and reclaim them if necessary.

Each function executing on the pod is wrapped with a library provided by our extended serverless framework, abstracting the workflow coordination from the function developers. It will call RMMAP primitives to enable (de)serialization-free state sharing between functions (see Figure 6). The framework as well as the user function is executed on our extended language runtime (§4.3), supporting accessing objects in multiple heaps. Finally, the language runtime is layered on our extended kernel that supports the RMMAP primitive (§4.1).

Security and network model. Our security model is the same as that of containers: we trust the OS and the hardware, but assumes malicious functions may exist. Besides, to use RMMAP, the producer function must trust the consumer function, i.e., the consumer function is allowed to read the producer's memory. This is a common case since most workflow functions are from the same application. If not, developers can annotate their workflow graph so that we can fallback to traditional methods, i.e., messaging. Finally, we assume that machines executing serverless functions are connected via RDMA, which is widely deployed in modern data centers [37, 83].

4 Detailed Design

4.1 The RMMAP operating system primitive

Major interfaces (Table 1). Producer function calls `register_mem` to register its memory to the RMMAP kernel. To uniquely identify the registered memory, we require a function ID and a key from the caller. These fields will be further stored at the kernel for future authentication. A successful registration returns the information required for a later `rmap`, e.g., the registered virtual address

Table 1: *Main interfaces exposed by* **RMMAP** *kernel.*

System call interface	Description	Caller in Serverless
`register_mem(id, key, vm_start, vm_end)->vm_meta`	Register a virtual memory range of the caller to the kernel. Returns an identifier of the registered memory. Note that the memory range will be marked as copy-on-write.	Function
`rmap(mac_addr, id, key, vm_start, vm_end)-> result`	Map the remote virtual memory at the caller process.	Function
`deregister_mem(job_id, key)-> result`	Reclaim the registered memory from the kernel.	Framework
`set_segment(seg, vm_start, vm_end)`	Set the segment of of the process in a fixed memory range.	Function

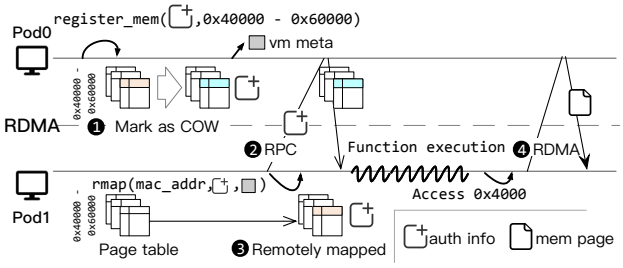

Figure 8. *An illustration of major* **RMMAP** *system call (see Table 1) execution, i.e.,* ***`register_mem`*** *and* ***`rmap`****.*

space range. Note that our kernel will keep the registered memory even if the caller exits.

Given the information of the registered memory, the consumer can call `rmap` to map the remote memory to its own address space. The ID and key are also needed for the authentication. Such a metadata can be known by exchanging messages with the coordinator (see Figure 6). `rmap` may fail due to failed authentication or memory space conflicts, i.e., some addresses between `vm_start` and `vm_end` has been mapped by the consumer container. If `rmap` succeeds, the consumer can access the pointers in the remotely mapped range just like its local memory.

`set_segment` allows setting the container segment (e.g., stack or heap) of the registered memory to a fixed position. This gives the coordinator the ability to assign disjoint address spaces to functions to prevent conflicts. Note that though we can leverage the link script to set the address of segments like `.txt` or `.data`, setting the ranges of heap and stack still requires kernel involvement.

Finally, we don't reclaim the registered memory unless some authenticated process (e.g., serverless framework) explicitly calls `deregister_mem`. The invocation process must be authenticated because this system call will deregister memory of other processes.

Coherency. Since a function's memory can be shared by the others via RMMAP, we must provide a coherency model, i.e., what will the consumer see if the produce modifies the registered memory. Providing strong coherency is slow due to extensive synchronization over the network [39, 47, 52]. For efficiency, RMMAP provides a simple copy-on-write (CoW) model: after registering the memory, we will mark the memory pages as CoW to isolate the modifications of the producer from the consumers. This model is sufficient for serverless computing because the consumer only accesses the state of the producer in a read-only fashion.

Detailed execution flow (Figure 8). After the producer calls the `register_mem`, its hosting kernel will mark the corresponding page table entries as CoW (❶). Besides, we also store the ID and key (`auth info`) in kernel for future authentication. When consumer invokes `rmap`, its kernel first issues an RPC to the producer machine (`mac_addr`) for authentication (❷). The `auth info` is piggybacked in the RPC so that remote kernel can validate the request. If the authentication succeeds, the remote kernel sends an acknowledgement back, and we will create a new virtual memory area (VMA) for the consumer container (❸) with a special (logical) device. This device hooks the page fault handler for this VMA to handle remote page read (❹). Specifically, if the consumer touches a page in the mapped memory, the kernel will call in the hooked handler and the device will send a network request to read remote physical page.

RDMA-based remote paging. The key requirement of RMMAP is high performance. For the producer, the overhead is small because it only involves marking page table entries (1–5 ms). Therefore, the dominating performance factor is the networked request to read the memory pages (❹ in Figure 8), which includes the following two aspects: First, we should efficiently read the memory pages through the network. Second, we should quickly establish a network connection between two machines.

RMMAP chooses a solution based on kernel-space RDMA. RDMA is a high speed (up to 400 Gbps [60]) and low latency (lower than 2 μs) networking feature supporting reading the memory of remote machine in a CPU-bypassing way. Specifically, given the physical address of the remote machine, the kernel can directly access this address with it [83, 91]. Therefore, using RDMA to read remote pages when handling page faults is extremely efficient. Kernel-space RDMA further supports establishing RDMA connection within 10 μs, which is orders of magnitude faster than user-space RDMA (10 ms) [90]. As a result, the cost of network connection is negligible with it.

To read a remote page with the best performance of RDMA (i.e., one-sided RDMA), the consumer kernel needs to know its physical address. Hence, we further retrieve the page table corresponding to the registered memory during the authentication RPC (❷ in Figure 8). With the help of RDMA, reading a 4 KB page only takes 3.7 μs in RMMAP, which is comparable with the time of handling a page fault (1.7 μs).

Note that unlike user-space RDMA, kernel-space RDMA does not need memory registration [83]. To validate the access permission of remote memory accesses, we follow MITOSIS [91] and adopt a connection-based permission control mechanism to isolate access from different functions.

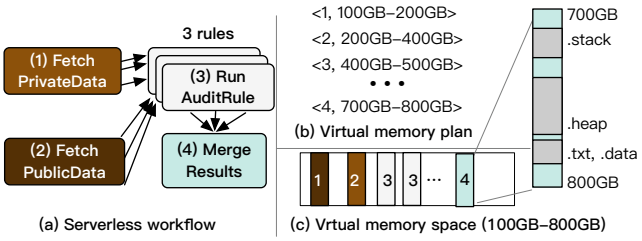

Figure 9. ***An illustration of generating virtual memory (VM) plan (b) given a serverless workflow (a). The plan will partition a virtual memory space for different functions (c).***

Management of the producer's memory lifecycle. In a distributed setting, the OS on the producer's host machine alone cannot solely manage the lifecycle of the registered memory, i.e., deciding when the CoWed pages can be reclaimed or not. This is because the memory can be accessed by another function on a different machine using RDMA, which bypasses the CPU (and thus, the OS). Meanwhile, after the producer container exits, the registered memory can still be accessed by the others. Therefore, we always keep the registered memory alive by recording a shadow copy of the registered pages in the kernel upon calling `register_memory`. On Linux, the shadow copy can be efficiently implemented by increasing the reference counter in `page_t`. These shadow copies are stored in the kernel and can be explicitly freed later by the `deregister_mem`.

4.2 Virtual memory plan and memory reclamation

Static virtual memory address space planning. The platform generates a plan for a workflow after it is uploaded. The plan contains a list of $\langle ID, Range \rangle$ pairs (Figure 9): the ID represents the function type and the *range* specifies that the virtual memory space of the container executing the function.

Since the (maximum) memory requirement of each function must be configured or determined by the platform [9, 11], it is straightforward to generate the plan. First we traverse the DAG of the workflow to find all type of functions that need state transfer between them. Note that the same type of function can be concurrently invoked (the `RunAuditRule` in Figure 9): they must be assigned different address space if they have the same downstream function (the `MergeResults`). A problem is that the detailed concurrency can be determined at runtime. To this end, we take a conservative approach that count the maximum concurrency for each function type. After determining the number of space to divide, we choose a maximum memory budget (say 100 GB) for each function type, and equally assign partition an entire address space to them. Such a partition can support up to 2,814 different types of functions on 64-bit servers[5], which is sufficient for all known serverless workflows. After the plan has been generated, we will store it together with the workflow to facilitate future executions.

Realizing the plan. Each machine must ensure the virtual address space of the container running the function conforms to the plan. We achieve so by compiling the binaries of each function type statically with the augmented link script, i.e., set the base address of the elf file to the start address of the range. Though linking can place the code or data segments to appropriate address, the OS may dynamically assign the heap and stack. Therefore, we also leverage the `set_segment` interface described in §4.1 such that the OS can adjust the heap and stack to the space within the range.

Static vs. Dynamic. Choosing a static plan generation strategy is our explicit design choice. Dynamic plan generation—generate the plan after the coordinator have received the workflow requests—cannot work with the caching techniques [5, 13, 22, 35, 42, 43, 63, 74] widely adopted by serverless platforms. With caching, the coordinator will reuse a container that previously runs the same type of function to hide the startup cost. If the reused container has an overlapped address space with the producer function, it cannot leverage remote memory map to bypass the (de)serialization. With static plan generation, we can ensure the reused container will have a distinct address space. Note that static planning is always possible on current platforms since they require a static configuration of function's maximum memory usage and concurrency [26, 30, 73].

Registered memory reclamation. Since we onload the memory management of the producer to the serverless framework, the coordinator needs to track the execution status of workflow functions to reclaim their registered memory in time. Specifically, if a function calls `register_mem`, it will send the key and metadata of the memory to the coordinator so that it can calls `deregister_mem` when needed. If the coordinator finds the registered memory is no longer needed, e.g., all its dependent function reports completion, we send an RPC to the pod holding the memory to reclaim it via `deregister_mem`.

One issue of a coordinator-centric resource reclamation is dealing with the coordinator failure. Persisting the recorded heap on the disk may significantly slowdown the execution of the coordinator. Observing the fact that serverless function has a maximum lifetime [7], we adopt a simple solution to avoid the overhead of the tolerating coordinator failure: we let each pod periodically scan registered memory that stayed longer than a threshold (the maximum lifetime plus a grace period). If they find one, they will directly reclaim it. By doing so, we no longer need to persist or replicate the coordinator's states.

Discussion on extra memory usage. To use RMMAP, we keep the producer function's memory alive for a slightly longer time, which results in additional memory usage. Fortunately, the caching technique widely used by serverless platforms can hide the extra memory usage [5, 13, 22, 35, 42, 43, 63, 74]. Specifically, after a function finishes execution, the platform will cache its host container in memory for a period to accelerate future function invocations. Therefore, even without RMMAP, the producer's memory will be retained in memory. Since the cache period is typically much longer than the ephemeral function execution time, we found no additional memory usage for RMMAP empirically (§5.6).

4.3 Supporting high-level language

This section describes how to co-design language runtime with RMMAP. We assume the producer and consumer functions share the same version of language runtime.

Remote object management. A key aspect when considering remote mapped memory is how to manage the remote object—the

[5]For example, x86-64 servers support an 2^{48} B user virtual address space.

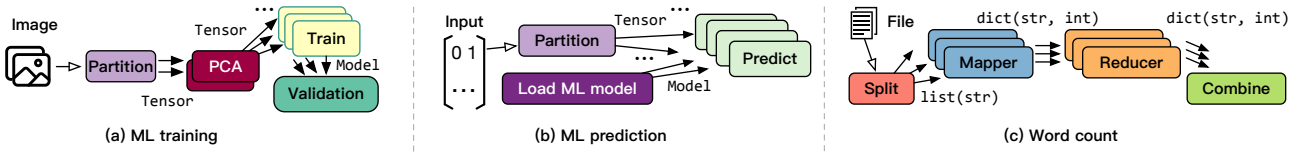

Figure 10. *The DAGs of our evaluated workflows. The DAG of FINRA is shown in Figure 1.*

object stored on the remote heap. For example, if the remote object is no longer used, we must release its local memory to avoid memory leakage. Existing runtime adopts garbage collection (GC) for this purpose. However, naively executing GC on the remote heap is inefficient since the GC states, e.g., tracing set is also on the remote heap. To conduct the GC, we must additionally fetch these states with RDMA, causing additional overhead.

Our observation is that for the consumer, we only need to care about the *lifecycle of the transferred state*, not the others. Therefore, we can manage the remote heap in a coarse-grained way: we can release all memory related to the remote heap if the root object of the transferred state is no longer used. The GC for the remote heap is executed as follows: after mapping the remote memory and acquire the state pointer, we will create a special object on the local heap pointing to the root object of the state. If this object is destroyed later, we will unmap the remote heap from the consumer. This scheme enables zero-cost GC on the remote heap. Note that if the local GC traces a object on the remote heap, we will simply skip it.

The above scheme is unable to handle a corner case where a remote sub-object is assigned to a local object. If the root of the sub-object is deleted, the local reference will then point to an invalid heap. Fortunately, such assignments are rare in serverless workflows. Therefore, we adopt a simple solution: when assigning a remote object locally, we will make a copy of it onto the local heap.

Type safety. To ensure the type safety of the accessed remotely mapped object, we share the type metadata (e.g., Java klass) among different function instances. For statically-typed language (e.g., Java), we leverage class data sharing (CDS) technique [6] that maps type metadata to the same address for all functions, similar to a priori work [93]. For dynamically-typed languages (e.g., Python), since their type metadata is stored on the heap and is accessed via shared memory, we can directly use RMMAP's on-demand paging to access it.

4.4 Prefetching and cascading state transfer

Semantic-aware prefetching. RMMAP trades network operations for (de)serialization: for each remote page access, the function will trap into the kernel and issue one RDMA to read the page. The overhead of RDMA and page faults will accumulate if an object's sub-objects span multiple memory pages, even RDMA is extremely fast. To hide these cost, we leverage prefetch to read all the sub-object pages in advance. The challenge is determining what to prefetch: reading the entire memory space to the consumer not only waste network bandwidth but also waste the memory.

Our observation is that for functions written in a high-level language (common in serverless), we can leverage the language runtime to precisely calculated the pages that store a given object. This can be done by traversing objects pointed by the root object of the state at the producer. With the traversed objects, we can derive the virtual pages they belong and send their addresses to the consumers. Based on these addresses, consumer can leverage doorbell batching [45]—an RDMA technique to read multiple memory pages in one roundtrip—to prefetch the pages efficiently.

Note that in most cases, we don't require developers writing extra codes for semantic-aware prefetching. For example, all Python built-in objects have provided `__iter__` and `__next__` functions to facilitate the object traversal. Nevertheless, third-party objects may miss such an implementation (e.g., Python numpy). In such a case, we fallback to non-prefetch mode or ask the developers to provide the implementation. The additional development cost is typically small. For example, numpy arrays have already implemented an internal iterator so we only need to wrap it, which has only 12 LoC modifications.

Finally, an interesting phenomenon we found is that prefetching is not always better (§5.2), because traversing the objects incurs additional computation costs at the producer-side, i.e., traversing the objects. Note that prefetching will add no computation overhead at the consumer-side. To balance the overhead for calculating objects to prefetch, we can set a threshold to limit the number of objects traversed. The detailed policy to set the threshold is left as our future work.

Handling cascading state transfer. The state transfer cascades if the transferred state is forwarded to another function. For example, suppose we have a workflow $A \rightarrow B \rightarrow C$, and the state from $B \rightarrow C$ may depend on objects stored on A. One naive solution would be map A's virtual memory to C's address space, but it would make memory management more complex.

We adopt a simple copy scheme similar to how we handle remote GC object described in the previous section. Since it is common that the transferred object is created from scratch, we will copy the remote object to the local heap if it is assigned to a local object. Note that if the same object is passed between the function chain (e.g., B passes A's input to C), we will also copy A's input to a local object of B in order to serve as C's input. In evaluations, we found no copy is needed in our workloads (§5). If the copy overhead is significant, we can adopt a multi-hops remote memory map design, e.g., by assigning unique IDs to remote addresses, similar to how MITOSIS [91] handles multi-hops remote fork. We leave the detailed design as our future work.

5 Evaluation

Our evaluations aim to answer the following questions:

- Can RMMAP reduce the serialization and deserialization costs for transferring various states? (§5.2)
- How effective is RMMAP to serverless workflows? (§5.3)
- How workflow configurations affect RMMAP? (§5.4)
- What are the performance trade-off of RMMAP? (§5.5)

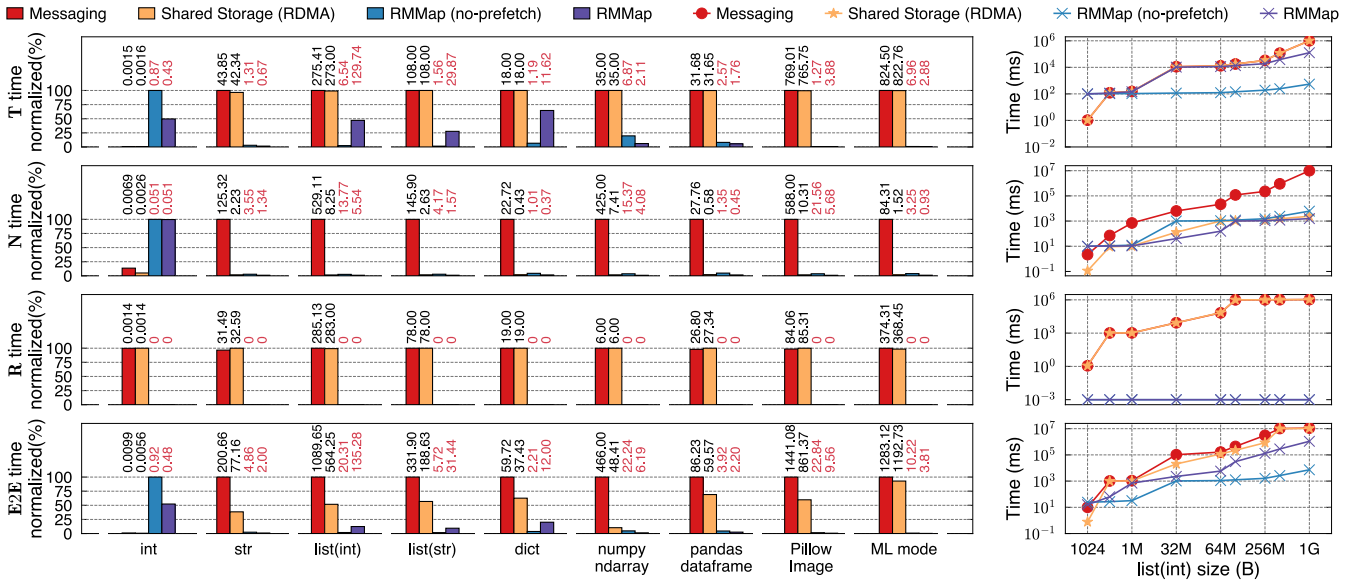

Figure 11. ***Latency breakdown for (a) different data types and (b) different payload sizes under a Python dataframe type. T: The time to transform an object to be ready to send. N: The network time to send the transformed the object. R: The time to reconstruct the object. E2E: summed up time of T, N and R. Note that the units of the labeled number in (a) are millisecond and the y-axis in (b) is log-scale.***

5.1 Evaluation setup

We implement the core RMMAP primitive as a loadable Linux module (on 4.15.0-46-generic) in 3,600 LoC rust code, excluding tests, benchmark code and RDMA support. We use KRCore [90] to support efficient kernel-space RDMA. We also extend Python 3.7 [1] and JDK 11.0.18 [14] runtime to support RMMAP, with about 2,300 and 1,800 LoC, respectively. We execute serverless workflows on Knative with cloudevent [10] to support workflow coordination, with about 1,000 LoC to support RMMAP-aware function execution.

Evaluated worfkflow applications. Figure 10 shows the DAGs of our evaluated workflows. **FINRA** [2] is a real word financial workflow that validates 3.5 MB trades data (stored via Python dataframe objects) based on rules. We configure it to run 200 `RunAuditRules` as reported by the AWS [12]. **ML training** [27, 57] is adopted from ORION [57]. It is a machine learning training workflow aiming to train a random forest on images. The training proceeds proceeds in three phases: image set partition, PCA for feature extraction, random forest training and validation. We follow the setup of ORION and run with 2 PCA function and 8 parallel training functions. The overall workflow is trained on 10 K images (total 42 MB) (from MNIST [51]) and will combine 64 decision trees as a random forest. The model is trained via the LightGBM [16] library. The **ML prediction** is a model serving workflow that utilize the above trained model to do the prediction. It will first partition the input images (total 30 MB) to 16 partitions and then run 16 parallel predictors for the prediction. The prediction results are then combined for all images. Finally, **WordCount** is a serverless MapReduce workflow adopted from FunctionBench [48], which counts the frequency of words in a book. We configure 8 mapper and one reducer function. The input is a French version of Oliver Twist (13 MB).

Without explicit notation, we implement all the workflows in Python because first, Python is the dominant serverless language [31] and second, some workflow requires specific python library to run (e.g., numpy, pandas and LightGBM). We also evaluate the RMMAP on a Java workflow in §5.7.

Comparing targets. We compare RMMAP with: **Messaging**: Functions share states with cloudevents [10], the built-in messaging primitive in Knative. The state is serialized and deserialized with pickle [18], the de facto (de)serialization library in Python. **Shared storage**: Functions transfer states with Pocket [49], the state of the art storage for serverless. It also uses pickle to serialize/deserialize states to the underlying storage. **Shared storage (RDMA)**: since Pocket is not optimized for RDMA, we evaluate an optimized shared storage with DrTM-KV [92], a state-of-the-art distributed key-value store on RDMA. DrTM-KV is 64.6× faster than Pocket, which we believe it can serve as the optimal performance for transferring state with shared storage. Since shared storage (RDMA) is much faster than without RDMA, we will only discuss it and leave the results without RDMA as a reference. Note that we can't optimize messaging with RDMA because it is currently tightly coupled with the Knative runtime. Finally, RMMAP (no-prefetch) and RMMAP (prefetch) are the variants of RMMAP without and with our prefetch optimization, respectively.

Testbed setup. We evaluate all the approaches on a Knative cluster deployed on a K8S cluster with 10 physical machines. Each machine is equipped with two 12 core Intel Xeon E5-2650 v4 processors and two 100 Gbps ConnectX-4 MCX455A InfiniBand NICs. To prevent the interference from coldstart, we pre-warm all the functions without explicit notation.

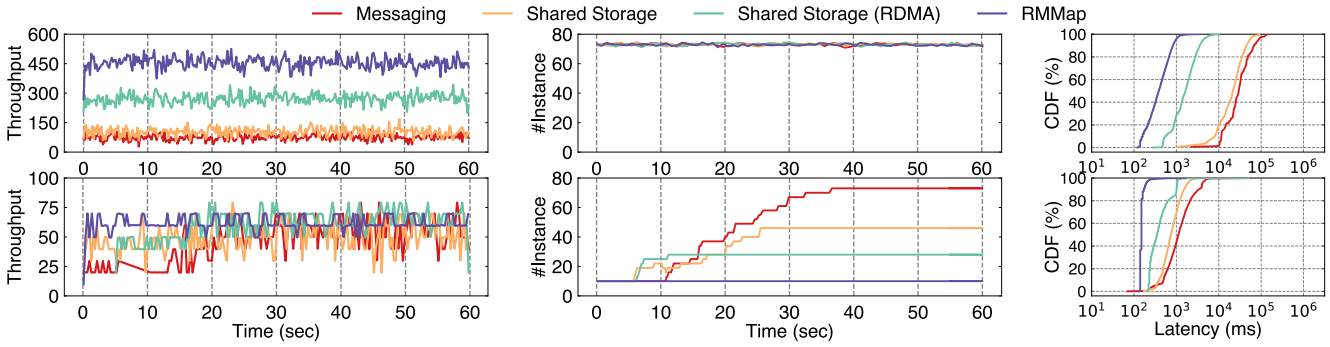

Figure 12. *(a) Throughput, (b) resource usage and (c) latency CDF of ML prediction workflow. Upper row: results with all resources utilized and bottom row: results with a fixed 70 request rate.*

5.2 Microbenchmark performance

First, we conduct a microbenchmark that compares the latency of transferring different python objects. Figure 11 (a) shows the latency breakdown of the end to end (**E2E**) transfer time, which we breakdown it into **t**ransform, **n**etwork and **r**econstruct stages. For variants of RMMAP, **T** includes the time to mark its memory as copy-on-write. For others, **T** is the serialization time. The **N** in RMMAP means the time to fetch the remote mapping information as well as reading pages with RDMA. For messaging, it is the time to transfer the message from the producer to the consumer in Knative. For storage, it is the time to put and read the serialized data from the storage. Finally, the **R** in RMMAP is the time to do the remote mapping, while it is the deserialization time in other approaches.

Performance with various data types. The (de)serialization time is sensitive to data types, e.g., simple data like integer has a trivial or no (de)serialization cost. We first compare different approaches to transfer common python data types in Figure 11 (a). The representative types (also used in the evaluated workflows) are: `str` is a 13 MB string, which is the input used in WordCount. `list(str)` is generated by delimiting the above 13 MB string with `\n`. `dict` is a nested map with a depth of six (total 380 B). `numpy ndarray` is a data matrix with shape (7000, 785). We reshape the `numpy ndarray` into one-dimensional to generate `list(int)`. `pandas dataframe` is a dataframe that encapsulates a (524,28, 8) shape CSV file. `Pillow Image` is obtained by utilizing PIL library with images size of 5.3 MB. Finally, `ML model` is a pre-trained LightGBM tree (8.6 MB) used in ML inference.

At the transform stage, RMMAP (with or without prefetch) is 19.8–99.8% faster than messaging and shared storage (with or without RDMA) except for `int`, since serializing it has negligible overhead. The time of RMMAP (no prefetch) is dominated by mark the process memory as copy-on-write. Some data type needs a longer transform time since their used virtual memory is larger due to a large dependent library. When adding the prefetch (RMMAP), we can reduce the copy-on-write overheads since we can precisely trace which pages are needed. However, the cost of its transformation is higher due to additional object traversals.

At the network stage, RMMAP (no prefetch) is 86.5–96.4% faster than messaging, except for `int`, since piggybacking a small integer in the cloudevent message is trivial. Nevertheless, for large data, messaging is slow because the data must passes many system components in Knative, which has non-trivial software cost. Compared to shared storage (RDMA), RMMAP is slower without prefetch and is faster with prefetch. RMMAP (no prefetch) is slower due to the extra page faults and extra network roundtrips to read the data.

At the reconstruct stage, variants of RMMAP has nearly no cost while others takes a non-trivial time for deserialization the data. For example, de-serializing a 3.2 MB dataframe object takes 12 ms, which is significant compared to the execution time of its used function (0.3 ms in the FINRA workflow).

For the end to end time, RMMAP (no prefetch) is 54.1–99.2% faster than other approaches except for `int`. Add prefetching further improves RMMAP by 43.8–72.2% except for `list(int)`, `list(str)`, `dict`. For these types, we need to traverse multiple objects to find which page to prefetch, e.g., traversing a 50 MB `list(int)` will touch 5,000,000 objects. Compared to the others, the performance benefit of RMMAP mainly comes from the reduced serialization cost at the transform stage, as well as no deserialization at the reconstruct stage. On the other hand, we should mention that RMMAP is not beneficial for transferring simple objects like `int`. This is because it is trivial to serialize or deserialize them. Thus, the copy-on-write and RPC overhead in RMMAP will dominate the state transfer. For such objects, workflows can directly use messaging to transfer them.

Performance with different data paylaod sizes. Another important factor for state transfer is the data payload size, especially for collection types (e.g., lists or dataframes). We present the performance of different approaches by varying the data payload sizes in Figure 11 (b). We choose `list(int)` as our evaluating type and vary the entry number to change the payload. The results for other types are similar.

The takeaways from Figure 11 (b) are twofolds. First, RMMAP has a smaller per-sub-object overhead than others for payloads larger than 1 KB: it is 66.4–82.3% faster shan Storage (RDMA) in the end-to-end time thanks to the eliminated (de)serialization. Nevertheless, RMMAP has a relative large startup cost—issuing an additional RPC to fetch the page table (10 μs) as well as trapping in the kernel to mark the pages as copy-on-write (1 μs). Thus, it is 9.2 × slower on shared storage (RDMA) with less than 1 KB payloads.

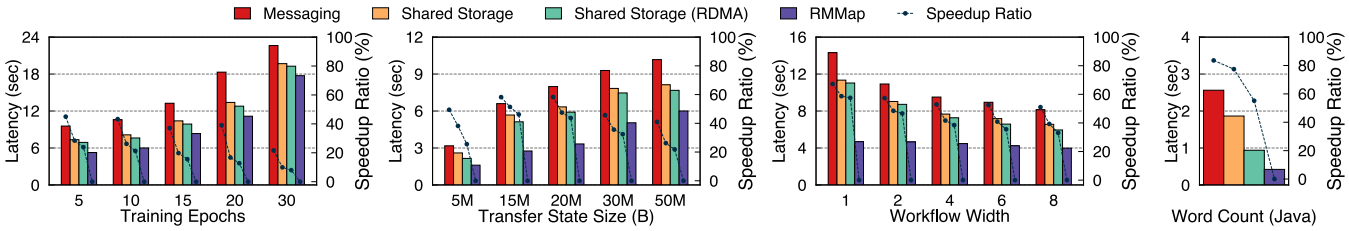

Figure 13. ***Sensitive analysis about (a) transferred data ratio, (b) workflow width and (c) training epochs. (d) The Java workflow performance.***

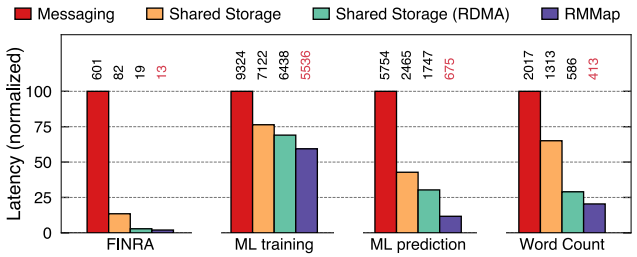

Figure 14. ***Execution time of different serverless workflows.***

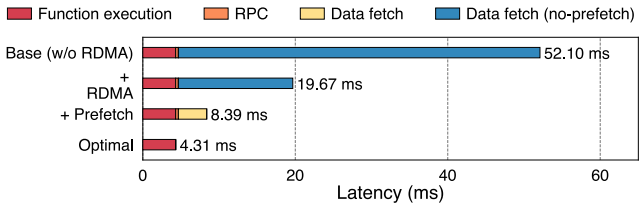

Figure 15. ***Factor analysis of the transferring state between the PCA and partition functions in the ML training workflow.***

5.3 End-to-end application performance

End-to-end latency. Figure 14 compares the average latency of different serverless workflows. To rule out interference from resource contentions, we first run one workflow at a time and measure its execution time. Overall, RMMAP is faster than messaging and variants of shared storage, reducing the execution time of different workflows by 14–97.8%. For brevity, we focus on discussing how RMMAP compares with shared storage (RDMA), as it is the fastest competitor. The benefit of RMMAP mainly comes from the eliminated (de)serialization, which takes 24.6%, 16.2%, 68%, 28.7% of the end-to-end time for FINRA, ML training, ML prediction, WordCount, respectively. Meanwhile, the additional cost (e.g., RPC) is negligible compared to the function execution time.

Throughput, resource usage and latency CDF. The throughput of workflows depends on two factors: the number of machine instances (pods) used and the request rate of the clients. The first row in Figure 12 presents the throughput of workflow with all machines saturated. The second row shows the throughput with a fixed 70 request rate. To saturate all the machines, we increasing the number of clients until the measured throughput won't increase. To fix the request rate, we deploy one client with an open-loop harness for the evaluation. For the throughput timelines, we spawn the configured clients at time 0 to emulate the elasticity of serverless requests. Finally, we only present the ML prediction results for brevity as others are similar.

When the machines are saturated (see the upper row (a)), RMMAP has a 1.2–1.6× higher peak throughput than the other counterparts due to the lower per-workflow execution time. Moreover, when different approaches are under the same request rate (see the lower row), RMMAP has a much better resource utilization. If the rate is smaller than the minimum peak throughput of different approaches, all of them reach the same throughput. Nevertheless, RMMAP only utilizes 64.3–86.3% of the available pods since it is much faster. Note that other approaches gradually increase resource utilization (see (c) of the lower row) because Knative will scale more pods to handle the incoming requests. Besides, the medium, 90^{th} and 99^{th} latencies of RMMAP are 66.1–99.2%, 69.3–99.4%, and 72.6–99.5% lower comparing to the others.

5.4 Sensitive analysis

The speedup of RMMAP can be affected by several factors, e.g., function execution time, transferred data payload and the concurrency of executed functions (workflow width). This section evaluates the sensitiveness of these configurations using the ML training since we can adjust its execution time by varying the number of training epochs.

Figure 13 (a) first shows the ML training time with various training epochs. When increasing the epochs from 5 to 30, the performance improvement of RMMAP (compared with Storage (RDMA)) decreases from 23.9% to 8%. A longer execution time will amortize the cost of (de)serialization so the benefits of RMMAP diminishes. Figure 13 (b) further evaluates the workflow's sensitiveness with the transferred tensor size. When increasing the payload, the improvement of RMMAP does not always increase or decrease: a larger payload is more costly for traditional approach for (de)serialization; yet, it also enlarge the workflow execution time (more data to train). We draw a similar observation in Figure 13 (c).

5.5 Factor analysis

RMMAP can slowdown the function execution due to RPC latency to map the remote pages as well as RDMA to read these pages. This section conducts a factor analysis of the above overheads by factoring out the execution of train function in ML training. The results are shown in Figure 15.

Compared with the optimal case where the function completely reads a local state, the end-to-end execution time of RMMAP is 1.4× and 1.7× longer with and without prefetch, respectively. First, the overheads are dominated by the RDMA to read the data pages, as remote memory read is still orders of magnitude slower than local read even with fast networking [32]. Second, prefetching significantly reduces the data read time due to (1) reduced page fault and (2) sending RDMA requests in batches is more CPU friendly [45].

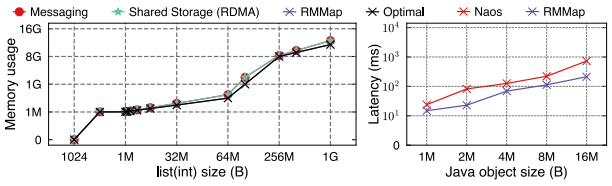

Figure 16. ***(a) Analysis of memory consumptions and (b) comparison with Naos [80], a library for optimizing (de)serialization in Java.***

Third, RPC to pull the remote memory information to setup the page table has negligible overhead. Finally, not using RDMA—implement remote page fetch with RPC slowdown the execution of RMMAP by 62.2%, showing the necessities of co-designing with RDMA. Note that we have adopted an optimized RPC on the same RDMA network fabric (Fasst-RPC [46]) to read pages.

5.6 Memory consumption

As discussed in §4.2, RMMAP may cause additional memory consumption. However, our empirical evaluation, as shown in Figure 16, indicates that RMMAP consumes negligible extra memory. In this experiment, we measured the peak memory usage during a workflow execution. The workflow involved transferring a Python `list(int)` with varying payloads, using one producer and one consumer. The extra memory used by RMMAP is at most 4% of the optimal, where the producer does not transfer the state. Since the platform caches the producer container after it finishes, it helps to amortize the extra memory left by the RMMAP after the producer finishes. Interestingly, RMMAP even requires up to 20% less memory compared to messaging and shared storage, as they need additional memory to store the message buffers.

5.7 Performance on Java

Application performance. We evaluate how RMMAP performs for a Java workflow that implements WordCount (§5.1) in Figure 13 (d). Other workflows have a similar result. As expected, the results of RMMAP is similar to that of Python: it is 77.4%, 55.2%, 39.0% faster than messaging, storage with and without RDMA, respectively. RMMAP is a general design that is agnostic to the language.

Comparison to Naos [80]. Finally, we compare RMMAP with Naos, a Java library optimized for (de)serialization. Since Naos does not support serverless, we use the microbenchmark described in his paper to compare the performance of transferring a Java map that stores (`Integer`, `char[]`) pairs. The integers are 4 B and the `char[]`s are 5 B, whose setup is the same as Nao's original paper. As we can see in Figure 16 (b), RMMAP outperforms Naos by 42–64% because Naos still needs to traverse and modify pointers of Java objects, while we completely eliminate this step.

6 Discussion and Limitation

RMMAP does not compromise on small object performance. As we have shown in §5.2, the overhead of small and simple types (e.g., `int`) is negligible. Therefore, the overhead of executing system calls (as well as RDMA) outpaces the benefits of eliminating (de)serialization for such objects. Nevertheless, since RMMAP is compatible with existing techniques like messaging, we can fallback to messaging for them to hide the overhead of RMMAP on small objects. Determining which types to fallback is trivial since we can leverage the semantics given by the language runtime.

Support functions written in different languages. Eliminate the (de)serialization for workflow functions written in different languages is not supported by RMMAP. It is more challenging since the object layout of different languages is different. Though common serverless workflow is composed of functions written in the same language, RMMAP can fallback to (de)serialization if not so.

Comparison to specialized libraries for exchanging objects. Although libraries like Apache Arrow [4] can enhance portability between different languages and improve efficiency of (de)serialization, they still need to transform a regular runtime object to its internal layout (and vice versa). In contrast, RMMAP completely eliminates the need for such transformation.

Map the heap vs. Map the whole address space. An initial version of RMMAP only maps the heap, since most of the state is allocated on the heap. However, we found complex objects in Python may span on multiple locations (e.g., `.txt` to store the callbacks) other than the heap. So we eventually fallback to map the whole address space. The drawback of our choice is that it needs to transfer extra page table and unnecessary marked copy-on-write pages. This overhead is significant when the producer imports a huge dynamic library. Techniques like on-demand page table access [100] can possibly help and we will explore them in the future.

Data compression in (de)serialization. Several libraries will aggressively compress the data to reduce network bandwidth usage [19]. Supporting compression needs extra computation, which we believe is not suitable for serverless since compressing and de-compressing the data on the function's critical path. We leave compression support as our future work.

Security of RDMA. RMMAP leverages RDMA for high performance state share. However, current RDMA implementations lack security enhancements, such as encryption [81]. Securing RDMA is an orthogonal to our design and is currently an active research topic [71, 81, 94]. We believe works on RDMA security will further complement RMMAP.

7 Related work

Optimizing serverless computing. Many works have been conducted on optimizing applications that run in a serverless paradigm. They can be categorized as accelerating function startups [22, 28, 34, 63, 72, 76, 85, 86, 91], supporting stateful serverless functions [42, 66, 97], accelerating workflow execution strategies [57, 58] and others [20, 33, 43, 44, 54, 55, 65, 70, 77, 78, 82, 87, 95, 99, 101]. Most of them focus on optimizing a single function, an orthogonal topic to us. Orion [57] proposes various strategies for faster workflow execution. Our focus is on the mechanism their strategies complement RMMAP. A recent work MITOSIS eliminates (de)serialization via remote fork. However, fork cannot support state transfer if the consumer function need to read states from multiple producers, which is now supported by RMMAP.

Optimizing (de)serialization. Reducing the (de)serialization overhead is an active research topic beyond serverless [41, 62, 69, 80, 84, 93, 96, 98]. For example, Hgum [98] offloads the (de)serialization to

the FPGA for acceleration. We completely eliminate the (de)serialization by co-designing the serverless framework with a new remote memory map primitive. RMMAP does not rely on specialized hardware, and we believe RMMAP can be made even faster by offloading several parts of the RMMAP to the programmable hardware. ZCOT [93] als avoids (de)serialization for Java-based big data applications with a global exchange space. RMMAP instead provides a remote memory map abstraction for any languages and tackle the problem of adapting the remote map to serverless workflows.

RDMA-based remote paging. Reading pages from remote hosts via RDMA is not a so new technique in modern OSes [21, 23, 36, 59, 75, 91]. MITOSIS [91] co-designs RDMA to realize a fast remote fork primitive. RMMAP further builds an efficient remote memory map primitive to eliminate (de)serialization for serverless workflows.

8 Conclusion and Future Work

We present RMMAP, an operating system primitive to enable serialization and deserialization-free state transfer in serverless workflows with a distributed shared memory-like abstraction. We show that such an abstraction can be made efficient and feasible by co-designing with the serverless platform, the language runtime, the OS and fast interconnect like RDMA. Specifically, unmodified real-world serverless workflows can enjoy up to 2.6× speedup and 86.3% better resource utilizations on Knative.

Although RMMAP currently focuses on serverless computing, we believe that several of its main techniques, including RDMA-based remote memory map for data transfer and semantic-aware prefetching, are generally applicable. On the other hand, when applying them in other scenarios, system designers must derive mechanisms for correct address planning, efficient remote memory reclamation, and effective garbage collection strategies. We will explore how RMMAP can be effectively applied in scenarios other than serverless in the future.

Acknowledgment

We sincerely thank our shepherd Marcos Aguilera and the anonymous reviewers, whose reviews and suggestions greatly strengthened our work. We also thank Zhe Li and Hongtao Lv for porting applications on the Java, Hongrui Xie for proof reading, and Yuhan Yang for discussing the assumptions made by serverless platforms. This work was supported in part by the National Key Research & Development Program of China (No. 2022YFB4500700), the National Natural Science Foundation of China (No. 62202291, 62272291, 61925206), the HighTech Support Program from STCSM (No. 22511106200) as well as research grants from Huawei Technologies and Shanghai Artificial Intelligence Laboratory. Corresponding author: Xingda Wei (wxdwfc@sjtu.edu.cn).

References

[1] Python3.7. https://github.com/python/cpython/tree/3.7, 2018.
[2] United States Financial Industry Regulatory Authority. https://aws.amazon.com/cn/solutions/case-studies/finra-data-validation/, 2022.
[3] Amazon s3. https://aws.amazon.com/s3, 2023.
[4] Apache arrow. https://arrow.apache.org, 2023.
[5] Apache openwhisk website. https://openwhisk.apache.org, 2023.
[6] Application class-data sharing. https://openjdk.org/jeps/310, 2023.
[7] AWS Lambda FAQs. https://aws.amazon.com/en/lambda/faqs/, 2023.
[8] AWS Step Functions. https://aws.amazon.com/step-functions/, 2023.
[9] Cloud functions pricing. https://cloud.google.com/functions/pricing, 2023.
[10] cloudevents. https://github.com/cloudevents/spec, 2023.
[11] Configuring lambda function options. https://docs.aws.amazon.com/lambda/latest/dg/configuration-function-common.html, 2023.
[12] FINRA adopts AWS to perform 500 billion validation checks daily. https://aws.amazon.com/solutions/case-studies/finra-data-validation/, 2023.
[13] Fn project website. https://fnproject.io, 2023.
[14] JDK 11.0.18. https://www.oracle.com/java/technologies/javase/11-0-18-relnotes.html, 2023.
[15] Knative. https://knative.dev, 2023.
[16] Lightgbm. https://github.com/microsoft/LightGBM, 2023.
[17] pandas. https://pandas.pydata.org, 2023.
[18] pickle. https://github.com/python/cpython/blob/main/Lib/pickle.py, 2023.
[19] Protocol Buffers. https://protobuf.dev, 2023.
[20] AGACHE, A., BROOKER, M., IORDACHE, A., LIGUORI, A., NEUGEBAUER, R., PIWONKA, P., AND POPA, D. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020* (2020), R. Bhagwan and G. Porter, Eds., USENIX Association, pp. 419–434.
[21] AGUILERA, M. K., AMIT, N., CALCIU, I., DEGUILLARD, X., GANDHI, J., NOVAKOVIC, S., RAMANATHAN, A., SUBRAHMANYAM, P., SURESH, L., TATI, K., VENKATASUBRAMANIAN, R., AND WEI, M. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018* (2018), H. S. Gunawi and B. Reed, Eds., USENIX Association, pp. 775–787.
[22] AKKUS, I. E., CHEN, R., RIMAC, I., STEIN, M., SATZKE, K., BECK, A., ADITYA, P., AND HILT, V. SAND: towards high-performance serverless computing. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018* (2018), H. S. Gunawi and B. Reed, Eds., USENIX Association, pp. 923–935.
[23] AMARO, E., BRANNER-AUGMON, C., LUO, Z., OUSTERHOUT, A., AGUILERA, M. K., PANDA, A., RATNASAMY, S., AND SHENKER, S. Can far memory improve job throughput? In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020* (2020), A. Bilas, K. Magoutis, E. P. Markatos, D. Kostic, and M. I. Seltzer, Eds., ACM, pp. 14:1–14:16.
[24] AWS. Aws lambda. https://aws.amazon.com/lambda, 2023.
[25] AWS. Aws step functions limits. https://docs.aws.amazon.com/step-functions/latest/dg/limits-overview.html, 2023.
[26] AZURE, M. Azure functions hosting options. https://learn.microsoft.com/en-us/azure/azure-functions/functions-scale, 2023.
[27] CARREIRA, J., FONSECA, P., TUMANOV, A., ZHANG, A., AND KATZ, R. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2019), SoCC '19, Association for Computing Machinery, p. 13–24.
[28] CARREIRA, J., KOHLI, S., BRUNO, R., AND FONSECA, P. From warm to hot starts: leveraging runtimes for the serverless era. In *HotOS '21: Workshop on Hot Topics in Operating Systems, Ann Arbor, Michigan, USA, June, 1-3, 2021* (2021), S. Angel, B. Kasikci, and E. Kohler, Eds., ACM, pp. 58–64.
[29] CLOUD, A. Alibaba serverless application engine. https://www.aliyun.com/product/aliware/sae, 2023.
[30] CLOUD, A. Manage functions. https://www.alibabacloud.com/help/en/fc/manage-functions?spm=a2c63.p38356.0.0.14dd3213mted5w, 2023.
[31] DATADOG. The state of serverless. https://www.datadoghq.com/state-of-serverless/, 2022.
[32] DRAGOJEVIC, A., NARAYANAN, D., CASTRO, M., AND HODSON, O. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014* (2014), R. Mahajan and I. Stoica, Eds., USENIX Association, pp. 401–414.
[33] DU, D., LIU, Q., JIANG, X., XIA, Y., ZANG, B., AND CHEN, H. Serverless computing on heterogeneous computers. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022* (2022), B. Falsafi, M. Ferdman, S. Lu, and T. F. Wenisch, Eds., ACM, pp. 797–813.
[34] DU, D., YU, T., XIA, Y., ZANG, B., YAN, G., QIN, C., WU, Q., AND CHEN, H. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020* (2020), J. R. Larus, L. Ceze, and K. Strauss, Eds., ACM, pp. 467–481.
[35] FOR AWS LAMBDA CONTAINER REUSE, B. P. https://medium.com/capital-one-tech/best-practices-for-aws-lambda-container-reuse-6ec45c74b67e, 2022.
[36] GU, J., LEE, Y., ZHANG, Y., CHOWDHURY, M., AND SHIN, K. G. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017* (2017), A. Akella and J. Howell, Eds., USENIX Association, pp. 649–667.

[37] Guo, C., Wu, H., Deng, Z., Soni, G., Ye, J., Padhye, J., and Lipshteyn, M. RDMA over commodity ethernet at scale. In *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016* (2016), M. P. Barcellos, J. Crowcroft, A. Vahdat, and S. Katti, Eds., ACM, pp. 202–215.

[38] Hellerstein, J. M., Faleiro, J. M., Gonzalez, J., Schleier-Smith, J., Sreekanti, V., Tumanov, A., and Wu, C. Serverless computing: One step forward, two steps back. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings* (2019), www.cidrdb.org.

[39] Hong, Y., Zheng, Y., Yang, F., Zang, B., Guan, H., and Chen, H. Scaling out numa-aware applications with rdma-based distributed shared memory. *J. Comput. Sci. Technol. 34*, 1 (2019), 94–112.

[40] Huawei. Huawei clound functions. https://developer.huawei.com/consumer/en/agconnect/cloud-function/, 2023.

[41] Jang, J., Jung, S., Jeong, S., Heo, J., Shin, H., Ham, T. J., and Lee, J. W. A specialized architecture for object serialization with applications to big data analytics. In *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020* (2020), IEEE, pp. 322–334.

[42] Jia, Z., and Witchel, E. Boki: Stateful serverless computing with shared logs. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021* (2021), R. van Renesse and N. Zeldovich, Eds., ACM, pp. 691–707.

[43] Jia, Z., and Witchel, E. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021* (2021), T. Sherwood, E. D. Berger, and C. Kozyrakis, Eds., ACM, pp. 152–166.

[44] Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N. J., Gonzalez, J. E., Popa, R. A., Stoica, I., and Patterson, D. A. Cloud programming simplified: A berkeley view on serverless computing. *CoRR abs/1902.03383* (2019).

[45] Kalia, A., Kaminsky, M., and Andersen, D. G. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016* (2016), A. Gulati and H. Weatherspoon, Eds., USENIX Association, pp. 437–450.

[46] Kalia, A., Kaminsky, M., and Andersen, D. G. Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016* (2016), K. Keeton and T. Roscoe, Eds., USENIX Association, pp. 185–201.

[47] Keleher, P. J., Cox, A. L., Dwarkadas, S., and Zwaenepoel, W. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *USENIX Winter 1994 Technical Conference, San Francisco, California, USA, January 17-21, 1994, Conference Proceedings* (1994), USENIX Association, pp. 115–132.

[48] Kim, J., and Lee, K. Practical cloud workloads for serverless faas. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019* (2019), ACM, p. 477.

[49] Klimovic, A., Wang, Y., Stuedi, P., Trivedi, A., Pfefferle, J., and Kozyrakis, C. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (Carlsbad, CA, Oct. 2018), USENIX Association, pp. 427–444.

[50] Kotni, S., Nayak, A., Ganapathy, V., and Basu, A. Faastlane: Accelerating Function-as-a-Service workflows. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)* (July 2021), USENIX Association, pp. 805–820.

[51] LeCun, Y., and Cortes, C. MNIST handwritten digit database.

[52] Li, K. IVY: A shared virtual memory system for parallel computing. In *Proceedings of the International Conference on Parallel Processing, ICPP '88, The Pennsylvania State University, University Park, PA, USA, August 1988. Volume 2: Software* (1988), Pennsylvania State University Press, pp. 94–101.

[53] Li, Z., Liu, Y., Guo, L., Chen, Q., Cheng, J., Zheng, W., and Guo, M. Faasflow: enable efficient workflow execution for function-as-a-service. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022* (2022), B. Falsafi, M. Ferdman, S. Lu, and T. F. Wenisch, Eds., ACM, pp. 782–796.

[54] Lykhenko, T., Soares, R., and Rodrigues, L. Faastcc: Efficient transactional causal consistency for serverless computing. In *Proceedings of the 22nd International Middleware Conference* (New York, NY, USA, 2021), Middleware '21, Association for Computing Machinery, p. 159–171.

[55] Lyu, X., Cherkasova, L., Aitken, R. C., Parmer, G., and Wood, T. Towards efficient processing of latency-sensitive serverless dags at the edge. In *EdgeSys@EuroSys 2022: Proceedings of the 5th International Workshop on Edge Systems, Analytics and Networking, Rennes, France, April 5 - 8, 2022* (2022), A. Y. Ding and V. Hilt, Eds., ACM, pp. 49–54.

[56] Mahgoub, A., Shankar, K., Mitra, S., Klimovic, A., Chaterji, S., and Bagchi, S. SONIC: application-aware data passing for chained serverless applications. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021* (2021), I. Calciu and G. Kuenning, Eds., USENIX Association, pp. 285–301.

[57] Mahgoub, A., Yi, E. B., Shankar, K., Elnikety, S., Chaterji, S., and Bagchi, S. ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)* (Carlsbad, CA, July 2022), USENIX Association, pp. 303–320.

[58] Mahgoub, A., Yi, E. B., Shankar, K., Minocha, E., Elnikety, S., Bagchi, S., and Chaterji, S. WISEFUSE: workload characterization and DAG transformation for serverless workflows. In *SIGMETRICS/PERFORMANCE '22: ACM SIGMETRICS/IFIP PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, Mumbai, India, June 6 - 10, 2022* (2022), D. Manjunath, J. Nair, N. Carlsson, E. Cohen, and P. Robert, Eds., ACM, pp. 57–58.

[59] Maruf, H. A., and Chowdhury, M. Effectively prefetching remote memory with leap. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020* (2020), A. Gavrilovska and E. Zadok, Eds., USENIX Association, pp. 843–857.

[60] Mellanox. ConnectX-7 product brief. https://www.nvidia.com/content/dam/en-zz/Solutions/networking/ethernet-adapters/connectx-7-datasheet-Final.pdf, 2023.

[61] Microsoft. Azure functions. https://azure.microsoft.com/en-us/services/functions/, 2023.

[62] Nguyen, K., Fang, L., Navasca, C., Xu, G., Demsky, B., and Lu, S. Skyway: Connecting managed heaps in distributed big data systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018* (2018), X. Shen, J. Tuck, R. Bianchini, and V. Sarkar, Eds., ACM, pp. 56–69.

[63] Oakes, E., Yang, L., Zhou, D., Houck, K., Harter, T., Arpaci-Dusseau, A., and Arpaci-Dusseau, R. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, July 2018), USENIX Association, pp. 57–70.

[64] Oracle. Java. https://www.java.com/, 2023.

[65] Pu, Q., Venkataraman, S., and Stoica, I. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019* (2019), J. R. Lorch and M. Yu, Eds., USENIX Association, pp. 193–206.

[66] Qi, S., Liu, X., and Jin, X. Halfmoon: Log-optimal fault-tolerant stateful serverless computing. In *Proceedings of the 29th Symposium on Operating Systems Principles* (New York, NY, USA, 2023), SOSP '23, Association for Computing Machinery, p. 314–330.

[67] Qi, S., Monis, L., Zeng, Z., Wang, I., and Ramakrishnan, K. K. SPRIGHT: extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing. In *SIGCOMM '22: ACM SIGCOMM 2022 Conference, Amsterdam, The Netherlands, August 22 - 26, 2022* (2022), F. Kuipers and A. Orda, Eds., ACM, pp. 780–794.

[68] Qingyuan, L., Dong, D., Yubin, X., Ping, Z., and Haibo, C. The gap between serverless research and real-world systems. In *Proceedings of the 14th Symposium on Cloud Computing* (New York, NY, USA, 2023), SoCC '23, Association for Computing Machinery.

[69] Raghavan, D., Ravi, S., Yuan, G., Thaker, P., Srivastava, S., Murray, M., Penna, P. H., Ousterhout, A., Levis, P., Zaharia, M., and Zhang, I. Cornflakes: Zero-copy serialization for microsecond-scale networking. In *Proceedings of the 29th Symposium on Operating Systems Principles* (New York, NY, USA, 2023), SOSP '23, Association for Computing Machinery, p. 200–215.

[70] Romero, F., Chaudhry, G. I., Goiri, I., Gopa, P., Batum, P., Yadwadkar, N. J., Fonseca, R., Kozyrakis, C., and Bianchini, R. Faa$t: A transparent auto-scaling cache for serverless applications. In *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021* (2021), C. Curino, G. Koutrika, and R. Netravali, Eds., ACM, pp. 122–137.

[71] Rothenberger, B., Taranov, K., Perrig, A., and Hoefler, T. Redmark: Bypassing RDMA security mechanisms. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021* (2021), M. Bailey and R. Greenstadt, Eds., USENIX Association, pp. 4277–4292.

[72] Saxena, D., Ji, T., Singhvi, A., Khalid, J., and Akella, A. Memory deduplication for serverless computing with medes. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022* (2022), Y. Bromberg, A. Kermarrec, and C. Kozyrakis, Eds., ACM, pp. 714–729.

[73] Services, A. W. Configuring function memory (console). https://docs.aws.amazon.com/lambda/latest/dg/configuration-function-common.html#configuration-memory-console, 2023.

[74] Shahrad, M., Fonseca, R., Goiri, I., Chaudhry, G., Batum, P., Cooke, J., Laureano, E., Tresness, C., Russinovich, M., and Bianchini, R.

Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020* (2020), A. Gavrilovska and E. Zadok, Eds., USENIX Association, pp. 205–218.

[75] SHAN, Y., HUANG, Y., CHEN, Y., AND ZHANG, Y. Legoos: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018* (2018), A. C. Arpaci-Dusseau and G. Voelker, Eds., USENIX Association, pp. 69–87.

[76] SHILLAKER, S., AND PIETZUCH, P. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (July 2020), USENIX Association, pp. 419–433.

[77] SINGHVI, A., BALASUBRAMANIAN, A., HOUCK, K., SHAIKH, M. D., VENKATARAMAN, S., AND AKELLA, A. Atoll: A scalable low-latency serverless platform. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2021), SoCC '21, Association for Computing Machinery, p. 138–152.

[78] SREEKANTI, V., WU, C., CHHATRAPATI, S., GONZALEZ, J. E., HELLERSTEIN, J. M., AND FALEIRO, J. M. A fault-tolerance shim for serverless computing. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020* (2020), A. Bilas, K. Magoutis, E. P. Markatos, D. Kostic, and M. I. Seltzer, Eds., ACM, pp. 15:1–15:15.

[79] SREEKANTI, V., WU, C., LIN, X. C., SCHLEIER-SMITH, J., GONZALEZ, J. E., HELLERSTEIN, J. M., AND TUMANOV, A. Cloudburst: Stateful functions-as-a-service. *Proc. VLDB Endow. 13*, 12 (jul 2020), 2438–2452.

[80] TARANOV, K., BRUNO, R., ALONSO, G., AND HOEFLER, T. Naos: Serialization-free RDMA networking in java. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)* (July 2021), USENIX Association, pp. 1–14.

[81] TARANOV, K., ROTHENBERGER, B., PERRIG, A., AND HOEFLER, T. srdma - efficient nic-based authentication and encryption for remote direct memory access. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020* (2020), A. Gavrilovska and E. Zadok, Eds., USENIX Association, pp. 691–704.

[82] THORPE, J., QIAO, Y., EYOLFSON, J., TENG, S., HU, G., JIA, Z., WEI, J., VORA, K., NETRAVALI, R., KIM, M., AND XU, G. H. Dorylus: Affordable, scalable, and accurate GNN training with distributed CPU servers and serverless threads. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021* (2021), A. D. Brown and J. R. Lorch, Eds., USENIX Association, pp. 495–514.

[83] TSAI, S.-Y., AND ZHANG, Y. Lite kernel rdma support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, Association for Computing Machinery, p. 306–324.

[84] TSENG, H., ZHAO, Q., ZHOU, Y., GAHAGAN, M., AND SWANSON, S. Morpheus: Creating application objects efficiently for heterogeneous computing. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016* (2016), IEEE Computer Society, pp. 53–65.

[85] USTIUGOV, D., PETROV, P., KOGIAS, M., BUGNION, E., AND GROT, B. Benchmarking, analysis, and optimization of serverless function snapshots. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021* (2021), T. Sherwood, E. D. Berger, and C. Kozyrakis, Eds., ACM, pp. 559–572.

[86] WANG, A., CHANG, S., TIAN, H., WANG, H., YANG, H., LI, H., DU, R., AND CHENG, Y. Faasnet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021* (2021), I. Calciu and G. Kuenning, Eds., USENIX Association, pp. 443–457.

[87] WANG, A., ZHANG, J., MA, X., ANWAR, A., RUPPRECHT, L., SKOURTIS, D., TARASOV, V., YAN, F., AND CHENG, Y. InfiniCache: Exploiting ephemeral serverless functions to build a Cost-Effective memory cache. In *18th USENIX Conference on File and Storage Technologies (FAST 20)* (Santa Clara, CA, Feb. 2020), USENIX Association, pp. 267–281.

[88] WANG, C., MA, H., LIU, S., LI, Y., RUAN, Z., NGUYEN, K., BOND, M. D., NETRAVALI, R., KIM, M., AND XU, G. H. Semeru: A memory-disaggregated managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020* (2020), USENIX Association, pp. 261–280.

[89] WEI, X., DONG, Z., CHEN, R., AND CHEN, H. Deconstructing rdma-enabled distributed transactions: Hybrid is better! In *13th USENIX Symposium on Operating Systems Design and Implementation* (2018), OSDI '18, pp. 233–251.

[90] WEI, X., LU, F., CHEN, R., AND CHEN, H. KRCORE: A microsecond-scale RDMA control plane for elastic computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)* (Carlsbad, CA, July 2022), USENIX Association, pp. 121–136.

[91] WEI, X., LU, F., WANG, T., GU, J., YANG, Y., CHEN, R., AND CHEN, H. No provisioned concurrency: Fast RDMA-codesigned remote fork for serverless computing. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)* (Boston, MA, July 2023), USENIX Association, pp. 497–517.

[92] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 87–104.

[93] WU, M., WANG, S., CHEN, H., AND ZANG, B. Zero-Change object transmission for distributed big data analytics. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)* (Carlsbad, CA, July 2022), USENIX Association, pp. 137–150.

[94] XING, J., HSU, K., QIU, Y., YANG, Z., LIU, H., AND CHEN, A. Bedrock: Programmable network support for secure RDMA systems. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022* (2022), K. R. B. Butler and K. Thomas, Eds., USENIX Association, pp. 2585–2600.

[95] YANG, Y., ZHAO, L., LI, Y., ZHANG, H., LI, J., ZHAO, M., CHEN, X., AND LI, K. Infless: a native serverless system for low-latency, high-throughput inference. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022* (2022), B. Falsafi, M. Ferdman, S. Lu, and T. F. Wenisch, Eds., ACM, pp. 768–781.

[96] ZARANDI, A. P., GUPTA, S., KASSIR, H., SUTHERLAND, M., TIAN, Z., DRUMOND, M. P., FALSAFI, B., AND KOCH, C. Optimus prime: Accelerating data transformation in servers. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020* (2020), J. R. Larus, L. Ceze, and K. Strauss, Eds., ACM, pp. 1203–1216.

[97] ZHANG, H., CARDOZA, A., CHEN, P. B., ANGEL, S., AND LIU, V. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020* (2020), USENIX Association, pp. 1187–1204.

[98] ZHANG, S., ANGEPAT, H., AND CHIOU, D. Hgum: Messaging framework for hardware accelerators. *CoRR abs/1801.06541* (2018).

[99] ZHANG, W., FANG, V., PANDA, A., AND SHENKER, S. Kappa: a programming framework for serverless computing. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020* (2020), R. Fonseca, C. Delimitrou, and B. C. Ooi, Eds., ACM, pp. 328–343.

[100] ZHAO, K., GONG, S., AND FONSECA, P. On-demand-fork: A microsecond fork for memory-intensive and latency-sensitive applications. In *Proceedings of the Sixteenth European Conference on Computer Systems* (New York, NY, USA, 2021), EuroSys '21, Association for Computing Machinery, p. 540–555.

[101] ZHAO, L., YANG, Y., LI, Y., ZHOU, X., AND LI, K. Understanding, predicting and scheduling serverless workloads under partial interference. In *SC '21: The International Conference for High Performance Computing, Networking, Storage and Analysis, St. Louis, Missouri, USA, November 14 - 19, 2021* (2021), B. R. de Supinski, M. W. Hall, and T. Gamblin, Eds., ACM, pp. 22:1–22:15.

A Artifact Appendix

A.1 Abstract

The artifact for the EuroSys 2024 paper—"Serialization/Deserialization-free State Transfer in Serverless Workflows"—provides the necessary resources for reproducing the main experiment results presented in the paper. It includes the source code and setup and reproduce instructions. The DOI of the artifact is at https://zenodo.org/doi/10.5281/zenodo.10078917.

A.2 Description

A.2.1 How to access All the source code and instructions can be accessed through the following git repository: https://github.com/ProjectMitosisOS/dmerge-eurosys24-ae.

A.2.2 Hardware dependencies The experiments require multi-core server machines equipped with RDMA NICs. InfiniBand is the preferred RDMA protocol.

A.2.3 Software dependencies The RMMAP artifact can only run on the Linux platform. Several software packages are required for compiling and running the experiments, as detailed in the *README.md*.

A.2.4 Benchmarks No special benchmarks are required. The artifact includes four different benchmark applications, as described in §5.1.

A.3 Setup

The artifact is self-contained and includes all necessary steps for hardware and software preparations, as documented in the *README.md*. The *docs/exp.md* file provides instructions for setting up the experiment environment, including the integration of the four workflow applications described in §5.1.

A.4 Experiment Workflow

The process for reproducing the results of all experiments is documented in the *docs/exp.md*. By following the provided instructions, experiment results in §5.2–§5.7 can be successfully reproduced.