TrEnv: Transparently Share Serverless Execution Environments Across Different Functions and Nodes

Jialiang Huang[♠♡] Mingxing Zhang[♠+] Teng Ma[♡+] Zheng Liu[♡] Sixing Lin[♠♣] Kang Chen[♠] Jinlei Jiang[♠] Xia Liao[♠] Yingdi Shan[♠] Ning Zhang[♡] Mengting Lu[♡] Tao Ma[♡] Haifeng Gong[◇] Yongwei Wu[♠+]
[♠]Tsinghua University [♡]Alibaba Group [♣]Wuhan University [◇]Intel

Abstract

Serverless computing is renowned for its computation elasticity, yet its full potential is often constrained by the requirement for functions to operate within local and dedicated background environments, resulting in limited memory elasticity. To address this limitation, this paper introduces TrEnv, a co-designed integration of the serverless platform with the operating system and CXL/RDMA-based remote memory pools in two key areas. Firstly, TrEnv introduces repurposable sandboxes, which can be shared across different functions and hence, substantially decrease the overhead associated with creating isolation sandboxes. Secondly, it augments the OS with "memory templates" that enable rapid restoration of function states stored on remote memory. These innovations allow TrEnv to facilitate rapid transitions between instances of different functions and enable memory sharing across multiple nodes. Our evaluations using a variety of representative and real-world workloads demonstrate that TrEnv can initiate a container within 10 milliseconds, achieving up to a 7× speedup in P99 end-to-end latency and reducing memory usage by 48% on average compared to state-of-the-art on-demand restoring systems.

CCS Concepts: • **Computer systems organization** → **Cloud computing**; • **Software and its engineering** → **Memory management**.

Keywords: Serverless, Cold Start, CXL, Remote Memory

ACM Reference Format:
Jialiang Huang[♠♡] Mingxing Zhang[♠+] Teng Ma[♡+] Zheng Liu[♡] Sixing Lin[♠♣] Kang Chen[♠] Jinlei Jiang[♠], Xia Liao[♠] Yingdi Shan[♠] Ning Zhang[♡] Mengting Lu[♡] Tao Ma[♡] Haifeng Gong[◇] Yongwei Wu[♠+]. 2024. **TrEnv: Transparently Share Serverless Execution Environments Across Different Functions and Nodes. In *ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24), November 4–6, 2024, Austin, TX, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3694715.3695967**

+: Corresponding author.

SOSP '24, November 4–6, 2024, Austin, TX, USA

ACM ISBN 979-8-4007-1251-7/24/11
https://doi.org/10.1145/3694715.3695967

1 Introduction

1.1 Motivation

Serverless computing, renowned for its fine-grained resource allocation and billing, enables applications to scale efficiently while optimizing resource utilization. Given its transformative potential, serverless computing is now available across all major cloud service providers [21, 24, 44] and is applied across a broad range of sectors [5, 25, 33, 47, 60].

However, despite the utopian vision of serverless computing, its real-world implementations face many practical limitations. Although serverless models suggest that functions could be transparently scheduled across data centers to maximize resource utilization, this idealized elasticity is often compromised by the functions' need for efficient access to local and dedicated background environments. This **contradiction between computational elasticity and environment localization** poses challenges to the full realization of serverless's potential.

Precisely, the execution of a serverless function typically unfolds in three phases. *(1)* The sandbox creation phase establishes an isolated sandbox (e.g., a container); *(2)* The bootstrapping phase initializes the function, which may involve steps like launching Python/Java virtual machines; and finally, *(3)* The execution phase executes instance-specific logic according to the inputs of this invocation. The first two phases are essential for setting up an execution environment for each function invocation. However, they do not directly contribute to processing user inputs, thus their overheads (in both CPU time and memory consumption) should be minimized.

Unfortunately, studies show that the time required to create such dedicated environments can significantly exceed the actual execution time of a function, leading to the notorious "cold start" problem in serverless computing [39, 66]. This issue has prompted the development of various caching techniques aimed at keeping functions warm and ready for immediate reuse [6, 20, 28, 59]. However, these mechanisms necessitate the reservation of local resources, particularly memory, introducing a trade-off between initialization speed and resource costs, thereby diminishing the system's overall elasticity. For instance, commercial solutions like AWS Lambda's

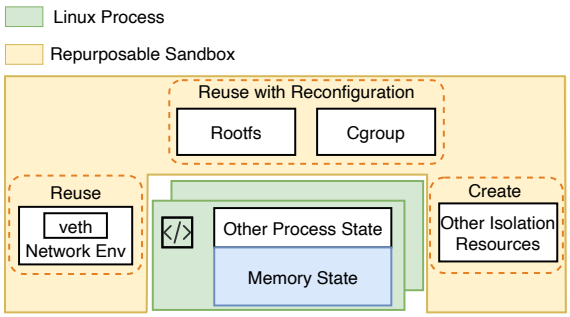


Figure 1. TrEnv's view of a container. It consists of two parts: a repurposable sandbox, which is comprised of isolation resources such as the root filesystem (rootfs); and a group of processes, the critical component of which is the memory state.

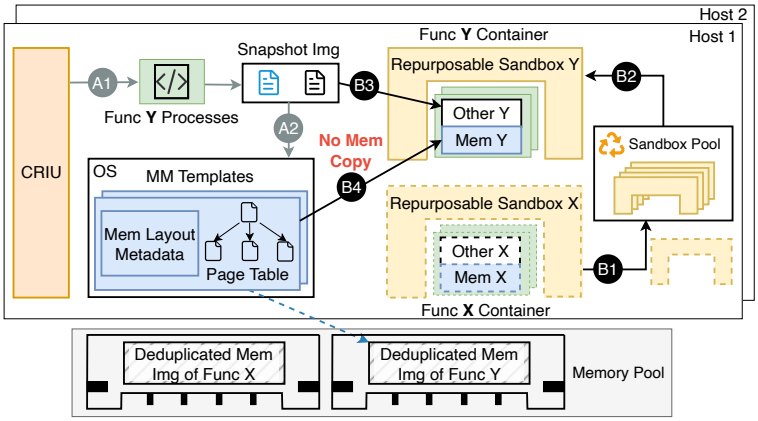


Figure 2. Overview of TrEnv. Steps A1-A2 refer to offline preprocessing, while steps B1-B4 refer to online repurposing (detailed in §3).

Table 1. The core components in current containers.

	Unit	Description	Overhead	TrEnv's Solution
Sandbox	Network	Isolated network environment, such as independent ports, including a network namespace and a virtual ethernet device.	80 ms ~10 s	Direct reuse as it does not leak any data produced during processing (§5.1.1).
	Rootfs	Root filesystem for containers, including a mount namespace and necessary filesystems such as sysfs under /sys.	10 ~800 ms	Reuse with reconfigurations, at a lower cost than creating (§4.2).
	Cgroup	Resource isolation, such as CPU and memory usage control.	30 ~400 ms	
	Other	Other isolated resources, such as time and pid namespaces.	<1 ms	Create with low overhead.
Process	Memory	Post-initialized state of functions, including loaded language runtime, imported libraries and user code.	>300 ms	Introduce new kernel features, bypass costly memory copy (§4.1).
	Other	Multi-thread context, open file descriptors, socket, etc.	3 ~15ms.	Handled by CRIU with strong generality.

Provisioned Concurrency [28], which provide caching, deviate from the ideal pay-as-you-go model of cloud computing by charging for these reserved resources.

The requirement for execution environments also gives rise to other issues, such as "memory stranding" [51] and "state duplication" [63]. Memory stranding occurs when a server's computational resources are fully engaged, but its memory remains underutilized. State duplication arises when concurrent serverless functions require but cannot efficiently share identical states due to existing isolation mechanisms. Recent studies have indicated that these challenges lead to significant inefficiencies, with up to 50% of memory resources being underutilized in the cloud [41, 46] and an 80% occurrence of state duplication [63]. These issues collectively pose significant obstacles to achieving genuine elasticity in serverless computing.

1.2 Our Contribution

In this paper, we present TrEnv, a serverless computing platform crafted to minimize CPU time and memory costs associated with serverless execution environments. The main idea is to share components of these environments across different functions and nodes as much as possible. As we can see from Figure 1, TrEnv partitions the environment of a function instance into a sandbox and a group of processes, with the memory state being their most critical component. Rather than discarding this execution environment upon the completion of a function instance, TrEnv cleanses and reallocates it to a repurposable sandbox pool. Subsequent functions can reuse these sandboxes by attaching function-specific memory states, which are offloaded to a shared CXL or RDMA-based memory pool. This process, termed "repurposing" in TrEnv, is depicted in Figure 2. Through innovative enhancements at the operating system (OS) and core libraries like Checkpoint/Restore In Userspace (CRIU) [8], this procedure typically takes less than 10 ms, and each invocation only requires additional memory for updated pages, with all read-only pages being shared transparently.

TrEnv's approach of container repurposing, while bearing similarities to existing caching solutions, distinguishes itself in two significant ways. Firstly, it enables transparent transitions of containers between different types of functions, eliminating the need for function-specific resource reservations and thereby maximizing resource elasticity. To achieve this, we have identified crucial kernel objects that contribute to the overhead of creating isolated sandboxes.

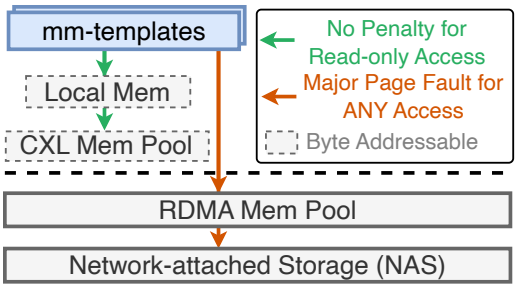


Figure 3. Multi-layer architecture of mm-template.

Some of these components are directly reusable across different functions, while others necessitate a tailored reconfiguration that upholds cost-effectiveness without compromising isolation properties relative to previous container-based systems. These components are listed in Table 1 and will be discussed in detail in §3.1 and §4.2.

Secondly, TrEnv offloads function-specific memory states to a shared memory pool. This strategy shares the states across multiple server nodes, effectively amortizing the overall expense. TrEnv also supports an effective deduplication process that eliminates redundant states across different functions, potentially reducing memory usage by up to 48% in our evaluated functions (Table 2 in §6.1).

However, operating system modifications are required to rapidly instantiate serverless functions from snapshots stored in remote memory pools. This process resembles performing a "fork" but includes capabilities beyond those offered by current operating systems. To support this, we have extended the kernel with an *mm-template API*, enabling the creation of custom memory templates for each serverless function. These templates consist of reserved page tables that map to remote, potentially overlapping, non-contiguous memory segments within the pool, representing deduplicated snapshots of serverless functions.

The flexibility of our design is one of its key strengths. As demonstrated in Figure 3, TrEnv integrates mm-templates with both CXL and RDMA memory pools, each offering distinct advantages. Notably, to leverage the byte-addressable capabilities of CXL, mm-templates enable direct reads for read-only pages, which introduces zero additional software-level overhead during execution. Given our analysis showing that 24% to 90% of memory accesses are to read-only pages in serverless functions, this technique substantially accelerates processing times by avoiding unnecessary data copying or page faults, thereby significantly reducing latency. Despite this, existing OS primarily utilizes CXL for memory expansion as per the capabilities of CXL 1.1, thus lacking support for memory sharing and deduplication required in multi-node environments introduced since CXL 2.0. To overcome this limitation, we have enhanced the OS to allow precise control over the mapping between virtual and physical addresses in CXL memory and supported both file-backed and anonymous memory mappings to maintain a process's complete state, including heap and stack areas.

Different from CXL, integrating RDMA-based remote memory follows a lazy paging strategy, where any access to remote memory triggers a major page fault that fetches a 4KB block online. Since all states in the memory pool are read-only, with write operations managed through copy-on-write mechanisms, a multi-layered architecture integrates seamlessly with our approach. This allows for the strategic placement of hot pages in the upper layers, such as CXL or even local memory, and cold pages in the lower layers, such as RDMA or even network-attached storage. The specific number of layers, as well as the cache eviction or page promotion strategies, are orthogonal to our core implementation.

Moreover, the ease of use is the most important attraction of serverless computing. To avoid the need for users to modify their code or recompile their applications, TrEnv decides to *transparently* incorporate repurposable sandboxes and mm-templates into the serverless platform. It implements a new "repurpose" command in CRIU, based on the existing "restore" command. This facilitates the seamless integration with many existing serverless platforms due to CRIU's widespread adoption in mainstream container runtimes like Docker[57] and Podman[13].

We have developed TrEnv atop faasd [40], a widely-used serverless platform. TrEnv can initiate many function instances in about 10 ms, and even complex applications containing more than 140 threads in just 18 ms. Importantly, these latencies are achieved while maintaining the same or stronger isolation than previous container-based solutions. Our evaluations of TrEnv with both Python and Node.js functions, conducted under both representative and industrial workloads, have shown significant improvements:

(1) TrEnv achieves up to a 7× and 18× speedup in P99 end-to-end (E2E) latency under concurrent loads compared to REAP [72] and FaaSnap [26], two state-of-the-art (SOTA) lazy restoration methods, respectively.
(2) TrEnv can reduce peak memory usage by 48% on average compared to SOTA.
(3) A detailed comparison between real-world CXL and RDMA memory pools reveals that CXL provides more stable performance, particularly at P99, and offers up to a 3.51× speedup on execution time over RDMA.

In summary, TrEnv significantly enhances serverless computing by reducing P99 E2E latency and minimizing memory overhead for each function through two key strategies: (1) It transparently shares repurposable sandboxes across different types of functions, thus avoiding the overhead associated with reserving resources for each individual function type. (2) It transparently shares the initialized memory states on remote memory pools via mm-template, substantially reducing restoration time and effectively amortizing memory consumption. Both functionalities are supported by low-level kernel and CRIU modifications, enabling their transparent

integration into the initialization of multi-threaded/process and even multi-language serverless applications. TrEnv is available at https://github.com/switch-container.

2 Background and Motivation

2.1 Dissaggregated Memory

CXL Compute Express Link (CXL) [3] is emerging as a promising technology in the realm of high-speed server interconnects, notable for its low latency [79], high bandwidth [71], and ability to enable shared memory access between servers. Its adoption by industry giants [52, 56, 79] also underscores its increasing significance and application in modern computing infrastructure.

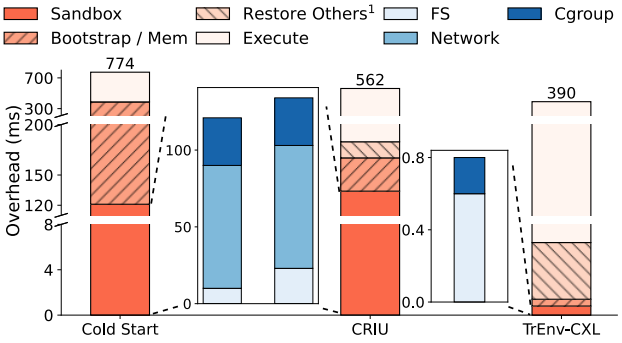
Our research primarily concentrates on CXL's capability of sharing memory across different machines, which has been supported by multi-headed devices (MHD) even with CXL 2.0. Real-world support for this capability has been demonstrated in previous research [79]. Collaborations between many companies [9, 11, 14, 16] have also led to a commercial solution that enabled up to 7.5 TB CXL-attached memory pool shared by up to 12 servers [15]. We only need read-only sharing capabilities, so a Type-3 device with multiple CXL 2.0 interfaces (MHD) is sufficient without the strong consistency guarantees introduced in CXL 3.0 [2].

CXL's fine-grained and efficient access capability means only the needed cache lines are retrieved, not entire pages. Thus, it provides a potentially faster alternative to RDMA.

RDMA Prior to CXL, RDMA was the standard protocol for building disaggregated memory pools, which can be grouped into two main categories: Firstly, solutions such as Infiniswap [45] and Fastswap [23] enable efficient swapping over RDMA. Secondly, frameworks and runtimes like AsymNVM [55] and FaRM [37] offer coarse-grained abstractions that resemble key-value stores or file interfaces. TrEnv tries to implement transparent and extensible in-kernel functionality to support various remote memory, including but not limited to RDMA.

2.2 Serverless Computing

The very idea of serverless is fragmenting applications into smaller, independent tasks that can be dispatched to whichever node has available CPU resources, thereby enhancing resource utilization. However, the pursuit of this optimal elasticity often encounters two predominant barriers: (1) a target node may sometimes lack sufficient memory, preventing task dispatch, due to the tight coupling between computational and memory resources, and (2) additional overheads introduced by this fragmentation, such as cold starts. The adoption of disaggregated memory has emerged as a viable solution to the first issue, with studies like Fastswap [23] and Pond [52] exploring its benefits. Our research pivots towards harnessing CXL/RDMA-based shared memory pool to tackle the second barrier.



[1] CRIU restores other states of processes, such as recovering multi-thread context, reopening files, and re-establishing sockets.

Figure 4. Breakdown of the latency for a Python-based function, highlighting the overhead associated with the sandbox.

Referring to the "Cold Start" bar in Figure 4, we identify two significant costs: (1) The preparation of an isolated sandbox, including tasks like creating the rootfs, and (2) The cost involved in bootstrapping the function's processes, such as launching interpreters and importing libraries. These ancillary overheads overshadow the proportion of the function's actual productive execution time, given that many studies report that most serverless functions execute in less than one second [48, 64].

2.2.1 Existing Research Historically, caching mechanisms have been extensively studied and deployed to mitigate cold start overheads [42, 62, 64]. However, as discussed earlier, they introduce a trade-off between expediting initialization and reserving resources, which in turn impedes system elasticity.

Prior research efforts have thus been directed towards reducing the cost of caching by eschewing the reservation of active containers. Instead, they created snapshots or templates of a function's complete state immediately after initialization and stored them as files [10, 66]. When a new instance needed to be started, it was restored from the snapshot, bypassing the bootstrapping phase.

Nevertheless, these strategies fail to fully conquer the aforementioned challenges. As highlighted by the CRIU bar in Figure 4:

(1) the need to establish a new isolated sandbox still exists, thereby sustaining the associated overheads. Our findings also indicate that isolation costs escalate with concurrent cold starts. For example, concurrently initiating 15 instances results in a network setup time of 400 ms, aligning with prior studies [58, 59].

(2) the memory restoration overhead ("Mem" in Figure 4) is still non-negligible due to costly data copying. In Figure 4, we store the snapshot files in a local DRAM-based tmpfs to avoid the overhead of disk or network I/O. However, as we can see from the figure, memory copying alone takes over 60 ms during container bootstrapping, even for its small 60MB memory image. This overhead *scales* with the size of the

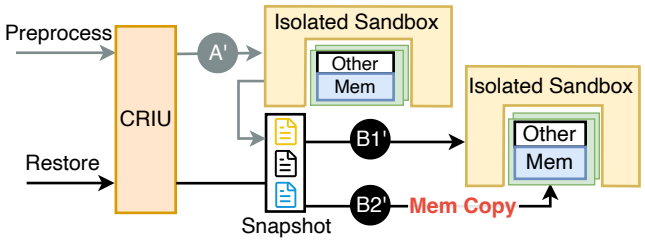


Figure 5. Two phases of CRIU in current container runtime.

memory image; for instance, a 360MB memory image would take over 220 ms to restore.

Further optimizations have been suggested to alleviate these challenges. But, according to our investigation, a holistic solution remains elusive. Lightweight container technologies [59], for instance, mitigate the overhead of sandbox setup but do so at the cost of compromised isolation, making them unsuitable in multi-tenant contexts (more in §4.2 and §5.1). In practice, serverless platforms, such as OpenWhisk [4] and faasd, still resort to standard containers for a better level of isolation, or even opt for secure containers (e.g., VM) [10, 19, 53], which could entail higher initialization costs [75].

Additionally, several "lazy restore" techniques have been proposed to hide the latency of data copying, such as recording and prefetch-based replay [26, 72] or restoring state through efficient RDMA reads in a manner akin to a remote on-demand "fork" [76]. Nonetheless, these methodologies (1) merely defer the restoration overhead to the execution phase rather than reduce it, (2) do not leverage the potential of various remote memory pools, (3) focus on memory restoration but overlook the overhead associated with isolated sandboxes, such as namespaces and cgroups.

3 Overview: From Restore to Repurpose

As discussed earlier with Figure 2, the key idea of TrEnv is an innovative repurposing strategy for reusing sandboxes and memory states of functions. In a traditional caching strategy, the cached container is limited to being reused by the same function. TrEnv diverges from this by enabling an efficient transition from an idle function instance to any one of the pending functions, regardless of its type. Thus, it achieves a universal sandbox pool. By moving away from the rigid, function-type-specific model to a more flexible, function-type-agnostic one, our design substantially enhances the elasticity of serverless computing. Crucially, TrEnv accomplishes this with the same or stronger level of security and isolation compared with previous container-based systems.

Implementing such a transparent and efficient state transition necessitates OS kernel changes and updates to vital serverless infrastructures, such as CRIU. To further elucidate our concept and its accompanying challenges, Figure 5 and Figure 2 present a detailed comparison of the function initialization workflows with the adoption of CRIU before and after our proposed enhancement.

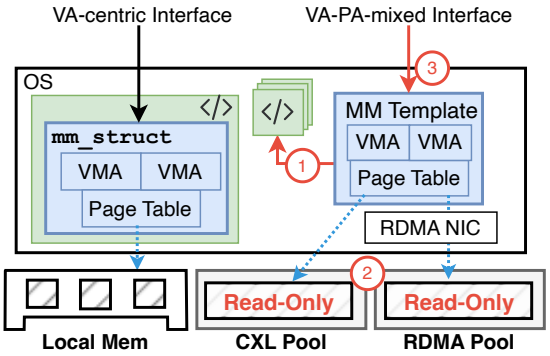


Figure 6. Three highlights in mm-template compared to the `mm_struct` in Linux. ① Not bound to a particular process but can be attached dynamically into any active process. ② Treat all remote memory read-only and enable copy-on-write. ③ Fine-grained control over page tables to map virtual address (VA) to physical address (PA) on remote memory.

The current workflow involves two phases: a preprocessing phase and an online restoration phase. As depicted in step A′ of Figure 5, the preprocessing phase involves creating snapshots for each function and storing them as files. These snapshots, representing the post-initialization state of a function, are used for subsequent restorations. When a function is invoked, the system performs two steps to start a new container: (B1′) it recreates a new sandbox (e.g., rootfs and cgroup) based on metadata in the snapshot image; and then (B2′) it restores the processes states. A significant challenge in step B2′ is restoring the memory state, which requires numerous `mmap()` system calls to recover virtual memory layouts and costly data copying to reload memory contents.

While maintaining both phases, TrEnv introduces improvements to streamline the restoration phase into the more efficient repurposing phase. In the preprocessing phase, TrEnv generates snapshots for each function (step A1 in Figure 2) akin to the original CRIU. Yet, rather than saving memory states as files, TrEnv first deduplicates them into consolidated images stored on remote memory pools, then constructs one memory template (mm-template) *per Linux process* in the function, based on these snapshots (step A2).

As shown in Figure 6, each mm-template is an in-kernel object that has a structure similar to how traditional memory state is maintained by the kernel for each process (i.e., `mm_struct`). It only contains the metadata of the memory state, such as a page table and virtual memory layouts (i.e., `vm_area_struct`). Hence, its size is small (e.g., < 1 MB). By only copying this metadata instead of large memory images during function restoration, the overhead is largely reduced. Additionally, copying from the same mm-template enables seamless memory sharing among the instances of the same function and even across different hosts.

TrEnv leverages these mm-templates to expedite function transitions. As shown in Figure 2, the online restoration phase in TrEnv consists of four steps.

(B1) Upon completion of an instance of function X, TrEnv cleans its sandbox and puts it into a pool instead of discarding it. This step includes terminating existing processes within the container, ensuring any residual state from the predecessor will not be kept for security reasons.
(B2) When a function Y's invocation is pending, TrEnv selects a sandbox from the pool and repurposes it into the pending type Y. This includes applying a unique overlay filesystem of function Y (more in §4.2.1) and restoring the corresponding cgroup limits. Note that TrEnv still offers all isolation components (e.g., cgroup and namespaces) as with a standard Docker container to every function instance.
(B3) TrEnv issues a "repurpose" request to CRIU. CRIU helps the restored processes of function Y to join the repurposed sandbox and recover other process states except for the memory state, such as calling `clone()` to restore multi-thread context. This repurpose-and-join pattern allows efficient reuse of sandbox units like namespaces and cgroups.
(B4) TrEnv attaches the mm-template to the restored process of function Y, efficiently restoring its memory state. Note that TrEnv executes the aforementioned steps *transparently*, requiring no code modifications or recompilation from users.

However, the implementation of the above steps each has its challenges.

3.1 Challenge on Reusing Isolated Sandbox

As shown by the breakdown analysis in Figure 4, the cost of creating a new sandbox predominantly consists of three parts, namely for the rootfs, network environment, and cgroup. While network namespaces and virtual network devices can be safely reused without leaking private data from execution, reusing other resources presents unique challenges.

Each function comes with unique dependencies in its rootfs, such as specific libraries or files. This diversity, along with the requirement of isolation, complicates the possibility of a universal sandbox that can be seamlessly repurposed across functions. Consequently, there is a need for a mechanism that allows rapid filesystem transitions to accommodate these varying dependencies while minimizing storage overhead. Additionally, it is essential to ensure write access to all paths in the rootfs, which is crucial for a practical serverless platform, as emphasized by Alibaba [53] and Amazon [30].

Cgroups play a critical role for container-based resource isolation. Its typical workflow involves three steps: (1) create a cgroup for each container (2) spawn container processes, and (3) move the processes into the newly created cgroup. The third step is referred to as cgroup migration in Linux. Prior studies focused on alleviating the bottlenecks in the first step, such as maintaining a cgroup pool to amortize the creation overhead [53, 59]. Our investigations reveal that the cgroup migration typically incurs latency ranging from 10 to 50 ms, while the cgroup creation latency ranges from 16 ms to 32 ms. Therefore, there is a significant need to address and minimize the overhead stemming from cgroup migration.

3.2 Challenge on Implementing mm-template

The implementation of memory templates necessitates substantial modifications to existing OS memory management interfaces. In this section, we highlight three challenges associated with the implementation. First, mm-template must be extensible and capable of **transparently** supporting access interfaces of different types of remote memory. For example, CPUs can directly access CXL memory via load/store instructions, RDMA employs a message queue model, and NAS uses a block-based I/O interface. The mm-template should accommodate these different interfaces, establishing an extensible framework that can utilize multi-layer memory pools.

Second, the system should **minimize** additional overhead. Traditional memory systems often adopt a lazy or on-demand approach, triggering memory loading during execution via page faults [23] or a userspace daemon [72] via userfaultfd. These approaches are acceptable in previous implementations because the predominant component of their latency remains in I/O (e.g., 60 µs for SSD and 6 µs for RDMA). However, the low-latency, byte-addressable features of CXL memory make it crucial to avoid additional overheads, such as page faults, which could drastically degrade performance and obscure its advantages.

Finally, the OS needs to be extended to **support shared CXL memory**. Current OS memory interfaces are designed primarily for single-host memory expansion (as for CXL 1.0) and hence fall short in supporting cross-host sharing and diverse memory mappings that TrEnv requires.

Presently, two main approaches are used for incorporating CXL memory in Linux: (1) Expose the CXL memory as a CPU-less NUMA node [52, 71]. (2) Designate CXL memory as a special device file (e.g., `/dev/dax0.1`) and pass it to `mmap()`, with the help of direct access (DAX)[1] drivers.

The CPU-less NUMA approach is typically suitable only for memory expansion. Due to the Linux memory allocator's design for physical address transparency, it prevents the specification of physical addresses during allocation. Hence, it is challenging to coordinate multiple hosts to share CXL memory pages. Although the DAX method permits setting physical offsets on CXL memory devices via `mmap`, it has its own set of limitations:

(1) The DAX driver does not support private mappings, which means it does not support copy-on-write for CXL memory pages.
(2) DAX is incompatible with file-backed and some anonymous memory mappings, which are essential for critical process regions like heap and stack.

For example, enforcing mm-template to restore heap areas through the DAX method could inadvertently cause heap

```
// create a new mm-template, return an identifier of mm-template in `id`
int mmt_create(int *id);
// add a memory region to mm-template, most arguments are the same as `mmap()`
int mmt_add_map(int id, void *start, size_t len, int prot, int flags,
    int fd, off_t offset);
// setup page table for a particular memory region at `phys_offset` on
// remote memory. `pool` denotes the type of memory pool (e.g., CXL or RDMA)
int mmt_setup_pt(int id, void *start, size_t len, off_t phys_offset,
    enum mem_pool_t pool);
// attach the mm-template into process identified by `pid`
int mmt_attach(int id, int pid);
```

Figure 7. Core API of mm-template.

growth (e.g., `brk`) to jump into adjacent CXL memory ranges, posing risks of memory disclosure or data corruption.

4 TrEnv Design

In this section, we describe how TrEnv solves the challenges described in §3 with kernel extensions.

4.1 Memory Template Design

To enhance transparency, we have implemented the mm-template within the OS kernel and integrated it into CRIU. This integration allows serverless applications to leverage the advantages of memory pools without the need for code modifications or recompilation. For extensibility, the mm-template supports various memory pool backends via the page table and the page fault mechanism. During the preprocessing phase, mm-template saves essential information in its page table entries (PTEs), including the type of memory pool, remote addresses, and a special bit. For slower backends like RDMA, TrEnv adopts a lazy approach, marking the corresponding PTEs as invalid. Thus, page faults are triggered during execution, allowing the kernel to identify these pages via the special bit in the PTE and route them to appropriate memory pool backends. These backends then use the remote address in the PTE to allocate local pages and load content from remote pools through their specific interfaces and implementations.

In contrast, for low-latency, byte-addressable memory pools like CXL, mm-template preconfigures *valid PTEs*, directly pointing to shared snapshot images stored in CXL memory. This achieves zero software-level cost (instead of on-demand) for read-only access during execution. The CPU can directly access the CXL memory using load instructions, thereby avoiding the additional overhead associated with context switches and minor page faults. After analyzing the functions in Table 2, we found that 24% to 90% of the pages are read-only, consistent with prior research results [22, 69, 72]. Thus, employing a copy-on-write strategy for write accesses in TrEnv is feasible to maintain the integrity of a single copy of memory images on shared memory pools.

Furthermore, to facilitate sharing of CXL memory, TrEnv implements a new driver in the kernel to overcome the limitations of the existing DAX driver. According to our investigation, the drawbacks of the DAX method stem from its tight coupling of the CXL device (e.g., `/dev/dax0.1`) and the virtual memory mappings. Users must create virtual memory mappings through `mmap` and bind those mappings with the CXL device. This (1) conflicts with some anonymous mappings like heap areas, and (2) prevents binding other file with the mapping. By closely integrating the pre-created page tables mentioned earlier, our driver enables virtual memory mappings on CXL without associating DAX devices.

As an illustration of the core mm-template API shown in Figure 7, in the preprocessing phase, TrEnv will *explicitly* set PTEs (via `mmt_setup_pt`) in mm-template, after creating the virtual memory area (via `mmt_add_map`). The new driver helps `mmt_setup_pt` to (1) translate the offsets into physical addresses on CXL memory, (2) set the valid PTE corresponding to those virtual addresses, and (3) *pin* the CXL memory pages in memory. After that, there will be no reliance on the DAX device and its driver to allocate pages or set PTE during execution. Moreover, unlike the DAX method, the new driver can seamlessly support copy-on-write by clearing the writable bit in the PTE. Users of mm-template could create virtual memory mappings backed by CXL memory without binding any special file or device. Thus, mm-template can support both other file-backed mappings and anonymous mappings. For instance, heap areas will reside on CXL memory after attaching mm-template via `mmap_attach`. As there is no binding DAX device anymore, subsequent heap growth during execution will default to local memory allocations, avoiding disrupting other CXL memory regions.

4.1.1 Usage of mm-template Due to the transparency of mm-template, using CXL and RDMA only differs in the `` `pool` `` argument passed to `mmt_setup_pt` for PTE preconfiguration. Most of the complexity is hidden in the kernel. Figure 8 shows the workflow to use mm-template, involving two single-process functions. Initially, during the *offline* preprocessing phase, the system generates a snapshot for each function using CRIU. TrEnv then deduplicates these snapshots, resulting in a consolidated image stored in memory pools (step ① in Fig 8). As shown in Fig 8, the snapshots for functions X and Y include a duplicated region (R2 and R2′, respectively) mapped to the same Block 2 on remote memory. Additionally, TrEnv logs the physical start offset of each memory block on the memory pool, such as 0x88000 for Block 2, which comprises, for example, 4 pages. This physical offset acts as a machine-independent pointer.

Subsequently, each host constructs one mm-template for each process in the function (step ②). For example, TrEnv calls `mmt_create(&X)` to initiate a memory template for function X. This template is then populated with specific virtual memory mappings using the `mmt_add_map` API (step ③). For example, to allocate the private read-only anonymous mapping R2 in X's template, TrEnv executes `mmt_add_map(X, 0x7FFF4000, 4*PAGE_SIZE, PROT_READ, MAP_PRIVATE, -1, 0)`. The address 0x7FFF4000, derived from the snapshot image, represents the original virtual memory address of R2 in the checkpointed process of X. The virtual memory mappings are then linked to the remote memory pool via

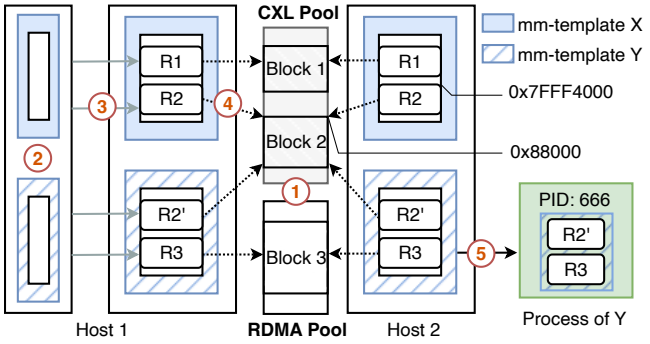


Figure 8. A simplified example of using mm-template.

`mmt_setup_pt` API (step ④). For example, `mmt_setup_pt(X, 0x7FFF4000, 4*PAGE_SIZE, 0x88000, CXL)` associates R2 in X's template with Block 2 on CXL memory. Our new kernel driver helps to convert the offset 0x88000 to the corresponding physical address on CXL memory and install valid PTEs. For the RDMA pool, such as R3 in mm-template Y, it installs invalid PTEs with the address in the RDMA memory pools, then loads memory pages in subsequent page faults.

During the repurposing phase (i.e., critical path), TrEnv attaches the mm-template to the process to be restored, via `mmt_attach` API (step ⑤). Each mm-template can be attached *multiple times*, and it only copies the metadata, e.g., page tables, instead of the memory pages. For instance, to restore the memory state of a process of function Y whose process ID (PID) is 666, TrEnv executes `mmt_attach(Y, 666)`.

4.2 Repurpose Isolated Sandbox

Two challenges related to isolated sandbox were mentioned in §3.1. The first is that varying functions require different dependencies, making the reuse of rootfs non-trivial. The second is that the overhead of cgroup migration is non-negligible and cannot be solved by current approaches.

4.2.1 Rootfs Reconfiguration The standard container rootfs typically consists of multiple mountpoints within a per-container mount namespace, often using a base union filesystem like overlayfs for the root directory. Notably, many dependencies (e.g., glibc, language interpreters) are common across functions. Instead of switching the entire rootfs, our approach focuses on swapping only the function-specific dependencies. By exploiting the flexibility of Linux mountpoints, we overmount another union filesystem atop an existing path, effectively "replacing" its contents, as shown in Figure 9. More precisely, after purging modifications from previous instances, TrEnv dynamically mounts and unmounts separate overlay filesystems tailored to specific function dependencies. TrEnv further enhances this procedure by maintaining a pool of function-specific overlayfs, instead of discarding them after unmounting.

Compared with Cold Start. Typically, preparing a rootfs for containers from scratch requires many system calls, including more than 9 `mount`, 6 `mkdev`, 6 `mknod` and 1 `pivot_root` calls to Linux. TrEnv, however, requires only 2 `mount` at

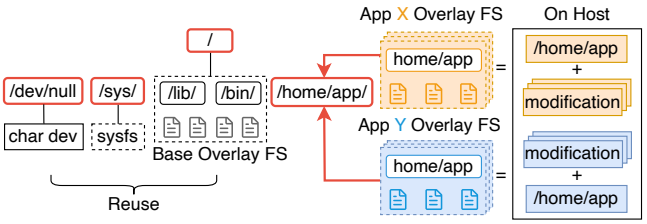


Figure 9. An example of rootfs reconfiguration. The red box indicates the mountpoint.

minimum for function-specific dependencies and `/proc`, efficiently reusing other mountpoints.

Compared with Lightweight Container. Many prior studies have adopted lightweight containers to overcome the isolation bottleneck. For example, SOCK[59] has proposed a *lean container*, using `chroot` and read-only bind mounts to set up its rootfs rapidly. However, according to the Linux manual page [7], `chroot` is not intended for full process sandboxing, and Sun *et al.* [70] have underscored its security limitations, which is susceptible to several potential "escape" attacks. In contrast, TrEnv still adopts the mount namespace as standard containers, which provides more robust isolation.

Compared with Zygote Container. Prior research, such as Pagurus [54], suggests a "zygote" container that can help various functions. In Pagurus, each function-specific directory is assigned a unique owner, and the user ID of the function process is dynamically adjusted to access the corresponding directory. However, it merely provides *read-only* access.TrEnv, conversely, leverages the copy-on-write feature of overlayfs to emulate an ideal single-tenant environment with no restrictions on writable paths. Moreover, zygote containers require additional initialization steps when helping other functions, such as loading unique libraries and user codes, leading to longer latency than TrEnv.

4.2.2 New Cgroup Feature Utilization As discussed in §3.1, cgroup migration suffers from significant latency. To understand the root cause of this latency, we employed ftrace [29] to analyze the call graph in Linux during migration. We find that two global read-write semaphores in the `cgroup_attach_lock()` and `cpuset_can_attach()` are the leading cause. The underlying RCU (Read-Copy Update) synchronization, integral to semaphore implementation, is inherently time-consuming as it necessitates waiting for a grace period to ensure successful write locking.

TrEnv employs the `CLONE_INTO_CGROUP` feature, a recent Linux enhancement for task creation. It significantly speeds up the procedure by assigning a specific cgroup to a process at the time of spawning, rather than post-creation. Now the workflow in TrEnv is: (1) create the cgroup and (2) assign the cgroup directly while spawning container processes. Since the process is still invisible to other OS components while spawning, the kernel can bypass the costly synchronization in cgroup migration. Despite its efficiency, this feature has

yet to be adopted in mainstream container runtimes (e.g., runc). In our evaluation, it typically takes only 100 to 300 µs.

5 Discussion

In this section, we further discuss the security and deployment costs of TrEnv.

(1) Security: TrEnv involves reusing or repurposing certain container components, as outlined in Table 1. It is essential to assess the extent to which this reuse may introduce potential security vulnerabilities.

(2) Deployment Costs: TrEnv optimizes memory restoration and utilization by leveraging emerging hardware technologies (e.g., CXL and RDMA). This approach presents a tradeoff between the benefits gained and the additional deployment costs incurred.

5.1 Security

The mm-template API is implemented through a set of `ioctl` calls on a pseudo-device driver. To ensure its usage is controlled, only users with root privileges can access that device. For repurposable sandboxes, the design aims to provide a level of security and isolation that is equivalent to or exceeds that of existing container-based systems. In the following, we first examine the security implications associated with the reused kernel objects (i.e., container components). We then analyze other security limitations inherent in the current implementation of TrEnv. Finally, we explore the potential for extending TrEnv to work with the VM to meet the higher security requirement in production systems.

5.1.1 Reused Kernel Objects

Network namespace (netns). Lightweight containers (e.g., used in Mitosis [76]) do not employ netns, leading all instances to share the same network environment. In contrast, TrEnv assigns a separate netns to each instance. During repurposing, the previously opened network connections will be forcibly terminated to prevent data leakage. However, certain states, including the network configurations (e.g., firewall rules and routing tables) and statistics (e.g., received bytes of the veth interfaces), are preserved. Many serverless functions, including all evaluated in our study, do not modify network configurations. Therefore, these residual states do not expose data generated during function execution. For other functions that customized the network, these states can be reset as needed to ensure security.

Rootfs. Each rootfs contains a mount namespace and a union filesystem. Its security implications has been discussed in §4.2.1. During repurposing, TrEnv first kills the processes, and then purges the file modifications of the previous instance. Thus, TrEnv does not leak any data, including memory or files, produced by the processing of the last function.

Cgroup. The reuse of cgroup is confirmed and adopted by production systems like Rund [53]. Still, TrEnv provides a cgroup per instance, as standard Docker containers.

A "share" strategy has been proposed by Pagurus [54]. A pool of zygote containers is maintained to mitigate cold start overhead. Nevertheless, all child containers forked from the same zygote container share the isolated components (e.g., rootfs and netns), resulting in poorer isolation compared with TrEnv.

5.1.2 Security Limitations

Although TrEnv prevents data leakage and enforces the proper level of isolation during repurposing, there are other potential security limitations.

(1) ASLR (Address Space Layout Randomization). In TrEnv, all restored container instances share the same memory layout (e.g., the virtual address of the stack) as the mm-template they are attached during restoring, preventing ASLR from introducing randomness. This issue is prevalent in all Checkpoint/Restore and fork-based schemes, where restored or forked containers inherit the virtual address layout of the snapshot or parent process.

(2) Side-channel attacks related to memory deduplication across functions [78]. A possible solution is to only enable it for functions from the same users.

(3) Data protection during transfer. For CXL, the 2.0 specification propose security features, such as IDE (Integrity and Data Encryption). For RDMA, it is possible to encrypt the memory images during transfers.

5.1.3 Applying TrEnv to VMs

Containers are generally considered less secure than (micro-)VMs, due to their reliance on a shared kernel across all execution environments and the host. However, the methodology of TrEnv can be adapted to VMs, for example by creating a pool of repurposable VMs, to meet the security requirements necessary for production environments.

(1) Many hypervisors, including RunD and Cloud Hypervisor [17], support virtio-fs [18], a shared filesystem for VMs. It allows the rootfs within the guest to be a directory on the host. Thus, the rootfs of VMs can be reconfigured on the host by TrEnv.

(2) The hypervisor still needs to execute in an isolated environment on the host. For example, Firecracker [19] needs to run inside a "jail", with independent cgroup and namespaces. The relevant cgroup optimization and netns reusing in TrEnv can be applied directly.

(3) VM's lazy restore approach can be further optimized by pre-populating the EPT (extended page table) for hot memory regions with the help of mm-templates. This avoids triggering a VM exit due to a page fault on first access, thereby enhancing execution efficiency.

5.2 Deployment Cost

For RDMA-based deployment, there is no additional cost since cloud providers already offer RDMA devices (e.g., eRDMA [32], EFA [82]) in production. For CXL-based deployment, a quantitative cost analysis is currently not feasible because CXL 2.0 switches are still in the demo phase. However, we are confirmed from manufacturers that memory expanders' and switches' price will be comparable to DDR5 DIMMs and IB switches, respectively. Previous works [27]

showed that the cost of switches and multi-headed memory controllers are within 5% of the original servers, at the scale of 16 sockets. Thus, we anticipate rack-level deployment of CXL memory pools will be available at an acceptable cost in the near future, motivating our work. A rack-level mini-cluster with around 10 machines and 20TB memory would be sufficient to capitalize on the benefits of TrEnv.

Regarding the energy cost for occupying DRAM and the comparison with pre-warm/keep-alive approaches, TrEnv reduces the overall memory footprint by enabling cross-machine-intra-rack deduplication. Only one copy is needed per rack if it is read-only, reducing the cost by a factor of the number of machines (~10). In contrast, each kept-warm container requires an independent copy, leading to excessive duplication and memory costs.

For larger clusters, it is possible to blend CXL (intra-rack) and RDMA (inter-rack), by adjusting the mm-template, to further enhance the scalability.

6 Evaluations

6.1 Methodology

TrEnv's implementation includes 2900 lines of code (LoC) modifications to CRIU, 3500 LoC alterations to v6.1 Linux, and an RDMA server with 700 LoC. The key addition in CRIU is the "repurpose" command, which extends the existing "restore" command and is integrated into faasd [40].

Testbed. We conducted evaluations of TrEnv using memory pools based on both CXL and RDMA technologies, termed T-CXL and T-RDMA in the following sections, respectively. The test platform was equipped with dual 32-core Intel Xeon Gold 6454S CPUs, 256 GB of RAM, and a 7 TB Samsung PM9A3 SSD. Additionally, the platform was connected to a 128 GB experimental Samsung CXL memory device and a Soft-RoCE RDMA device. In our tests, the latency for accessing remote memory was 641.1 ns for CXL and 6 µs for RDMA.

Evaluated Functions. Our method is universally applicable across diverse language runtimes and is able to handle multi-threaded and multi-process scenarios. Thus, for our evaluation, we selected a wide range of applications from SeBS [35] and Function-Bench [49] (Table 2). Given the prevalent use of Node.js and Python by AWS Lambda developers[36], we also ported several Python (Py) functions to Node.js (Node). This further highlights TrEnv's ability to repurpose between heterogeneous languages.

Table 2. The evaluated functions. The last column shows the number of threads that need to be restored.

Func	Lang	Description	Mem Size	# Thread
DH	Py	Dynamic web pages generating.	50.4M	14
JS	Py	Deserialize and serialize json.	94.9M	14
PR	Py	Pagrank algorithm.	116M	395
IR	Py	Deep learning (ResNet).	855M	141
IP	Py	Image rotating and flipping.	67.1M	15
VP	Py	Gray-scale effect on video.	324M	204
CH	Py	HTML tables rendering.	94.9M	38
CR	Node	AES encryption algorithm.	124M	16
JJS	Node	Similar to JS.	111M	21
IFR	Node	Similar to IP.	253M	21

Baselines. TrEnv is based on faasd, a widely used serverless platform that not only provides a baseline for our comparative analysis but also supports CRIU natively.

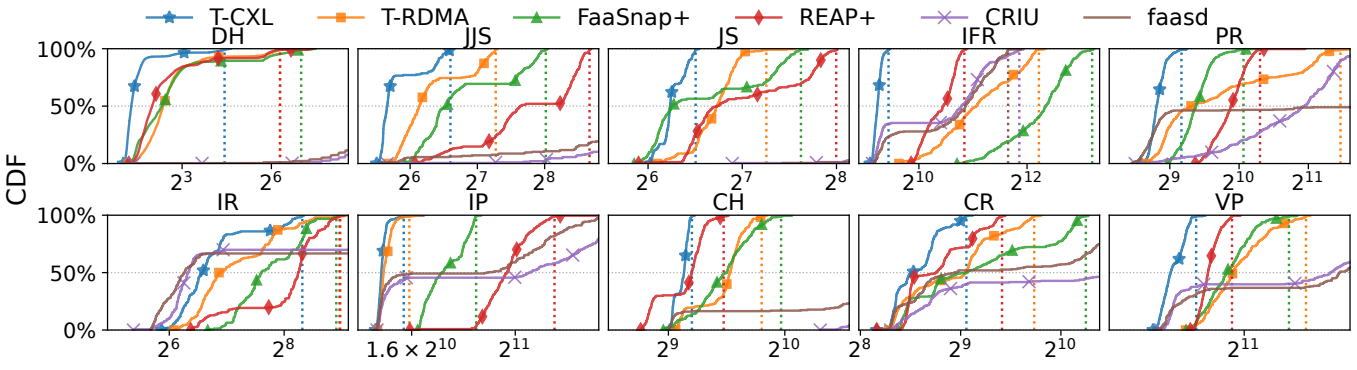
Additionally, the current predominant approach to mitigating cold start issues in serverless computing is "lazy restoration", which prioritizes the recovery of essential states and delays other tasks (e.g., memory content restoration) until necessary. Thus, we also compare TrEnv with REAP and FaaSnap, two state-of-the-art lazy restoration methods that utilize Firecracker. Notably, FaaSnap builds on top of REAP by introducing an asynchronous prefetch policy. Each guest for FaaSnap and REAP has 2 vCPUs and 2 GB memory.

During our evaluation, we noted that the network configuration phase during Firecracker initialization introduces significant overhead, which can reach up to 600 ms under scenarios of high load and concurrency. This overhead allows TrEnv to outperform them substantially. To facilitate a more meaningful comparison, we developed a network namespace pool for both REAP and FaaSnap. After a virtual machine (VM) is terminated, its network environment is recycled into this pool for later reuse, analogous to our repurposable sandbox pool. We refer to these enhanced implementations as REAP+ and FaaSnap+, respectively.

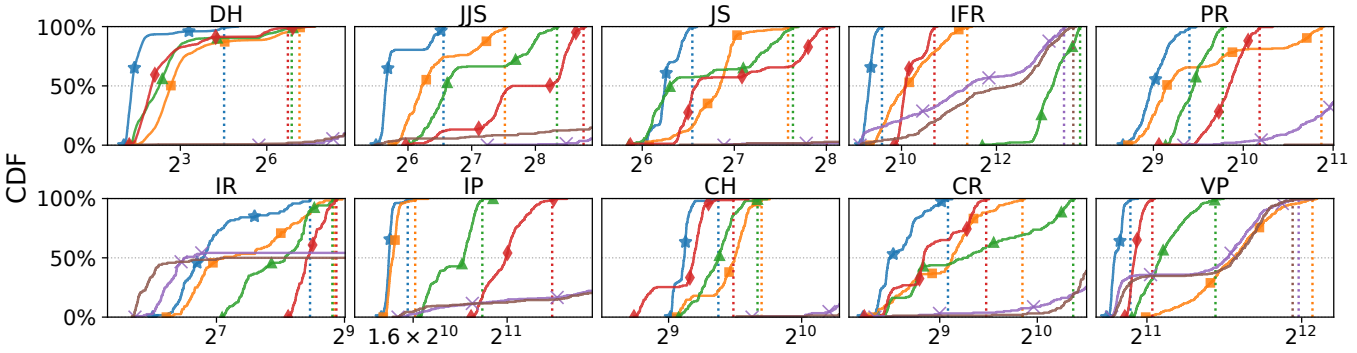
Schedule Policy. We implement a widely used scheduling policy across all evaluated methods. After the invocation, instances are retained in a container pool for a fixed duration (e.g., 10 minutes) akin to Openwhisk [12], commonly referred to as keep-alive. This approach allows for the immediate reutilization of cached instances for new invocations of the *same function*. The container pool works as a Least Recently Used (LRU) list, organized by the most recent activity.

Workloads. We assessed the performance of TrEnv using both synthetic and real-world workload traces. Specifically, bursty loads and diurnal patterns are the two most commonly observed patterns in real-world serverless platforms that lead to load instability and thus diminish the effectiveness of the keep-alive strategy [42, 54, 62, 64, 67, 73]. To accurately simulate these conditions, we designed two specific workloads: W1 and W2. W1 replicates bursty traffic patterns, with intervals between consecutive bursts *longer than the keep-alive threshold*. In contrast, W2 emulates diurnal traffic fluctuations, cycling through various functions under *tight memory limits* (a soft memory cap of 32GB is applied in W2, compared to the 64GB used in other tests). To further substantiate TrEnv's utility in practical scenarios, we also incorporated industry traces from Azure [64] and Huawei [48] into our evaluations.

To ensure the effectiveness of traditional caching mechanisms, all tests include a warm-up phase of about 5 minutes. Additionally, snapshot images of CRIU, REAP, and FaaSnap



(a) W1: bursty loads with intervals longer than the keep-alive threshold.



(b) W2: diurnal and heavy traffic under tight memory limits.

Figure 10. CDF of different functions' E2E latency under two representative workloads, W1 and W2. The vertical dotted line indicates the P99 E2E latency. Note that x-axis denotes the E2E latency in milliseconds on a logarithmic scale with a base of 2.

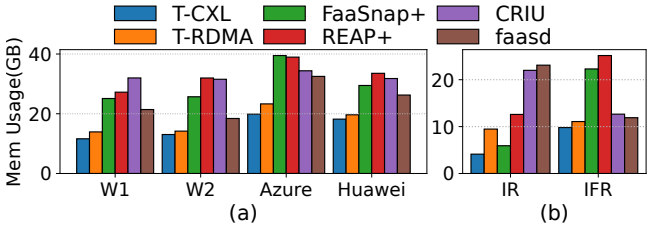


Figure 11. (a) Peak memory usage during four workload tests. (b) The memory usage when starting 50 instances of IR and IFR, respectively.

are stored on a CXL-memory-backed tmpfs to eliminate disk overhead, ensuring a fair comparison with T-CXL.

In the following evaluation, we focus on the following aspects. (1) **Performance** (§6.2, §6.3): What is the benefit of TrEnv on *end-to-end* latency, especially under concurrency and the P99 latency? (2) **Memory utilization** (§6.2, §6.3): What is the benefit of TrEnv on the reduction of *memory usage*? (3) **Optimization Breakdown** (§6.4): What is the contribution of each optimization proposed by TrEnv? (4) **Memory Pool Comparison** (§6.5): What is the difference between CXL and RDMA memory pool?

6.2 Representative Workload

Figure 10 displays the cumulative distribution function (CDF) of end-to-end (E2E) latency for functions within the W1 and W2 workloads, capturing over 4k invocations in 30 minutes. For clarity, the x-axis shows the logarithmic scale (base 2) latency, and the results for CRIU and faasd are partially truncated due to their relatively longer latencies.

As a summary, in terms of tail latency, T-CXL consistently outperforms all other solutions, achieving a speedup ranging from 1.11×-5.69× (1.17×-18×) for P99 latency and up to 5.9× (8.6×) for median latency, compared with REAP+ (FaaSnap+). The superior performance primarily stems from T-CXL's reduced execution time, as FaaSnap+ and REAP+ incur more context switches and on-demand restoration overheads, particularly under conditions of high concurrency. T-RDMA also shows promising performance; however, it does not perform as well in certain scenarios due to the higher latency associated with RDMA access compared to the CXL-backed tmpfs used in REAP+ and FaaSnap+.

Regarding memory utilization, as detailed in Figure 11a, T-CXL achieves an average reduction in memory usage relative to faasd, CRIU, REAP+, and FaaSnap+ in W1 and W2 by 37.4%, 61.2%, 58.2%, and 51.5%, respectively. T-RDMA exhibits a comparable level of memory savings. These improvements are attributed to memory sharing, deduplication, and reduced number of instances to keep in memory, thanks to our repurposing technique that allows for the effective share of execution environments across various functions.

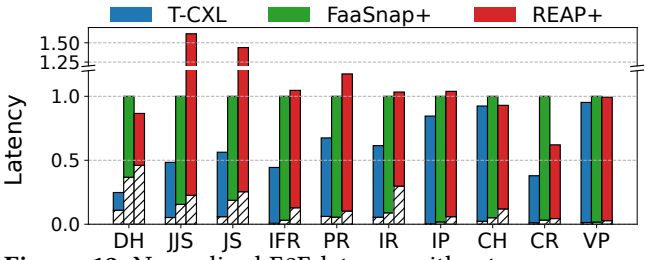


Figure 12. Normalized E2E latency without concurrency, while the hatched region denotes the startup time.

6.2.1 Comparing with CRIU and faasd The suboptimal performance of CRIU and faasd stems from their high startup latency. As given in Table 1, creating isolation environments can take over one second under conditions of heavy loads and high concurrency. Additionally, faasd encounters delays due to application initialization during cold starts, whereas CRIU's restoration process is hampered by the expensive copy-based memory restoration. For instance, launching a CR instance takes 1.7 s at P99, whereas its execution time is only about 500 ms. Thanks to our repurposable sandboxes and mm-templates, TrEnv significantly reduces the costs associated with isolation environment creation and memory restoration. It takes merely 15 milliseconds for T-CXL to start the same CR instance at P99, representing a reduction of more than 100×. In §6.4, we will explore the individual contributions of the repurposing and mm-template techniques.

T-CXL relies on CXL, which has higher latency than local DRAM, resulting in degraded execution performance. This explains CRIU's better P50 IR performance in Fig 10. For instance, T-CXL nearly doubles the execution time of DH and IR due to their short runtimes (<100 ms), while other functions see about a 10% increase on average. However, during cold starts, T-CXL's E2E latency is significantly lower than CRIU's, thanks to its efficient startup process. Additionally, performance can be improved by configuring mm-templates to store hot regions of memory image in local DRAM.

6.2.2 Comparing with REAP+ and FaaSnap+ Both REAP+ and FaaSnap+ are based on Firecracker. While the primary focus is on enhancing security, the additional layer of the hypervisor facilitates efficient state restoration. Despite the reduced gap, TrEnv, which is container-based, still achieves better start latency. For example, it takes only 13 ms to start an instance of CH at P99, compared to 49 (90) ms for FaaSnap+ (REAP+). Moreover, Firecracker leads to significantly higher memory consumption (as shown in Figure 11a) than containers, due to each guest OS maintaining exclusive resources, such as page caches. A microbenchmark, illustrated in Figure 11b, further highlights this issue. When starting 50 function instances, we observed that FaaSnap and REAP even doubled the memory usage compared to T-CXL.

In addition to startup latency, TrEnv achieves lower tail latency than both REAP+ and FaaSnap+ due to its much more stable and **shorter execution time**. REAP and FaaSnap use a lazy restore approach that only delays, rather than eliminates, restoration overheads during execution, especially for memory. For instance, handling each page fault still requires several microseconds by the OS, even when their snapshots are stored on a CXL-based tmpfs. In contrast, TrEnv eagerly restores most states instead of on-demand, such as opened files. Further, by leveraging CXL's byte addressability, mm-template avoids page faults for read-only pages. According to our investigations, about 24% to 90% of the pages used during execution are read-only. Thus, as illustrated in Figure 12, the execution time for TrEnv is considerably shorter than that for REAP+ and FaaSnap+.

6.2.3 The effect of function characteristics Different applications exhibit distinct characteristics that influence their performance. The functions we evaluated can be categorized into three groups. (1) Memory-insensitive applications, such as compute-intensive (like VP and IP) and I/O-intensive (like CH) applications. The latency for these applications does not show a significant disparity among different methods, as they are primarily CPU-bound or I/O-bound rather than memory-bound. As a result, the differences in E2E latency between CXL, RDMA, and even userfaultfd-based page faults (used in REAP) are not evident. (2) Applications with a large memory footprint and complex memory access patterns, including IR, PR, and IFR. They lead to more minor page faults in REAP+ and FaaSnap+, resulting in longer execution time than T-CXL. T-RDMA can also experiences longer tail latency, especially during burst, due to unstable P99 latency of RDMA under high request rates. (3) Applications with brief execution time, which is common in serverless scenarios, including DH, JS, CR and JJS. Any extra overhead, such as minor page faults in FaaSnap or REAP, and longer latency for RDMA, significantly impacts their E2E latency. The shorter startup latency and zero cost for read accesses in T-CXL demonstrate more pronounced effects for these functions.

6.3 Real-world Workload.

To further justify the effectiveness of our designs, we tested TrEnv under two industrial and complex workloads (Figure 13). As both datasets only record the number of invocations per minute, we randomly distributed those within each minute, with a probability of creating skew or bursty loads to imitate real-world conditions.

In summary, for tail latency, T-CXL achieves speedups ranging from 1.06×-7.00× and 1.16×-9.25× compared with REAP+ and FaaSnap+, due to our shorter execution time under heavy loads. T-RDMA falls behind REAP+ and FaaSnap+ in JS, VP, CH, CR, and PR. The unstable P99 memory access latency of RDMA still incurs a delay in these heavy-load scenarios. Nevertheless, T-RDMA still achieves a speedup of 1.29×-4.28× and 1.11×-4.64× against REAP+ and FaaSnap+ for other functions. As shown in Figure 11, T-CXL and T-RDMA reduce memory usage by over 25% compared to all baselines in both workloads. Specifically, T-CXL reduces

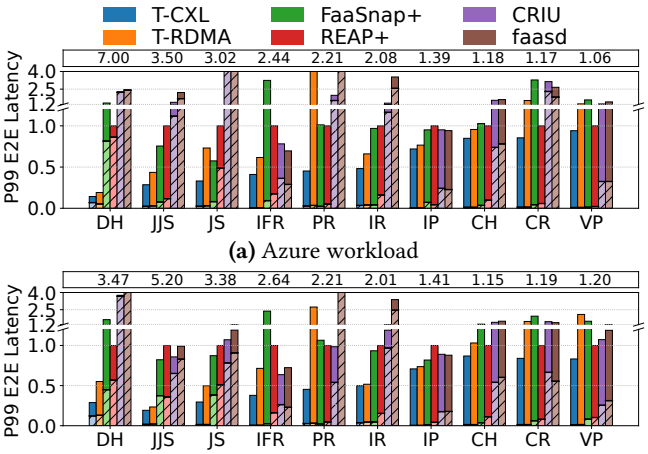


Figure 13. P99 E2E latency for read-world workloads, normalized against REAP latency. Each bar is divided into two segments: the upper un-hatched represents execution time, and the lower hatched indicates startup time. Top numbers are the speedup of T-CXL compared to REAP+.

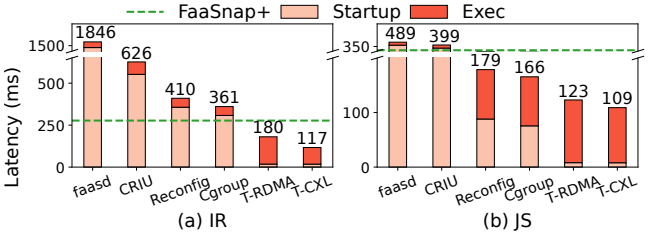


Figure 14. Optimization steps and its effects to E2E latency. The green line indicates the E2E latency of FaaSnap+.

memory usage by up to 49% relative to REAP+ and FaaSnap+. T-RDMA consumes, on average, 10% more memory than T-CXL.

6.4 Breakdown of Optimization Steps

To delve deeper into the contribution of each optimization, we analyzed the E2E latency by enabling different optimizations in TrEnv step by step.

In Figure 14, the "Reconfig" optimization applies sandbox repurposing without cgroup optimization, reducing startup latency by approximately 200 ms. Regarding rootfs reconfiguration, one subtask of the repurpose, it takes more than 30 ms to restore the rootfs and mount namespace in CRIU. In contrast, TrEnv's reconfiguration process involves only two system calls and typically completes rootfs preparation in less than 1 ms. Additionally, the "Cgroup" optimization employs the `CLONE_INTO_CGROUP` to bypass synchronization delays within the kernel, further reducing startup latency by 49 ms for IR and 13 ms for JS.

After that, the function's startup latency is predominantly governed by memory restoration (>85%). TrEnv reduces this cost through mm-template. First, it only needs to copy a small amount of metadata rather than the memory pages.

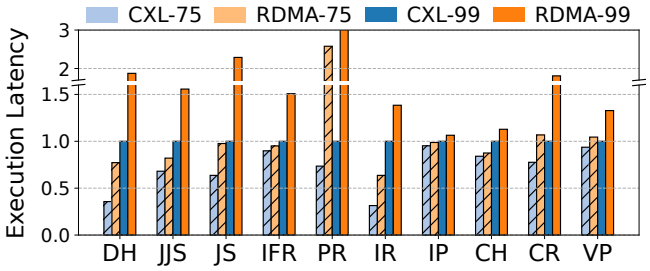


Figure 15. Normalized execution latency of T-CXL and T-RDMA. The hatched bar indicates P75 latency, while the un-hatched bar indicates P99 latency.

For example, the metadata is less than 400 KB, while the memory contents exceed 70 MB for JS. Second, it eliminates the need for reconstructing the virtual memory layout since it is already preserved in the mm-template, thus avoiding numerous `mmap()` system calls in CRIU. The larger the memory footprint, the greater the reduction in startup latency achieved by mm-template. The mm-template alone reduces startup latency by 290 ms for IR and 67 ms for JS, allowing TrEnv to launch an IR and JS instance in just 18 ms and 8 ms, respectively.

Utilizing the mm-template with a remote memory pool leads to a slight increase in execution time compared to when local DRAM is used, because of the inherently longer access latencies associated with CXL and RDMA memory. Specifically, an additional 24 ms (88 ms) for IR and 11 ms (25 ms) for JS, are observed for T-CXL (T-RDMA), compared to CRIU. Nonetheless, the overall E2E latency still experiences a substantial reduction.

We also evaluated the "Cgroup" (i.e., enable repurposable sandboxes but without mm-template) with industrial workloads: T-CXL still outperforms "Cgroup" in all functions, achieving the speedup ranging from 1.04× to 2.32×.

6.5 TrEnv-CXL vs. TrEnv-RDMA

The flexibility of TrEnv enables it to leverage various interconnect technologies. However, the differing physical characteristics of CXL and RDMA lead to disparities in execution time. As shown in Figure 15, T-CXL outperforms T-RDMA across all functions, with speed increases ranging from 1.04× to 3.51× for P75 latency. This improvement is attributed to (1) the faster access latency of CXL memory, and (2) the utilization of CXL's byte-addressable feature. The zero additional software-level cost for read accesses with mm-templates allows T-CXL to eliminate page faults that T-RDMA encounters for read-only pages. Analysis of our evaluated functions reveals that a significant portion of the memory pages are read-only, ranging from 24% to 90%.

Furthermore, the disparity in P99 latency between T-CXL and T-RDMA is even more pronounced than at P75. Under extreme loads at P99, the heavy RDMA traffic exacerbates CPU load and flow interference, and increases contention for network resources such as the switch [81] and RDMA

NIC [34]. Several works report similar observations, with this performance cliff being nearly fivefold in instances of burst traffic [50]. In contrast, CXL memory offers higher IOPS and more stable latency at P99. CPU usage monitored during two industrial tests with `mpstat` shows that T-RDMA's total CPU usage is 1.24× and 1.23× higher than T-CXL for Azure and Huawei traces, respectively.

Figure 11b also demonstrates that for read-heavy functions like IR, T-CXL primarily accesses remote memory directly, avoiding the allocation of local pages. It substantially saves memory (43.5%) compared to T-RDMA. Conversely, for write-heavy functions like IFR, T-CXL shows lesser memory gains (13%) as both trigger COW for write access.

Nonetheless, since all states in the memory pool are read-only, a multi-layered architecture that strategically places hot pages in CXL and cold pages in RDMA, integrates seamlessly with our approach. The specific cache eviction strategies are orthogonal to our core implementation.

7 Related Work

Sandboxing techniques. Prior research has explored various sandboxing techniques like micro-VMs [10, 19, 53], lightweight containers [59], unikernels [31], and WebAssembly-based sandboxes [43, 65, 80]. The current implementation of TrEnv primarily focuses on containers.

Caching and pre-warm. Prior studies have sought to reduce the overhead of cold starts by employing caching or pre-warming techniques. After execution, the platform may keep the container alive for a certain duration, reusing it if the same function is invoked again within that period. Besides, the platform may pre-start containers for functions that are likely to be executed soon. Some platforms utilize fixed keep-alive times or static pre-warm strategies [12, 75], while recent research has explored heuristic approaches [42, 62, 64], including the use of machine learning models [77]. TrEnv takes a different approach by directly reducing cold start overhead, thereby eliminating the need for designing those complex strategies. Groundhog [22], a lightweight sequential request isolation system, restores memory to a "clean" state before reuse in its caching strategy. In contrast, TrEnv focuses on mitigating cold start overhead by sharing resources across different functions and hosts. Additionally, TrEnv is capable of providing functionality similar to Groundhog by fully restoring the container state after each execution.

Utilization of "fork". Several studies have investigated the use of the OS 'fork' primitive for efficiently initiating new instances from a cached container or zygote [38, 39, 54, 59]. Moreover, MITOSIS [76] enhances the OS by implementing a remote fork primitive that enables the use of states from remote machines. While TrEnv shares some similarities with MITOSIS, there are significant differences.

Firstly, MITOSIS utilizes less-isolated, lightweight containers to reduce isolation costs, which results in compromised isolation levels, as discussed in Section 4.2.1 and 5.1.1. Secondly, MITOSIS is currently limited to supporting single-thread functions. Overcoming this limitation typically requires specialized solutions that involve both the language runtime and the OS levels, complicating the development of a universal "fork" solution. In contrast, TrEnv is built upon CRIU, which supports the restoration of multi-threaded and multi-process environments more robustly. Although MITOSIS achieves startup latencies of less than 4 ms for many single-threaded functions, initializing real-world serverless functions that involve multiple processes and threads is inherently more complex and costly. Furthermore, while both MITOSIS and TrEnv utilize kernel-space RDMA, MITOSIS depends on a single container (i.e., the parent) to fork from, thus focusing primarily on optimizing RDMA's scalability. In contrast, TrEnv leverages disaggregated memory, focusing on how memory is transparently shared. Moreover, TrEnv is designed to accommodate various types of disaggregated memory technologies, including RDMA and CXL.

Storage Systems. Many researchers have tried to optimize serverless computing by enabling more efficient state transfer [61, 68, 74]. They build scalable and distributed storage or cache systems. These works are orthogonal to TrEnv and can be integrated to enhance TrEnv's I/O performance.

8 Conclusion

We introduced TrEnv, a serverless platform providing extreme elasticity through sharing as many resources as possible. It makes use of repurposable sandboxes to reduce the isolation environment overhead, and mm-templates to enable rapid restoration from the disaggregated memory. The evaluation of TrEnv shows that memory consumption can be reduced by 48% on average, and P99 startup latency can be accelerated by up to 7×, compared to SOTA lazy restore approaches (FaaSnap and REAP).

Acknowledgments

We thank the anonymous reviewers and our shepherd, Prof. Michael Stumm, for their valuable feedback. The authors affiliated with Tsinghua University are all in the Department of Computer Science and Technology, Beijing National Research Center for Information Science and Technology (BNRist), Tsinghua University, China. This work is supported by National Key Research & Development Program of China (2022YFB4502004), Natural Science Foundation of China (62141216) and Tsinghua University Initiative Scientific Research Program, Young Elite Scientists Sponsorship Program by CAST (2022QNRC001), and Beijing HaiZhi XingTu Technology Co., Ltd. This work was also supported by Alibaba Group through Alibaba Innovative Research Program. Correspondence to: Mingxing Zhang (zhang_mingxing@mail.tsinghua.edu.cn), Teng Ma (sima.mt@alibaba-inc.com), and Yongwei Wu (wuyw@tsinghua.edu.cn).

References

[1] 2014. DAX: Page cache bypass for filesystems on memory storage. https://lwn.net/Articles/618064/.

[2] 2022. Compute Express Link 3.0. https://www.computeexpresslink.org/_files/ugd/0c1418_a8713008916044ae9604405d10a7773b.pdf.

[3] 2022. Compute Express Link™: The Breakthrough CPU-to-Device Interconnect. https://www.computeexpresslink.org/home.

[4] 2023. Apache OpenWhisk. https://openwhisk.apache.org/.

[5] 2023. AWS Lambda Customer Case Studies. https://aws.amazon.com/lambda/resources/customer-case-studies/.

[6] 2023. Azure Functions Premium plan. https://learn.microsoft.com/en-us/azure/azure-functions/functions-premium-plan?tabs=portal#eliminate-cold-starts.

[7] 2023. Chroot—Linux manual page. https://man7.org/linux/man-pages/man2/chroot.2.html.

[8] 2023. CRIU. https://www.criu.org/Main_Page.

[9] 2023. CXL 2.0 Memory Pooling Solution. https://www.h3platform.com/product-detail/overview/35.

[10] 2023. Google gVisor: Container Runtime Sandbox. https://github.com/google/gvisor.

[11] 2023. MemVerge CXL Technology. https://memverge.com/cxl-technology/.

[12] 2023. Openwhisk Warmed Containers. https://github.com/apache/openwhisk/blob/master/docs/warmed-containers.md.

[13] 2023. Podman. https://docs.podman.io/.

[14] 2023. Samsung Develops Industry's First CXL DRAM Supporting CXL 2.0. https://news.samsung.com/global/samsung-develops-industrys-first-cxl-dram-supporting-cxl-2-0.

[15] 2023. Samsung, MemVerge, H3 Platform, and XConn Demonstrate Memory Pooling and Sharing for "Endless Memory". https://memverge.com/samsung-memverge-h3-platform-and-xconn-demonstrate-memory-pooling-and-sharing-for-endless-memory/.

[16] 2023. Xconn Technologies. https://www.xconn-tech.com/.

[17] 2024. Cloud Hypervisor. https://www.cloudhypervisor.org/.

[18] 2024. Virtiofs. https://virtio-fs.gitlab.io/.

[19] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*. 419–434.

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

[21] Alibaba. 2023. Alibaba Function Compute. https://alibabacloud.com/product/function-compute.

[22] Mohamed Alzayat, Jonathan Mace, Peter Druschel, and Deepak Garg. 2023. Groundhog: Efficient request isolation in FaaS. In *18th European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 398–415. https://doi.org/10.1145/3552326.3567503

[23] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can far memory improve job throughput?. In *15th European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 14, 16 pages. https://doi.org/10.1145/3342195.3387522

[24] Amazon. 2023. AWS Lambda. https://aws.amazon.com/lambda/.

[25] Lixiang Ao, Liz Izhikevich, Geoffrey M Voelker, and George Porter. 2018. Sprocket: A serverless video processing framework. In *2018 ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. 263–274.

[26] Lixiang Ao, George Porter, and Geoffrey M Voelker. 2022. Faasnap: Faas made fast using snapshot-based VMs. In *17th European Conference on Computer Systems*. 730–746.

[27] Daniel S. Berger, Daniel Ernst, Huaicheng Li, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Lisa Hsu, Ishwar Agarwal, Mark D. Hill, and Ricardo Bianchini. 2023. Design Tradeoffs in CXL-Based Memory Pools for Public Cloud Platforms. *IEEE Micro* 43, 2 (2023), 30–38. https://doi.org/10.1109/MM.2023.3241586

[28] James Beswick. 2023. New for AWS Lambda – Predictable start-up times with Provisioned Concurrency. https://aws.amazon.com/cn/blogs/compute/new-for-aws-lambda-predictable-start-up-times-with-provisioned-concurrency/.

[29] Tim Bird. 2009. Measuring function duration with ftrace. In *the Linux Symposium*, Vol. 1.

[30] Marc Brooker, Mike Danilov, Chris Greenwood, and Phil Piwonka. 2023. On-demand container loading in AWS Lambda. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 315–328. https://www.usenix.org/conference/atc23/presentation/brooker

[31] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: Skip redundant paths to make serverless fast. In *15th European Conference on Computer Systems*. 1–15.

[32] Hang Cao, Cheng Xu, Yunqi Han, Muhui Lin, Kai Shen, Geng Wang, Jinhu Li, Xiangzheng Sun, Ronghui He, Liang You, Hang Yang, and Xiantao Zhang. 2024. An efficient cloud-based elastic RDMA protocol for HPC applications. *CCF Transactions on High Performance Computing* 6, 1 (2024), 45–53. https://doi.org/10.1007/s42514-023-00170-y

[33] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2019. Cirrus: A serverless framework for end-to-end ML workflows. In *2019 ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. 13–24.

[34] Youmin Chen, Youyou Lu, and Jiwu Shu. 2019. Scalable RDMA RPC on reliable connection with efficient resource sharing. In *14th EuroSys Conference 2019*. 1–14.

[35] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. 2021. SeBS: A serverless benchmark suite for function-as-a-service computing. In *22nd International Middleware Conference* (Québec city, Canada) *(Middleware '21)*. Association for Computing Machinery, New York, NY, USA, 64–78. https://doi.org/10.1145/3464298.3476133

[36] Datadog. 2023. The State of Serverless. https://www.datadoghq.com/state-of-serverless/.

[37] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 401–414.

[38] Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. 2022. Serverless computing on heterogeneous computers. In *27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. 797–813.

[39] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. 467–481.

[40] Alex Ellis. 2023. faasd - a lightweight & portable faas engine. https://github.com/openfaas/faasd.

[41] Alexander Fuerst, Stanko Novaković, Íñigo Goiri, Gohar Irfan Chaudhry, Prateek Sharma, Kapil Arya, Kevin Broas, Eugene Bak, Mehmet Iyigun, and Ricardo Bianchini. 2022. Memory-harvesting VMs in cloud platforms. In *27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 583–594. https://doi.org/10.1145/3503222.3507725

[42] Alexander Fuerst and Prateek Sharma. 2021. FaasCache: Keeping serverless computing alive with greedy-dual caching. In *26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. 386–400.

[43] Phani Kishore Gadepalli, Sean McBride, Gregor Peach, Ludmila Cherkasova, and Gabriel Parmer. 2020. Sledge: A serverless-first, light-weight wasm runtime for the edge. In *21st International Middleware Conference*. 265–279.

[44] Google. 2023. Google Cloud Functions. https://cloud.google.com/functions.

[45] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient memory disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 649–667. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/gu

[46] Seyyed Ahmad Javadi, Amoghavarsha Suresh, Muhammad Wajahat, and Anshul Gandhi. 2019. Scavenger: A black-box batch workload resource manager for improving utilization in cloud environments. In *2019 ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 272–285. https://doi.org/10.1145/3357223.3362734

[47] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the cloud: Distributed computing for the 99%. In *2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. 445–451.

[48] Artjom Joosen, Ahmed Hassan, Martin Asenov, Rajkarn Singh, Luke Darlow, Jianfeng Wang, and Adam Barker. 2023. How does it function? Characterizing long-term trends in production serverless workloads. In *2023 ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '23)*. Association for Computing Machinery, New York, NY, USA, 443–458. https://doi.org/10.1145/3620678.3624783

[49] Jeongchul Kim and Kyungyong Lee. 2019. Functionbench: A suite of workloads for serverless cloud function service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 502–504.

[50] Xinhao Kong, Jingrong Chen, Wei Bai, Yechen Xu, Mahmoud Elhaddad, Shachar Raindel, Jitendra Padhye, Alvin R Lebeck, and Danyang Zhuo. 2023. Understanding RDMA microarchitecture resources for performance isolation. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 31–48.

[51] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, et al. 2019. Software-defined far memory in warehouse-scale computers. In *24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. 317–330.

[52] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. 2023. Pond: CXL-based memory pooling systems for cloud platforms. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '23)*. 574–587.

[53] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. 2022. RunD: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 53–68.

[54] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. 2022. Help rather than recycle: Alleviating cold startup in serverless computing through inter-function container sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 69–84. https://www.usenix.org/conference/atc22/presentation/li-zijun-help

[55] Teng Ma, Mingxing Zhang, Kang Chen, Zhuo Song, Yongwei Wu, and Xuehai Qian. 2020. AsymNVM: An efficient framework for implementing persistent data structures on asymmetric NVM architecture. In *25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 757–773. https://doi.org/10.1145/3373376.3378511

[56] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent page placement for CXL-enabled tiered-memory. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23)*. 742–755.

[57] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.* 2014, 239, Article 2 (mar 2014).

[58] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. 2019. Agile cold starts for scalable serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. USENIX Association, Renton, WA. https://www.usenix.org/conference/hotcloud19/presentation/mohan

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

[60] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 193–206. https://www.usenix.org/conference/nsdi19/presentation/pu

[61] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. 2021. Faa$T: A transparent auto-scaling cache for serverless applications. In *2021 ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 122–137. https://doi.org/10.1145/3472883.3486974

[62] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2022. Icebreaker: Warming serverless functions better with heterogeneity. In *27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. 753–767.

[63] Divyanshu Saxena, Tao Ji, Arjun Singhvi, Junaid Khalid, and Aditya Akella. 2022. Memory deduplication for serverless computing with medes. In *17th European Conference on Computer Systems*. 714–729.

[64] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 205–218.

[65] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 419–433. https://www.usenix.org/conference/atc20/presentation/shillaker

[66] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. 2020. Prebaking functions to warm the serverless cold start. In *21st International Middleware Conference*. 1–13.

[67] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. 2021. Atoll: A scalable low-latency serverless platform. In *2021 ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 138–152. https://doi.org/10.1145/3472883.3486981

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

[69] Jovan Stojkovic, Tianyin Xu, Hubertus Franke, and Josep Torrellas. 2023. MXFaaS: Resource sharing in serverless environments for parallelism and efficiency. In *50th Annual International Symposium on Computer Architecture*. 1–15.

[70] Yuqiong Sun, David Safford, Mimi Zohar, Dimitrios Pendarakis, Zhongshu Gu, and Trent Jaeger. 2018. Security namespace: Making Linux security frameworks available to containers. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 1423–1439. https://www.usenix.org/conference/usenixsecurity18/presentation/sun

[71] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Ipoom Jeong, Ren Wang, and Nam Sung Kim. 2023. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. *arXiv preprint arXiv:2303.15375* (2023).

[72] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, analysis, and optimization of serverless function snapshots. In *26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. 559–572.

[73] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. 2021. FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at Alibaba Cloud Function Compute. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 443–457. https://www.usenix.org/conference/atc21/presentation/wang-ao

[74] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. 2020. InfiniCache: Exploiting ephemeral serverless functions to build a cost-effective memory cache. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 267–281. https://www.usenix.org/conference/fast20/presentation/wang-ao

[75] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 133–146. https://www.usenix.org/conference/atc18/presentation/wang-liang

[76] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. 2023. No provisioned concurrency: Fast RDMA-codesigned remote fork for serverless computing. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 497–517.

[77] Zhengjun Xu, Haitao Zhang, Xin Geng, Qiong Wu, and Huadong Ma. 2019. Adaptive function launching acceleration in serverless computing platforms. In *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*. 9–16. https://doi.org/10.1109/ICPADS47876.2019.00011

[78] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 719–732. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

[79] Mingxing Zhang, Teng Ma, Jinqi Hua, Zheng Liu, Kang Chen, Ning Ding, Fan Du, Jinlei Jiang, Tao Ma, and Yongwei Wu. 2023. Partial failure resilient memory management system for (CXL-based) distributed shared memory. In *29th Symposium on Operating Systems Principles*. 658–674.

[80] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. 2019. Narrowing the gap between serverless and its state with storage functions. In *2019 ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. 1–12.

[81] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion control for large-scale RDMA deployments. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 523–536.

[82] Tobias Ziegler, Dwarakanandan Bindiganavile Mohan, Viktor Leis, and Carsten Binnig. 2022. EFA: A viable alternative to RDMA over InfiniBand for DBMSs?. In *18th International Workshop on Data Management on New Hardware* (Philadelphia, PA, USA) *(DaMoN '22)*. Association for Computing Machinery, New York, NY, USA, Article 10, 5 pages. https://doi.org/10.1145/3533737.3538506