# CXLfork: Fast Remote Fork over CXL Fabrics

Chloe Alverti
University of Illinois
Urbana-Champaign
Champaign, IL, USA
xalverti@illinois.edu

Stratos Psomadakis
National Technical
University of Athens
Athens, Greece
psomas@cslab.ece.ntua.gr

Burak Ocalan
University of Illinois
Urbana-Champaign
Champaign, IL, USA
bocalan2@illinois.edu

Shashwat Jaiswal
University of Illinois
Urbana-Champaign
Champaign, IL, USA
sj74@illinois.edu

Tianyin Xu
University of Illinois
Urbana-Champaign
Champaign, IL, USA
tyxu@illinois.edu

Josep Torrellas
University of Illinois
Urbana-Champaign
Champaign, IL, USA
torrella@illinois.edu

## Abstract

The shared and distributed memory capabilities of the emerging Compute Express Link (CXL) interconnect urge us to rethink the traditional interfaces of system software. In this paper, we explore one such interface: *remote fork* using CXL-attached shared memory for cluster-wide process cloning. We present *CXLfork*, a remote fork interface that realizes close to zero-serialization, zero-copy process cloning across nodes over CXL fabrics. CXLfork utilizes globally-shared CXL memory for cluster-wide deduplication of process states. It also enables fine-grained control of state tiering between local and CXL memory. We use CXLfork to develop *CXLporter*, an efficient horizontal autoscaler for serverless functions deployed on CXL fabrics. CXLfork minimizes cold-start overhead without sacrificing local memory. CXLfork attains restore latency close to that of a local fork, outperforming state-of-practice by 2.26x on average, and reducing local memory consumption by 87% on average.

***CCS Concepts:*** • **Computer systems organization** → **Cloud computing**; • **Software and its engineering** → **Checkpoint / restart**; **Distributed memory**.

***Keywords:*** CXL, Process forking, Remote memory, Checkpoint restore, Serverless computing, Cold start

**ACM Reference Format:**
Chloe Alverti, Stratos Psomadakis, Burak Ocalan, Shashwat Jaiswal, Tianyin Xu, and Josep Torrellas. 2025. CXLfork: Fast Remote Fork over CXL Fabrics. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3676641.3715988



*ASPLOS '25, Rotterdam, Netherlands*

ACM ISBN 979-8-4007-1079-7/25/03
https://doi.org/10.1145/3676641.3715988

## 1 Introduction

The emerging Compute Express Link (CXL) interconnect [7, 21, 68] provides a byte-addressable interface for low-latency access to remote memory. The latest versions of the standard, i.e., CXL 3.0 and beyond, also enable rack-scale, cache-coherent memory sharing across compute nodes [26]. These new capabilities urge us to rethink the traditional system software interfaces of distributed systems (e.g., [1, 44, 62]). In particular, byte-addressable shared memory can potentially speed-up cluster-wide process cloning with a *remote fork* interface—a key building block for high-performance, cloud-native serverless computing.

Unfortunately, existing remote fork mechanisms, namely CRIU (Checkpoint and Restore in Userspace) [25, 54] and Mitosis [75], are fundamentally limited to effectively harness the benefits of shared CXL memory, as they are designed for disconnected memory. CRIU serializes process state to files, including the entire process memory footprint, as well as the operating system (OS)-maintained process state. It then transfers and deserializes this checkpointed state on the remote node that clones the process. Mitosis implements a remote fork primitive that targets RDMA-capable interconnects (e.g., Infiniband [18]). It avoids the cost of (de)serializing process memory by using RDMA to lazily copy from the parent node the pages that the cloned process accesses. However, its data copies incur significant overhead (§2.3). In addition, Mitosis still needs to serialize and copy the OS-managed state, which induces non-negligible overhead.

To utilize the capabilities of CXL memory, we introduce *CXLfork*, a new remote fork interface. Our insight is that checkpointing and restoring process state from CXL shared memory mostly obviates the need for state serialization and state transfer. Hence, we can realize a close to *zero-copy, zero-serialization remote fork primitive* that speeds-up process cloning across nodes and enables cluster-wide memory deduplication via state sharing. This can significantly benefit the performance of modern applications, especially in cloud-native, serverless paradigms [5, 8–11, 59]. Specifically, CXLfork's fast cluster-wide process cloning alleviates the

overhead of cold starts of serverless functions [61, 72] and relaxes the need to keep idle containers warm in memory for a long time [2, 22, 50, 78] across the cluster. Combined with the rack-scale deduplication of function footprints, CXLfork eases the memory pressure in serverless systems, increasing system throughput and improving their responsiveness to load spikes.

The design of CXLfork faces two key challenges: (1) how to checkpoint the process state to CXL memory without resorting to serialization, and (2) how to efficiently share the checkpointed state between concurrent cloned processes across the cluster without impacting performance.

To address the first challenge, CXLfork checkpoints process data and most of the OS-maintained process state (e.g., page tables) to CXL memory as-is, using memory copies. Note that the checkpointed OS structures need to be decoupled from the OS instance that created the checkpoint, in order to allow other nodes to concurrently use them. To that end, CXLfork traverses the checkpointed OS structures, after copying them to CXL memory, and *rebases* them on the CXL physical memory address space. Pointers in these structures are modified to index CXL memory, so that different OS instances can remap and dereference them.

For the second challenge, CXLfork, by default, directly maps the CXL-checkpointed process state to the cloned processes on the remote nodes, without *copying* it to local memory. It then employs Copy-on-Write at runtime to handle modifications, while the read-only state remains in CXL memory, shared among all cloned processes in the cluster. While this enables almost constant-time process cloning, and effectively deduplicates memory over the CXL fabric, accessing state stored in CXL memory incurs a latency overhead of hundreds of nanoseconds [40, 43, 68]. Hence, retaining frequently-accessed data on the CXL memory might affect performance, especially for applications with large working sets. To reduce the overhead, CXLfork allows fine-grained tiering control of the checkpointed state with tailored placement policies.

We integrate CXLfork with Docker and an OpenWhisk-based serverless platform, and design *CXLporter*, a horizontal autoscaler designed for serverless functions deployed on CXL-interconnected clusters. We maintain a pool of empty (ghost) containers that consume neither CPU nor memory resources. CXLporter uses CXLfork to clone serverless functions into these empty containers on demand [75].

We evaluate CXLfork and CXLporter on an Intel Sapphire Rapids platform with an Intel Agilex FPGA-based CXL memory device. Our results show that CXLfork improves remote fork performance over state-of-practice and state-of-the-art by 2.26x and 1.40x, respectively, on average, closely matching local fork performance, while reducing memory consumption by 87% and 61%, respectively, on average. CXLporter leverages CXLfork's capabilities to reduce tail latency and increase resource utilization.

This paper makes the following contributions:

- We design CXLfork, the first remote fork interface for shared CXL memory to realize close to *zero-serialization, zero-copy, cluster-wide process cloning*.
- We leverage CXLfork's fine-grained control over the checkpointed state to devise tailored tiering policies that can balance state deduplication and memory savings with runtime performance.
- We develop CXLporter, a horizontal autoscaler for serverless workloads that uses CXLfork to unlock increased concurrency and memory efficiency for serverless systems.
- We evaluate CXLfork and CXLporter with various serverless workloads on an experimental CXL platform.

## 2 Background and Motivation

### 2.1 Compute Express Link (CXL)

The Compute Express Link (CXL) [7, 21, 68] is an emerging cache-coherent interconnect based on PCI Express (PCIe) [63]. Its latest revisions, i.e., CXL 3.0 and beyond, incorporate support for coherent access to shared remote memory at the cache-line granularity by multiple nodes [21, 26]. This ability of CXL to support cluster-wide, direct, shared memory access opens-up the possibility to rethink system interfaces for distributed computing. For example, recent works [45, 46, 73] leverage it to realize pass-by-reference Remote Procedure Calls (RPCs) to minimize data movement. In this work, we examine its applicability for fast inter-node process cloning.

### 2.2 Process Cloning for Serverless Workloads

Fast cluster-wide process cloning is especially important in Function-as-a-Service (FaaS) (i.e., serverless) environments [5, 10, 11, 59], where function instances are frequently spawned and discarded. Spawning a new function instance has a costly cold-start overhead due to the initialization of the function's state. The standard mitigation technique is to keep inactive function instances alive (cached in memory) for a fixed keep-alive time window [2, 22, 50, 78], to avoid future cold starts—at the expense of idling resources and pressing memory capacity. Within the limits of a single node, *fork* semantics have been proposed as a mitigation technique. They provide fast instantiation and seamless resource sharing [2, 67] across sibling instances. Unfortunately, load spikes commonly necessitate the creation of multiple function instances across cluster nodes, which is a costly process.

With the capabilities provided by CXL, however, inter-node (i.e., remote) fork semantics can potentially offer the same fast instantiation and memory sharing benefits as a local fork, but across nodes. To understand the potential of this approach, we have analyzed the access pattern of common FaaS workloads (§6). We spawn a function, invoke it 128 times with a different input in each request, and examine the footprint of each invocation. The results are shown in Figure 1. The figure breaks down each function's footprint

into: a) data that are used for function initialization and are rarely accessed during function execution (*Init*) [78], b) data that are only read during function execution (*Read-only*), and c) data that are written and possibly read during function execution (*Read/Write*). On average, these categories account for 72.2%, 23%, and 4.8% of the footprint, respectively.

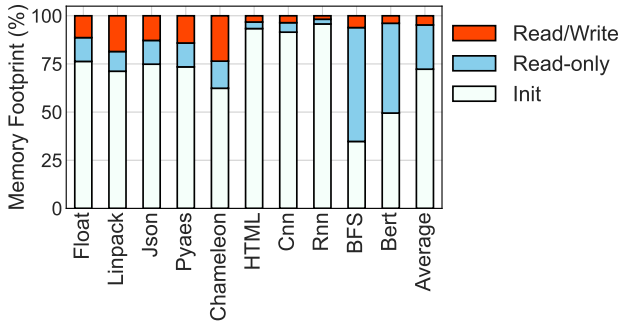



**Figure 1.** Breakdown of the memory footprint of different FaaS functions. Init and Read-only data dominate.

These results imply that inter-node (remote) forks can substantially benefit from storing most of the *Init* and *Read-only* state in the CXL shared memory. Sibling function instances running in different nodes can share such state. This facilitates cluster-wide memory deduplication, potentially increasing the number of function instances that can run on a fixed local memory budget. Since the working set of serverless functions is typically small [66, 68], the local hardware caches of the compute nodes may be able to intercept most of the requests to such data, amortizing the increased latency of CXL accesses. The high-level design of the envisioned approach is shown in Figure 2.

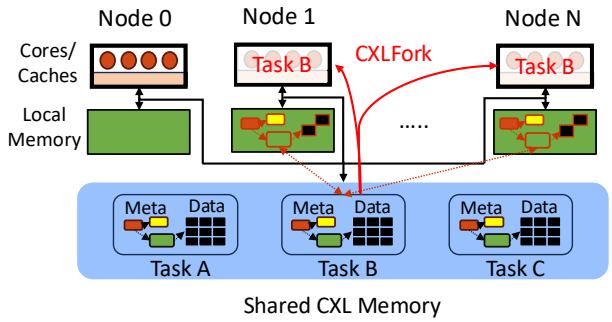



**Figure 2.** Envisioned system that enables fast remote process cloning and data sharing over CXL memory.

### 2.3 Existing Remote Fork Designs

There are two existing mechanisms that can fork a process to a remote node in a cluster: CRIU (Checkpoint and Restore In Userspace) [25, 54] and Mitosis [75]. In this section, we describe how they could be adapted to use CXL shared memory, and evaluate their performance using an experimental setup of two virtual machines that access an FPGA-based CXL memory prototype (described in §6).

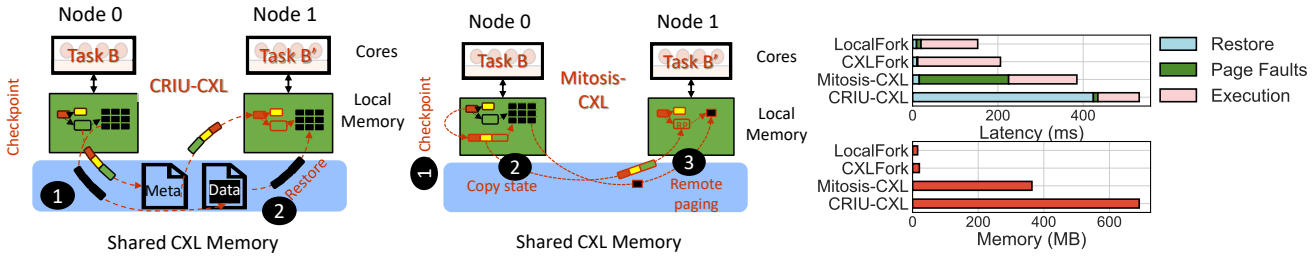
**2.3.1 Checkpoint and Restore in Userspace (CRIU).** CRIU is the state-of-practice framework for transferring process state across nodes. CRIU does not directly leverage the network fabric in a cluster. Instead, its remote fork copies a process' state from one node to another indirectly via storage. It uses Protocol Buffers [56] to serialize the OS state of a running process (e.g., its virtual memory areas, page tables, open files, namespaces, and CPU registers) and the process' memory pages to files. With CXL, we can cache these files in shared CXL memory (Figure 3a). This is the *checkpoint* phase (❶ in Figure 3a). In the *restore* phase ❷, these files are deserialized to the target node and used by CRIU to create a cloned child, restoring the entire checkpointed state and resuming execution.

Our evaluation reinforces recent studies [75] and shows that CRIU is inefficient for the FaaS use case even if we port it to CXL. Figure 3c shows the CRIU latency and memory overhead when it forks a BERT [58, 78] function instance to a new node and runs an inference task. The latency plot assumes that the checkpoint files are already in the CXL memory. We observe that just the restore phase of CRIU takes 2.7x longer than forking a BERT instance using local fork and its execution. Moreover CRIU consumes 42x more local memory than local fork, since parent and child processes in different nodes share no state.

**2.3.2 Mitosis.** Mitosis [75] is the state-of-the-art framework to fork processes to remote nodes. It utilizes the RDMA capabilities of network fabrics (e.g., Infiniband) to accelerate remote fork. Mitosis creates a shadow immutable copy of the parent process in the memory of the same node, while serializing the OS state. This is the checkpoint phase. Then, it transfers the serialized OS state to the remote node using one-sided RDMA operations, and deserializes it to create a new process. This is the restore phase. By default, the forked process is resumed without copying the parent's memory pages. As the forked process executes, it triggers special page faults that copy such pages from the parent node lazily, with remote paging [44].

To implement Mitosis with CXL support, we replace the RDMA operations with page copies over the shared CXL memory. The checkpoint phase (❶ in Figure 3b) creates the checkpoint locally. The restore phase ❷ transfers the OS state over CXL memory, and the remote page faults of the child process are served with copies over CXL ❸ (instead of one-sided RDMA). Since there is no way currently to serve inter-node faults with CXL, we test Mitosis-CXL within the same node. Note that CXL 3.0 will introduce such support with Global Integrated Memory (GIM) [7], so Mitosis could then be properly evaluated.

Despite the CXL-provided speedup, our evaluation shows that Mitosis also has substantial inefficiencies. As shown in




**(a)** CRIU over CXL **(b)** Mitosis over CXL **(c)** Fork-and-execute overhead for a BERT function

**Figure 3.** State-of-practice (CRIU) (a) and state-of-the-art (Mitosis) (b) remote fork mechanisms, and their latency and memory overhead (c). The latency figure assumes that the state is already checkpointed.

Figure 3c, Mitosis' latency to fork and execute the BERT function is high. Although Mitosis performs better than CRIU, the total execution time is still 2.6x longer than forking an instance with local fork and executing it. The dominant overhead stems from copying process memory during page faults. For the BERT function, the restore overhead of transferring and de-serializing the OS state is not high. However, for other functions, such overhead can be substantial (§7). Finally, the local memory consumption increases by 24x over a local fork. While copying all accessed data locally is reasonable for a remote fork based on RDMA with expensive remote memory accesses, it misses the benefits of low-latency shared direct access to CXL memory.

*Existing remote fork mechanisms introduce substantial overheads because they are inherently designed for disconnected memory and thus need to (de)serialize and transfer OS state and process data across nodes.*

## 3 Designing Remote Fork for CXL Fabrics

We design *CXLfork*, a remote fork interface that enables fast cluster-wide process cloning and effective memory state deduplication over shared CXL memory. CXLfork leverages the global shared memory capabilities of CXL to:

1. Minimize the software overheads of process state serialization and transfer between nodes, providing close to *zero-serialization*, *zero-copy* remote fork.
2. Enable the seamless, efficient, and controlled cluster-wide sharing of read-only state between remote sibling processes via an *enriched* remote fork interface.

### 3.1 Challenges and Opportunities

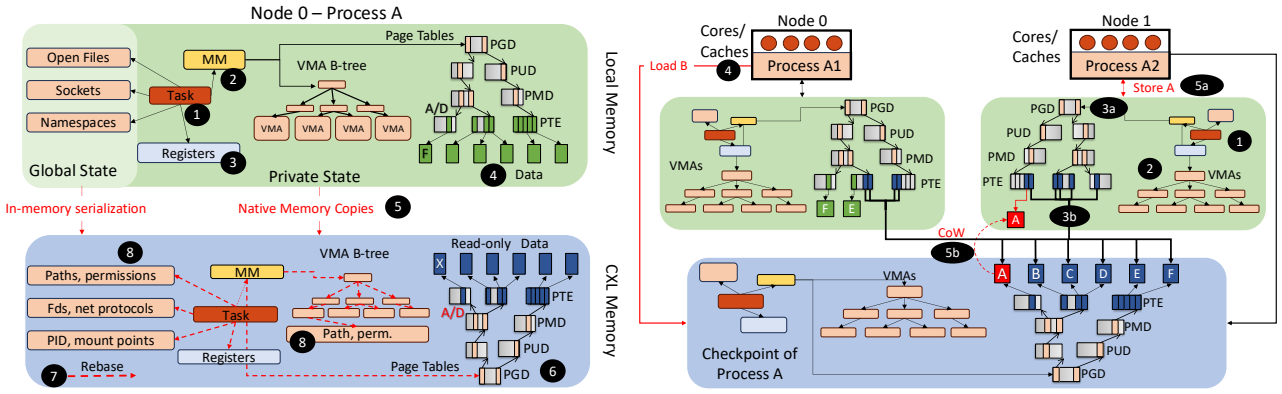
CXLfork adheres to the standard checkpoint-and-restore interface of remote fork as in prior art [54, 75]. To achieve the above goals, CXLfork addresses three main challenges.

**How to Store the Checkpointed State?** As discussed in §2, existing remote fork mechanisms either checkpoint process state to files [54] or keep it in the memory of the OS instance that initiates the checkpoint [75]. The first approach (CRIU) decouples the checkpointed state from the OS instance that created it—i.e., all references (pointers or dependencies) to the checkpoint-initiating OS have been stripped and the checkpoint files can be stored on, transferred to, and used by any node in the cluster. However, process restoration suffers from significant deserialization overhead.

The second approach (Mitosis) skips serialization for process data, but couples the checkpointed state with the node and process that created it. One implication of this design approach is that the parent process cannot exit until all its descendants (i.e., remote children) have terminated, which complicates process lifecycle management [75]. Thus, the node where the parent process and the checkpoint reside, acts as a point of failure [51, 80, 83]. Such a design also limits the potential to exploit shared-memory fabrics, as the state needs to be copied from the parent node to different children nodes. The parent node acts as a point of congestion and its network uplink is a potential bottleneck. Overall, retaining non-serialized process data in the checkpoint initiator trades-off lower checkpoint latency for increased overhead and more constraints in the restoration path. However, remote fork performance is typically biased toward restores, as we see a checkpoint-once-restore-multiple pattern—especially in a serverless environment.

In contrast, CXLfork leverages direct, low-latency access to shared memory enabled by CXL to achieve the best of both worlds. Process state is decoupled from the OS instance that initiated the remote fork and is placed as-is (i.e., mostly avoiding serialization) on the shared CXL medium. Other nodes can concurrently access the checkpointed state from there, without the need to fetch it locally (i.e., avoiding the bulk of deserialization and copying).

To efficiently decouple process state and make it available for concurrent use by other OS instances, CXLfork distinguishes between *private* and *global* process state. Private state is self-contained—it does not link to OS structures outside the process. For private state, CXLfork needs to manage its internal pointers, so that they point to and index the physical address space in CXL memory where the checkpointed state now resides. On the other hand, global state requires explicit unplugging (detach) and plugging (attach) for checkpoint and restore respectively, to keep the OS metadata consistent.




**(a)** CXLfork checkpoint **(b)** CXLfork restore

**Figure 4.** Workflow of CXLfork operations: (a) checkpoint and (b) restore.

**How to Efficiently Share the Checkpointed State?** To realize close to zero-copy restore, CXLfork retains both metadata and data for the cloned process on CXL memory. CXLfork thus skips the overheads of reconstructing OS state, such as process page tables and virtual memory layout, during restore. The checkpointed process memory also resides in CXL and can be readily attached to the cloned process. CXLfork employs a Copy-On-Write (CoW) approach to handle writes via faults that fetch pages from CXL memory to local memory (*migrate-on-write*) and ensure checkpointed (meta)data immutability. Using CoW instead of fetching pages to local memory on loads via faults [75] a) minimizes restore latency by avoiding copies, and b) enables the seamless sharing of the read-only state, which remains stored in CXL and is automatically and dynamically cached in the hardware caches of the different nodes in the cluster.

**How to Tier Checkpointed State?** In some cases, relying only on the *migrate-on-write* strategy to bring pages to the node running the cloned process may not be optimal. There is a trade-off between data tiering, data deduplication, and remote fork performance. The challenge derives from the fact that the shared CXL memory tier is slower than local memory. If the working set of the cloned process is not captured by the hardware caches in the node where the process is running, load accesses may miss in the caches and access data from the CXL tier, slowing down execution.

To handle this case, we design mechanisms in CXLfork to track the working set of cloned processes. Further, we enrich the CXLfork interface to allow fine-grained control of checkpointed state tiering between local memory and CXL shared memory. Such an interface enables dynamically trading off performance for local memory savings under different conditions of access locality and memory pressure.

# 4 CXLfork Design and Implementation

We now present the design and implementation of CXLfork. CXLfork targets clusters of nodes interconnected via CXL fabrics where, like in related work [54, 75], the nodes run standalone instances of the same OS image and use a shared (distributed) file system.

## 4.1 CXLfork Checkpoint

Figure 4a shows the checkpoint workflow. For optimal operation, CXLfork distinguishes between *private* and *global* process state, and checkpoints each differently.

**Private State** is all the data and metadata uniquely owned by a process, which can therefore be decoupled from the rest of the OS and checkpointed. It includes: the process Task structure (❶ in Figure 4a); the memory descriptor (MM) ❷, which comprises the Virtual Memory Area (VMA) tree and the page tables that hold the process address space; the CPU register contents ❸; and the physical pages that comprise the process private memory and the private file mappings (e.g., libraries) ❹. All these OS data structures and the process data pages are checkpointed to CXL memory ❺ using native memory copies. Unlike prior art [54], CXLfork also checkpoints private clean file-backed memory pages, i.e., for files that are privately mapped by the process such as libraries, trading off checkpoint size for restore performance—since faulting in file pages on a remote node on restore is expensive.

The checkpointed page table tree in the CXL memory is not a simple copy of the original; it has to be modified, since CXLfork uses it to index checkpointed data without the need for the metadata serialization used in prior art [54, 75]. CXLfork updates the checkpointed page table entries (PTEs) to map the new CXL physical locations where the checkpointed pages reside, and marks them as read-only. For example, the virtual page that was mapped to physical page $F$ on node 0 in Figure 4a is now mapped via the checkpointed page tables to CXL page $X$, which stores $F$'s replica. CXLfork also preserves the Access (A) and Dirty (D) bits of the original tables, which encode the access pattern of the checkpointed process. The design choice to checkpoint the page table tree in CXL also accelerates CXLfork restore as discussed in § 4.2.

Once the checkpoint has been copied to CXL memory, CXLfork rebases ❼ the internal pointers of the checkpointed OS structures into the corresponding (machine-independent) offsets on the CXL device [80, 83]. This is done to allow other OS instances to remap and dereference them.

**Global State** is the OS state shared by multiple processes running on the same node, e.g., open files, sockets, and namespaces. It typically contains pointers to global OS data structures like the filesystem layer (e.g., inodes), which cannot be checkpointed; they are neither standalone nor portable.

CXLfork serializes into CXL memory the information necessary to re-instantiate global state on a remote node. During restore, it de-serializes it and redoes operations to restore global system state (§ 4.2). CXLfork could fall back to serializing the global state to storage like CRIU [54]. However, this would induce non negligible de-serialization overheads on restore. Figure 4a ❽ showcases how open or memory-mapped files (e.g., libraries) are handled. CXLfork serializes their paths and permissions in the checkpointed data structures. Note that, similarly to existing remote fork interfaces [54, 75], CXLfork assumes that the root file system is identical across nodes (e.g., as in the case of a container image). Hence the file paths are the same across nodes. Sockets and other file descriptors are handled similar to files. CXLfork does not currently support shared anonymous memory mappings, i.e., memory shared between processes.

Some of the private and global state is *reconfigured* during the restore operation. Such state comprises metadata that, while shared between parent and child in a local fork, can change in the cloned process in a remote fork. For example, scheduling policies such as CPU or NUMA affinity, and namespace and cgroup configuration can be reset on the new node after the remote fork. From the reconfigurable state, CXLfork only serializes and checkpoints mount points and the process identifier (PID) namespaces. For the rest of the metadata, it supports restoring execution into new namespaces [53].

### 4.2 CXLfork Restore

CXLfork uses the checkpointed data and metadata on CXL memory to clone processes across cluster nodes. Figure 4b shows the restore workflow. On the right, it shows how CXLfork restores process *A*2 from the checkpoint of process *A* on CXL memory. On the left, it shows an already restored process *A*1 from the same checkpoint. To perform the restore, CXLfork first creates a new process on the new node that calls CXLfork-restore ❶. CXLfork-restore re-constructs the process virtual memory using the checkpointed metadata ❷ and maps the checkpointed physical memory to the new process' address space 3a,3b. Finally, it restores the global state and resumes execution using the checkpointed hardware context.

**Zero-Copy, Seamless State Sharing.** CXLfork *avoids copying any process data on restore*. Instead, it maps the checkpointed data, as read-only, into the new process address space. The simplest way to achieve this is by naively copying the checkpointed page table entries to local memory 3a, as they already store the CXL addresses of the checkpointed data and map them as read-only 3b (§ 4.1). However, we describe later in §4.2.1 how CXLfork can initialize the page table tree of the child more efficiently, avoiding most of these copies. The process then resumes execution instantly.

All the load instructions of the restored process that miss in the hardware caches of the CPU fetch directly the checkpointed data from CXL memory, bypassing the local memory of the node (e.g., load *B* in *A*1 ❹). Store instructions 5a trigger CoW faults 5b that copy the target page to local memory and perform the update locally. This approach ensures that the checkpoint in CXL remains pristine and readily reusable. This design also enables the seamless sharing of read-only data across nodes. Figure 4b shows how processes *A*1 and *A*2 directly map the same CXL-checkpointed pages *B*, *C*, and *D* while running on different nodes.

**Restoring Global and Reconfigurable State.** During restore, CXLfork redoes all the necessary operations to reconstruct global OS state, using the lightly serialized checkpointed state (§ 4.1) For example, it reopens all the file descriptors that the parent process held open (e.g., files and sockets) using checkpointed paths and permissions. Similarly, CXLfork restores private memory mappings (e.g., libraries) for files that are no longer open. CXLfork deserializes the checkpointed metadata (e.g., the file paths) stored on the CXL-checkpointed VMA tree (Figure 4a) and sets up the necessary data structures (e.g., *file struct*) while registering their callbacks to the local file system layer. Finally, CXLfork restores the mount points and PID namespaces of the child process using the CXL checkpoint. For the rest of the metadata, such as network and cgroup configurations, CXLfork uses a hybrid approach. It inherits them from the process that calls the CXLfork API on the new node, rather than from the checkpointed process [53]. This way, CXLfork can clone processes directly into new containers.

#### 4.2.1 Restore Efficiency.

A naïve implementation of restore in CXLfork may copy all the checkpointed private OS state (i.e., the memory descriptor, the virtual address space layout, and the page table tree) from CXL to local memory and re-construct all global OS state synchronously. However, we measured that copying and re-instantiating the OS state on the remote node on restore can take several milliseconds. This incurs a non-negligible start-up delay, especially for latency-sensitive serverless applications.

**Attaching OS State in Constant Time.** To reduce this overhead, CXLfork directly maps the checkpointed OS state instead of copying it to local memory. CXLfork allocates and

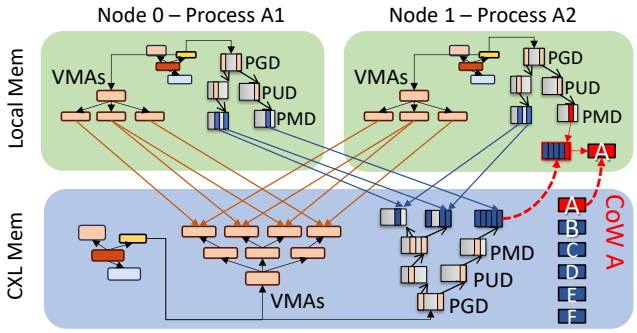



**Figure 5.** CXLfork restores OS state in constant time by allocating and initializing only the upper levels of the page table and VMA trees, and directly attaching the checkpointed leaves of the trees.

initializes only the upper levels of the page table tree of the new process in local memory. It then directly attaches the checkpointed leaves (i.e., the last-level PTEs) to the cloned process' upper page table levels (Figure 5). These PTEs already map the CXL physical addresses of the checkpointed data as read-only. This approach enables the re-instantiation of the cloned process page tables in almost constant time [3].

CXLfork prevents the OS from modifying the attached (checkpointed) CXL PTEs. This is done by using an unused bit in the PTE structure to track any OS attempt to update them. On such an event, it lazily copies the entire leaf to local memory—similar to CoW faults but for page table entries [81]. In this way, CXLfork shares fractions of page table trees across processes and nodes. For example, processes *A*1 and *A*2 in Figure 5 share the same page table leaves via CXL.

The same optimized procedure that CXLfork uses to restore page tables is also used to restore the virtual memory area (VMA) tree. CXLfork allocates and initializes only the upper levels of the VMA tree and attaches the checkpointed leaves (Figure 5). We observe that address spaces are frequently populated by numerous VMAs that privately map libraries and runtime modules. In the case of serverless functions, their number grows to the order of hundreds, due to the many dependencies of popular FaaS languages such as Python, which makes the reconstruction of the VMA tree a costly operation. At the same time, most of these VMAs rarely change, in either size or access permissions. Thus, attaching their checkpointed version is sufficient to restore execution; in the rare case of an update, CXLfork copies the corresponding leaf to local memory lazily. This strategy also re-constructs the global state for these VMAs on demand, i.e., copying the VMA to local memory and registering call backs to the file system only during a fault. Note that CXLfork checkpoints/restores the clean pages for private file mappings to/from CXL memory. Therefore, faults for the VMAs of these mappings are not frequent (§ 4.1, § 4.2).

**Optimizing CXL Page Faults.** By directly attaching checkpointed PTEs, CXLfork eliminates all read faults, similarly to local fork. However, it still pays the penalty of CoW, which copies data from the CXL tier to local memory. In our system, described in §6, we measure that such faults cost 2.5 $\mu s$ on average, while a regular fault that allocates an anonymous page from local memory costs less than 1 $\mu s$. We also measure that, on a CXL CoW fault, $\approx$1.3 $\mu s$ are spent on data movement and $\approx$500 *ns* on maintaining TLB coherence. To reduce the TLB shootdown overhead, CXLfork *opportunistically prefetches* into local memory those pages that are marked as dirty in the checkpointed page tables. The intuition is that, when fork is used to clone processes, and when checkpoints are taken judiciously [34], the Access (*A*) and Dirty (*D*) bits reflect the memory access patterns of the process. For the serverless use case, we profile forked functions and, indeed, over 95% of the pages that were written by the parent are also written by its children.

### 4.3 CXLfork Tiering

CXLfork extends the fork semantics to expose the trade-off between data sharing, restore performance, and potential runtime penalty due to slow CXL memory. CXLfork supports three different policies to copy checkpointed read-only pages from the CXL tier to local memory: (1) migrate-on-write tiering (default), (2) migrate-on-access (no tiering), and (3) hybrid tiering. CXLfork relies on information saved in the checkpointed page tables to do this efficiently, and exposes these policies to user-space controllers.

**Migrate-on-Write** is the default policy that lazily copies pages from CXL to local memory on stores and opportunistically prefetches dirty pages (§4.2.1).

**Migrate-on-Access (no tiering)** copies pages to local memory on access, like in [75, 78]. The first access to a checkpointed page triggers a special CXL page fault that copies the page from CXL to local memory. When using this policy, CXLfork does not attach the CXL page table leaves to the cloned process page table tree during restore (§4.2). It does not populate the entries at all, letting accesses to trigger CXL faults that set the PTEs of the cloned process.

**Hybrid Tiering** relies on the *A* bits in the checkpointed page tables to decide which pages to fetch into local memory on access. As discussed in §4.1, CXLfork checkpoints the *A*/*D* status of the page table of the parent process. From that point on, no OS instance in the cluster changes the checkpointed bits, as we exclude CXL memory from OS memory reclamation (i.e., from LRU lists). As CXL memory is shared between nodes, individual OS instances should not reclaim pages without coordinating with other sharers.

In hybrid tiering, CXLfork uses the *A* bit to select how pages will be placed in different tiers. Specifically, a CXL page with a clear *A* bit is assumed not to be heavily-accessed and, therefore, on access, is not fetched to local memory. Conversely, an access to a page with the *A* bit set fetches the page to local memory. An alternative approach would be to

prefetch the pages with the *A* bit set synchronously during restore. However, we find that such a design, which trades-off remote fork tail latency for fewer CXL faults, generally delivers lower performance.

**Continuous Update of Access Patterns.** Hybrid tiering is effective only if the *A* bits of checkpointed page tables effectively capture the "hot" pages of the workload's footprint. Ideally, hot pages are the read-only pages that cause most of the cache misses and, hence, most of the accesses to the CXL tier; in practice, we have to settle for the read-only pages that are accessed the most. Once hot pages are identified, they should be fetched into local memory on access.

We can identify hot pages by relying on: (1) the hardware-driven update of the *A* bits in the checkpointed PTEs in CXL memory, and (2) their reset from user-space. Indeed, when a restored process that has attached the checkpointed page tables does access the checkpointed pages, its page-table walks will update the *A* bits on the CXL PTEs. Note that *D* bits are never updated, as these pages are attached as read-only. Furthermore, CXLfork allows user-space to reset the checkpointed *A* bits via a dedicated interface. In the next section, we show how a runtime system can use this ability to periodically clear the *A* bits in the checkpointed PTEs to continuously monitor working sets similarly to [31].

**User-Identified Hot Pages.** CXLfork also allows users to explicitly declare hot pages. Specifically, user-space profilers [39, 48, 76] identify hot pages and, through a dedicated interface, save this information in an unused PTE bit in the checkpointed CXL page tables. Such information can then be used to optimize future remote forks.

## 5 CXLporter: Exploiting CXLfork for FaaS

To exploit CXLfork in a FaaS environment, we have built *CXLporter*, a horizontal autoscaler for FaaS. CXLporter efficiently scales up and down the number of function instances running in a CXL-interconnected cluster using CXLfork. We integrate CXLporter with an OpenWhisk-based serverless runtime using Docker, similar to [78]. CXLporter performs the following operations: (1) takes appropriately-timed checkpoints of functions, (2) maintains an object store of checkpoints, (3) maintains a pool of ghost containers, (4) controls the CXLfork tiering policies, and (5) dynamically adjusts keep-alive windows. Next, we discuss each operation.

**Function Checkpointing.** Prior work [34, 64] shows that checkpoints must be taken only after functions have been invoked for at least a few times. This is because the languages commonly used for FaaS workloads use JIT compilers, which need a few invocations to optimize the code and reach a steady state. For this reason, CXLporter checkpoints functions after their 16th invocation. Further, as discussed in §4.3, checkpointing a function includes capturing its memory access patterns with the *A* and *D* bits of its page table entries, and saving them in the checkpointed page tables

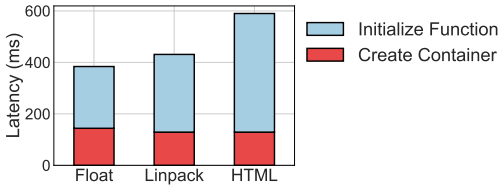



**Figure 6.** Latency of cold-starting a serverless function.

stored in CXL memory. We want these bits to capture the pattern in the steady state, not in the initialization phase. Hence, CXLporter also clears the A/D bits in the function's local page tables after the function's first invocation.

**Object Store of Checkpoints.** CXLporter maintains a distributed object store in the CXL fabric [80], that associates unique tuples of `<user,function>` with checkpoint identifiers (*CIDs*) of CXL-stored checkpoints, similarly to [34]. CXLporter uses this storage to (1) store new CIDs after checkpointing a function and (2) query CIDs before performing a CXLfork restore. CXLporter is also responsible for reclaiming checkpoints under CXL memory pressure.

**Ghost Container Pool.** We measure the overheads involved in cold-starting a function on a local node using Docker. Figure 6 shows the average overhead for several functions, broken down into State Initialization and Container Creation. The former includes initializing the runtime and the function's private data (e.g., weights of an ML model). This latency depends on the function and, in our experiments, is 250–500 *ms*. CXLporter minimizes this overhead plus any inter-node network latency with CXLfork-restore.

The second latency, container creation, is the overhead of setting up a new container to deploy a function instance [2, 47, 50]. It includes setting up the container network [50], namespaces, and cgroups. Similar to prior studies [47], we find that this overhead is ≈130ms. From the figure, we see that this overhead changes little across functions, irrespective of their image or footprint size. We also measure that a bare container with no deployed function consumes only 512KB of memory.

To eliminate the container creation overhead, CXLporter builds on the idea of Zygote template containers [50] and CXLfork's ability to restore function state, and introduces the notion of *Ghost Containers*. CXLporter provisions and caches a few configured but empty containers per function, which wait for "function restoration requests". Each of them occupies only 512KB of memory. When a new function instance is to be restored, CXLporter triggers the control socket of one of these containers to issue the restore request and clone the target function within the empty container by attaching its CXL checkpointed state.

**CXLfork Tiering Policies.** When the serverless runtime decides to deploy a function (e.g., during a spike in incoming load), CXLporter queries the checkpoint object store for an available checkpoint of the function. If a checkpoint is found,

CXLporter uses CXLfork-restore to clone the function. If no checkpoint is found for the requested function, CXLporter defaults to the regular cold-start mechanisms.

When restoring a function from a checkpoint, CXLporter controls how the checkpointed state is tiered between local and CXL memory. The decision is based on collected performance metrics from previous runs of the function, and on the available memory on the node. By default, CXLporter uses the migrate-on-write tiering policy (§4.3). This maximizes local memory savings and deduplication across nodes but can penalize function performance. Hence, CXLporter monitors the tail and average latency of function instances. If they are close to or over the user-defined Service-Level-Objectives (SLOs), it switches the function policy to hybrid-tiering. Hybrid tiering copies to local memory the pages that are estimated to be hot because their *A* bit is set in the checkpointed page tables. CXLporter also continuously monitors the local memory usage. If memory usage reaches a *High-Mem* threshold, no more functions are promoted to hybrid tiering. Moreover, CXLporter periodically resets the *A* bit on the checkpointed page tables to re-estimate hot pages (§ 4.3).

**Keep-Alive Windows.** Serverless runtimes keep idle functions cached in memory for a *keep-alive window* of several minutes [30] to minimize near-future cold-starts. CXLporter leverages the low cold-start latency of CXLfork and, when the memory pressure in the nodes increases, it dynamically shortens keep-alive windows to 10 seconds to reclaim memory faster. We consider studying different window sizes for different functions as future work.

Overall, CXLporter takes advantage of the low overhead of function cold-starts with CXLfork and the memory savings enabled by state deduplication in CXL memory to increase function density and attain higher throughput with the same memory budget.

# 6 Methodology

## 6.1 Experimental Setup

**CXL Hardware Setup.** We use a dual-socket 64-core Intel Sapphire Rapids server [13, 14, 20] as the host. Each socket has a 64MB L3 cache and a 128GB local DDR5 memory, and supports CXL v1.1-attached memory. We use an Intel Agilex 7 FPGA [15, 16] equipped with a 16GB DDR4 DIMM as the CXL memory device. We attach the CXL memory to the host as a CPU-less NUMA node [40]. Because we only have a single host, we set up a virtual machine in each of the two sockets to model a two-node distributed machine with shared CXL memory. We measure the round-trip latency from a core to the CXL memory to be 391ns on average [12].

**System Software.** We implement CXLfork on Linux v6.6. We use QEMU/KVM to spawn the two VMs on the host, and attach the CXL memory as a regular QEMU memory region.

| Function | Description | Footprint (MB) |
|---|---|---|
| Float | Sin, Cos, and Sqrt on floats | 24 |
| Linpack | Linear algebra solver for matrices | 33 |
| Json | JSON serialization & deserialization | 24 |
| Pyaes | Python AES encryption of a string | 24 |
| Chameleon | HTML table rendering | 27 |
| HTML | HTML web service | 256 |
| Cnn | JPEG classification CNN | 265 |
| Rnn | Generating natural language sentences | 190 |
| BFS | Breadth-first search | 125 |
| Bert | BERT-based ML inference | 630 |

**Table 1.** Serverless functions used in the evaluation.

**Simulation.** To assess how CXLfork would perform with CXL devices with a latency different from our system [17, 19], we integrate QEMU with the SST simulator [57, 74]. After we calibrate the simulator with our CXL hardware, we perform simulations with different access latencies to CXL memory.

**Workloads.** We use the serverless functions shown in Table 1. They are the CPU and memory functions from FunctionBench [32] and three real-world functions from [78] (HTML, BFS, and Bert). We invoke these functions according to real-world Azure serverless traces [61], to generate a realistic load for our system [28, 60].

## 6.2 Evaluation Scenarios

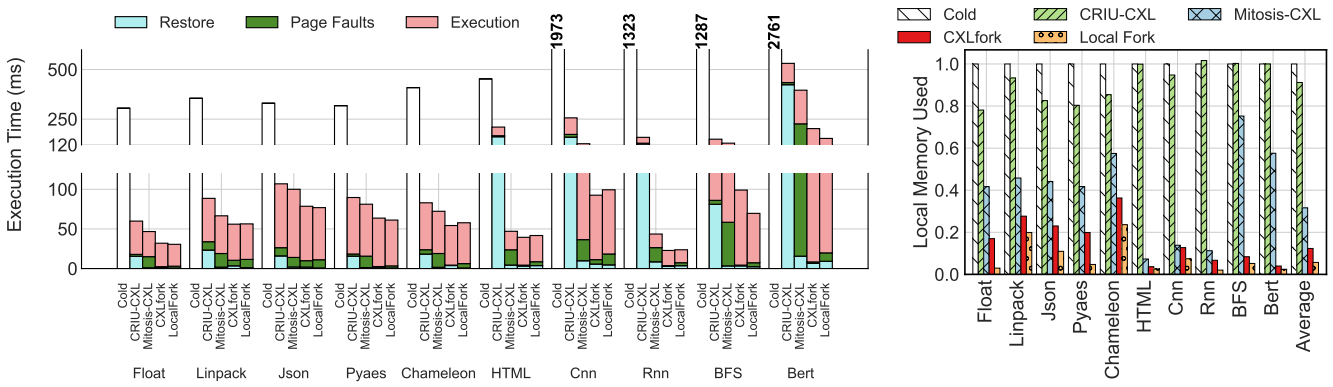
We evaluate two different scenarios.

**Performance of CXLfork.** We evaluate the cold-start execution of functions that are remote-forked to serve an incoming request on another node. To focus on the remote fork (rfork) overhead itself, we run the functions unsandboxed—i.e., without containers. We compare the checkpoint latency and the restore latency with CXLfork, CRIU-CXL, and Mitosis-CXL.

For CRIU-CXL, we create an in-CXL-memory filesystem which we share between the two VMs. The first VM serializes checkpoint files on the shared filesystem, which the second VM deserializes to clone a new function instance. This CRIU setup leverages CXL shared memory to avoid file copies, unlike prior studies [75].

For Mitosis-CXL, we face the challenge that there is no existing support for remote faults over CXL memory as in RDMA (§2.3.2). So, we port Mitosis [55] to our system and use only one of our two VMs. We mimic remote forking within the same VM, by replacing RDMA operations with memory copies over the CXL fabric. Each "remote" fault thus includes the latency to store and fetch data from CXL memory.

**Performance of CXLfork Bursts with CXLporter.** We implement variations of CXLporter that use CXLfork, CRIU-CXL, or Mitosis-CXL as rfork mechanisms. To generate a realistic load for our system, we invoke our serverless functions of Table 1 following Azure serverless traces [61]. For this set of experiments, we use rfork to spawn new function instances both across and within nodes. We use rfork-restore




**(a)** Execution time **(b)** Normalized local memory consumption

**Figure 7.** Remote fork performance under cold-start execution and normalized local memory consumption.

within a node, similar to [24, 34], to approximate an environment with invocation bursts in our limited two-node platform. This correctly estimates the overheads of remote forking except for the network overheads that would appear in a large distributed system. We use ghost containers to sandbox new function instances in both CXLfork and Mitosis-CXL. CRIU-CXL is not compatible with ghost containers, as it stores/deserializes checkpoints to/from a file system. We set the *HighMem* tiering threshold to 90% (§5).

**Metrics.** We use two metrics to evaluate CXLfork: (1) the average and tail latency of function execution and (2) the amount of local memory consumed. We report these metrics with both standalone CXLfork and CXLfork integrated in CXLporter.

## 7 Evaluation

### 7.1 Performance of CXLfork

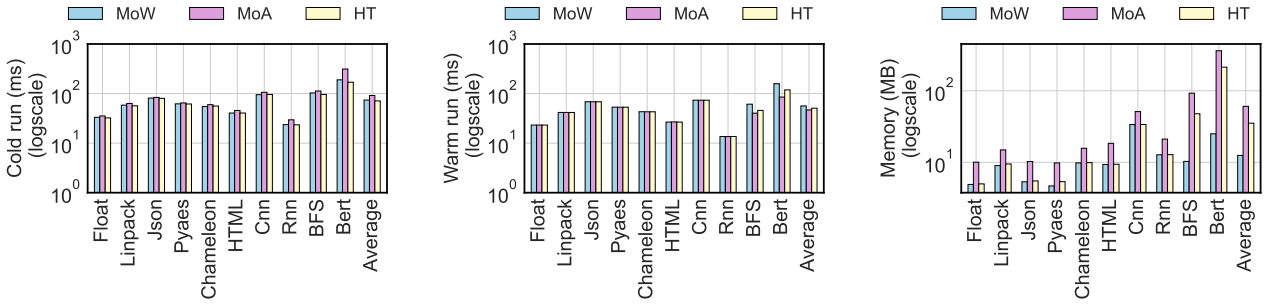
Figure 7a shows the end-to-end cold-start execution time of our functions under different rfork scenarios: *CRIU-CXL*, *Mitosis-CXL*, and *CXLfork*. The execution time is broken down into: the restore phase (*Restore*), page fault overhead (*Page Faults*), and the rest of function execution (*Execution*). For the page fault overhead, we include all types of faults (i.e., minor, major, and CoW); we profile them by instrumenting the kernel to capture their latencies. We do not include the checkpointing phase because it is typically performed only once, while restore is executed many times. For reference, we also show bars for vanilla cold execution (*Cold*) without a breakdown, and for forking a new instance within the same node (*LocalFork*).

**Restore Latency.** CRIU-CXL suffers from long restore latency, ranging between 16–423 *ms*, because it has to deserialize the checkpointed state and copy all data to local memory. Mitosis-CXL effectively reduces this latency, by avoiding most deserialization and performing no copies during the restore. However, it can still take up to 15 *ms* (Bert) and can account for up to 19% of the end-to-end latency (Rnn). The main overhead comes from transferring and de-serializing OS state, e.g., the page tables of the parent process. CXLfork restores a new function in 1.2-6.1 *ms*. CXLfork performs minimal deserialization (global state) and zero data copies. It also attaches, instead of re-constructing, checkpointed OS state—i.e., the checkpointed page table and VMA tree leaves.

*During the restore, CXLfork eliminates the deserialization and data copy overheads of other rfork designs.*

**End-to-end Latency.** *Cold* takes much longer to complete than any of the rfork scenarios, e.g., it is on average 11x slower than CXLfork. Among the rfork scenarios, *CRIU-CXL* is the slowest. On average, functions take 2.6x longer to complete than with *LocalFork*. This is primarily due to its long restore overhead. *Mitosis-CXL* performs better, as functions take on average 1.5x longer to complete than with *LocalFork*. It is particularly beneficial for functions with long initialization phases and small active working sets (e.g., Rnn), as it only copies accessed data. However, its benefits are limited for smaller functions (e.g., Float and Chameleon) due to page faults that copy mainly runtime pages and can impact cold-start performance. *Mitosis-CXL* benefits are also limited for large functions due to the costly page faults that copy process pages over CXL. For example, page faults cost 42% and 54% of BFS and Bert total execution, respectively.

*CXLfork* minimizes page fault overheads by (1) attaching checkpointed page tables for read-only data (eliminating faults) and (2) optimizing CoW faults (§4.2). The latter combined with the fact that *CXLfork* checkpoints and restores private file mappings (e.g., libraries), while *LocalFork* re-populates them lazily for the child, makes *CXLfork*'s page fault overheads (and potentially the whole execution time) occasionally lower than *LocalFork*'s. Overall, *CXLfork* is the fastest rfork scenario and is on average only 14% slower than *LocalFork*. It is 2.26x faster than CRIU-CXL and 1.40x faster than Mitosis-CXL on average. In fact it can be shown




**(a)** Cold execution time **(b)** Warm execution time **(c)** Local memory consumption

**Figure 8.** CXLfork tiering policies and their trade-offs between cold and warm execution time, and memory consumption.

that cold execution with CXLfork takes on average only 30% longer than a warmed-up (cached) instance execution.

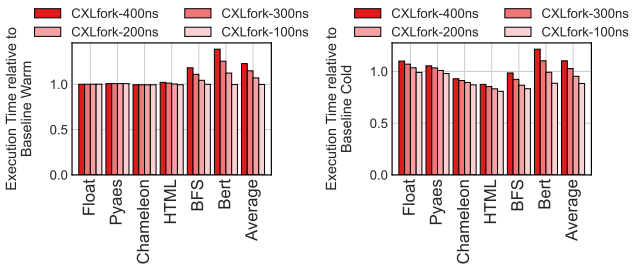
*Cold function execution with CXLfork takes on average only 14% longer than with local fork.*

**Local Memory Consumption.** Figure 7b shows the local memory consumed by a child function spawned with different rfork scenarios. The results are normalized to *Cold*. *CRIU-CXL* has a footprint similar to *Cold*, as it deserializes and copies all checkpointed state to local memory. It occasionally has a smaller footprint due to not checkpointing/restoring the clean pages of file mappings, but lazily faults on a subset of their pages during child execution. *Mitosis-CXL* uses 60% less memory than *CRIU-CXL*, on average, as it does not copy unnecessary data to local memory. However, memory consumption is high for functions with large working sets (e.g., Bert and BFS). *CXLfork* further reduces memory consumption by 87% over CRIU-CXL and by 61% over *Mitosis-CXL* on average, as it avoids copying read-only state to local memory, but shares it through CXL memory across function instances and across nodes.

*Execution with CXLfork requires only 13% of the local memory of a cold-started function on average, while speeding-up the cold-started execution by 11x on average.*

**Tiering.** Figure 8 compares our three CXLfork tiering policies: Migrate-on-Write (*MoW*), Migrate-on-Access (*MoA*), and Hybrid Tiering (*HT*). We measure the cold function execution time (Fig 8a), warm function execution time (Figure 8b), and local memory consumption (Figure 8c). Note that the Y axes are in log scale.

*MoW* copies data to local memory only on a write. It keeps cold-start overheads and local memory consumption modest, as it does not move read-only data; it shares it. However, it may penalize the latency of loads issued by the child process. This is because read-only data is fetched from the slower CXL tier. This effect is best seen in the warm execution time. While the majority of functions are not affected by this




**(a)** Warm **(b)** Cold

**Figure 9.** Sensitivity of (a) warm and (b) cold function execution with CXLfork to CXL latency via simulation.

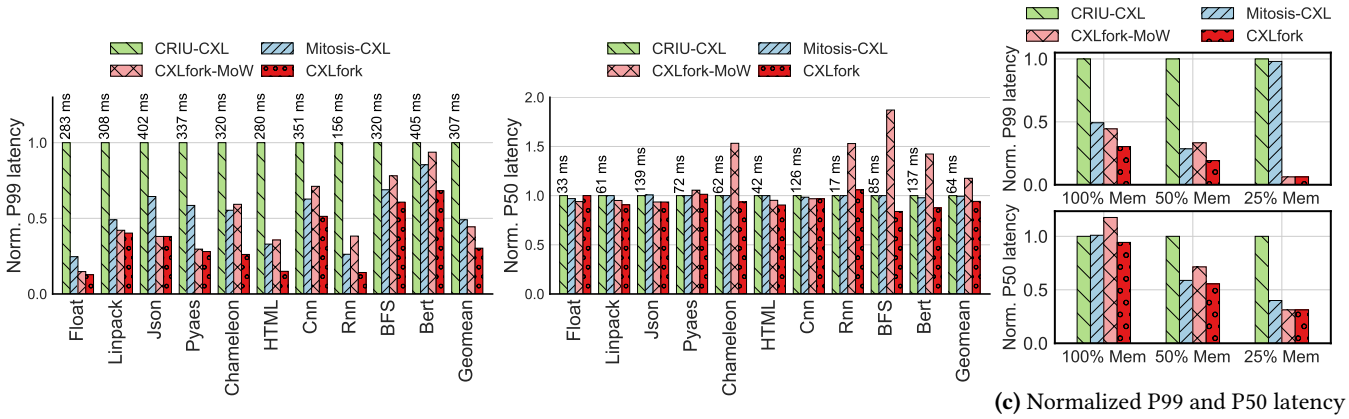
because their working sets fit in the CPU core caches [68], the performance of BFS and Bert is substantially hurt.

*MoA* copies pages on access. On average, it reduces warm execution time by 11% but penalizes cold execution time by 14% and, importantly, increases the child's memory footprint by 250%. Finally, *HT*, which copies pages based on their *A* bit, achieves a middle ground. It has comparable or better cold execution time than *MoW*, and intermediate warm execution time and memory footprint in BFS and Bert.

*CXLfork's tiering policies enable trade-offs between memory consumption, cold-start and warmed-up execution of functions with different footprints and access patterns.*

**Sensitivity to CXL Latency.** We use simulations (§6) to study the sensitivity of CXLfork performance to the CXL device latency. We first calibrate the simulator to match our real system. Then, we vary the round trip latency to the CXL memory from 400 ns (close to the 391 ns of our system) to 100 ns (close to the round trip to our local memory). For space reasons, we show only the most representative functions; we exclude functions with identical behavior.

Figure 9a shows the warm execution time of functions with CXLfork relative to the warm execution time with local fork in an environment without CXL memory. We observe that lower CXL access latency improves performance for BFS and Bert and does not affect the rest—whose working sets fit in the caches. However, even when CXL latency is 200 ns (2x




**(a)** Normalized P99 latency (abundant memory) **(b)** Normalized P50 latency (abundant memory) **(c)** Normalized P99 and P50 latency (limited memory)

**Figure 10.** P99 and P50 latency of function execution under CXLporter with abundant and limited memory.

the latency of local memory), performance is still penalized. This underlines the need to manage tiering in combination to remote forking.

Figure 9b shows the cold execution time of functions with CXLfork relative to the cold execution time with local fork in an environment without CXL memory. As CXL latency drops, CXLfork performance improves. In many cases, CXLfork performs equally or better than the baseline, which uses a local fork. CXLfork performs better because it avoids the overheads of duplicating OS state (e.g., process page table and VMA tree leaves) and, instead, attaches these structures from the checkpoints. It also attaches checkpointed private file mappings, which the local fork faults in lazily.

**Checkpoint Performance.** It can be shown that Mitosis and CXLfork perform checkpointing one order of magnitude faster than CRIU because they avoid data serialization. In turn, Mitosis checkpoints 1.5x faster than CXLfork because CXLfork checkpoints the data in CXL memory, while Mitosis checkpoints it in local memory. However, since Mitosis keeps the data in the parent node, it cannot share it and has to copy it across nodes on every restore operation (§2). Note that the checkpoint phase, especially for FaaS, is off the critical path of the remote fork, as a function is checkpointed once but restored multiple times [75].

### 7.2 Performance of CXLporter

We evaluate CXLporter with different rfork designs to clone functions upon load spikes. We compare CRIU-CXL, Mitosis-CXL, CXLfork using the migrate-on-write tiering policy statically (*CXLfork-MoW*), and CXLfork where CXLporter dynamically adjusts the tiering policy based on past performance and memory pressure as described in §5 (*CXLfork*). We run experiments using the Azure traces of bursty functions under a total load of 150 Requests Per Second (RPS) on average [67].

**Results with abundance of memory.** Figures 10a and 10b show the P99 and P50 end-to-end function latency when nodes have ample memory to serve incoming requests. The bars are normalized to the latency with CRIU-CXL, whose absolute latency is shown on top of the bars. In these experiments, the benefit of rfork comes from mitigating cold starts when the runtime needs to create new function instances.

For P99, Mitosis-CXL and CXLfork reduce the latency over CRIU-CXL by an average of 51% and 70%, respectively. One reason why Mitosis-CXL and CXLfork have lower latency is the effect of ghost containers, which alleviate cold-start containerization overhead in these two designs. Another parameter that affects rfork performance is the effect of the bursts of requests, which feeds on itself. Specifically, the rfork designs that are intrinsically slower in cold starts (as shown in Figure 7a), take longer time to spawn a new function instance and, during that time, more function requests arrive. Such new requests end up being processed as *cold*-start functions, while they could have been processed as *warm*-start functions with faster rforks. This effect amplifies P99 latency difference between CRIU-CXL, Mitosis-CXL, and CXLfork. For P50, CRIU-CXL, Mitosis-CXL, and CXLfork perform similarly, as P50 latency generally reflects warm execution time, where rfork overhead has less of an effect.

*CXLfork-MoW* has significantly longer latencies than CXLfork and sometimes even Mitosis-CXL for both P99 and P50. The reason is the higher access latency to read-only data residing in CXL. This fact has a stronger effect for P50. These results underline the importance of dynamic tier management when forking over CXL.

**Results on memory-constrained nodes.** We re-run the same experiments but reduce the memory of the VMs to 50% and 25% of the size in the previous experiment. Now, the runtime scheduler has to recycle containers to serve requests, so the performance of the different rforks is affected by their local memory consumption. Figure 10c shows the

normalized P99 (bottom) and P50 (top) end-to-end function latency across all functions with 100%, 50%, and 25% local memory. We see that, as the available memory decreases, the relative latency of CXLfork decreases and, therefore, CXLfork becomes more attractive. This is because, as shown in Figure 7b, CXLfork needs less local memory than Mitosis-CXL and much less than CRIU-CXL. As a result, more function instances can be executing with CXLfork.

In the 25% memory environment, CXLfork reduces the P99 latency by ≈16x over Mitosis-CXL and CRIU-CXL. It can be shown that CXLfork also achieves a ≈2x throughput increase over these two other environments. The impact is a bit lower for P50. Finally, we see that, for 25% memory, CXLfork and *CXLfork-MoW* have the same latency. The reason is that when memory is limited, CXLfork dynamically adjusts its tiering algorithm to be the same as *CXLfork-MoW*.

*CXLporter leverages CXLfork's fast remote fork to significantly reduce FaaS tail latency arising from cold start effects. It also leverages the memory deduplication enabled by CXLfork to increase the number of functions that can be concurrently alive in the cluster.*

## 8 Discussion

In this section, we provide a discussion regarding some implementation and performance implications of the design choices we made for CXLfork and CXLporter.

**Hardware Requirements of CXLfork.** CXLfork uses non-temporal stores to copy (meta)data to CXL memory during checkpointing, bypassing the CPU caches of the checkpoint-initiating node. This approach delivers high performance [68] and does not introduce any data coherence issues because the restored processes *only read* the checkpointed (meta)data—recall that when a restored process performs a write, CXLfork uses CoW. In the presence of hardware support for cache coherence in CXL, e.g., with devices implementing CXL 3.0 and beyond, the checkpoint-initiating node could use normal stores for the checkpoint, as the hardware would ensure cache coherence.

**CXLfork for write-heavy workloads.** CXLfork mainly targets serverless functions, which tend to be dominated by read-heavy access patterns (§ 2). CXLfork enables the sharing and deduplication of read-only state across sibling instances at the cluster-level. Nonetheless, even write-heavy workloads benefit from CXLfork's instant process cloning across nodes, ensuring high availability. However, in this case, CXLfork's memory savings are blunted, as eventually much of the workload's memory will be lazily copied to the local memory of the remote node via Copy-on-Write faults triggered as the workload modifies its footprint at runtime.

**CXLporter for FaaS Workflows.** CXLporter uses CXLfork to speed-up the cold-start performance of individual FaaS functions (§7). We expect similar benefits for complex FaaS applications with workflows that comprise multiple functions. Since the functions of each workflow are deployed independently, they benefit from on-demand rapid remote forking [2, 50, 67]. CXLporter can further leverage the CXL fabric to accelerate inter-function communication by minimizing data movement [35, 41]—e.g., by using CXL-tailored RPC schemes [46, 73, 80] or by extending CXLfork to provide shared-memory semantics over CXL for communication.

**Scalability to a high number of nodes.** Due to the state of CXL hardware prototypes, we cannot study CXLfork on a distributed system with many nodes (§7). In a large cluster, we anticipate that limited CXL bandwidth may be a bottleneck. In this case, our current CXLporter and CXLfork tiering policies may not be the most appropriate ones, as they are mainly driven by access latencies. We plan to extend our tiering policies to take CXL memory bandwidth into consideration [78].

## 9 Related Work

**Checkpoint-restore.** Checkpoint-restore techniques for both process migration and process cloning have been extensively studied [49]. Earlier studies either focus on distributed OSes employing distributed memory management and inter-process communication mechanisms [79], or resort to a file-based checkpoint-restore interface [42, 65] similar to CRIU [54]. Other works focus on single-node checkpoint-restore [6, 36, 42, 52] or VM cloning [37]. VAS-CRIU [71] adds multiple virtual address spaces (MVAS) [23] to Linux to replace file-based memory checkpointing. While this mitigates serialization costs, it is confined to a single node.

**Serverless Scaling.** Scaling serverless functions has been extensively studied both for container and VM sandboxing. Some works [2, 50, 67] employ local fork to accelerate intra-node containerized function scaling, while other works use CRIU [34] or focus on improving function scaling from VM-based snapshots [4, 22, 38, 64, 70]. For example, Catalyzer [22] checkpoints and restores virtualization-based sandboxes via storage, and (s)forks these v-sandboxes within a node. In contrast, CXLfork checkpoints native functions running inside OS-containers and exploits the shared CXL fabric to fork them to remote nodes, unlocking their fast inter-node scaling, while efficiently deduplicating their read-only state at the cluster-level.

TrEnv [24] is a recent proposal developed concurrently with CXLfork , that relies, partially, on checkpointing, restoring, and sharing function data over CXL to optimize FaaS scaling. It is a CRIU-based solution optimized for intra-node scaling that does not provide remote fork semantics. Instead, it requires an expensive pre-processing step before remote nodes can spawn functions that can access checkpointed data on CXL memory. Specifically, *for each function on each remote node*, it requires de-serializing CRIU metadata in order to

generate dedicated local OS data structures (i.e., memory templates) that functions will then attach and use to access the checkpointed data on CXL memory. In contrast, CXLfork enables the rapid cloning of functions on any remote node without requiring any pre-processing or idling local data structures. Our preliminary results show that, in the absence of pre-created memory templates, CXLfork remote-forks functions are 1.8x faster than TrEnv on average. Moreover, unlike TrEnv, CXLfork provides tiering policies to control CXL overhead, and further enables the sharing of OS state, such as page tables and VMA trees, across nodes over the CXL fabric.

**Memory Tiering.** Several works study the potential of CXL-attached memory for memory tiering [27, 31, 33, 39, 43, 48, 69, 76, 77, 82]. A recent work [78] shows that FaaS can benefit from memory tiering to significantly reduce local memory usage. It uses Linux's multi-generational LRU (MGLRU) [29] to identify idle pages of serverless functions and migrate them to far memory via RDMA. CXLfork's design automatically and transparently stores the idle memory of functions on the CXL tier, without any specialized detection mechanism. CXLfork also shares these data across nodes, deduplicating footprints at the cluster level, significantly reducing local memory consumption across all nodes. In contrast to [78], which needs to fetch pages locally via RDMA before being able to access them, the idle memory of the CXL tier can be directly and concurrently accessed from any node.

## 10 Conclusion

We design CXLfork, a CXL-tailored remote fork interface, that realizes close to zero-serialization, zero-copy cluster-wide process cloning and enables the controlled deduplication and tiering of checkpointed state over shared CXL memory. CXLfork outperforms both state-of-practice and state-of-the-art approaches, while reducing memory consumption. We use CXLfork to build CXLporter, a horizontal FaaS autoscaler that unlocks increased system throughput and memory efficiency by exploiting CXLfork's fast, memory-frugal remote forking.

## Acknowledgments

We thank the anonymous reviewers and the paper's shepherd, Pedro Fonseca, for their valuable comments. We also thank all the members of the i-acoma group at UIUC for their constant feedback. This work was supported in part by NSF under grants CNS 1956007 and CCF 2107470; by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA; and by the IBM-Illinois Discovery Accelerator Institute.

## References

[1] Marcos K Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2018. Remote regions: a simple abstraction for remote memory. In *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC)*. https://www.usenix.org/conference/atc18/presentation/aguilera

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

[3] Chloe Alverti, Vasileios Karakostas, Nikhita Kunati, Georgios Goumas, and Michael Swift. 2022. DaxVM: Stressing the Limits of Memory as a File Interface. In *Proceedings of the 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1109/MICRO56248.2022.00037

[4] Lixiang Ao, George Porter, and Geoffrey M Voelker. 2022. FaaSnap: FaaS Made Fast Using Snapshot-based VMs. In *Proceedings of the 17th European Conference on Computer Systems (EuroSys)*. https://doi.org/10.1145/3492321.3524270

[5] Microsoft Azure. 2025. Azure functions. https://azure.microsoft.com/en-us/products/functions

[6] Edouard Bugnion, Vitaly Chipounov, and George Candea. 2013. Lightweight Snapshots and System-level Backtracking. In *Proceedings of the 14th Workshop on Hot Topics in Operating Systems (HotOS)*. https://doi.org/10.5555/2490483.2490506

[7] Compute Express Link Consortium. 2024. CXL Specification. https://computeexpresslink.org/cxl-specification/

[8] Alibaba Corporation. 2025. Alibaba Serverless Application Engine. https://www.aliyun.com/product/aliware/sae

[9] Cloudflare Corporation. 2025. Cloudflare Workers. https://workers.cloudflare.com/

[10] Google Corporation. 2025. Google Serverless Computing. https://cloud.google.com/serverless

[11] Huawei Corporation. 2025. Huawei Cloud Functions. https://developer.huawei.com/consumer/en/agconnect/cloud-function/

[12] Intel Corporation. 2021. Intel® Memory Latency Checker v3.11b. https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html

[13] Intel Corporation. 2022. Intel Xeon Gold 6430. https://ark.intel.com/content/www/us/en/ark/products/231737/intel-xeon-gold-6430-processor-60m-cache-2-10-ghz.html

[14] Intel Corporation. 2023. Intel Sapphire Rapids. https://www.intel.com/content/www/us/en/newsroom/news/4th-gen-xeon-scalable-processors-max-series-cpus-gpus.html

[15] Intel Corporation. 2024. Intel CXL FPGA IP. https://www.intel.com/content/www/us/en/products/details/fpga/intellectual-property/interface-protocols/cxl-ip.html

[16] Intel Corporation. 2025. Intel Agilex7 FPGA. https://www.intel.com/content/www/us/en/products/details/fpga/development-kits/agilex/agi027.html

[17] Micron Corporation. 2024. https://www.micron.com/products/memory/cxl-memory

[18] NVIDIA Corporation. 2024. Introduction to Infiniband. https://network.nvidia.com/pdf/whitepapers/IB_Intro_WP_190.pdf

[19] Samsung Corporation. 2023. https://semiconductor.samsung.com/news-events/news/samsung-develops-industrys-first-cxl-dram-supporting-cxl-2-0/

[20] Supermicro Corporation. 2024. Supermicro Hyper SuperServer SYS-221H-TNR. https://www.supermicro.com/en/products/system/hyper/2u/sys-221h-tnr

[21] Debendra Das Sharma, Robert Blankenship, and Daniel Berger. 2023. An Introduction to the Compute Express Link (CXL) Interconnect. *ACM Computing Surveys (CSUR)* (2023). https://doi.org/10.1145/3669900

[22] 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 *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/3373376.3378512

[23] Izzat El Hajj, Alexander Merritt, Gerd Zellweger, Dejan Milojicic, Reto Achermann, Paolo Faraboschi, Wen-mei Hwu, Timothy Roscoe, and Karsten Schwan. 2016. SpaceJMP: Programming with Multiple Virtual Address Spaces. In *Proceedings of the 21th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org10.1145/2872362.2872366

[24] 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, and YongWei Wu. 2024. TrEnv: Transparently Share Serverless Execution Environments Across Different Functions and Nodes. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24)*. https://doi.org/10.1145/3694715.3695967

[25] Docker Inc. 2024. https://docs.docker.com/reference/cli/docker/checkpoint/

[26] Sunita Jain, Nagaradhesh Yeleswarapu, Hasan Al Maruf, and Rita Gupta. 2024. Memory Sharing with CXL: Hardware and Software Design Approaches. *arXiv preprint arXiv:2404.03245* (2024). https://arxiv.org/abs/2404.03245

[27] Sepehr Jalalian, Shaurya Patel, Milad Rezaei Hajidehi, Margo Seltzer, and Alexandra Fedorova. 2024. ExtMem: Enabling Application-Aware Virtual Memory Management for Data-Intensive Applications. In *Proceedings of the 2024 USENIX Annual Technical Conference (USENIX ATC)*. https://www.usenix.org/conference/atc24/presentation/jalalian

[28] Christos Katsakioris, Chloe Alverti, Konstantinos Nikas, Dimitrios Siakavaras, Stratos Psomadakis, and Nectarios Koziris. 2024. FaaSRail: Employing Real Workloads to Generate Representative Load for Serverless Research. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. https://doi.org/10.1145/3625549.3658684

[29] Linux kernel documentation. 2023. Multi-Gen LRU. https://docs.kernel.org/admin-guide/mm/multigen_lru.html

[30] Dong Kyoung Kim and Hyun-Gul Roh. 2021. Scheduling Containers Rather Than Functions for Function-as-a-Service. In *Proceedings of the 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. https://doi.org/10.1109/CCGrid51090.2021.00056

[31] Honggyu Kim, Hyeongtak Ji, and Rakie Kim. 2024. DAMON CXL Tiering. https://lore.kernel.org/linux-mm/20240405101316.2890-1-honggyu.kim@sk.com/T/

[32] Jeongchul Kim and Kyungyong Lee. 2019. FunctionBench: A Suite of Workloads for Serverless Cloud Function Service. In *Proceedings of the 2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. https://doi.org/10.1109/CLOUD.2019.00091

[33] Kyungsan Kim, Hyunseok Kim, Jinin So, Wonjae Lee, Junhyuk Im, Sungjoo Park, Jeonghyeon Cho, and Hoyoung Song. 2023. SMT: Software-defined memory tiering for heterogeneous computing systems with CXL memory expander. *IEEE Micro* 43, 2 (2023). https://doi.org/10.1109/MM.2023.3240774

[34] Sumer Kohli, Shreyas Kharbanda, Rodrigo Bruno, Joao Carreira, and Pedro Fonseca. 2024. Pronghorn: Effective Checkpoint Orchestration for Serverless Hot-Starts. In *Proceedings of the 19th European Conference on Computer Systems (EuroSys)*. https://doi.org/10.1145/3627703.3629556

[35] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. 2021. Faastlane: Accelerating Function-as-a-Service Workflows. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. https://www.usenix.org/conference/atc21/presentation/kotni

[36] Oren Laadan and Jason Nieh. 2007. Transparent Checkpoint-Restart of Multiple Processes on Commodity Operating Systems. In *Proceedings of the USENIX 2007 Annual Technical Conference (USENIX ATC)*. https://www.usenix.net/legacy/events/usenix07/tech/full_papers/laadan/laadan.pdf

[37] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M Rumble, Eyal De Lara, Michael Brudno, and Mahadev Satyanarayanan. 2009. SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys)*. https://doi.org/10.1145/1519065.1519067

[38] Nikita Lazarev, Varun Gohil, James Tsai, Andy Anderson, Bhushan Chitlur, Zhiru Zhang, and Christina Delimitrou. 2024. Sabre: Hardware-Accelerated Snapshot Compression for Serverless MicroVMs. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. https://www.usenix.org/conference/osdi24/presentation/lazarev

[39] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. 2023. MEMTIS: Efficient Memory Tiering with Dynamic Page Classification and Page Size Determination. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*. https://doi.org/10.1145/3600006.3613167

[40] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/3575693.3578835

[41] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. 2022. FaaSFlow: enable efficient workflow execution for function-as-a-service. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. https://doi.org/10.5281/zenodo.5900766

[42] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. 1997. *Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System*. Technical Report. University of Wisconsin-Madison Department of Computer Sciences. https://minds.wisconsin.edu/bitstream/handle/1793/60116/TR1346.pdf

[43] Jinshu Liu, Hamid Hadian, Hanchen Xu, Daniel S Berger, and Huaicheng Li. 2024. Dissecting CXL Memory Performance at Scale: Analysis, Modeling, and Optimization. *arXiv preprint arXiv:2409.14317* (2024). https://arxiv.org/abs/2409.14317

[44] Fangming Lu, Xingda Wei, Zhuobin Huang, Rong Chen, Minyu Wu, and Haibo Chen. 2024. Serialization/Deserialization-free State Transfer in Serverless Workflows. In *Proceedings of the Nineteenth European Conference on Computer Systems*. https://doi.org/10.1145/3627703.3629568

[45] Teng Ma, Zheng Liu, Chengkun Wei, Jialiang Huang, Youwei Zhuo, Haoyu Li, Ning Zhang, Yijin Guan, Dimin Niu, Mingxing Zhang, and Tao Ma. 2024. HydraRPC: RPC in the CXL Era. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. https://www.usenix.org/conference/atc24/presentation/ma

[46] Suyash Mahar, Ehsan Hajyjasini, Seungjin Lee, Zifeng Zhang, Mingyao Shen, and Steven Swanson. 2024. Telepathic Datacenters: Fast RPCs using Shared CXL Memory. *arXiv preprint arXiv:2408.11325* (2024). https://arxiv.org/abs/2408.11325

[47] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*. https://doi.org/10.1145/3132747.3132763

[48] 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 *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/3582016.3582063

[49] Dejan S Milojičić, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. 2000. Process Migration. *ACM Computing Surveys (CSUR)* 32, 3 (2000). https://doi.org/10.1145/367701.367728

[50] 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 *Proceedings of the 2018 USENIX annual technical conference (USENIX ATC)*. https://www.usenix.org/conference/atc18/presentation/oakes

[51] Adarsh Patil, Vijay Nagarajan, Nikos Nikoleris, and Nicolai Oswald. 2023. Āpta: Fault-tolerant object-granular CXL disaggregated memory for accelerating FaaS. In *Proceedings of the 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '23)*. https://doi.org/10.1109/DSN58367.2023.00030

[52] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. 1995. Libckpt: Transparent Checkpointing under UNIX. In *Proceedings of the 1995 USENIX Technical Conference (USENIX)*. https://www.usenix.org/conference/usenix-1995-technical-conference/libckpt-transparent-checkpointing-under-unix

[53] CRIU project. 2019. CR in namespace. https://criu.org/CR_in_namespace

[54] CRIU Project. 2025. CRIU - Checkpoint and Restore in Userspace. https://github.com/checkpoint-restore/criu

[55] Mitosis project. 2023. MITOSIS: An OS primitive of fast remote fork. https://github.com/ProjectMitosisOS/mitosis-core

[56] Protobuf project. 2025. Protocol Buffers. https://protobuf.dev/

[57] SST project. 2025. The Structural Simulation Toolkit. https://sst-simulator.org/

[58] Amazon Web Services. 2021. Hosting Hugging Face models on AWS Lambda for serverless inference. https://aws.amazon.com/blogs/compute/hosting-hugging-face-models-on-aws-lambda/

[59] Amazon Web Services. 2025. AWS Lambda. https://aws.amazon.com/lambda

[60] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. 2019. Architectural Implications of Function-as-a-Service Computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1145/3352460.3358296

[61] 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 *Proceedings of 2020 USENIX Annual Technical Conference (USENIX ATC)*. https://www.usenix.org/conference/atc20/presentation/shahrad

[62] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. https://www.usenix.org/conference/osdi18/presentation/shan

[63] Debendra Das Sharma. 2020. PCI Express® 6.0 Specification at 64.0 GT/s with PAM-4 signaling: a Low Latency, High Bandwidth, High Reliability and Cost-Effective Interconnect. In *Proceedings of the 2020 IEEE Symposium on High-Performance Interconnects (HOTI)*. https://doi.org/10.1109/HOTI51249.2020.00016

[64] Wonseok Shin, Wook-Hee Kim, and Changwoo Min. 2022. Fireworks: A Fast, Efficient, and Safe Serverless Framework using VM-level post-JIT Snapshot. In *Proceedings of the 17th European Conference on Computer Systems (EuroSys)*. https://doi.org/10.1145/3492321.3519581

[65] Jonathan M Smith and John Ioannidis. 1987. *Implementing remote fork() with checkpoint/restart.* Department of Computer Science, Columbia University. https://www.cis.upenn.edu/~jms/TCOS.pdf

[66] Jovan Stojkovic, Esha Choukse, Enrique Saurez, Íñigo Goiri, and Josep Torrellas. 2024. Mosaic: Harnessing the Micro-Architectural Resources of Servers in Serverless Environments. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. https://doi.ieeecomputersociety.org/10.1109/MICRO61859.2024.00103

[67] Jovan Stojkovic, Tianyin Xu, Hubertus Franke, and Josep Torrellas. 2023. MXFaaS: Resource Sharing in Serverless Environments for Parallelism and Efficiency. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1145/3579371.3589069

[68] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. 2023. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1145/3613424.3614256

[69] Bijan Tabatabai, James Sorenson, and Michael M. Swift. 2024. FBMM: Making Memory Management Extensible With Filesystems. In *Proceedings of the 2024 USENIX Annual Technical Conference (USENIX ATC)*. https://www.usenix.org/conference/atc24/presentation/tabatabai

[70] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, Analysis, and Optimization of Serverless Function Snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/3445814.3446714

[71] Ranjan Sarpangala Venkatesh, Till Smejkal, Dejan S Milojicic, and Ada Gavrilovska. 2019. Fast In-Memory CRIU for Docker Containers. In *Proceedings of the International Symposium on Memory Systems (MEMSYS)*. https://doi.org/10.1145/3357526.3357542

[72] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *Proceedings of the 2018 USENIX annual technical conference (USENIX ATC)*. https://www.usenix.org/conference/atc18/presentation/wang-liang

[73] Stephanie Wang, Benjamin Hindman, and Ion Stoica. 2021. In Reference to RPC: It's Time to Add Distributed Memory. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*. https://doi.org/10.1145/3458336.3465302

[74] Ziqi Wang, Kaiyang Zhao, Pei Li, Andrew Jacob, Michael Kozuch, Todd Mowry, and Dimitrios Skarlatos. 2023. Memento: Architectural Support for Ephemeral Memory Management in Serverless Environments. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1145/3613424.3623795

[75] 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 *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. https://www.usenix.org/conference/osdi23/presentation/wei-rdma

[76] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. 2022. TMO: Transparent Memory Offloading in Datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/3503222.3507731

[77] Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan, and Ren Wang. 2024. Nomad: Non-Exclusive Memory Tiering via Transactional Page Migration. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

https://www.usenix.org/conference/osdi24/presentation/xiang

[78] Chuhao Xu, Yiyu Liu, Zijun Li, Quan Chen, Han Zhao, Deze Zeng, Qian Peng, Xueqi Wu, Haifeng Zhao, Senbo Fu, and Minyi Guo. 2024. FaaSMem: Improving Memory Efficiency of Serverless Computing with Memory Pool Architecture. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/3620666.3651355

[79] Roman Zajcew, Paul Roy, L David, Chris Peak, Paulo Guedes, Bradford Kemp, John LoVerso, Michael Leibensperger, Michael Barnett, Faramarz Rabii, Faramarz Rabii, and Durriya Netterwala. 1993. An OSF/1 UNIX for Massively Parallel Multicomputers. In *Proceedings of the 1993 USENIX Winter Conference (USENIX Winter)*. https://www.usenix.org/conference/usenix-winter-1993-conference/osf1-unix-massively-parallel-multicomputers

[80] 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 *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*. https://doi.org/10.1145/3600006.3613135

[81] Kaiyang Zhao, Sishuai Gong, and Pedro Fonseca. 2021. On-demand-fork: A Microsecond Fork for Memory-Intensive and Latency-Sensitive Applications. In *Proceedings of the 16th European Conference on Computer Systems (EuroSys)*. https://doi.org/10.1145/3447786.3456258

[82] Yuhong Zhong, Daniel S Berger, Carl Waldspurger, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D Hill, Mosharaf Chowdhury, and Asaf Cidon. 2024. Managing Memory Tiers with CXL in Virtualized Environments. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. https://www.usenix.org/conference/osdi24/presentation/zhong-yuhong

[83] Zhiting Zhu, Newton Ni, Yibo Huang, Yan Sun, Zhipeng Jia, Nam Sung Kim, and Emmett Witchel. 2024. Lupin: Tolerating Partial Failures in a CXL Pod. In *Proceedings of the 2nd Workshop on Disruptive Memory Systems (DIMES '24)*. https://doi.org/10.1145/3698783.3699377