

Accelerating Burst Parallelism of SigmaOS processes with CRIU

by

Frederick Tang

B.S. Computer Science and Engineering, MIT, 2025

Submitted to the MIT Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER
SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2025

© 2025 Frederick Tang. All rights reserved.

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable,
royalty-free license to exercise any and all rights under copyright, including to
reproduce, preserve, distribute and publicly display copies of the thesis, or
release the thesis under an open-access license.

Authored by: Frederick Tang
Department of Computer Science
August 29, 2025

Certified by: Ariel Szekely
Doctoral Candidate, Thesis Supervisor

Certified by: Frans Kaashoek
Charles Piper Professor of Electrical Engineering and Computer Science, Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Accelerating Burst Parallelism of SigmaOS processes with CRIU

by

Frederick Tang

Submitted to the MIT Department of Electrical Engineering and Computer Science
on August 29, 2025 in Partial Fulfillment of the Requirements for the Degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER
SCIENCE

ABSTRACT

σ OS is a multi-tenant cloud operating system designed to integrate the agility of serverless environments with the interactivity of microservices. A goal of achieving this integration is the ability to start new instances of server processes quickly. However, σ OS only handles σ container initialization, and does not assist with runtime and app initialization costs. One approach to overcome this challenge is to checkpoint processes using Checkpoint/Restore in Userspace (CRIU). CRIU is a linux toolset which can start new server instances by restoring them from a saved checkpointed state, avoiding the full cost of reinitialization and setup. This thesis introduces σ CRIU, which adapts CRIU for burst-parallel spawning of microservices in σ OS. σ CRIU implements a number of optimizations: compressing checkpointed proc metadata to reduce network communication costs, implementing demand-paging using a lazy page service, and caching kernel metadata to reduce CRIU's restore operation latency. These optimizations allow σ CRIU to start new microservices on remote machines quickly while still making use of CRIU's existing proven checkpoint and restore technology.

Thesis Supervisor: Ariel Szekely
Title: Doctoral Candidate

Thesis Supervisor: Frans Kaashoek
Title: Charles Piper Professor of Electrical Engineering and Computer Science

1 Introduction

1.1 σ OS Background

σ OS, built in the Go Programming Language, is a multi-tenant cloud operating system that integrates the strengths of serverless and microservice frameworks [5]. Modern cloud applications often come in two categories, and each differ in the system support they rely on. Short-lived tasks, known as *serverless tasks*, that take advantage of burst-parallelism, are supported by services with stateless computation. Long-running and stateful tasks, known as *microservices*, need to communicate and hold state across requests. Current systems either excel at one or the other but fail to unify both. σ OS addresses this gap by allowing tasks (called procs) to have the fast start capabilities of a serverless environment while still being able to communicate and be long-lived like microservices. Procs are Linux process each run in separate σ containers that have a restricted set of system calls and no local filesystem, making them light weight, but procs under the same tenant still have the ability to communicate with each other through the σ OS API.

1.2 The Fast Start Challenge

Tasks requiring serverless computing are often invoked in parallel bursts, meaning many copies of the same process are spawned on remote machines at once. Define the start time as the time difference from when a process is requested to be spawned to when it is able to start doing meaningful work. Slow start times for processes spawned in parallel bursts would degrade user experience and diminish the performance benefits that serverless platforms promise. Thus, the ability to start procs rapidly and in parallel is one of σ OS's goals. A common technique to speed up process initialization is starting a process warm instead of *cold*.

A cold start is defined as when a process is initialized for the first time on a machine, requiring the system to fetch the process's binary from shared storage (e.g., S3), which results in higher latency. In contrast, a warm start happens when a process of the same type has been run recently on the same machines. In this case, The binaries are typically cached from the previous execution, allowing the proc to skip the binary download and resulting in faster start times.

Although warm starts improve proc startup latency, there are instances where *application initialization* requires a significant amount of time to execute, for example from loading a runtime like the JVM or reading state from a file. This can push initialization latency to the order of hundred of milliseconds, measured on various serverless services [1]. Additionally, σ OS handles container start times, not application start time, so currently does not address this issue. This thesis introduces σ CRIU as a solution.

1.3 CRIU And σ CRIU

Checkpoint/Restore in Userspace (CRIU) is a Linux tool that allows running processes to be checkpointed and saved to a snapshot, called an image dump, and later restored to continue execution from the same state. CRIU can capture a process's memory, file descriptors, network connections, and other execution details, enabling it to resume exactly as it was at the time of the snapshot. This thesis proposes the integration of CRIU into σ OS, and introduces optimizations to

CRIU's restore to speed up burst-parallel spawning times of checkpointed procs in σ OS.

1.3.1 σ CRIU

σ CRIU is a toolset in σ OS that uses CRIU's Checkpoint and Restoring capabilities to act as a *warm start* when burst spawning procs while overcoming the problem of slow initialization. Users of σ CRIU would checkpoint a single instance of a proc right after initialization. Many copies of the proc could then run by restoring from the image dump, thereby bypassing the lengthy execution of the initialization code.

Using Checkpoint/Restore makes the startup latency of the proc no longer dependent on the slow initialization code, and only on the speed of the restore operation. Thus, σ CRIU's focus is on optimizing restore time. The main optimizations σ CRIU introduces are compression of a checkpointed proc's image metadata in S3 storage to reduce the network communication cost in restore, providing a lazy page service that allows demand paging of a restored proc's memory image, implementing a prefetching algorithm that reduces the latency of handling page faults using the lazy page service, and caching kernel metadata to eliminate recomputing it during CRIU's restore operation.

1.4 Hotel-Geod

A case study called Hotel-Geod, from DeathStartBench[3], represents a realistic use case of σ CRIU. The Hotel-Geod test is a compute bound microservice with a configurable amount of soft state, and involves issuing a slow file read operation during initialization. In the event that many of these procs need to be run in parallel, restoring after the checkpoint would save a significant amount of redundant work.

```
// Run starts the server
func RunGeoSrv(job string, cktpn string, nidx int,
    maxSearchRadius int, maxSearchResults int) error {
    geo := &Geo{
        maxSearchRadius: float64(maxSearchRadius),
        maxSearchResults: maxSearchResults,
    }
    geo.idx = NewGeoIndexes(nidx, "data/geo.json")

    if cktpn != "" {
        CheckpointMe(cktpn)
    }
}
```

Figure 1: Hotel Geo Server code before Checkpoint

The Hotel-Geod proc in σ OS consists of a call to the RunGeoSrv function in Figure 1. RunGeoSrv uses a file of point locations, "data/geo.json", to compute *nidx* indexes of hotels (the function NewGeoIndexes), and then sets up an rpc server able to answer nearest neighbor queries on these indexes. This initialization of the server state has a high latency cost. Since the test case checkpoints right after generating the indexes any restored Hotel-Geod proc can skip this initialization cost and immediately start serving requests upon spawning.

The *nidx* parameter acts as a control for how large the initialization cost will be before checkpointing, and we adjusted this in our evaluation of σ CRIU to find where checkpointing will be beneficial to do over spawning the proc from scratch.

1.5 Thesis outline

The remainder of this thesis is organized as follows.

- **Section 2** – Pre-existing research done on using checkpoint and restore for fast startup of processes and ideas that σ CRIU borrowed
- **Section 3** – Overview of σ CRIU’s design: how σ CRIU checkpoints and restores a proc, the optimizations σ CRIU introduces during the restore operation, and how σ CRIU handles failures.
- **Section 4** – Evaluation of memory usage and latency σ CRIU using the Hotel-Geod benchmark, including a comparison of σ CRIU restoration against spawning from scratch and the

contribution of each optimization to reducing restore time.

- **Section 5** – Summary of thesis and outline of future directions for improving σ CRIU

2 Related Work

Much work has been done on using the idea of checkpoint/restore to migrate processes or start them quickly.

Nu: Achieving Microsecond-Scale Resource Fungibility with Logical Processes, Ruan et al. (2023)[4] This paper presents the Nu system, which focuses on improving the efficiency of resource utilization on clusters of servers running many processes. The main innovation was process fungibility, breaking a process into units called proplets. Proplets for a single process can be spread across multiple machines. In this way, processes can be split in order to load balance. This ability to break up processes also makes them flexible and simplifies the migration process, which just consists of gradually migrating all proplets from one machine to another. Since proplets are fine grained, it is fast to migrate individual proplets across machines.

"No Provisioned Concurrency: Fast RDMA-codedigned Remote Fork for Serverless Computing," Wei et al. (2023) [7] This paper introduces MITOSIS, an operating system primitive designed to enhance the efficiency of remote forking in serverless computing environments. The authors address the critical trade-off between container startup time and provisioned concurrency, and specifically examined speeds of warm versus cold starts. By using Remote Direct Memory Access (RDMA) technology, MITOSIS enables fast forking of processes (called remote fork) from *warm* machines. RDMA enables the child process to read the physical memory of its parent, and combining RDMA's ability to bypass the operating system with copy on write techniques, MITOSIS significantly reducing function tail latency and improving execution time for state transfer.

"Fast In-Memory CRIU for Docker Containers," Venkatesh et al. (2019) This paper propose VAS-CRIU, an innovative mechanism designed to enhance the snapshot and restore processes for Docker containers by leveraging Multiple Virtual Address Spaces (MVAS). The authors identify limitations on vanilla CRIU in Linux, particularly potential costly filesystem operations on memory images. VAS-CRIU uses MVAS to store application memory as a separate snapshot address space in DRAM, instead of on disk. This approach eliminates the costly file I/O operations typically required by CRIU when reading/writing the image dump, and the paper shows that VAS-CRIU significantly reduces the time required for both snapshotting and restoring container memory.

"Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting," Du et al. (2020) [1] This paper proposes Catalyzer, a system for running serverless tasks which addresses the challenges of startup latency in serverless computing environments. The

authors introduce an initialization-less booting approach that uses checkpoint and restore techniques to significantly reduce startup times. Catalyzer also introduces a new operating system primitive, *sfork* (sandbox fork), to efficiently reuse the state of running sandbox instances, thereby minimizing the overhead associated with application initialization on *warm* tasks.

The current works mentioned above either built their own checkpoint/restore tools from scratch or require modifying linux and the hardware on their machines. They also don't support microservices and multi tenancy in their systems. Instead of building a new tool set from the ground up, σ CRIU aims to use CRIU's existing high quality checkpoint and restore technology while still achieving fast proc restore latency.

"Breaking the Bottleneck: Fast, Efficient Snapshot Restores for Serverless Functions," [6] This last paper benchmarked the spawn latency of restoring various serverless functions from a checkpoint. Their study shows that restoring functions from snapshots with Firecracker and Containerd significantly reduces memory footprint compared to cold booting, but performance still suffers from thousands of page faults during execution due to lazy paging. To address this, they propose Record-and-Prefetch (REAP), which records the working set of memory pages accessed during the first restore of a process and prefetches this set for subsequent restores. REAP eliminates most page faults, delivering a $3.7\times$ average speedup over baseline snapshotting without increasing memory usage, and this idea motivated our use of prefetching in σ CRIU.

3 Design and Implementation

σ CRIU introduces several optimizations to CRIU that accelerate proc restoration. It leverages CRIU's option for demand page to avoid the upfront cost of loading the full memory image, employs a prefetching algorithm that can accurately predict and preload the data for future page faults, caches computation of kernel-data parameters to eliminate redundant CRIU restore phase setup, parallelizes CRIU operation with lazy page server setup, and compresses proc image metadata to minimize network transfer overhead.

3.1 σ OS Overview

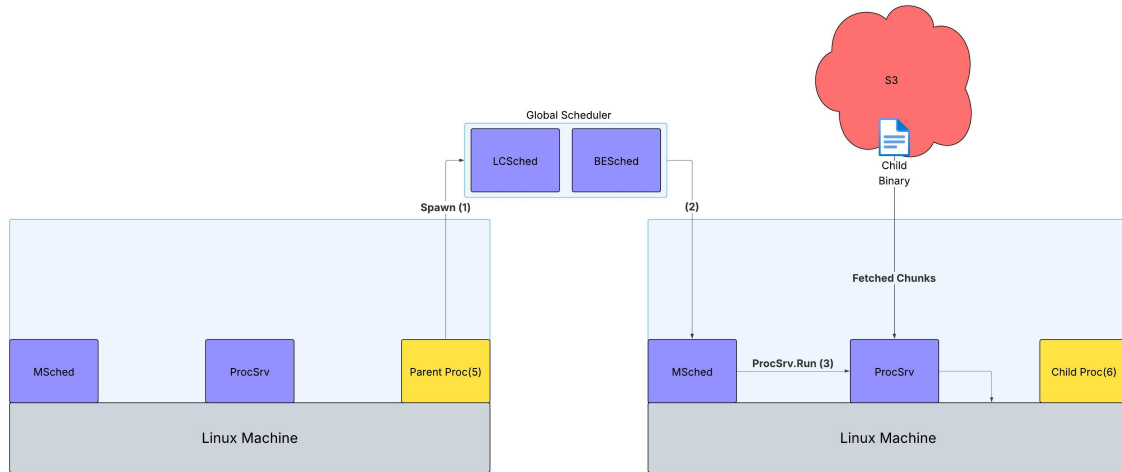


Figure 2: Proc Spawn Flow of Control in σ OS. Purple boxes represent σ OS kernel processes and yellow boxes represent user processes

The above diagram is a high level overview of the components of σ OS relevant to σ CRIU, demonstrated in the context of a parent proc(5) spawning a proc child proc(6) (without Checkpoint/Restore).

σ OS introduces the concept of a realm, which is a name space that is given to each tenant on σ OS. Realms may encompass multiple machines, and procs in the same realm can communicate with each other. The diagram assumes we are working within a single realm. Each machine in the realm runs Linux and has a service called ProcSrv, which is responsible for setting up procs, maintaining them, and cleaning up the environment when they exit. There is also a cluster scheduling service run on a set of machines, which has processes called BESched (Best Effort Scheduler) and LCSched (Latency Critical Scheduler), which are responsible for scheduling procs to be run on the machines in the cluster. The cluster scheduler can communicate with the Linux machines using MSched, a service run on each machine.

The parent proc can request to spawn the child proc via a Spawn RPC(1) to the cluster scheduler. The cluster scheduler will eventually contact the MSched on a machine, in this case a different machine than the one running the parent proc, to run the child proc by a Run RPC(2) (possibly on a different machine). MSched will issue an RPC to ProcSrv (3), which will create the child proc. All proc binaries live on S3, and are loaded in chunk by chunk lazily, so ProcSrv is also responsible for handling chunk fetching requests.

The parent proc can monitor the status of the child proc through this same path (Scheduler -> MSched -> ProcSrv) for events like when the child proc has started or terminated.

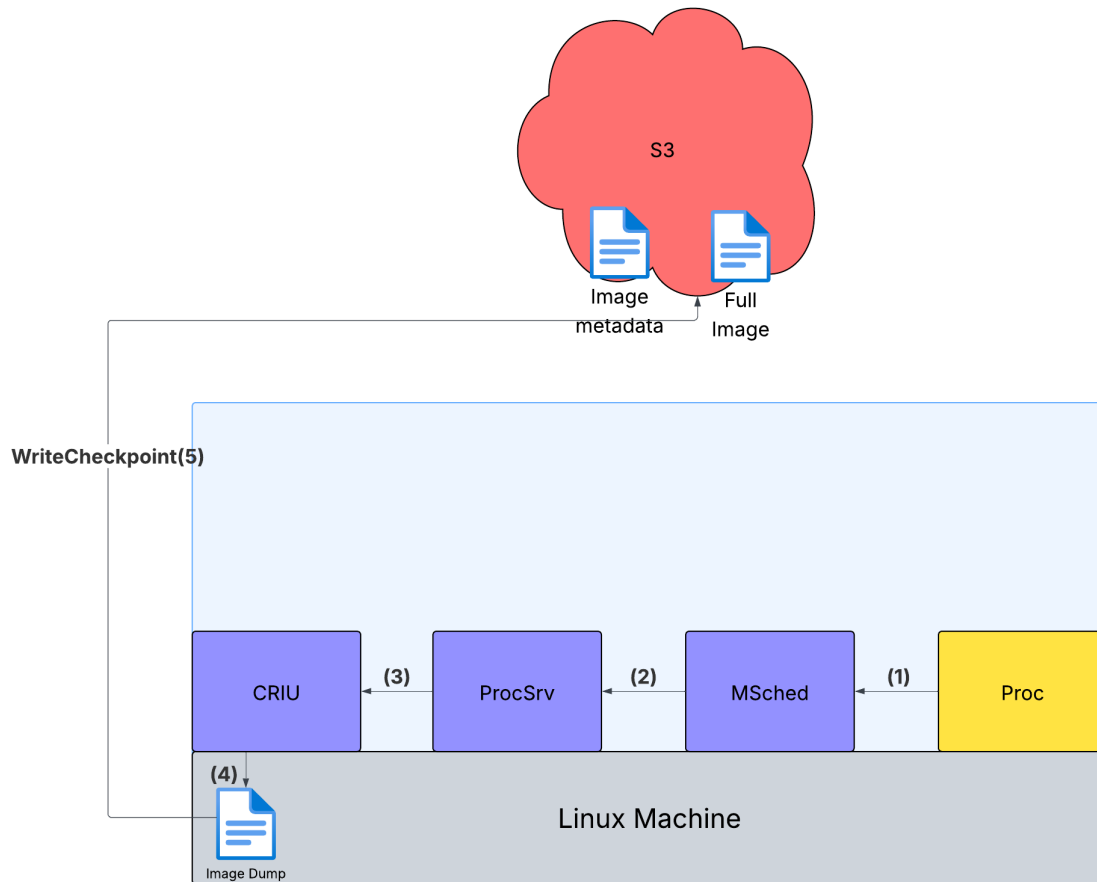


Figure 3: Checkpointing Proc Path

3.2 Checkpointing a Proc

A proc wishing to checkpoint itself issues a checkpoint (1) RPC to the MSched service running on the same machine. MSched then issues a checkpoint RPC(2) to the machine’s ProcSrv. ProcSrv will issue a Checkpoint RPC (4) to a CRIU server running on the machine, which will write the image dump to a local directory. This last step is done because the machine that restores the proc might be different than the machine that checkpoints it, so storing the image in S3 allows any machine to restore the proc.

The image metadata is actually written twice, one in full and the other to a metadata directory also residing in S3. The metadata directory’s pages.img file (which contains the entire memory image of the checkpointed proc) is empty, and so is much smaller than the full image (up to a hundred times smaller in our test cases). σ CRIU at restore time will only read the image metadata, saving significant time from having to transfer megabytes of data over the network. σ CRIU will fill

in the proc's memory image lazily after restore, which is discussed below.

```

string version = 30;
string fail = 31;
string checkpointLocation = 32;
}

```

Figure 4: Modification on Proc Struct for checkpointing

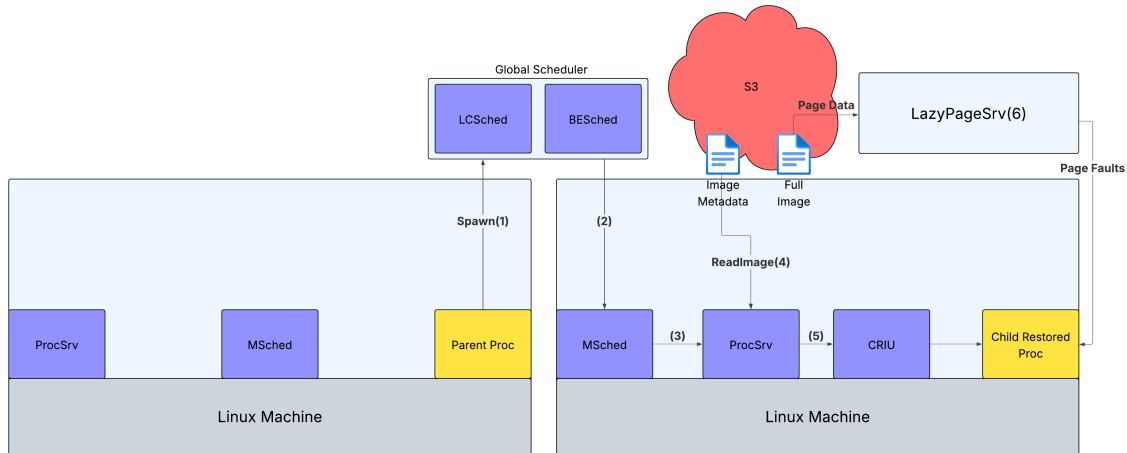


Figure 5: Proc Restoration path

Figure 5 is a diagram of how a parent proc would request restoration of child proc.

3.3 Restoring a Proc

Much like spawning a regular proc, the parent would send a Spawn RPC (1) to the cluster scheduler. However, the Proc struct that the Spawn function takes in has a field "CheckpointLocation," which if supplied represents the σ OS file path of the image dump on S3. After the cluster scheduler contacts an MSched (2) to spawn the child, MSched will recognize that the proc should be restored, and will divert from the usual path, calling RestoreProc (3) on ProcSrv.

RestoreProc is a function that reads all the image dump files from the lazy directory in S3 (4) and makes an RPC (5) to the CRIU server, which restores the child proc. Because the full memory image of a checkpointed proc can be large (over 50 Mb in Hotel-Geod), σ CRIU avoids the cost of having to read the entire file by implementing demand paging using a service called LazyPageSrv.

LazyPageSrv (6), which runs next to ProcSrv, handles the proc's page faults in userspace using the Linux system call userfaultfd. Userfaultfd takes in a process and supplies a file descriptor, and any page faults from the process get sent to the file descriptor instead of the kernel. LazyPageSrv will read the file descriptor, fetch the faulting page, and use the system call UFFDIO_COPY or

UFFDIO_ZERO to copy in/zero out the faulting page.

3.4 σ CRIU Optimizations

3.4.1 Compressing the Image Dump

When restoring a process, ProcSrv's first step is to read all the image metadata files from S3, shown at (4) in Figure 5. There are 25 files that make up the metadata, and reading this data over the network can account for a large portion of restore latency. An optimization that σ CRIU makes is to compress the image metadata in storage. For our test case Hotel-Geod this reduces the metadata size from 516Kb to 56Kb, which significantly decreases the amount of data needed to be transferred over the network between ProcSrv and S3. Different compression algorithms were tested, such as Klauspost and LZ4, but the effect on latency was negligible, so we used Klauspost "Best Compression," it being the most popular golang compression library.

3.4.2 Memory Metadata Preparation

When a ProcSrv starts restoring a proc, it will issue an RPC to LazyPageSrv to register the proc to be restored with a lazypagesid. LazyPageSrv will then create a Page Fault Handler, which will be the proc that handles the restored proc's page faults using Userfaultfd. The Page Fault Handler uses the mm (memory map) and pagemap files of the image metadata to map groups of pages in the proc's checkpoint to sections of virtual memory. The memory map file contains all the proc's VMA's, which are sections of virtual memory that have the same permissions. Using these two files, the Page Fault Handler will first create an array of what CRIU calls IOV's. IOV's are defined as a contiguous region obtained by overlaying pagemap entries onto VMAs. Each IOV corresponds to a segment of virtual memory that ends either at a VMA boundary or a pagemap entry boundary. This is done because the UFFDIO system call, which is used by LazyPageSrv, cannot copy over pages from different VMAs in a single call. Lastly, the Page Fault Handler sets up the Prefetcher, which will be explained later.

LazyPageSrv has a socket that it reads, and this socket is passed to every CRIU restore operation. When the CRIU restoring process is ready to start demand-paging, it uses LazyPageSrv's socket to request a connection by sending its lazypagesid and the file descriptor it will use to send page faults through. LazyPageSrv will find the Page Fault Handler associated with the lazypagesid, and assign the file descriptor to the Page Fault Handler, which the Page Fault Handler will then start listening to for page faults.

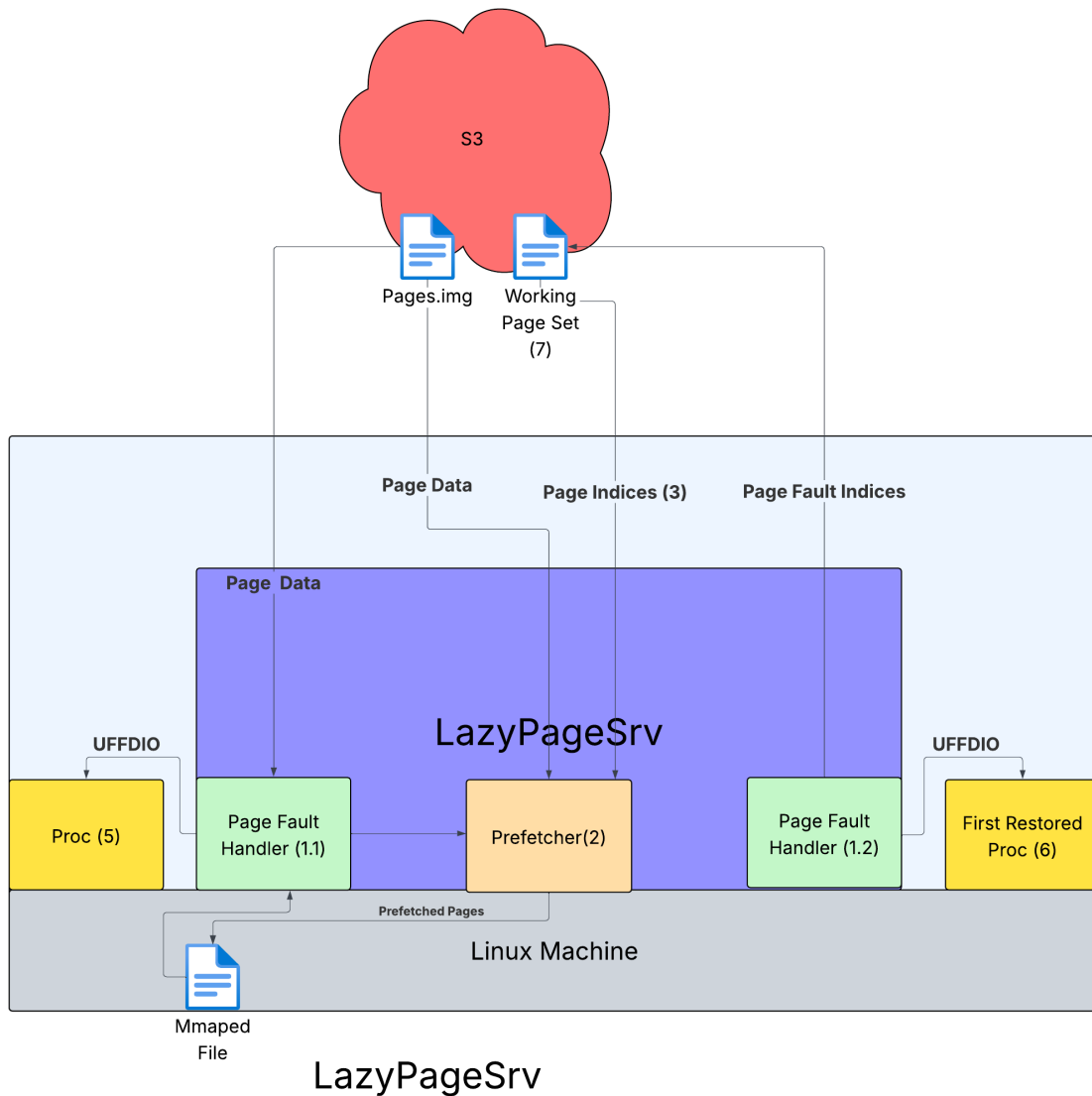


Figure 6: LazyPageService Page Fault Handling Workflow

3.4.3 Prefetching

Figure 6 demonstrates how the Page Fault Handler handles page faults. Since the restored proc's full memory image file lives on S3, each faulting page's data needs to be sent over the network. In testing, we found that the majority of the latency of page fault handling was due to this network communication cost. Thus, σ CRIU introduces the Prefetcher, which is a goroutine that prefetches pages from S3 for the restored proc and stores them locally on an mmaped file. The goal is that any time a page fault occurs, the page was already prefetched and the Page Fault Handler can avoid

querying S3, copying data all locally.

The Prefecher(2) continuously polls a queue for page indexes and prefetches these pages from the proc's image in S3. The fetched pages are then written to the mmaped file, which will be read by the main Page Fault Handler goroutine when page faults actually occur. In our experiments, copying a page from the mmaped file takes no more than a dozen microseconds, whereas reading a page from S3 can take up to hundreds of microseconds due to network latency. In our Hotel-Geod test case, there are over two hundred page faults occurring per restore, so a good prefetching algorithm can save a lot of latency, especially when the golang runtime is starting up (which is responsible for the majority of page faults).

3.4.4 Prefetching Algorithm

σ CRIU uses an algorithm centered around memory spatial locality to decide which pages around the faulting page should be copied over in the Page Fault Handler. the Page Fault Handler will first find the faulting page's IOV using a binary search of the IOVS list. Binary search was used because the the number of IOV's can be in the hundreds, and since the IOV list is assumed to be sorted, binary search will reduce the number of checks to less than 10.

The Page Fault Handler will then check the mmaped file for if the faulting page was prefetched via querying the IOV cache. The IOV cache contains whether each page in the IOV has been fetched yet. If not, which is the worst case, the Page Fault Handler will fetch the faulting page from S3. Another check will then be performed to calculate whether the pages surrounding the faulting page have been prefetched, and if so, up to `N_PREFETCH` (a tunable parameter) pages will be copied to the restored proc's memory using `UFFD_IO`. `N_PREFETCH` was experimentally shown to be optimal at 16, balancing both copying cost and the likelihood of future page faults needing the additionally copied pages. After copying the page, the Page Fault Handler will append up to `N_PREFETCH` pages to the Prefetching Queue from the pages surrounding the page fault, anticipating that the proc may fault on nearby pages in the future.

3.4.5 Prefetching the working Set

As shown in Figure 6, the first time a proc is restored (6), the Page Fault Handler (1.2) will record every page fault that occurs, in the order that they occur, and log it in a working page set file (7), which is stored with all the other dump files in S3. Subsequent restorations of the proc will read the Working Set File from S3 when retrieving the image dump. `LazyPageSrv` loads in and starts prefetching the working set when the Page Fault Handler for a proc is registered and recreated. Registration happens right before `CRIU Restore` is called, so the Prefetching Queue will be populated with all pages in the Working Set File in parallel with the restore operation. By the time the restored proc calls `userfaultfd` and starts faulting in pages, all the requested pages should be in the mmaped file, and the majority of the latency should then be copy time, not network communication time.

3.4.6 CRIU Kernel Data Caching

A goal of σ CRIU was to use the proven existing checkpoint/restore technology of CRIU to checkpoint processes rather than developing a new system from scratch; consequently, we aimed to modify CRIU as little as possible while making it compatible with σ OS. One optimization σ OS used from CRIU is in the computation of the kernel data file. CRIU needs a lot of information about the kernel it is running on, such as if particular system calls that it wants to use in operation (e.g. clone3) are available. It also takes information that is not constant across different reboots or machines. For example, every Linux machine on bootup will assign a random device number to represent regions of anonymous shared memory, and CRIU needs this device number to restore memory correctly. Therefore, on each restore, CRIU will compute a file called criu.kdat with all the kernel data. The computation of this file is actually responsible for most of the latency in restore operations. Fortunately, because σ OS procs are restricted, and do not support features like anonymous shared memory, σ CRIU will compute this file once on every machine at startup and cache it. Then any CRIU operations in the future will check the cached location and use it, cutting restore time in half (See Section 4.2).

3.5 Implementation details of σ CRIU

3.5.1 DialProxyConn

σ CRIU must do some preparation work before checkpointing and after restoring a proc in order to be compatible with CRIU.

Streaming Sockets: CRIU does not support open unix streaming sockets in dumped procs, so procs must close these sockets before checkpointing and open them again at restore time. There exists a unix streaming socket called DialProxyConn in the σ OS API that every proc has access to, and allows the proc to communicate with the Dial Proxy Service.

Dial Proxy Service: The Dial proxy service is used by procs to create TCP connections. The connection for this socket should be closed on checkpoint and then reopened on restore. However, σ OS does not allow procs to close and open sockets, so σ CRIU takes responsibility in handling this.

Inherited Socket Pair: To communicate this information with a restored proc, σ CRIU creates a socket pair, which CRIU passes through to the restored proc from the restorer. Using this, ProcSrv can communicate to the restored proc, and therefore can pass the new DialProxyConn: after the restore operation returns, the restored proc will check in by writing 1 byte of data to σ CRIU using the socket pair. σ CRIU will then send the DialProxyConn and also the proc's procEnv through the socket pair. This setup results in increased transparency when checkpointing and restoring.

3.5.2 RestoreProc

As seen below in Figure 7 (3), RestoreProc is the function responsible for spawning a proc from a checkpointed image and setting up the Lazy Page Server for it. RestoreProc is designed to save latency by parallelizing these two tasks RestoreProc uncompresses the files, and then spawns 3 goroutines.

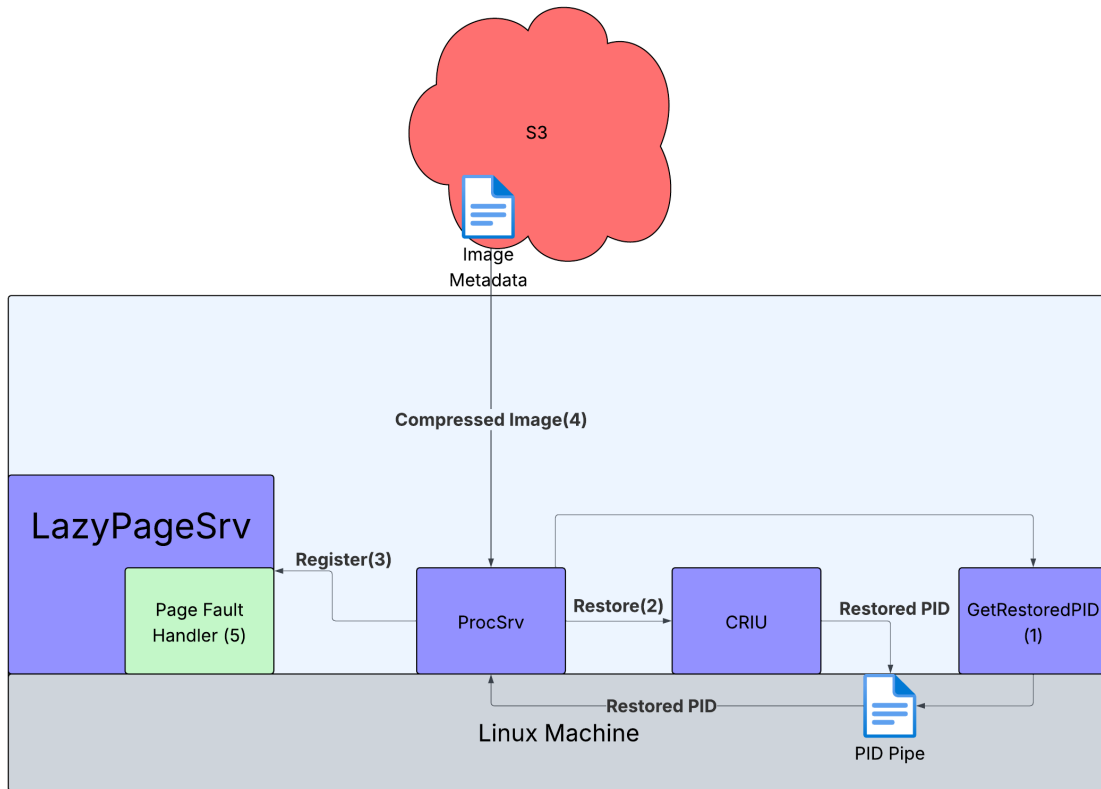


Figure 7: RestoreProc Path

3.5.3 GetRestoredPid(1)

As mentioned before σ OS proc binaries live on S3, and the binaries are loaded in chunk by chunk by ProcSrv lazily via fetch requests. ProcSrv thus keeps a mapping between all proc PIDs and their binary locations to serve these fetch requests. Native CRIU unfortunately cannot guarantee the pid of the restored proc, and the proc might issue a fetch request right when it starts running, so σ CRIU needs to obtain the new proc's PID as soon as possible (while CRIU is running). Note that CRIU works by forking itself, and then making the child morph into the restored process (forking if necessary to restore the entire process tree). Since σ OS procs are restricted to not have any children, as soon as CRIU forks for the first time, we know the child's PID must be the restored PID. Thus, GetRestoredPID creates a pipe (PIDpipe), and when CRIU forks, it writes its PID to the pipe. σ CRIU can then map this PID to the binary.

3.5.4 CRIU.Restore (2)

The second goroutine is the CRIU restore operation. First, RestoreProc will set up the jail directory that the proc's container will live in, including all necessary mounts and creating both a

new DialProxyConn and the socket pair discussed in the Section 3.5.1. RestoreProc then creates a criuclient using the gocriu library and then sends the Restore RPC to CRIU.

3.5.5 LazyPageSrv Register (3)

The last goroutine calls a "Register" RPC to LazyPageSrv with a generated id "lazypagesid." LazyPageSrv will create a Page Fault Handler proc corresponding to the id, which will lazy handle page faults while the restored proc is running. The CRIU.Restore operation, when connecting to the LazyPageSrv, will use lazypagesid to identify the Page Fault Handler (hopefully with a full prefetched working set of pages) the proc will send page faults to.

3.6 Handling Restore Failures

Once ProcSrv calls the CRIU Dump operation, the proc is killed. The checkpointing proc will receive an error status code since the proc did not exit normally. This is used as an indication that the checkpoint finished and restore can be called. However, because this protocol makes no distinction between a failed Checkpoint (which also returns an error) and a successful one. More importantly, after CRIU.dump is called, ProcSrv still needs to write the files to S3, and only after that should the proc actually return. ProcSrv keeps proc metadata for each proc it is running, so a field called checkpointStatus was added to the metadata, which represents whether a proc is either not checkpointing, in the process of checkpointing, or successfully done checkpointing. This field will be updated before and at the end of the checkpoint operation, so ProcSrv can read the field to determine if the proc actually exited with an error or not.

4 Evaluation

We evaluated σ CRIU using the Hotel-Geod benchmark (introduced in Section 1.4) on an 8-node cluster on Cloudlab [2], with each node provisioned with 4 cores running Ubuntu 24.04.

The benchmarks aimed to answer 3 main questions: how much better latency wise is restoring a proc from a checkpoint compared to running it from the beginning (Section 4.1); how much memory does it take to store the checkpoint of a proc and to restore it (Section 4.1); how much latency reduction does each optimization in σ CRIU contribute (Section 4.2).

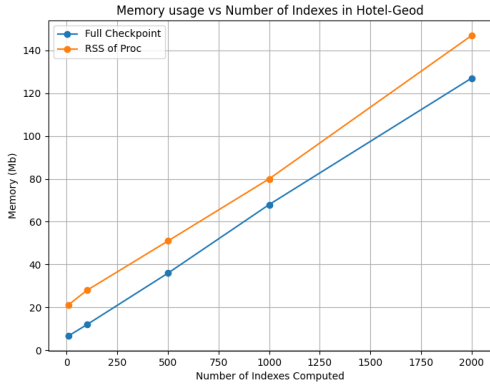


Figure 8: Memory usage of Hotel-Geod vs nidx

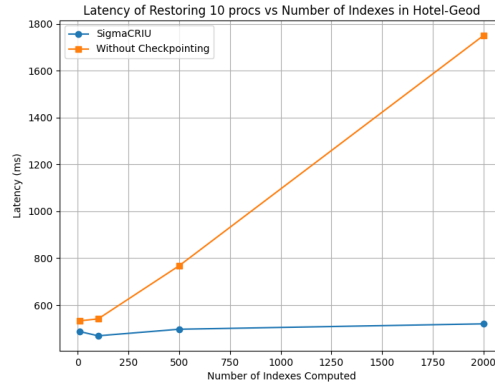


Figure 9: Latency of Restore vs nidx

4.1 σ CRIU Versus Baseline

Figure 8 shows the memory footprint of the Hotel Geod proc while varying `nidx`, the number of indexes generated before the checkpoint, which controls the size of the proc’s soft initialization state. The orange line represents Resident Set Size (RSS), which is the amount of physical RAM the proc used. The blue line is the size of the image dump when the proc is checkpointed. As shown, the memory footprint of the proc is directly correlated with the checkpoint size, and the storage requirement is just bit less than the RSS. We also measured the number of pages fetched from the memory image by `LazyPageSrv` after restore, which was consistently at around 200 1KB pages regardless of the size of the initialization state. For reference, the state size with 1000 indexes computed is 16700 pages. The number of pages fetched is relatively constant since most page faults during restore are from the go runtime, and the proc does not need to touch most of its state to start up.

Figure 9 measures the latency of restoring 10 Hotel-Geod processes while varying the number of indexes generated before the checkpoint. When spawning from scratch (orange), latency rises steeply as the amount of application state increases. The latency follows the same path as the memory usage in Figure 8, showing that the cost of initialization grows linearly as the state size increases. In contrast, σ CRIU (blue) maintains nearly constant latency regardless of the prework: the lowest latency (`nidx=100`) was 469ms while the highest latency (`nidx=2000`) was 519ms. As mentioned before, the number of page fetches done after restore is constant, which may explain why the latency was also constant.

These results demonstrate that σ CRIU effectively decouples restore latency from application state initialization and size, making it advantageous for applications which touch a fraction of their pages after restore.

4.2 σ CRIU Optimizations Evaluation

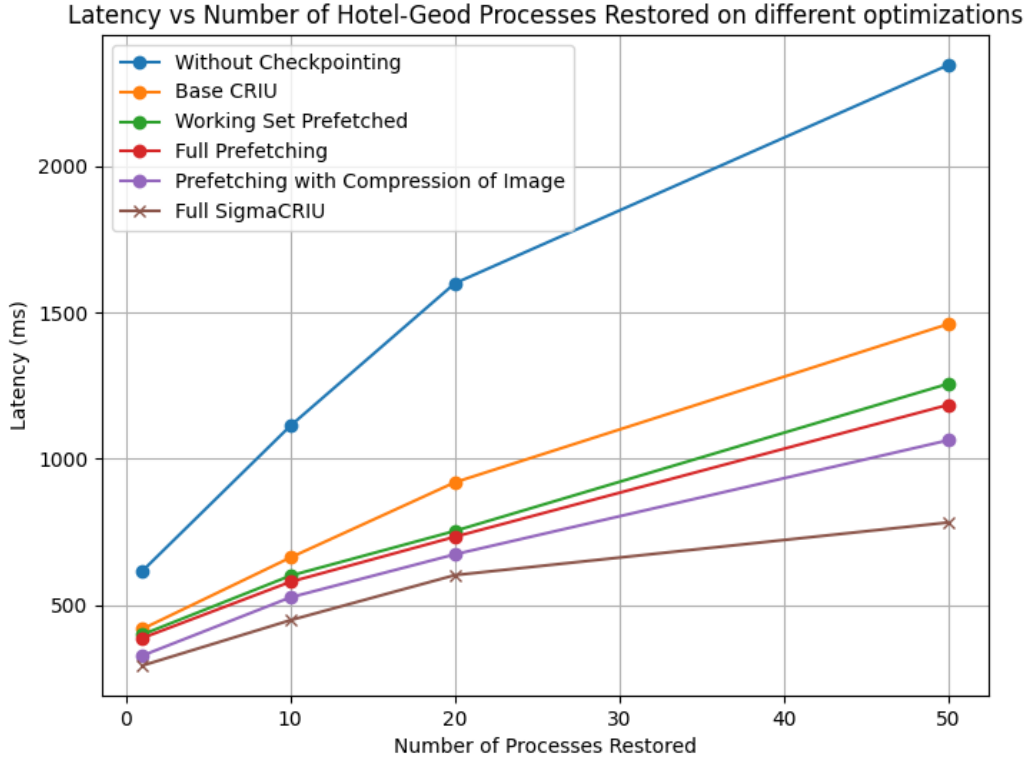


Figure 10: Latency reductions of each optimization

Figure 10 shows the latency of restoring Hotel-Geod processes with different optimizations in σ CRIU, building from spawning from scratch all the way to full σ CRIU. When restoring exactly one proc, σ CRIU (purple) is over 50% faster than spawning from scratch (blue). However, baseline CRIU is not much slower than σ CRIU, with an average spawn latency of 419 ms versus 294 ms.

As the number of procs scale, σ CRIU grows comparatively faster to both baseline CRIU and spawning from scratch: when spawning 50 procs, σ CRIU averaged 783 ms compared to 1361 ms for Base CRIU and 2346 ms(!) for spawning from scratch.

Prefetching the working set (green) appears to account for the greatest reduction in latency, while the improvement from only prefetching the working set to full prefetching (prefetching pages around a faulting page) is the least. We measured the latency of fetching from S3 versus copying from the mmaped file, and on average, fetching a page from S3 takes 117 microseconds, while copying from the mmaped file takes 0.8 microseconds. The 140x speedup in latency that occurs if

the page is local explains why prefetching the working set had such a significant improvement from having to fetch from S3. However, it seems that the spacial locality prefetching algorithm over-fetches pages that are never accessed, whereas the working-set is (and expectedly so) an excellent predictor for which pages will be fetched in the future, avoiding unnecessary I/O.

From checking individual logs, caching the kernal data file (which is the brown line compared to the purple line) reduces the CRIU restore operation from 66 ms to less than 30 ms in an individual proc, and this improvement resulted in the largest latency win at 50 procs spawned. We hypothesize that this is explained by the fact that caching the kernel data saves latency independent of the number of procs running, whereas the other optimizations may start to degrade with more parallelism, for example, *LazyPageSrv* may have to queue serving requests if it creates too many Page Fault Handlers at once.

5 Future Work and Conclusion

There remain several avenues for future work. One direction is to investigate local caching of restored process data, reducing the reliance on S3 and avoiding repeated remote fetches during subsequent restores. Another possible step is to expand the evaluation beyond Hotel-Geod by applying to a wider set of applications with different initialization patterns. This which would provide stronger evidence of its generality. Finally, while σ CRIU’s current prefetching algorithm is based on spatial locality, exploring more sophisticated prefetching algorithms may further reduce restore latency and improve overall performance.

By leveraging CRIU’s proven snapshotting capabilities and introducing optimizations such as lazy paging, compression, kernel metadata caching, and working-set prefetching, σ CRIU significantly reduces the latency of spawning new procs. Our evaluation on the Hotel-Geod benchmark demonstrates that these optimizations collectively allow σ CRIU to spawn procs in parallel bursts far more efficiently than baseline CRIU or fresh initialization. This makes σ CRIU an effective toolset in spawning microservices with large initialization cost in a latency-sensitive environment.

References

- [1] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 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 ’20*, page 467–481, New York, NY, USA, 2020. Association for Computing Machinery.
- [2] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

- [3] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19*, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery.
- [4] Zhenyuan Ruan, Seo Jin Park, Marcos K. Aguilera, Adam Belay, and Malte Schwarzkopf. Nu: Achieving Microsecond-Scale resource fungibility with logical processes. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1409–1427, Boston, MA, April 2023. USENIX Association.
- [5] Ariel Szekely, Adam Belay, Robert Morris, and M. Frans Kaashoek. Unifying serverless and microservice workloads with sigmaos. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles, SOSP '24*, page 385–402, New York, NY, USA, 2024. Association for Computing Machinery.
- [6] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 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 '21*, page 559–572, New York, NY, USA, 2021. Association for Computing Machinery.
- [7] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. No provisioned concurrency: Fast RDMA-codedigned remote fork for serverless computing. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 497–517, Boston, MA, July 2023. USENIX Association.