Metropolis: OS Support for Container FaaS Snapshots

Anonymous Author(s)
Submission Id: 100

Abstract

The use of snapshots for the cold start problem in serverless is well known and studied in virtualized environments. However, OS containers are increasingly being deployed in private and hybrid cloud serverless environments through emerging products and open source systems.

Metropolis is the first system to introduce novel serverless optimizations based on OS introspection in to the checkpointing mechanism for OS containers. Metropolis uses fine-grained sharing throughout storage and memory to provide performance and space savings to serverless. Metropolis has 10× lower invocation latency than CRIU based invokers, and creates functions 3× faster than Catalyzer, the state of the art in VM based snapshot invocations.

1 Introduction

The serverless paradigm provides scalable infrastructure on demand, fulfilling the longstanding vision of utility computing [22]. Developers write serverless code (i.e., functions) and upload them to cloud providers. Providers deploy instances of these functions inside sandboxes (e.g., VMs or OS containers) to service client requests (i.e., invocations). Providers assumes the responsibility of infrastructure management (hardware and software) and function autoscaling. Recent studies show most functions are shorted lived and often execute in under 1 s [34] leading to sandbox creation being a significant source of overhead. Optimizing on-demand sandbox creation is critical for efficient serverless.

A popular optimization for sandbox creation is to restore an already initialized function from an on-disk snapshot [19]. Serverless sandboxes like Amazon's Firecracker [17] require 100 ms to boot and even more to initialize the function runtime. For example, a Python script that imports NumPy, one of Python's most popular libraries, takes 270 ms to set up, more than the execution time of 25% of serverless functions [34]. Various systems have optimized the use of VM snapshots within serverless [18, 21, 37, 40].

Snapshotting is underexplored for OS container based sandboxes, which suffer from startup times that are as long as the function execution time.

OS containers are increasingly used in both public and private serverless infrastructure. In public clouds, containers are used to colocate functions in the same VM to share specialized hardware such as GPUs and TPUs [15, 26]. Private and hybrid clouds use containers to run serverless functions on owned infrastructure using products like Oracle Cloud Functions and Redhat Openshift Serverless [4, 5, 7, 8, 10–12, 16, 27]. However, none of these existings systems use snapshotting techniques to accelerate function invocations.

A few research systems [38, 41] use conventional checkpoint/restore (CR) approaches, but do not explore the unique benefits of OS containers to improve performance. These systems use CRIU, the standard Linux checkpoint/restore mechanism, to restore serverless instances. However, these systems do not investigate ways to optimize CRIU for serverless applications.

Containers allow us to go beyond typical snapshotting techniques through the use of OS introspection. Container snapshots can be manipulated during restore in ways that are not possible with VMs that appear as a black box. This fine-granularity allows for optimizations within serverless typically unavailable to other approaches, such as implicit dependency sharing and the manipulation of kernel objects in restored instances. We explore the use of OS introspection to optimize serverless container invocations.

We present Metropolis, a system that enables fast and efficient serverless invocations through container-based snapshots. Metropolis creates fine-grained images from container snapshots which describe containers as sets of OS resources, enabling faster restores. Metropolis consists of a kernel module that provides and manipulates the fine-grained container image using provided checkpointing and introspection mechanisms. The Metropolis invoker uses these images and manipulates restored state to minimize invocation latency, memory usage, and IO.

Metropolis uses the Aurora single level store (SLS) [39] as a checkpointing system that enables introspection. SLSes structure application images as groups of individually serialized OS objects (e.g., libraries, mappings, and process state). SLSes restore by deserializing the images within the kernel. However, Aurora is designed for application persistence and does not support serverless programming models. Aurora lacks an optimized restore path and does not implement any of the Metropolis mechanisms to support fast invocation of serverless containers.

Metropolis uses fine grained OS introspection on OS containers to enable submillisecond container restores and fast function executions. First, Metropolis supports fine-grained sharing of dependencies between images on-disk and in-memory. Second, sharing reduces the footprint of each function's image and reduces the amount of IO needed to bring in an image into memory. Third, Metropolis further uses OS introspection to automate the snapshotting of serverless functions, regardless of runtime, without arbitrary waiting. Fourth, Metropolis optimizes network communication through our *socket handoff* technique that injects a connected socket into a restoring function to bypass the builtin reverse proxy most local host invokers implement.

Metropolis reduces a major overhead common to snapshotting systems in which the cold start latency is proportional to the working set size of the function during its execution as this memory must be read in from disk. Metropolis does this through fine-grained sharing of dependencies on disk and in memory between functions without using a common base image. Metropolis caches common dependencies in memory, improving performance by minimizing IO during the functions execution. This technique, called *collaborative warmups*, provides up to 3× better execution times than alternative approaches (e.g., image overlays or function traces). Even seemingly unrelated functions benefit from collaborative warmups. For example when a JSON function runs before a video processing one, it results in a 100 ms speed-up to video processing (or a 25% reduction).

Sharing reduces the image size on-disk allowing Metropolis to cache many images on local disks rather than on cloud storage, which others have shown is beneficial [30, 40]. The effective image size is only the private code and data of that function's container and everything else is shared transparently. Metropolis can store 700 k no-op Python functions per terabyte of storage.

Metropolis uses an automatic snapshot creation mechanism and an invocation mechanism for efficiently backing new containers with image data. We observe that serverless functions always listen for incoming clients after initialization allowing us to snapshot them immediately after setup without userspace code. Our mechanism waits for functions to listen for clients, then transparently checkpoints them. This technique handles arbitrary runtimes and languages.

Metropolis uses OS introspection for other optimizations including *socket handoff*. Socket handoff directly connects function invocations with the client, allowing it to avoid the overhead of the builtin reverse proxy present in most invokers. This optimization reduces communication latency by up to 45%.

Our evaluation shows the performance benefits of OS support for snapshotting serverless containers(§ 5.1). Metropolis restores a running Python interpreter in 500 µs, 10× faster than CRIU-based systems. Metropolis achieves 5000 functions per second on a single socket. Compared to state-of-the-art VM-based solutions Metropolis achieves up to 7× faster application response times due to its sharing mechanism.

2 Background

Cold starts occur when a serverless function is run infrequently requiring restarting the service before processing requests. Cold starts are expensive as startup times are on the order of seconds. For example, a study of Azure's serverless infrastructure showed that a Python sandbox, which takes longer than the execution time of %40 of all functions run on Microsoft Azure [34]. This study further shows that cold starts are common as most functions are called rarely [34].

The main source of overhead when invoking a cold function from an on-disk snapshot is paging in the application memory during restore and execution. *Pagein* is the process of reading in pages from disk (or a cloud store) into the memory of a process. Invoking the snapshot requires restoring the sandbox state, retrieving the image and paging in its memory. Function image size scales with sandbox size, so optimizing the pagein process is important for good performance. A common technique is to selectively pagein all memory you expect to be used at once. Any unexpected memory accesses during the course of a function's execution will result in IO as this memory must be fetched from disk during runtime.

Paging Strategies. The pagein of image data is either *eager*, i.e., at invocation time, or *lazy*, i.e., on demand at runtime. Eager paging can be faster than lazy due to better IO locality, however, depends on the predictability of the function's working set to avoid overhead. Excessive paging wastes time doing IO during restore as unneeded data is paged, this also increases the memory use an instance. On the other hand, inaccurately predicting the working set leads to costly pagein during function execution.

Some systems improve performance by attempting to predict the working set of a function during its execution. One such technique is Record and Prefetch (REAP) [40], which uses the fact that instances of the same function often have similar working sets, and traces all pages accessed during a test invocation to a file. REAP eagerly pages in the trace file at invocation time using sequential IO.

However, VM snapshots are large and include the function's dependencies and sandbox state, which is unshareable between function images. Even small functions have large images by virtue of including interpreter state and common dependencies. For example, each Firecracker function has its own gRPC package, server and Python interpreter with standard dependencies. This leads to even small functions having a large on-disk and in-memory footprint which reduces local node capacity.

In general, snapshotting is expensive in terms of disk usage, with REAP trace files adding to this overhead. Table 1 shows a breakdown of the space overhead for a Hello World benchmark using REAP with Firecracker VMs. Firecracker snapshots are 256 MiB in size, equal to the VM memory size. However, Hello-World only has 1 KiB of Python code that is unique to the application, while the rest is common image data. Function state accounts for a major part of both the image and the trace.

Reducing Image Size. Reducing image size is crucial for invocation latency because of two major effects. First, reduction through sharing and/or deduplication results in shared state remaining warm, meaning less IO during subsequent restores [21]. Second, smaller images result in more images being able to be stored on local disk, which is orders of

	Firecracker
Snapshot	256 MiB
VM State	12 KiB
Docker Image	65 MiB
REAP Trace	7.9 MiB
Disk Total	329 MiB
Memory Total	166 MiB

Table 1. Breakdown of the Disk Usage for a Firecracker image and a HelloWorld application. The application is a less than 1 KiB of Python.

magnitude faster to read from than cloud storage [40]. For example, the latency to retrieve a small object from AWS S3 is 8 ms or higher [30]. For short lived serverless functions, 8 ms adds significant overhead to the invocation time.

SEUSS [21] uses differential snapshotting to reduce cold start latency. Implemented for serverless unikernels, SEUSS minimizes disk IO during invocations by using the same common in-memory image as a base for all functions with private data to the function in an on-disk diff. Function images are thus composed by stacking the in-memory base with the on-disk function private data. Functions shares dependencies only if it is imported in the language interpreter included in the base snapshot.

Common in-memory base images for OS containers lead to complex functions slowing simpler ones, which does not occur with VMs. In SEUSS, adding dependencies adds little overhead as VMs have nearly constant restore times proportional to memory size. This is because hypervisors view VM's as a list of memory pages. Containers are far more complex as the operating system views processes not as just pages of memory, but also the kernel objects that represents and aggregates these pages together. This view results in checkpoint/restore times for OS containers being proportional to the complexity of the process.

To illustrate this, we implemented SEUSS's strategy within Metropolis and used two functions as an example, a no-op Python function compared to that of a video processing one using OpenCV [29]. We found that while a no-op Python function's address space takes only 50 µs to create, we need 3000 µs for that of the video processing workload. Using a common base for both functions leads to the no-op benchmarks taking over 3000 µs to restore.

OS Container Snapshotting. Container Frameworks typically use CRIU [3], the popular Linux checkpoint restore system, to create function images. Such systems include Function Prebaking [38], which builds a snapshotting invoker based on CRIU, and Medes [33], which involves actively scanning memory to deduplicate memory within images. These systems which depend on conventional checkpointing mechanisms, like CRIU, are inherently limited by the speed with which the framework can deserialize images into running containers.

Deserializing a language interpreter like Python into a running application involves restoring the in-kernel state of the container, including all processes, threads, and individual mappings. The costs of restoring these primitives heavily penalizes cold starts due to the complexity of in-kernel container state. For example, a simple "hello world" Python program includes about 500 discrete address space mappings. For this reason, systems like Catalyzer consider OS container checkpointing fundamentally slow [24].

SLS: A Better Approach to Checkpointing. Single level stores (SLS) [35] unify volatile and persistent storage by continuously and incrementally checkpointing application state. Application objects like file descriptors or memory regions are stored as separate objects in the underlying object store. From the perspective of the SLS an on-disk file and a memory mapped object are the same. SLSes treat volatile memory as a cache for system state. Applications are persistent from a user's point of view and continue executing after a crash as if nothing happened.

Aurora [39] views checkpointing from the internal kernel perspective to faithfully checkpoint and restore any and all OS state, rather than the userspace perspective used by conventional checkpointing systems. Each kernel object is checkpointed and restored separately facilitating the manipulation and inspection of checkpoint state.

Metropolis: Our Approach to Serverless Containers. Metropolis uses the Aurora single level store's checkpointing system with persistence disabled and develops a new set of optimizations that accelerate restore and execution of serverless functions. This requires a significant engineering effort in optimizing the restore path, which is not performance critical for single level stores.

Metropolis introduces fine-grained sharing to save resources on-disk and in-memory while reducing cold start latency by reducing IO. Metropolis benefits from the view that dependencies are independent objects in the SLS and uses introspection to achieve this fine grained sharing. Unlike previous systems, Metropolis achieves an implicit common warm image without added OS state complexity.

Metropolis adds *lightweight cloning* to SLS objects to enable sharing. On-disk objects have copy-on-write semantics so functions can safely share dependencies without being able to modify each other's data. Similarly, when the object is mapped in memory the pages are shared copy-on-write, and any private modification breaks sharing to ensure data integrity. This occurs at the page granularity and is transparent to the running function instances.

The performance benefits of sharing subsumes any need for paging strategies, because function private state is typically small and often part of the working set. The remaining working set is in shared objects (code or data) that benefit from sharing. Hence, the *collaborative warmup* technique

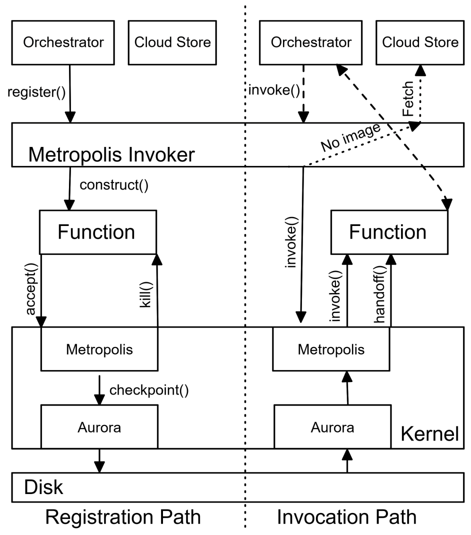

Figure 1. Overview of the Metropolis platform. The invoker on the local machine registers new functions by setting up the container instance, which automatically triggers a checkpoint after it is setup. On invocations it retrieves the image from the cloud if needed, restores it, and hands off a socket to it to directly receive arguments from the orchestrator. Metropolis applies uses introspection and sharing (§ 3) during the restore process to accelerate sharing.

amortizes pagein cost for shared objects across all functions using the object.

Metropolis further uses OS introspection for novel optimizations. For example, to mitigate networking costs Metropolis injects an already connected socket into a restoring function to allow direct communication to the client. This skips the need for the invoker to behave as a reverse proxy and the invoker uses OS to monitor the function's execution.

3 Design

3.1 Overview

Metropolis has three main design goals:

Low Latency: Servicing user requests with low latency is key for serverless frameworks. As serverless function chains grow they become more sensitive to high tail latencies [23], limiting their size. Low latency is crucial to enable developers to decompose applications into functions without incurring large overheads.

Transparency and Generality: Optimizations and mechanisms should support arbitrary runtimes and languages. Metropolis should snapshot and invoke serverless containers without the developer writing code for either operation.

High Image Density: High on-disk image density improves resource use on the local node and indirectly benefits performance. On-disk density minimizes overall application response latency by reducing the high latency IO done to cloud storage. Ensuring high image density in the local node increases the overall efficiency of the node.

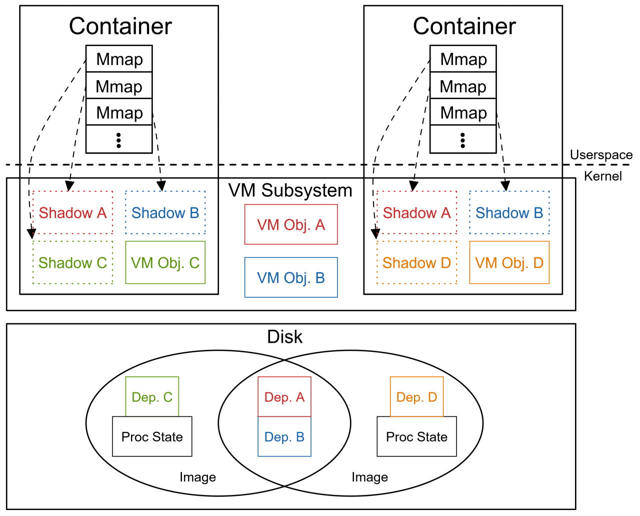

Figure 2. The Metropolis sharing mechanism.

Metropolis is a single node platform for fast serverless invocations It is comparable to invokers like containerd [9], and provides an API to cloud level orchestrators for invoking functions on the node. Metropolis delegates the placement, scheduling and routing of functions to orchestrators like Kubernetes and OpenWhisk. The cloud orchestrator communicates with an individual instance of the Metropolis invoker. Figure 1 shows the overall process of how functions are used within Metropolis.

Metropolis uses FreeBSD jails for OS containers, extended to conform to the OCI standard [6]. Metropolis natively supports Linux and FreeBSD containers, and arbitrary language runtimes. Metropolis runs FreeBSD binaries natively, and Linux applications using the Linux system call emulation layer [2]. This allows the deployment of Linux containers on FreeBSD in the same way Joyent [14] provides Linux containers on OpenSolaris. Functions either use Metropolis-provided language runtimes or provide their own.

Metropolis' uses an introspection-oriented design for performance, generality, and density. We use a fine-grained image format which describes functions as sets of OS primitives, and which allows us to use introspection during snapshotting and invocation. We take advantage of code patterns common in serverless to transparently trigger the snapshot creation process (§ 3.2). On invocations Metropolis individually restores each dependency and maps it into the function's address space (§ 3.3). Metropolis uses introspection to safely share dependencies between functions (§ 3.4). Functions share the read-only pages of dependencies, providing them with enough locality to keep them in memory and *speeding up each other's invocations.* Finally, Metropolis uses introspection to optimize the restore paths of OS primitives like vnodes and sockets for serverless by overloading their deserialization procedures (§ 3.5).

3.2 Triggering Snapshotting/Invocations

Metropolis uses introspection to register and invoke functions without coordinating with function instances or requiring a custom runtime. Programmers write a function as they would a serverful application. Metropolis functions are runnable as stateful servers without modification. Newly invoked instances start executing as if they just connected to a client, and receive their arguments through a connected socket. Metropolis thus simplifies the function development process and does not require adding runtime dependencies to the image for communicating with the invoker.

The invoker uses introspection when registering a function to snapshot the instance when it has finished setting up. We observe that serverless functions share a common code pattern, and use it to trigger snapshotting immediately when the instance is ready. Functions first set up their runtime, then wait for incoming clients. For this reason Metropolis treats an `accept` call from the container as a sign that the instance is fully set up. Metropolis ignores `accept` calls made on Unix domain sockets and IPs bound to localhost so as to not interfere with IPC done during setup.

Metropolis tracks calls by overloading the invocation's private system call vector. The FreeBSD kernel associates a system call vector with each process, and uses these vectors to emulate the ABI of other platforms. For example, FreeBSD runs unmodified Linux binaries by backing them with a system call vector which implements the Linux ABI. Metropolis uses its own system call vector with the `accept` system call replaced with code to call the snapshotting API.

Metropolis uses userspace logic to handle functions which do not call `accept` immediately. Some runtimes, like Node.JS, use `poll`, `select`, or `epoll`/`kqueue` to monitor a socket file descriptor, and call `accept` once a connection is ready. The Metropolis invoker forces such runtimes to call `accept` at registration time by calling `connect` to the serverless function's listening socket. The Metropolis kernel component relays to the invoker which network address to connect to by interposing on the function's `bind` calls.

Metropolis interposes `accept` calls in the kernel to ensure that tracking is general, efficient, and simple to implement. Metropolis handles runtimes like Go and Rust which issue system calls directly instead of using `libc`, and so cannot be intercepted with library interposition. We do not modify existing kernel paths, as we only use the Metropolis system call vector with serverless functions. Metropolis handles functions which use either TCP or UDP sockets to receive requests. For functions using UDP Metropolis uses the interposition mechanism to trigger snapshotting by tracking `recv` calls instead of `accept`.

3.3 Fast Address Space Creation

Metropolis safely backs function address spaces with image data using a copy-on-write (COW) mechanism to prevent modifications to the image. The underlying SLS normally destroys application images when restoring the application as it directly maps the image into the restored processes' address spaces. Metropolis adapts an existing COW primitive used by `fork` for COW between a function instance and its image. Instances retrieve image data by triggering the unmodified kernel page fault handler.

Metropolis avoids existing mapping techniques like CRIU's `userfaultfd` because they incur large overheads on minor page faults. Handling page faults in userspace has a latency of 6 *us*, not including the time it takes to copy over data from the image. By comparison, the kernel page fault handler has a total latency of 1 *us*. Even a no-op Python function takes 1000 minor faults during execution, so the overhead from minor page faults quickly adds up.

We build on top of existing virtual memory abstractions to implement the Metropolis invoker. FreeBSD's Mach-derived [25] VM subsystem does not directly attach pages to address space mappings. It instead backs mappings with groups of mappable physical pages called VM objects. The kernel maps data in a process by associating a VM object with an address range in the process' address space. The page fault handler populates the process' page table on a fault by finding the address mapping for the address, accessing the backing VM object, then retrieving the physical page. Metropolis images use the VM object abstraction to represent function data as a set of VM objects both on-disk and in-memory. Metropolis brings in function data by triggering paging from the image the VM object.

Metropolis applies COW between the function and the image through object shadowing, a mechanism used internally by `fork`. In object shadowing, top-level objects called *shadows* directly back process mappings and are themselves backed by a *parent* object. The parent holds all COW shared pages, while the shadow holds pages with data private to the process. On a page fault, the handler first searches the shadow VM object then the parent. On a page write, the handler creates a copy of the page and adds it to the shadow for the process to write to. Metropolis uses parent objects to represent function mappings in the image as parent object data is never modified.

Metropolis uses introspection to back function instances with image data in a COW fashion efficiently with minimal overhead at invocation time. Metropolis creates a parent object for each VM object in the image, and a shadow for each function invocation. The objects are initially fully unpopulated as to not waste IO by preemptively paging in data. Metropolis always pages in data to the parent object, and then creates a function-private copy on demand. Metropolis makes use of the Aurora pager to access the image on disk.

Metropolis applies COW with minimal overhead at invocation time and runtime. Metropolis creates shadows, parent objects, and mappings concurrently at invocation time for minimal overhead compared to using `fork` and `mmap`. The invocation uses the kernel page fault handler for minor page faults during execution, leading to minimal overhead. There are no inconsistencies in the invocation's VM state, and to the kernel it looks as if we set up the instance from scratch without snapshotting.

Metropolis uses the fine-grained mapping mechanism for function autoscaling. A parent VM obejct can have multiple shadows, so VM objects across instances of the same function share parent objects underneath. Instances thus share image data using COW, and if one instance brings in a page then other instances may reuse it to avoid disk IO. Metropolis shares through VM object shadowing, so no page table sharing takes place.

Metropolis uses the pager to deploy paging strategies common in serverless snapshotting. For eager paging Metropolis forces the pager to bring in in all image data from the disk at restore time. For lazy restores Metropolis uses the pager to retrieve data on-demand during execution.

We use Metropolis' fine-grained image format to implement sophisticated paging policies with minimal effort. We implement the record-and-prefetch (REAP [40]) algorithm which traces function executions to identify the working set of a function. Metropolis lazily invokes a function, and tracks the pages which the pager brings in during execution. Metropolis tracks page faults without adding latency to the running function. It then eagerly brings in these pages on subsequent invocations to avoid major page faults.

3.4 Collaborative Warmups

Metropolis uses introspection to expand shadowing so that if functions share a dependency, they COW share the dependency's image in memory. Sharing dependencies between functions significantly improves invocation latency because common dependencies are used often enough to keep them in memory. Individual functions do not exhibit good locality, so keeping whole images in memory is impractical. Metropolis' fine-grained image format allows it to cache only popular image components, e.g., dependencies, instead of whole images.

Metropolis uses object shadowing to share common dependencies in a fine-grained manner. COW sharing memory is normally only possible using `fork`, which works on the whole address space and incurs significant latency. Metropolis uses object shadowing to back shadows in multiple instances with the same parent object, so that the instances share the parent object's data. Object shadowing works with VM objects, allowing Metropolis to apply COW between functions for a single dependency.

Sharing allows functions to *warm each other up* using the same parent object to page in image data. The pager brings in image data from the disk and into the parent object, so if two instances request the same page only the first one will trigger an IO. Metropolis keeps or evicts function data using the VM page cache to track how warm each image page is.

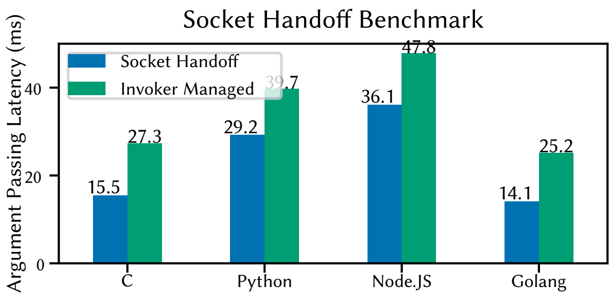

Figure 3. Performance gains for the socket handoff optimization for simple functions echoing a 6MB argument. Removing the invoker from the path between the orchestrator and the function reduces communication overhead by 45% for compiled languages (C, Go) and 25% for the Python interpreter and Node.JS JIT.

Commonly used dependencies like libraries are kept warm by multiple invocations, while function-private data quickly becomes cold and is evicted.

Sharing provides significant performance benefits because serverless functions are small, and dependencies are thus a major part of the total code. Many dependencies are also large and popular, e.g., NumPy for Python functions. Popular dependencies like OpenCV or even libc are even used by functions written in different languages through bindings. On an invocation Metropolis thus need only bring in private function data, which requires minimal IO.

Metropolis implements sharing on the disk by expanding function image semantics to share components, allowing components to be referenced by multiple images. Aurora images are self-contained to make checkpointing large stateful applications simpler. Aurora restores few applications at once, and that only after a power failure. Metropolis invokes functions constantly, executes invocations concurrently, and does not depend on fast checkpoints.

Metropolis sharing reduces both memory and disk usage, improving the overall performance of the node. Increasing memory density allow Metropolis to run more functions concurrently in a single node without risking paging. High on-disk density ensures that Metropolis rarely retrieves image data from the cloud store, which induces large latency.

Metropolis collaborative warmups work at a fine granularity without using a common base image for functions. Common base images prevent the developer from specifying their own custom root file system, and allow sharing only for dependencies already in the image. Using a base image for sharing also does not support multiple conflicting versions of the same dependency, which Metropolis effortlessly handles. Our techniques are not specific to container approaches. Paravirtualization enables sharing in virtualized environments using techniques similar to Disco [20].

3.5 Using Introspection

Socket Handoff. Socket handoff is a technique for reducing the overhead which the instance incurs when downloading large arguments. With this technique Metropolis reduces latency using introspection to connect invocations to the cloud orchestrator. The orchestrator sends large arguments directly to the function and thus bypasses the local invoker. Argument passing incurs significant latency costs, so optimizing it with socket handoff reduces overall function latency.

The orchestrator triggers socket handoff when it connects to a machine with a local Metropolis invoker. The invoker waits on a listening socket for incoming requests from the orchestrator. Once a request arrives, the invoker creates both a new connected socket and a new function instance. Metropolis passes the connected socket directly to the new instance, which then receives its arguments directly.

We implement socket handoff using introspection to pass the connected socket into the instance. Metropolis snapshots functions in the middle of an `accept` call. When it restores an instance it passes it ownership of the connected socket by attaching it to its file descriptor table. It then adjusts the instance's thread state to pass the file descriptor to the connected socket to userspace. Metropolis does this by passing the file descriptor as a return value to the thread which was waiting on the socket for arguments. The thread returns to userspace as if it finished an `accept` call, complete with the file descriptor of the new socket..

Socket handoff allows the local Metropolis invoker to track incoming requests and gather statistics. Functions close the connected socket every time they finish execution, and persist to serve warm invocations. On a subsequent request the invoker uses socket handoff again to create a new connection and again pass it to the function as a return value to `accept`. The invoker thus still counts incoming requests for autoscaling and load balancing. Establishing a connection on every new invocation does not add significant latency because the connection is between the orchestrator and the local invoker.

We demonstrate the performance improvements of socket handoff in Figure 3. We run four language echo servers each written in a different runtime, and send an argument from a client using TCP either through the invoker or directly using socket handoff. The server echoes the argument back to the client after receiving it. The argument is 6MB in size, the maximum amount AWS allows for direct argument passing. We measure the latency from the client.

Our results show that socket handoff provides a significant performance improvement over passing function arguments through the invoker. For the C and Golang which are combined there is a 45% reduction in latency, while for the Python and Node.JS the overhead reduction is 25%. Interpreted and JIT'ed runtimes incur overheads during warmup, which also explains why they have larger latency overall compared to C and Go. The latency which argument passing adds is large, 27.3 *ms* when passing through the invoker and 15.5 *ms* with socket handoff.

Introspection on Restore Paths. Metropolis uses the ephemeral nature of serverless functions to make operations like restoring memory mapped files faster. Mapping vnodes is traditionally an expensive operation relative to rest of the restore path. It require multiple operations, including opening the vnode and updating the access time on-disk.

Metropolis' ephemeral file system mappings allow us to avoid unnecessary restore operations. Metropolis uses introspection to replace files opening and maping routines with fast paths that only take a vnode reference. These optimizations improve the Python helloworld benchmark by an order of magnitude from 3.2 ms to 0.3 ms.

3.6 Security Performance Considerations

Metropolis' threat model depends on whether the provider collocates containers belonging to different users in the same (physical or virtual) machine. The threat model in a cloud which collocates user containers is an attacker who cannot escape the containerization mechanism. Users who to deploy serverless functions on infrastructure they own have as a threat model an attacker who cannot escape a virtual or physical machine.

Using interimage sharing mechanism with collocated users allows different users to infer whether library pages are in memory and thus have been recently used. Metropolis' default behavior is to not share security-critical dependencies, e.g., OpenSSL. Metropolis allows programmers to further break sharing for sensitive dependencies (§ 3). The Metropolis sharing mechanism is otherwise analogous to `fork`, and does not share page tables.

4 Implementation

Metropolis is built as a kernel module on FreeBSD 12.3 in addition to the Aurora code. Metropolis includes the shadowing mechanism, system call interposition layer, and code for sharing and collaborative warmups written in 2.8k SLOC of kernel code. Metropolis contains 1.2k SLOC that implement eager, lazy and REAP's paging strategies. Metropolis adds a 200 SLOC kernel patch to expose fast vnode mapping. Metropolis' userspace invoker consists of 675 SLOC.

5 Evaluation

Our evaluation focuses on two aspects of invocation performance: latency and density. We quantify the benefits of, *fast invocations*, *collaborative warmups*, and *socket handoff*. We break down latency into restore latency, paging latency, and execution time to explain the performance trade-offs made by different systems. Throughout the evaluation we use FunctionBench [29], a popular serverless benchmark that provides representative workloads.

	C	Python	NodeJS
Process creation	45 μs	43 μs	312 μs
Address space setup	87 μs	227 μs	481 μs
File table setup	121 μs	122 μs	256 μs
Other Metropolis	119 μs	110 μs	119 μs
Execution	501 μs	1397 μs	5892 μs
Retired instructions	0.7M	17.6M	28.5M
iTLB misses	0.2K	1.1K	7.1K
dTLB load misses	0.3K	5.1K	9.7K
dTLB store misses	0.1K	0.9K	1.1K
L2 misses	2.4K	63.0K	39.6K
User time (us)	0	6456	10536
System time (us)	510	2944	8035
Minor faults	30	1063	1028
Major faults	0	0	0

Table 2. Shows the breakdown of total response time into restore time and execution time for helloworld across three language runtimes.

We analyze Metropolis' invocation latency from memory and compare it to that of CRIU. Metropolis restores outperform those of CRIU by an order of magnitude and add minimal overhead to the invocation. We compare paging strategies with collaborative warmups, and measure their effect when running similar functions together. Instances of different functions warm each other up and provide a significant reduction in latency.

We measure image density both on disk and in memory. On-disk images include the entire file system state and the resumable function image. We show how Metropolis minimizes the size of on-disk images using sharing. In-memory images include only the function image and any files accessed during execution. We compare with the differential checkpointing method used by SEUSS.

We compare Metropolis' restore latencies to those of Catalyzer, is the state of the art in snapshot-based invocations. Metropolis outperforms Catalyzer's `sfork` by up to 3×. We further compare Metropolis against VM based systems using vHive.

For all measurements we use a server with dual Intel Xeon Silver 4116 CPUs (Skylake-SP) running at 2.1 GHz, 96 GiB of memory, and an Intel X722 10 Gb network card. The machine has hyperthreading enabled and turbo boost disabled. We use two 280 GB Intel 900P NVMe SSDs in a striped configuration with a 64 KiB stripe size as our storage. All experiments are the average of five or more runs and error bars are included where possible.

5.1 In-memory Invocation Times

Our first experiment shows that deserializing the image and creating the container adds minimal overhead to the end-to-end response time. We break down the end-to-end latency of three echo servers (i.e., helloworld from FunctionBench) written in C, Python, and Node.js. In all experiments we measure the end-to-end response time from a remote machine. We restore the function from a serialized in-memory image to isolate the cost of paging in the memory of the function.

Table 2 shows the latency for functions written in C, Python, and JavaScript. By comparing these three servers we quantify the restore and execution overheads caused by language runtimes. C is compiled and has a minimal runtime making it the fastest of the three. Python is interpreted and the process address space includes state related to the language interpreter. Node.js is a very complex runtime with 12 threads, 250 memory mapped regions, and 20 open pipes, kqueues and file descriptors.

Table 2 is broken into three parts. The first part breaks down the latency of the restore path until the function starts executing. The second part contains several microarchitectural performance counters including L2 misses and TLB misses to better understand execution latency. The last part contains OS accounting of user time, system time and page faults as measured by `getrusage`. For these simple workloads, the time spent in the kernel is from fixing up page tables and responding to the client through the socket.

The C function is the fastest of the three, with an end-to-end latency of around 800 μs. Restoring takes less than half that time, with most of it spent recreating the address space and open file descriptors. OS metrics show that the function spends all of its time in the kernel, because it executes `read` then `write` after resuming execution. The function completes after executing a few thousand instructions in userspace, which is far below a microsecond.

Python's larger address space and runtime lead to slower function restores (~500 us). The Python interpreter's larger code size and working set causes a ~150 us increase to restore latency. Conversely, the execution time is 3× that of the C function because the interpreter executes more code and thus induces more cache and TLB misses.

Node.js is the slowest and most complex application of the three. It takes 1.2 ms to restore Node.js, while its execution time is 5.9 ms. Even though the function is 3.5× slower than the Python function, both functions have a similar amount of TLB and cache misses. Python has 1.5× the L2 misses fo Node.js. The amount of instructions executed also does not merit the added execution time. The slowdown in Node.js is because of the disproportionate amount of iTLB misses compared to code size caused by JIT compilation.

The results show Metropolis is not bottlenecked by the restore mechanism. The mechanism scales well with function complexity, and the main overhead lies in function execution. Metropolis' snapshot invocation performance is sufficient to not contribute to the overhead.

Invocation Latency Comparison to CRIU. Figure 4 compares the restore times of the Metropolis and CRIU invokers for five language runtimes and for the Functionbench workloads. We snapshot each running server after its setup

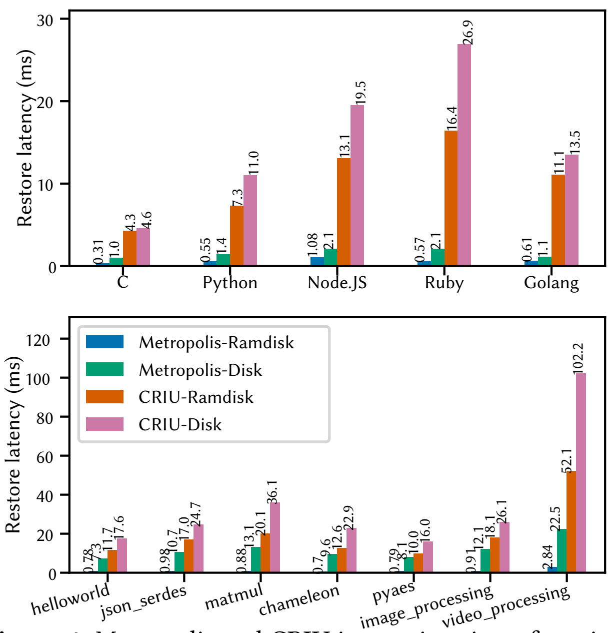

Figure 4. Metropolis and CRIU invocation times from images on a ramdisk and physical disk.

is complete, then measure the time it takes for CRIU and Metropolis to invoke it from the image. We use two configurations, one where the invoker retrieves image from a ramdisk, and one where it retrieves it from backing disk storage. We use eager restores for all configurations, which incurs paging related overhead at invocation time to compare the paging and deserialization costs of invocations.

Metropolis consistently outperforms CRIU by 12.5×–20× because of its mapping mechanism and introspection-based restore path. Metropolis invokes faster from on-disk images than CRIU does from a ramdisk. For CRIU, image deserialization is as costly as retrieving the image from the disk and dominates invocation time for small functions. Metropolis deserialization costs are below a millisecond for most workloads except Node.JS and `video_processing`. For these benchmarks, deserialization cost is 10% the total invocation time, so paging still dominates.

5.2 Paging Strategies

With deserialization times minimal, the major remaining source of restore overhead comes from paging in the memory of the application (i.e., anonymous memory, binaries, libraries, and data). Metropolis eliminates most of the paging latency using *collaborative warmups*.

This second experiment explores two trade-offs in the design space that we can make to optimize paging. First, paging in immediately or lazily paging in as the workload needs. Serverless is unlike other workloads because functions have a short lifespan. There is little benefit from lazy paging strategies, if the function uses most of the memory. Lazily paging often pays an increased overhead because the process incurs page fault latencies and does fewer smaller IOs than bringing in the entire process.

The only reason we may see benefits is when the function does not use it's entire image, which brings us to our second trade off. Selectively subsetting the image to bring in only the pages we need. Prefaulting strategies do this by tracing the function execution once and only paging in the used pages in subsequent calls.

We compare the effectiveness of different paging strategies for invocations from on-disk images for efficient cold starts. We measure the application response time of the FunctionBench workloads from a remote host for each paging strategy. Strategies include: *Eager* that pages in all memory during restore time; *Lazy* that does not page in at restore time, and pages in as needed during execution; *Prefault* uses a trace to preemptively page in data used on prior executions (i.e., a reimplementation of REAP [40]); and *Metropolis* that pages in anonymous pages at restore time and depends on collaborative warmups for vnode pages. We compare these strategies with restores from in-memory images, which do not do any paging. We flush the buffer cache between invocations, except for the Metropolis and memory strategies.

Figure 5 shows the results for each benchmark broken into restore and execution time. As we outlined earlier lazy restores are often the slowest, because of the short lifespans coupled with the need to page in a large fraction of the image. Each page fault incurs additional overhead in the page fault handler to bring in a few pages at a time. The function will be suspended by the operating system until the page is brought in, which adds to the application response time.

Generally eagerly paging in the entire image is better than lazy because of the characteristics of serverless environments. The `matmul` and `video_processing` workloads have a large application image with only a portion required for execution resulting in slower eagar restores.

Prefaulting often performs similar to eager restores for the same reason that much of image is used. Metropolis function images do not hold much unused function state like VM snapshots do. Large file system IO sizes prefetching amplify the total amount of data brought in when paging in libraries. Prefaulting thus avoids only a minor amount of IO for Metropolis images.

Metropolis is close to the performance of the in-memory restores, because the collaborative warmups reduces the total IO beyond whats possible with REAP's tracing. Strategies like collaborative sharing which keep libraries warm are critical for high performance.

Benefits of Collaborative Warmups. We breakdown the pairwise benefit of collaborative warmups across any two FunctionBench workloads. In each case, we flush the buffer cache, then invoke a warmup function eagerly. We then invoke a different function and measure its application response time. Each invocation brings in anonymous pages,

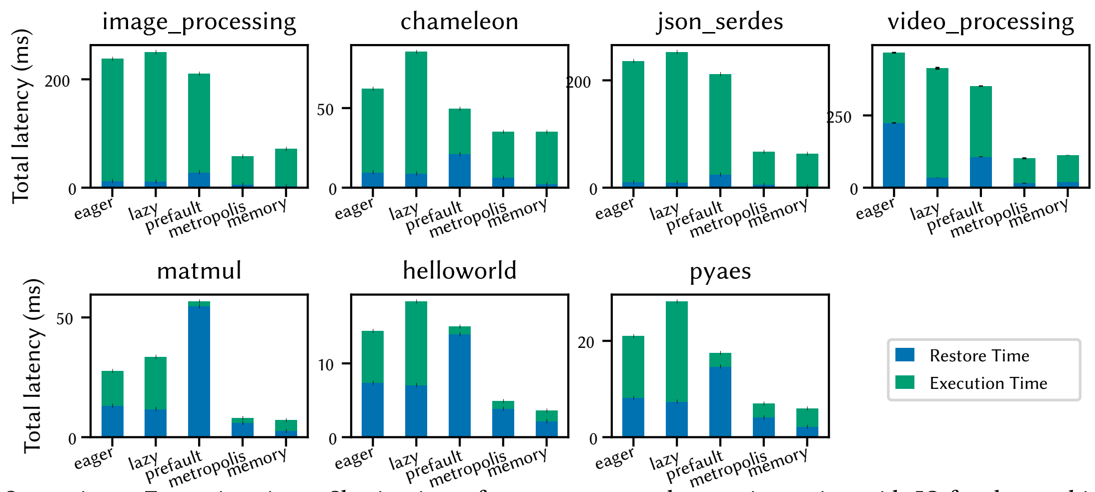

Figure 5. Paging Strategies vs Execution times. Sharing is performant compared to tracing as it avoids IO for the working set.

and warms up the cache for vnodes. If the vnode is already populated, the operation is a no-op.

Table 3 shows that even very different functions benefit from collaborative warmups. The functions do not share major dependencies, e.g., only `video_processing` uses OpenCV. Functions are invoked faster than eager restores in the presence of other non-trivial functions. For example, executing `image_processing` after `json_serdes` confers a 2× speedup. The source of this speedup is the inclusion of system libraries and the python runtime. Shared objects like `libc.so` are almost universal and reduce IO significantly.

5.3 Disk Density

Table 4 shows the space consumed by Metropolis image and the density we can pack of a given FunctionBench application per tebibyte. Metropolis images consume a fraction of the total amount of file system state accessible to the image. We reduce image size by 30–99% for up to 100× higher density, depending on the function's dependencies. The most complex applications (machine learning and video processing) have the highest savings.

All the dependencies of the FunctionBench workloads use common packages, e.g., OpenCV, Scikit, and Pandas, which are likely to be shared in any real production system. Metropolis shares these packages between images, so each has a constant overhead for the entire machine.

5.4 Sustained Invocation Throughput

Figure 6 shows the scalability of invocations per core of helloworld across two SSDs on a single socket. This is our worst case application as the helloworld function executes for just over 4 ms. Disk restores consume about 80% of the bandwidth of the SSDs before becoming latency bound. Memory restores contend on coarse grained locks that we have not optimizes, but we still saturate 20 cores with helloworld. Any other function would easily saturate the machine.

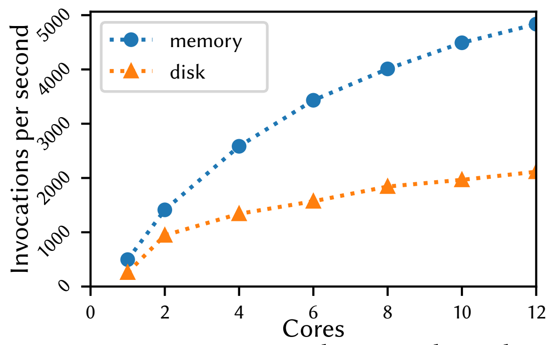

Figure 6. Invocations per second on a single machine with two PCIe SSDs. The disk restores achieve ~80% of the SSD throughput, because of some latency bound operations in our storage code. The memory case seems to be limited but requires 12 cores just processing helloworld to saturate.

5.5 Comparison to Other Systems

Finally, we compare against three state of the art systems. First, we compare against a reimplementation of SEUSS's delta checkpoints on Metropolis. Second, we compare against vHive with REAP running on Linux and show even after removing virtualization overheads we outperform them confirming our reimplementation of REAP. Third, we compare Catalyzer's sfork against Metropolis' fully cached restores.

SEUSS's Delta Checkpoints. In this experiment we demonstrate that incremental checkpointing is not effective in saving space for container based serverless functions. Incremental checkpoints are well established for unikernels, which hold OS and even file system related data in the function image. Only a small fraction of these images is actually taken up by the functions themselves. Metropolis in contrast produces lean images that mostly hold by function specific data.

Figure 7 shows the number of pages included in each checkpoint for full checkpoints, SEUSS's incremental checkpoints and Metropolis. Incremental checkpointing is effective for small functions where most of the state is from the Python interpreter.

2nd \ 1st	**helloworld**	**pyaes**	**matmul**	**chameleon**	**json_serdes**	**image_proc.**	**video_proc.**
helloworld	4.5 *ms*	6.0 *ms*	6.0 *ms*	6.0 *ms*	6.0 *ms*	5.9 *ms*	6.3 *ms*
pyaes	8.5 *ms*	6.7 *ms*	8.3 *ms*	8.0 *ms*	8.2 *ms*	8.4 *ms*	8.3 *ms*
matmul	47.5 *ms*	47.0 *ms*	8.4 *ms*	43.1 *ms*	43.2 *ms*	44.2 *ms*	11.0 *ms*
chameleon	39.6 *ms*	39.3 *ms*	36.4 *ms*	34.1 *ms*	35.9 *ms*	36.1 *ms*	35.9 *ms*
json_serdes	202.3 *ms*	202.7 *ms*	197.3 *ms*	199.5 *ms*	66.9 *ms*	71.8 *ms*	71.1 *ms*
image_proc.	198.1 *ms*	199.1 *ms*	193.3 *ms*	194.3 *ms*	67.3 *ms*	57.0 *ms*	66.1 *ms*
video_proc.	459.7 *ms*	463.9 *ms*	427.6 *ms*	458.7 *ms*	336.3 *ms*	333.2 *ms*	101.3 *ms*

Table 3. Shows the pairwise benefit of collaborative warmups across any two benchmarks in FunctionBench. The biggest benefit is for workloads warming up other instances of themselves, which share all libraries and most memory.

	Full		**Metropolis**		**%**
Function	*Size*	*per TB*	*Size*	*per TB*	*Saving*
helloworld	4.9 MiB	214.0 *k*	1.5 MiB	712.0 *k*	69.9%
pyaes	18.5 MiB	56.6 *k*	14.9 MiB	70.4 *k*	19.6%
chameleon	24.3 MiB	43.1 *k*	17.1 MiB	61.4 *k*	29.9%
json_serdes	39.5 MiB	26.6 *k*	18.9 MiB	55.8 *k*	52.3%
image_proc.	131.5 MiB	7.9 *k*	19.6 MiB	53.5 *k*	85.2%
video_proc.	3.0 GiB	0.35 *k*	32.6 MiB	32.1 *k*	98.9%

Table 4. Image sizes and density per TB of storage for FunctionBench functions, using full images versus Metropolis images. The sizes include the function snapshot and all dependencies. Metropolis' on-disk sharing scheme reduces the size of function images by up to 98.9%.

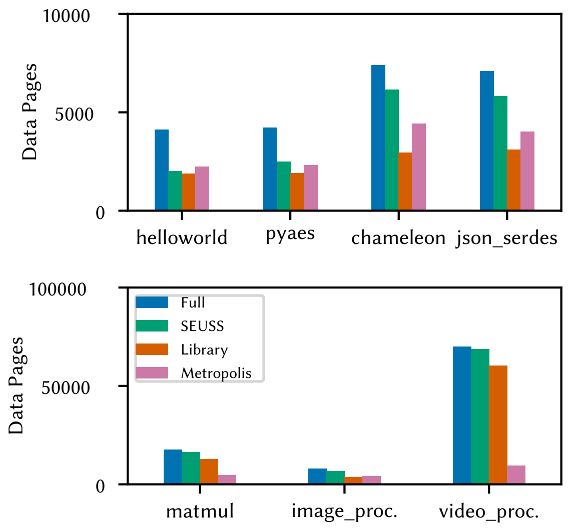

Figure 7. The size of a function's address space in the image. Metropolis' passive sharing technique measurably reduces the total checkpoint size. Delta checkpointing is effective for trivial functions (`helloworld`, `pyaes`), but it cannot reduce the size of large shared libraries in larger workloads like `video_processing`.

The effectiveness of incremental checkpointing decreases as functions become larger, which implies they contain more state private to that function. The larger state includes imported python libraries, native dynamic libraries and initial data in the function. Often in these benchmarks, the dependencies constitute the largest part of the address space, which reduces the benefit of sharing the initial runtime checkpoint.

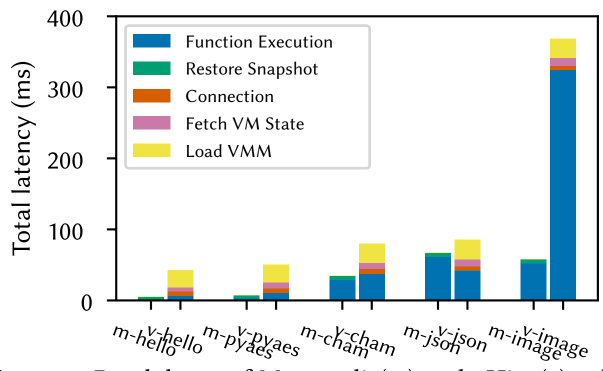

Figure 8. Breakdown of Metropolis(m) and vHive(v) cold start latency for five FunctionBench workloads. Metropolis has better performance both at invocation and execution time, except for `json_serdes` for which Metropolis is using an outdated version of the package.

Metropolis sharing is more effective, but has similar problems. The savings are more significant for small functions, and less for larger ones. The effectiveness of Metropolis sharing only decreases because of the relative increase of the total working set. Metropolis does not share anonymous pages, so its efficiency is not affected by package imports dirtying intepreter related pages.

Metropolis vs. vHive/REAP. Figure 8 shows a breakdown of the invocation latency for Metropolis and vHive with REAP. In this experiment we omit `video_processing` and `matmul`, because vHive did not have a runnable VM for them. We break down the total application response latency into its components to better understand the overheads.

vHive uses Firecracker [17] to run serverless functions inside of microVMs. vHive introduces REAP (Record and Prefetch) that traces a function's working set during an execution. vHive saves it to disk and eagerly restores it on subsequent invocations.

Metropolis optimizes image deserialization to achieve sub-millisecond restore times, and application response times for helloworld and pyaes of an order of magnitude lower than vHive. vHive in contrast, suffers from the fixed latency

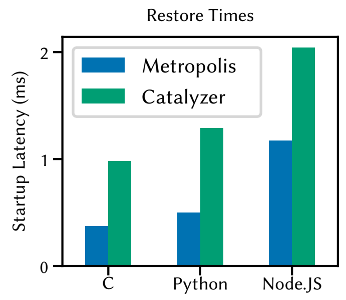

Figure 9. Cold starts for Metropolis snapshot restore modes. Metropolis consistently outperforms Catalyzer by 2–3× due to its optimized deserialization path.

cost of restoring the host-side microVM state. Setting up hypervisor state adds significant fixed latency to the process regradless of benchmark size. The VMM setup operation together with creating the connection between host and guest dominate total latency for smaller benchmarks.

Metropolis' restore time is similar to vHive fetching the function state from the disk, because of the similarity in working sets. Execution times Metropolis and vHive are similar, with Metropolis taking 5 ms less than vHive. This cost is due to vHive having to bring in more state that Metropolis shares many of the shared libraries and runtime state without needing to do IOs or page faults.

Metropolis outperforms vHive on larger benchmarks such as `image_rotate`. Long benchmarks have more opportunity for sharing and unpredictable access patterns across invocations, making prefaulting ineffective. The vHive outperforms on the JSON serdes benchmark because it is using a newer version of the serdes package.

Catalyzer sfork vs. Metropolis's Cached Restores. Finally, we compare our restore times against Catalyzer, the fastest known serverless framework in the literature. We specifically compare against Catalyzer's sfork mechanism that restores from a fully cached in memory image. The sfork mechanism caches initialized kernel data structures and uses them to create new instances from memory with. We compare against the fully cached Metropolis restore case. Figure 9 shows the results for Hello World programs written in C, Python, and Node.js.

The results suggest that Metropolis restores are faster than Catalyzer when doing so from memory. Catalyzer faces gVisor specific obstacles that add overhead to its `sfork` primitive, like serializing and deserializing thread state to implement multithreaded `fork`. Metropolis solves these issues at the system instead of the language level.

As Catalyzer is closed source, we note that they're results where on hardware superior to ours. The Catalyzer sfork does no IO so disks play no role. It was run an 4-core/8-thread Intel i7-7700 CPU running at 3.6 GHz versus our Intel Xeon 4116 CPU running at 2.1 GHz. Metropolis is not multithreaded and is does not benefit significantly from the increased CPU cache. Catalyzer benefits from the clock speed advantage of their hardware.

6 Related Work

OS containers are well established for serverless sandboxing. SOCK [32] expands on the OpenLambda [27] framework to optimize container creation throughput and latency. SOCK solves the Linux-specific scalability problems and focuses on Python. The authors report latencies of 100 ms for no-op functions when not using Zygotes, compared to Docker's 130 ms. RunD [13] similarly solves scalability issues for Linux with setting up Kata containers, and optimizes VM sandbox snapshot restores.

Function Zygotes are a popular technique that creates new functions by forking them out of template containers instead of constructing them from scratch. This technique improves invocation latency by skipping parts of the initialization process and is applicable to OS containers and sandboxes like gVisor [1]. SOCK uses Zygotes with popular dependencies already in the container to avoid having to install them at invocation time. This method further lowers invocation time to 10 ms. Catalyzer [24] brings the Zygote technique to gVisor sandboxes for restore times of 1.2 ms for Python no-op functions. Pagurus [31] turns containers of finished functions back into Zygotes to avoid creating.

Snapshot based restores have been optimized for various languages and runtimes. Replayable Execution [41] provides snapshot-based invocations for Java-based functions. Faasm [36] uses snapshotting for WASM based functions. Fireworks [37] optimizes snapshotting for functions with JIT runtimes. FaasSnap [18] optimizes Firecracker-based snapshot restores and expands on the REAP algorithm [40] for faster cold starts.

Various work uses optimized runtimes to improve execution performance for warm functions. Nightcore [28] enables microsecond latencies for functions using language based isolation. Medes [33] introduces deduplicated serverless containers for storage density, which are running containers whose data Pocket [30] provides a fast ephemeral storage stack for analytics-oriented functions.

7 Conclusion

We presented Metropolis, a serverless framework that uses sharing to improve function density and cold start times for container based serverless functions. Metropolis shares common dependencies between functions in memory and on-disk to improve density in the node. By maintaining sharing across snapshots we improve startup latency by reducing overall IO needed. Our experiments show that sharing is a powerful technique that simultaneously optimizes cold start invocations, memory and disk density.

References

[1] gvisor: Application kernel for containers. https://github.com/google/gvisor. Accessed: 2022-07-03.

[2] Linux binary compatibility. https://wiki.freebsd.org/Linuxulator. Accessed: 2022-07-03.

[3] CRIU website. https://www.criu.org/Main_Page, April 2019.

[4] Apache Openwhisk. https://openwhisk.apache.org/, December 2020.

[5] IBM Cloud Functions. https://cloud.ibm.com/functions/, December 2020.

[6] Open Container Initiative. https://opencontainer.org/, December 2020.

[7] Fission: Serverless Functions for Kubernetes. https://fission.io/, April 2021.

[8] Fn Project The Container Native Serverless Framework. https://fnproject.io/, April 2021.

[9] containerd An industry-standard container runtime. https://containerd.io/, October 2022.

[10] Home - Knative. https://knative.dev/docs/, October 2022.

[11] Official Red Hat OpenShift Documentation. https://docs.openshift.com/, October 2022.

[12] Oracle Cloud Infrastructure. https://cloud.oracle.com, October 2022.

[13] RunD: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, Carlsbad, CA, July 2022. USENIX Association.

[14] SmartOS - Joyent. https://www.tritondatacenter.com/smartos, October 2022.

[15] What is Azure Container Instances? https://learn.microsoft.com/en-us/azure/container-instances/container-instances-overview, October 2022.

[16] What is Dell APEX? https://www.wwt.com/article/what-is-dell-apex, October 2022.

[17] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, 2020.

[18] Lixiang Ao, George Porter, and Geoffrey M. Voelker. Faasnap: Faas made fast using snapshot-based vms. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 730–746, New York, NY, USA, 2022. Association for Computing Machinery.

[19] Marc Brooker, Adrian Costin Catangiu, Mike Danilov, Alexander Graf, Colm MacCárthaigh, and Andrei Sandu. Restoring uniqueness in microvm snapshots. *CoRR*, abs/2102.12892, 2021.

[20] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. *ACM Trans. Comput. Syst.*, 15(4):412–447, November 1997.

[21] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. Seuss: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 2020.

[22] Fernando J Corbató and Victor A Vyssotsky. Introduction and overview of the multics system. In *Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I*, pages 185–196, 1965.

[23] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Communications of the ACM*, 56:74–80, 2013.

[24] 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*, pages 467–481, 2020.

[25] David B Golub, Daniel P Julin, Richard F Rashid, Richard P Draves, Randall W Dean, Alessandro Forin, Joseph Barrera, Hideyuki Tokuda, Gerald Malan, and David Bohman. Microkernel operating system architecture and mach. In *In Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 11–30, 1992.

[26] Runsheng Guo, Victor Guo, Antonio Kim, Josh Hildred, and Khuzaima Daudjee. Hydrozoa: Dynamic hybrid-parallel dnn training on serverless containers. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 779–794, 2022.

[27] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless computation with openlambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver, CO, June 2016. USENIX Association.

[28] Zhipeng Jia and Emmett Witchel. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. 2021.

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

[30] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, Carlsbad, CA, October 2018. USENIX Association.

[31] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. Help rather than recycle: Alleviating cold startup in serverless computing through Inter-Function container sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 69–84, Carlsbad, CA, July 2022. USENIX Association.

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

[33] Divyanshu Saxena, Tao Ji, Arjun Singhvi, Junaid Khalid, and Aditya Akella. Memory deduplication for serverless computing with medes. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 714–729, New York, NY, USA, 2022. Association for Computing Machinery.

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

[35] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A Fast Capability System. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, pages 170–185, New York, NY, USA, 1999. ACM.

[36] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. *arXiv preprint arXiv:2002.09344*, 2020.

[37] Wonseok Shin, Wook-Hee Kim, and Changwoo Min. Fireworks: A fast, efficient, and safe serverless framework using vm-level post-jit snapshot. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 663–677, New York, NY, USA, 2022. Association for Computing Machinery.

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

[39] Emil Tsalapatis, Ryan Hancock, Tavian Barnes, and Ali José Mashtizadeh. The aurora single level store operating system. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*,

SOSP '21, page 788–803, New York, NY, USA, 2021. Association for Computing Machinery.

[40] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. *Benchmarking, Analysis, and Optimization of Serverless Function Snapshots*, page 5592̌013572. Association for Computing Machinery, New York, NY, USA, 2021.

[41] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. Replayable execution optimized for page sharing for a managed runtime environment. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.