Alto Snapshot/Restore: Enabling Stateful, Long-Lived Serverless Computation

James Larisch, Alisha Ukani, Vincent Viego
Harvard University
jameslarisch@g.harvard.edu, alisha@college.harvard.edu, veviego@college.harvard.edu

Abstract

As demand for the serverless computing paradigm increases, so does demand for a currently unavailable serverless architecture: stateful, long-lived serverless computation. In this paper, we re-introduce Alto, which provides such an architecture. Using Alto, developers can enjoy the autoscaling and fine-grained billing promised by the serverless platform while also enjoying the performance benefits of co-locating code and local data. This serverless paradigm makes developing existing applications easier and also enables new cloud application architectures. We explain why providing stateful, long-lived serverless computation requires fast snapshot and restore of computation. We explain how Alto virtualizes the managed runtime to provide fast snapshot and restore. We describe how Alto moves filesystem and network state from the kernel into the Alto runtime to improve performance. Finally, we evaluate an Alto prototype against Checkpoint-and-Restore In Userspace (CRIU), a fast process checkpoint-restore tool, and show that Alto snapshot and restore performance is comparable.

1 Introduction

Serverless platforms are becoming increasingly attractive for cloud application developers. In the serverless computing paradigm, tenants pay for computational resources used, rather than for the time computational resources were allocated. The cloud provider is responsible for allocating additional resources to each tenant's application automatically as the application's load increases, and releasing these resources when load drops. In the extreme case, when an application is not in use, the cloud provider allocates no resources and the tenant is not billed.

The most popular instance of serverless computing is Function-as-a-Service; though popularized by Amazon Lambda [2], other offerings include Google Cloud Functions [20] and Microsoft Azure Functions [28] . Using FaaS, developers deploy "functions" which respond to cloud platform-provided events such as HTTP requests, database triggers, and other function calls. Upon receiving an event, the cloud provider is responsible for deploying an "instance" of the function to respond to that event. In response to events, functions are free to create other events, query databases, and respond to the requesting client. Functions instance are, by specification, to be treated as ephemeral and stateless: instances have a maximum execution time (on the order of minutes) and once a function instance is finished executing (or killed), all in-memory state is discarded.

Clearly, the constraints imposed by FaaS make it unsuitable for many types of applications. An image conversion algorithm or ETL (Extract, Transform, Load) pipeline will operate well on an FaaS platform, but data-heavy computation like MapReduce and machine learning algorithms will not. However, FaaS is not the only exemplar of the serverless paradigm. Certain databases and datastores, such as Firebase Firestone [21] and Amazon S3 [3] provide the autoscaling and fine-grained billing properties which characterize a successful serverless platform, though for data storage. Using a serverless database or datastore, tenants pay for the amount of data they actually store and the queries performed on their data, rather than total storage allocated and database server uptime. There is also research on autoscaling databases designed specifically for serverless platforms [25].

Unfortunately there are types of computation not covered by these serverless offerings. Database-as-a-Service, or DaaS is built purely for data storage. Because FaaS instances are stateless, the co-location of code and data becomes impossible; instances are forced to store long-lived data at centralized locations, incurring a latency penalty whenever data is stored or retrieved. Hellerstein et al. [22] note that FaaS is holding distributed architectures back, since modern architectures are realizing the performance benefits of placing code near local data. Numpywren [29] builds a system for performing linear algebra operations, which retrieves intermediate state from S3 at coarse granularities to offset the high request latency. ExCamera [19] develops a distributed video encoder atop Lambda, but is limited by function instance's transience. Since a function instance may be killed at any time, the system must be fault tolerant. Both Numpywren and ExCamera shoehorn function instances into units akin to actors, treating properties like transience and statelessness as obstacles, rather than features.

In this work, we present Alto, a serverless platform that provides *stateful, long-lived* computation. Using Alto, developers can deploy types of serverless applications currently unsuitable for today's serverless platforms. Tenants specify their application once, much like in FaaS, and the platform is responsible for deploying instances of their applications called Clefs. In this new serverless model, Clefs include both code and data. Using Clefs, tenants pay for computation only when Clefs are active and running. However, unlike FaaS,

each Clef's data persists across multiple invocations. When a Clef is not in use, both code and data are saved, together, to disk. When needed, Clefs are restored from disk into memory. Tenants are billed, like FaaS, based on the execution time of their active Clefs. To be truly serverless, Alto must provide the ability to autoscale resources based on perceived load. Alto defines a more flexible approach to autoscaling than FaaS: tenants define criteria which, when met, cause Alto to spawn additional distinct Clefs.

This new serverless model, though it can help existing applications like ExCamera and Numpywren, also enables new, exciting architectures. For example, developing a social network application using Alto would be very different from using traditional deployment mechanisms. Using Alto, the developer would write their application as the code required to handle a *single user* of the social network. The developer specifies *autoscaling criteria*, which instructs Alto to create a new Clef using this code whenever a new social network user joins. In this way, *every user has their own personal Clef*. When the user logs out, their Clef is written to disk, and when they log back in, the Clef is read back into memory. This architecture has a number of benefits:

1. The developer pays based on the number of active users.
2. Because the user's data lives exclusively in in-memory data structures, access to the data is faster than it would be if the data was stored externally. (Note aggregate data should still be stored in serverless databases.)
3. Reasoning about each user's data and the application as a whole becomes much simpler. In particular, *moving* or *deleting* a user's data is as simple as moving or deleting the user's Clef.

In order to provide stateful, long-lived units of computation with high-performance, Alto must be able to snapshot and restore Clefs quickly. Alto could build Clefs atop traditional virtualization options like KVM or Docker, which provide snapshot and restore operations for VMs and containers, respectively. Unfortunately, snapshotting and restoring VMs is slow, since entire memory must be copied. Snapshotting containers is complex, since the underlying mechanism, Checkpoint Restore in Userspace (CRIU) [11], must extract both network and filesystem state from the kernel.

To facilitate fast, stateful snapshot and restore, Alto [26] moves the virtualization boundary to the level of a managed runtime. In this way, the Alto runtime acts as the hypervisor, and Clefs are composed of a running program's code and data that sit atop the runtime. Specifically, an Alto Clef includes the program state (such as the stack, in-memory data structures), the runtime state (such as garbage collector state), and the code. This state is saved to disk when the Clef is snapshotted, and is loaded from disk into the Alto runtime when a process is restored. To avoid the need to examine the kernel for system-specific state, Alto moves both filesystem and network state into the Alto runtime.

Alto presents a multitude of research challenges including multiplexing the runtime, specifying autoscaling criteria, and Clef-to-Clef communication. Here, we focus on just one: achieving fast snapshot and restore for Alto Clefs. Particularly, we demonstrate that both snapshotting and restoring an Alto program which includes both open TCP connections and open files is faster than snapshotting and restoring the containing process and data using Docker Checkpointing, powered by CRIU. (Mention specific result)

By providing low-latency snapshot and restore, Alto represents a new architectural model model that makes and enables new stateful, long-lived serverless computation.

2 Background

In this section we describe important technologies such as serverless computing, virutalization technologies, and snapshotting tools.

2.1 Serverless Computing

Serverless computing broadly refers to the notion of developers being able to write code and upload it to the cloud for deployment, without having to worry about infrastructure like managing servers [22]. The term is also commonly used to refer to *Function as a Service* (FaaS), in which developers upload short functions to run on certain triggers. These computations are short-lived; AWS Lambda, the most widely used FaaS service, may terminate running function instances (called *lambdas*) after 5 minutes [34]. In addition, FaaS does not provide any guarantees about preserving function state, meaning developers must write results to remote storage if they want those results to persist. Another limitation of FaaS is that function composition is difficult; in fact, Baldini et al. have shown that platforms like AWS Lambda cannot support function composition and achieve all three of: being billed only once for each function instance, compose functions without hard-coding one function into another, and ensure that function compositions are themselves functions [5].

A related subset of serverless computing is serverless databases, in which users can access a remote database management system without needing to manage the physical infrastructure. Example services include AWS Relational Database Service [**?**] and Google Datastore [**?**].

2.2 Virtualization Technologies

There are two primary mechanisms of virtualization today: virtual machines and containers. OS-level virtual machines allow all layers of the software stack above hardware to run without being aware of the specific underlying hardware architecture. A virtual machine is managed by a *hypervisor*,

which mediates interactions between guest operating systems and the hardware (e.g. by managing page tables). VMs are widely used because of their strong isolation guarantees; errors in one application cannot break another application becuase both are running on separate operating systems. VMs are widely used for FaaS because of this security guarantee. As Wang et al. describe, coresidency of different users' function invocations on the same VM is dangerous because it leaves users vulnerable to side channel attacks [34].

Linux containers, on the other hand, are groups of one or more processes that have isolated namespaces. As a result, processes in one container cannot address resources like files and PIDs in other containers and in the operating system [17]. Containers are advantageous because they are more lightweight than VMs; however, they also cannot provide as strong isolation guarantees. Docker is one of the most popular implementations of containers.

2.3 Snapshotting Tools

In this section we present tools used to *snapshot* different levels of the software stack. Snapshotting means serializing state into *image files* such that the entity can be restored from the images.

Starting at the highest level of the stack, CRIU is a tool that can snapshot and restore processes [11]. In order to ensure that processes can be restored, it must also capture some operating system state such as network connections and open file connections. CRIU is able to restore network connections; however, it can only restore file connections if the file exists on the machine where the file is restored [9]. So, the user may need to read the files from disk on the checkpoint machine, send them over a network, and then write them to disk on the restore machine.

Docker containers can be snapshotted in two different ways. First, Docker containers are generated from *Dockerfiles*, which are the commands that need to run to generate the container image [15]. So, a Docker container can be snapshotted by just migrating the Dockerfile to a new machine, and starting up a Docker container. However, this is not *true* snapshotting; rather, it lets one recreate the same initial environment. While this is useful for being able to deploy software in the same environment on different machines, it does not let one pause running computations and restart them on a different machine.

The `docker checkpoint` command can be used for true snapshotting, where a running Docker container is paused [14]. This command uses CRIU to checkpoint and restore the containers [24]. So, it has the same benefits and limitations as CRIU.

Finally, VMs are always stored as images, which allows them to be suspended and resumed easily. Clark et al. started research into live migration for virtual machines, aiming to minimize the VM's downtime (i.e. the period when the VM is not running on any physical machine) [7]. They developed the *pre-copy* approach, which starts with multiple rounds of sending the VM's pages over the networks (and re-sending pages that have been modified since they were first sent), and then a short period where the VM is stopped in order for the final set of pages to be sent over the network. This approach reduced VM downtime to as low as 60ms, with a worst-case downtime of 3.5 seconds. In general, a VM's downtime is proportional to the rate at which it dirties pages, the page size, the duration of the last pre-copy round, and inversely proportional to network bandwidth [30]. Total migration time is proportional to the number of live migration rounds.

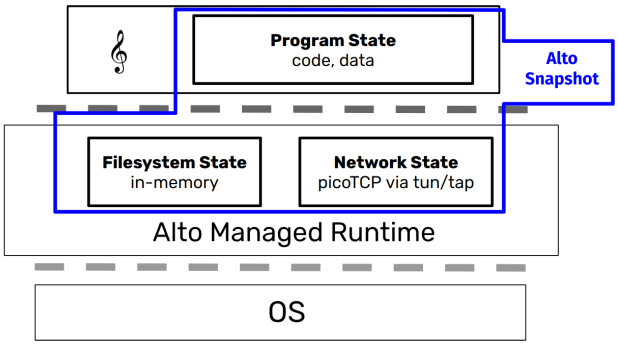


Figure 1. Alto design overview. To snapshot a Clef, Alto must enumerate program state, filesystem state, and network state. Alto moves filesystem and network state from the OS into the Alto runtime.

3 Design

We present the following three design goals for Alto:

1. Stateful autoscaling, ensuring Alto fits within the serverless paradigm
2. Fast snapshot and restore of above-runtime Clefs, enabling data-with-code application architectures
3. The ability to support more Clefs per machine than Lambdas per machine (i.e. tens of thousands of Clefs)

To achieve these design goals, Alto's key insight is to treat the runtime as a hypervisor, as presented in Figure 1. We follow the definition of hypervisors presented for Xen virtualization, in which the runtime "operates at a higher privilege level" than the programs running through the runtime [6]. Runtimes are already able to spin up new programs and tear them down, much like a traditional hypervisor does for virtual machines. But runtimes can also assist in migration of programs to different physical machines by storing necessary state.

The runtime must hold three types of states necessary for Clefs to be resumed on any machine: (1) local program state, such as the program's local and global variables, (2) file system state, such as the program's open file connections, and (3) network state, such as the program's open network

connections. If these three types of state can be serialized, then they can be resumed on a new physical machine, and the program should be able to continue running without interruptions. As a result, the Alto managed runtime must store this state, and thus the runtime becomes the unit of snapshotting and restoring.

This stored state is critical and allows Alto to meet its proposed system goals. For Goal 1, stateful computation is achieved because a program's memory can be snapshotted and later resumed, meaning all data in memory is effectively persistent. If developers store a user's state in memory, then the persistence of memory enables Clefs to autoscale for each user. Computation can be long-lived because long-running programs can be snapshotted and resumed on different physical machines.

Faster snapshot and restore in Goal 2 is achieved because the runtime is a smaller virtualization layer than virtual machines and containers. So, these operations deal with less state. In addition, the increased utility of in-memory data structures improves performance because it is faster to write data from memory than from disk. This persistent memory also enables the colocation of code and data.

And finally, more Clefs can be supported than Lambdas because long-running programs can be quickly snapshotted and resumed on different physical machines. This allows cloud computing providers to load-balance and decrease fragmentation, as they can migrate long-running programs to new machines to free up resources for a short-lived program.

Alto's design goals allows it to support new types of applications as well as improve existing ones. Alto is useful for stateful applications where computation can be sharded by some entity. For example, a ridesharing service could utilize Alto by creating a new Alto Clef per user. As users move in a car, the service could migrate the user and driver's Clefs to physical machines closer to their geolocations, without either the user or driver perceiving any interruptions in the app. And as mentioned earlier, social media platforms could utilize Alto by specifying the logic for each user's connection; a Clef would be rebooted when a user logs in, and then saved to disk when the user logs out. In addition, one could build FaaS platforms like AWS Lambda using Alto, by imposing a time limit for running computation and ignoring state. Thus, *Alto creates a more general platform for serverless computing*.

However, we note Alto doesn't serve all applications well; in particular, applications that read more data than will fit in main memory – such as database management systems – are not well-suited for Alto because Alto stores operating system in memory.

In this paper, we specifically focus on Goals 1 and 2. The following sections enumerate the state that Alto must track. We start with the runtime state, and then explain the network and file system state that must be stored.

3.1 Runtime State

In order to snapshot/restore an Alto Clef, the program's state must be extracted from the runtime and serialized to disk such that it can be deserialized and loaded into a new runtime. A program's state includes the following:

- **Global state:** values accessible to the entire program, which many include global variables, classes, and user-defined functions.
- **Local state:** values accessible only in a given context, such as instance variables, function-local variables, and function arguments.
- **Closures:** lexically-scoped references to values "closed over" by functions.
- **Call stack:** the chain of functions to be walked after the current function returns.
- **Threads:** the values and references accessible to different threads of execution.
- **Code:** the code itself, which may be referenced by function state.

However, the underlying runtime contains structures and mechanisms containing their own program-specific state. For example, a runtime may include:

- **Scheduler state:** information about which threads have been run and which threads should be prioritized next.
- **Garbage collector state:** depending on the garbage collection algorithm used, this may include information about reference counts, generations, objects marked for sweeping.
- **JIT information:** if the language contains a JIT compiler, information about which parts of the code have been compiled or not.
- **I/O:** file descriptors, open sockets, and other system-specific data.

Therefore, the above two lists enumerate all of the state relevant to an Alto Clef. Snapshotting an Alto Clef involves enumerating this state, serializing it, and flushing it to disk (or the network). Restoring an Alto Clef involves reading the state, deserializing it, then inserting it into the runtime's data structures. For instance, the managed runtime may store garbage collector information such as a linked list of objects to be swept. Alto snapshots must extract the information from this list, serialize it, and load it into another Alto runtime's to-sweep list. In this way, the runtime resembles a hypervisor, since it must abstract the resources (runtime data structures) from the multiplexed units of computation (Alto Clefs).

3.2 The Alto Runtime

We will now discuss the design of the Alto runtime and how the design facilitates simple, fast snapshot and restore. In this section we focus on runtime-specific structures and state,

and we describe the techniques used to snapshot files and network sockets in later sections.

The Alto runtime is based on Lua 5.3 [23]. It uses an interpreter which supports the following types:

- Boolean
- Integer
- String (immutable, interned)
- Table
- Function

Note that Tables, which are maps of keys to values, is the only mutable data structure. Functions are first-class, and both Functions and Tables are pass-by-reference, while all other data types are immutable and thus pass-by-value. Alto is lexically scoped and supports closures. Alto does not support threads or coroutines.

Alto uses Lua's mark-and-sweep garbage collector, which classifies objects as either live or dead based on whether or not a reference chain from the object to a globally accessible object. If no reference chain exists, the object is marked as dead and swept during the next GC cycle.

There are four pieces of state included in an Alto snapshot.

1. The set of global variables: stored in an internal table and thus easily enumerable.
2. The stack: the stack includes local variables, Function references, and Function arguments.
3. The call graph: a linked list of activation frames which each store metadata about the current Function level such as the Function name. Each node in the linked list references a Function on the stack. Restoring the call graph ensures that, upon restore, execution resumes from the point immediately before snapshot was called.
4. The code: In Lua, and therefore Alto, all chunks of code are represented as anonymous Functions. This means all code is automatically serialized when Functions are snapshotted.

Alto's simple runtime design facilitates fast, easy state snapshot and restore. Since Alto does not support threads, Alto snapshots need snapshot scheduler state. Alto does not include a JIT compiler. Alto *does* support the I/O structure (such as network and filesystem) state snapshots and describes the mechanisms required to do so in subsequent sections. Alto also includes a garbage collector. However, as stated, in order to perform a snapshot Alto must serialize global variables, the stack, and the call graph. So, by definition, Alto snapshots only values that are accessible by references from global variables (stack values are freed once the stack slot is popped), leaving dead objects behind. In this way *Alto snapshots perform implicit garbage collection.* Thus, there is no need for Alto to snapshot GC state, since all objects being restored are were definition live.

3.3 File System State

The following state must be stored for each file in order to recreate open file connections when Clefs are restored:

1. **File path.**
2. **Access mode** (e.g. "read-only" or "read and write"), if the file is currently open
3. **File offset.** Note that files are separate for each Clef, and Clefs are expected to run in single-threaded mode, so Alto can store the offset per-Clef, instead of per-file connection, as Linux does.
4. **Data buffer:** the file contents.

This last piece of state is motivated by Alto's goal of fast snapshot and restore times, as this allows cloud computing providers to migrate long-lived computation to different physical machines. Reading files from disk has a latency of 3ms, which is too slow for snapshotting and restoring computation that runs to completion in seconds [1]. So, Alto instead maintains all files as in-memory data structures.

3.4 Runtime Network Stack

Alto aims not only to enable efficient snapshot and restore, but also to perform migration in such a way as to minimize the perceived interruption of normal processing. It is therefore necessary to move the networking stack out of the kernel and into the runtime, so that it may be more easily tied to applications.

Nevertheless, not all network state is relevant in this context. In fact, only *transport layer* network state must be explicitly accounted for by Alto's network-specific snapshot and restore mechanisms. To see why this is the case, we briefly consider the other levels of the OSI network stack:

1. **Application Layer:** the relevant state amounts to either code or data, both of which will be captured by Alto's general runtime state snapshot-restore mechanisms.
2. **Network, Data Link, and Physical Layers:** transport layer protocols operate under the assumption that these layers are unreliable and, if necessary, are robust against failures in these layers.

Thus, Alto's runtime network stack and corresponding snapshot-restore mechanisms need only place explicit emphasis on transport layer state in order to successfully migrate applications. Furthermore, Alto can restrict its attention even further to just those transport layer protocols that are stateful (e.g., TCP); it can be assumed that applications built upon stateless protocols (e.g., UDP) are robust against transport failure and loss, and thus additional faultiness introduced by Alto migration should not prove problematic.

4 Implementation

4.1 Runtime

As noted in Section 3.2, the Alto runtime prototype is built using Lua 5.3. Lua provides a simple but expressive API for developing Lua libraries in C. Specifically, Lua stores all program state (including global variables, the stack, and the call graph) in a single `luaState` instance, and makes this state accessible to C libraries. (Alto does not use Lua coroutines.) In this way, implemented in C as a Lua library, Alto can simply serialize the relevant pieces of state to perform snapshots. To perform a restore, the Alto library deserializes the state and inserts it into the new `luaState` object.

The current Alto prototype is about 1,000 lines of library code, with less than 100 lines of modification to the actual Lua 5.3 runtime. The current iteration of the prototype can snapshot global variables (including closures), local variables, and the call graph. However, it only supports the restoration of global variables, including closures. As a result, both snapshots and restores are triggered by the programmer, rather than the runtime. Additionally, execution does not resume where the snapshotted program left off. After restore, all global variables captured by the snapshot are accessible to the restored program. For instance, here is an example of a program being snapshotted using the current prototype.

```
alto = require "alto"
-- globals
a = 10
b = "foo"
local c = 6
function f(arg1)
  -- c only captured because closed over
  return a + arg1 + c
end
alto.snapshot()
```

And the current program being restored:

```
alto = require "alto"
alto.restore()
print(f(11)) -- prints 27
print(c) -- prints nil
```

Lua's lack of complex, user-defined data (such as classes) also simplifies Alto's design: Tables are the closest thing Lua has to `structs`. Two variables may reference the same object *only* if the object is either a Table or Function. As a result, if Alto encounters a Function or Table while snapshotting, Alto must ensure that the object was not previously snapshotted, since another variable may have been pointing to the same object. Although closed-over values referenced by Functions (called upvalues in Lua) may appear to complicate this, a Function's upvalues are stored in a Table, which makes snapshotting and restoring Alto values a relatively painless task.

4.2 Network Stack

As discussed in 3.4, the Alto design calls for the direct integration of the network stack and the language runtime. To this end, we choose to integrate picoTCP, a small, modular networking stack with a clear internal separation of state along network layer and protocol boundaries [31], into the Alto runtime.

The current Alto prototype contains an API for serializing and deserializing picoTCP sockets (i.e., *struct pico_socket*) used in TCP connections. The prototype currently supports snapshotting applications with active TCP client connections and subsequently restoring these applications with the original connection intact and without loss of data that was in transmission or enqueued for transmission at the time of snapshotting.

During snapshotting, all application specific metadata associated with picoTCP sockets as well as the sockets' associated TCP state is serialized to disk using *stdio* file function calls. Most socket state is easily written to a file, in the order that it is laid out within the relevant picoTCP structs and without modification or additional data. The most notable exception to this is the serialization of sockets' outbound TCP queues, which are stored using red-black trees and thus require additional consideration and manipulation to serialize.

The chronology and mechanics of deserialization mostly parallel those of serialization. However, while picoTCP sockets encapsulate all state required to be serialized, it is not sufficient to reconstruct the original sockets during application restoration. Before resuming a snapshotted application we must create a temporary socket on the new picoTCP stack and connect this socket to an arbitrary server in order to properly initialize the relevant picoTCP structures. Subsequently, we replace all of the temporary socket metadata (including TCP state) with that from the serialized socket. Finally, we attempt to resend the packets from the serialized socket's outbound TCP queue. After this, the application can resume networking as if it had never been snapshotted.

Further development of the Alto prototype will result in the full integration of the picoTCP networking stack into the Lua runtime. Completing this requires only a small Lua wrapper around the serialization/deserialization API and possibly a set of POSIX style socket system calls (which are provided by compiling against the picoTCP-BSD library extension).

5 Evaluation

5.1 Runtime Snapshot/Restore

In this section, we evaluate the performance of snapshotting and restoring an Alto Clef using our Lua-based Alto prototype. We snapshot a simple program that does two things: 1) construct a 10KB globally accessible String, and 2) construct a global table of 10,000 anonymous Functions. We measure

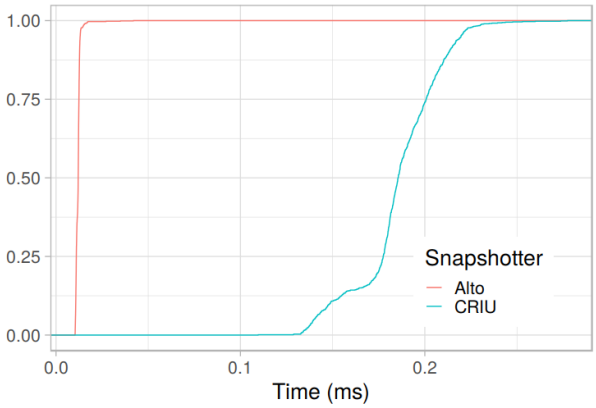


Figure 2. CDF of time taken to snapshot the **runtime** of an Alto Clef that creates a 10KB String and 10k functions.

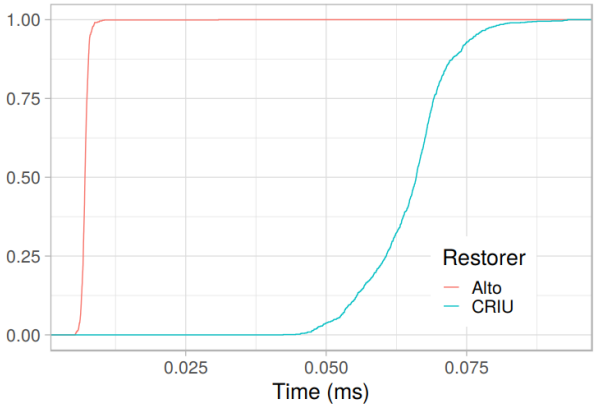


Figure 3. CDF of time taken to restore the **runtime** of an Alto Clef that created a 10KB String and 10k functions.

the time it takes to snapshot the program after this data has been initialized. We then measure the time it takes to restore that data into a new Lua program. We benchmark Alto's performance against CRIU. Using CRIU, we snapshot and restore the same program, but CRIU snapshots and restores the entire process rather than the runtime state. We performed each experiment 1,000 times. The experiments were run in an Ubuntu 18.06 VirtualBox VM with 8GB of memory atop a Dell XPS 15 with an Intel 2.8 GHz i7 processor.

The snapshot results are shown in Figure 2. Note that Alto's average snapshot time (0.0119ms) is better than CRIU's (0.1856ms). This is because Alto doesn't need to snapshot the entire process state (which includes all of the process's memory), but only needs to snapshot the relevant program state. The same can be said for restore performance, shown in 3. Alto's average restore time is 0.0071ms while CRIU's is 0.0649ms. However, note that Alto has an advantage here: currently restoration of the call graph does not work. This means CRIU successfully restores execution state whereas Alto does not.

These results indicate that snapshotting and restoring at a higher level than the process abstraction provides superior performance. This stems from the fact that Alto can snapshot only the state relevant to the program, given a runtime. By assuming a runtime (and a platform atop which the runtime executes), snapshots become smaller. Additionally, enumerating state is much easier, since it all lives within a single in-memory data structure.

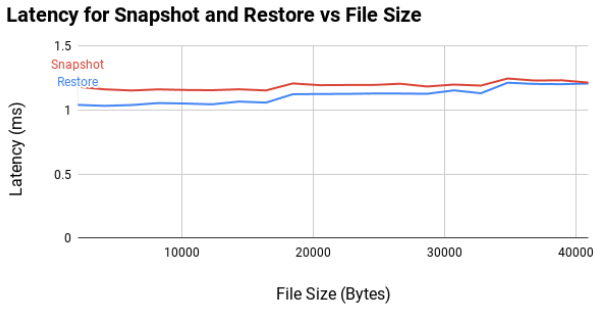
5.2 File System Microbenchmarks

Recall that Alto maintains an in-memory file system in order to support fast migration; alternative snapshotting tools would need to read files from disk, send them over the network, and write the data to disk on a new machine. So, our first file system microbenchmark compares the total time to snapshot and restore an Alto process with one open file of varying sizes against reading that file from disk and writing it to a new file on disk. The Alto operations were recorded by adding timestamps in the Alto snapshot and restore functions. The disk copy operation was recorded using the Linux `dd` tool, which evaluates the latency and bandwidth of disk operations.

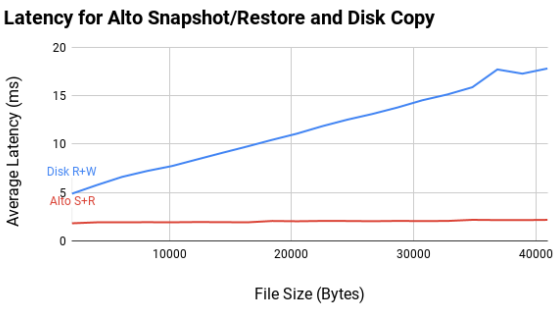
For both Alto and `dd`, we sync and flush the disk cache after every operation in order to ensure a fair comparison; that is, we sync and flush after each Alto snapshot, Alto restore, and `dd` operation. In addition, we used `setvbuf(3)` to ensure that Alto's serialized FS state is not cached by stdio, and is therefore read from disk [13]. We varied the file size opened by the snapshotted program from 2048 bytes to 40960 bytes in increments of 2048 bytes, and averaged 1000 trials. All experiments were run in a VirtualBox VM running atop a MacBook Pro with a 2.3 GHz processor, four cores, and 16 GB of RAM. The individual snapshot and restore times are found in Figure 4a, and the comparison is in Figure 4b.

We compared the combined snapshot and restore times against a disk copy operation because we are unable to isolate the time to write to the disk; Given that writing to disk is analogous to snapshotting, and reading from disk is analogous to restoring, a fair comparison is combined snapshotting and restoring against one disk copy operation. As Figure 4b shows, Alto is at best 8x faster than disk operations, and at worst 2x faster. We believe that Alto's performance should be faster than that of a disk copy operation because Alto performs a disk write from memory instead of a disk write from disk. However, this would only explain why Alto snapshotting is faster than a disk copy operation; we expect that restoring would have similar latencies to reading from disk, and therefore to increase proportionately to file size. While we see a slight increase, we are unable to explain why the restore latency is relatively flat.

We also performed a microbenchmark of the `open()` function. In Section **??**, we mentioned that Alto must prefetch existing disk files that are opened in a Clef. We measured



(a)



(b)

Figure 4. Microbenchmarks for Alto snapshotting and restoring as a function of the size of an open file.

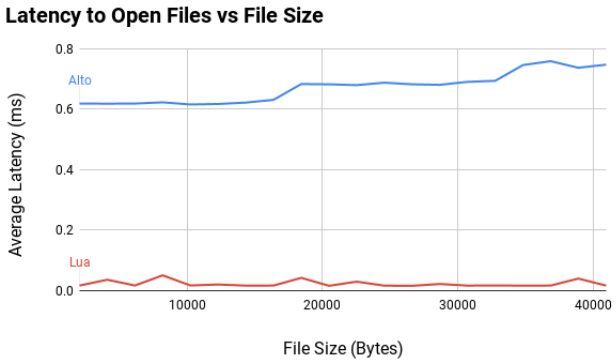


Figure 5. Comparison of latency to open a file in Alto and in Lua, for varying file sizes.

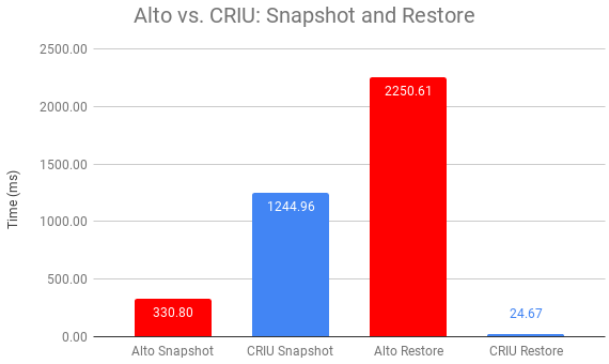


Figure 6. Comparison of time to snapshot and restore an application with 100 open TCP connections using Alto and CRIU.

the performance degradation of this decision against the unmodified Lua function to open files. Again, we varied the file size from 2048 bytes to 40960 bytes in 2048 byte increments, and averaged 1000 iterations of each trial. As the results in Figure 5 show, Alto is much slower than unmodified Lua because of the prefetching. However, we believe that latency smaller than 1ms is a sufficient trade-off for faster migration.

We were unable to present benchmarks against CRIU because CRIU snapshots much more information than just file system state; so a direct comparison would be unfair. Future work will include a macrobenchmark of the entire Alto system against CRIU for varying dimensions, including size of open files.

5.3 Network Stack Microbenchmarks

We evaluate the Alto network snapshot and restore API by comparison against CRIU. For both Alto and CRIU we snapshot a simple program that establishes 100 TCP connections with a netcat server and subsequently writes short, 30 byte messages across each of its connections until killed. For Alto, we measure the aggregate time spent in the Alto serialize and restore API functions for all 100 open sockets. For CRIU, we determine the time spent in the CRIU dump function (snapshot equivalent) and the time spent in the CRIU restore function both after the application has established its 100 TCP connections. We performed each experiment 100 times. The expeirmnets were run in an Ubuntu 18.06 VirtualBox VM with 8GB of memory atop a MacBook Pro with an Intel 3.3 GHz processor i7 processor.

The snapshot and restore comparison between Alto and CRIU is shown in Figure 6. We note that Alto's snapshot time (330.80 ms) is faster than CRIU's time (1244.96). This is likely because of the additional overhead incurred by CRIU in modifying connections to prevent FIN and RST packets from being sent or received while the application is suspended. Alto is not burdened by manipulating socket state during suspension because this information lives in the picoTCP stack rather than the kernel. However, we also note that Alto's restore time (2250.61 ms) is slower than CRIU's time (24.67 ms). This is almost certainly because of the way in which dummy connections are established as refernced in Section 4.2. A

further experiment (graph omitted) showed that the average time (across 1000 trials) to restore a single connection was 3.32 ms, thus suggesting parity with the time required to snapshot an application with 100 open connections.

These results show that snapshotting and restoring network state at a higher level than the proces abstraction can provide superior performance. Further work (specifically establishing dummy connections in parallel rather than serially) could provide greater insight into how to make Alto network state restoration competitive or better than that of CRIU.

6 Related Work

In this section we acknowledge the work of and differentiate Alto from previous lightweight virtualization and application snapshotting/restoring approaches.

6.1 Lightweight Virtualization

Throughout the literature, a range of new virtualization techniques can be found [12, 32, 33], ranging from iterative approaches to full hardware virtualization to sub-process level techniques.

In 2018 Amazon deployed Firecracker as a hardware level virtualization technology meant to reduce startup latency for their serverless computing service, AWS Lambda, by removing legacy support and emulating only the minimal hardware required for booting AWS Linux [33]. Akin to Xen's *Dom0*, AWS Firecracker includes an orchestration manager that invokes one-time-use "MicroVMs" via the Linux KVM API [33]. Notably, the approach taken by Firecracker differs from that taken by Alto in that Firecracker purposefully does not support live migration of VMs as it is Amazon's intention that FaaS instances are short-lived. Additionally, in choosing to keep the virtualization boundary at the level of hardware, Firecracker's solution is inherently "heavier" than Alto's approach.

In recent years, unikernel and libraryOS designs have regained the interest of their predecessors [16, 27]. One of the more recently developed systems is Graphene, which aims to reudce VM memory overhead and support multi-process applications by combining traditional elements of unikernel and microkernel design. Graphene kernels can run arbitrary, unmodified binaries and supports application snapshot and resume. The key difference between Alto and Graphene is that Graphene is still a "heavier" solution in that it can provide all necessary kernel functionalities to its applications via the traditional libraryOS module system. Thus, Graphene is required to support a greater number abstractions than Alto.

Perhaps the most virtualization technology most akin to that of Alto Clefs is proposed by Boucher et al. [18]. Aiming to minimize the size of FaaS computational units as well as their latency, Boucher et al. "virtualize" single-threader worker processes which are given microservices to run (via calling a registered function) by a centralized dispatcher. Ultimately, Alto differs from this approach in that it aims to provide long-lived, stateful units of computation, and thus established a slightly lower virtualization boundary (i.e., that of a managed runtime).

6.2 Snapshot and Restore

Both the literature and the open source community are replete with snapshot and restore solutions with support ranging from single-process applications to entire virtual machines.

Presented by Cully et al. in 2008, the Remus system was designed to provide high fault tolerance for unmodified kernels running as virtual machines atop commodity hardware [12]. By actively sending VM state across the network to actively running backups and running the primary guest slightly ahead via speculative execution, Remus is a highly fault tolerant system. Though Remus and Alto provide similar guarantees for the state of restored applications, Alto, owing to its inherently lightweight design, achieves these guarantees via operations that are two orders of magnitude faster.

Another existing solution is Distributed MultiThreaded Checkpointing (DMTCP) [4]. The approach taken by DMTCP is similar to that of Cully et al. though restricted to the application level. However, the major downside of DMTCP's solution is that any application that wishes to be snapshotted and resumed must be statically linked against DMTCP; additionally, DMTCP doesn't support all useful kernel features such as inotify() [8]. Thus, Alto differs from DMTCP in that Alto places no requirements on the applications that can be snapshotted and restored beyond that they be written for a supported runtime (currently Lua, though in future others are possible).

Perhaps the most innovative snapshot and restore solution is Checkpoint and Resume In Userspace (CRIU). CRIU relies on a set of invasive tools (e.g., ptrace) and parasitic behavior in order to snapshot and restore applications [10]. Like Alto, CRIU can snapshot and restore arbitrary applications; however, the major difference between the solutions is in their deployment. Alto is self-contained, providing both a mechanism for lightweight virtualization and snapshot/restore, whereas CRIU must be coupled with a separate virtualization technology (e.g., Docker containers) in order for deployment in an FaaS implementation.

7 Conclusion

In this paper, we described Alto, a system that fills the gap for stateful, long-lived computation in the serverless ecosystem. Alto's key insight is to treat the runtime as a hypervisor, storing all necessary state for running programs, including program, network, and file system state. We then present

our Alto prototype using a modified Lua runtime, which can successfully snapshot and restore running Lua programs. Our evaluation shows that Alto achieves fast snapshot and restore performance relative to CRIU and disk reads and writes.

While we presented some applications that work well for Alto, such as social media platforms and ride sharing applications, we hope that creating a new serverless architecture will allow developers to create new types of applications.

References

[1] Harvard CS 61. 2018. Storage 2: Cache Model. https://cs61.seas.harvard.edu/site/2018/Storage2/.

[2] Amazon. [n. d.]. Amazon Lambda. https://aws.amazon.com/lambda/.

[3] Amazon. [n. d.]. S3. https://aws.amazon.com/s3/.

[4] K. Arya, G. Cooperman, R. Garg, J. Cao, and A. Polyakov. 2019. DMTCP: Distributed MultiThreaded CheckPointing. https://lwn.net/Articles/775736/.

[5] Ioana Baldini, Perry Cheng, Stephen J. Fink, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter, and Olivier Tardieu. 2017. The Serverless Trilemma: Function Composition for Serverless Computing. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2017)*. ACM, New York, NY, USA, 89–103. https://doi.org/10.1145/3133850.3133855

[6] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*. ACM, New York, NY, USA, 164–177. https://doi.org/10.1145/945445.945462

[7] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. 2005. Live Migration of Virtual Machines. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2 (NSDI'05)*. USENIX Association, Berkeley, CA, USA, 273–286. http://dl.acm.org/citation.cfm?id=1251203.1251223

[8] CRIU. [n. d.]. *Comparison to Other CR Projects.* https://criu.org/Comparison_to_other_CR_projects.

[9] CRIU. [n. d.]. *Inheriting FDs on restore.* https://www.criu.org/Inheriting_FDs_on_restore.

[10] CRIU. [n. d.]. *TCP Connection.* https://criu.org/TCP_connection.

[11] CRIU. 2017. Checkpoint/Restore Tool. https://github.com/checkpoint-restore/criu.

[12] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. 2008. Remus: High Availability via Asynchronous Virtual Machine Replication. In *NSDI '08: 5th USENIX Symposium on Networked Systems Design and Implementation*. 161–174.

[13] die.net. [n. d.]. *setvbuf(3).* https://linux.die.net/man/3/setvbuf.

[14] Inc. Docker. [n. d.]. *docker checkpoint.* https://docs.docker.com/engine/reference/commandline/checkpoint/.

[15] Inc. Docker. [n. d.]. *Dockerfile reference.* https://docs.docker.com/engine/reference/builder/.

[16] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. 1995. Exokernel: An Operating System Architecture for Application-level Resource Management. *SIGOPS Oper. Syst. Rev.* 29, 5 (Dec. 1995), 251–266. https://doi.org/10.1145/224057.224076

[17] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. 2015. An updated performance comparison of virtual machines and Linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 171–172. https://doi.org/10.1109/ISPASS.2015.7095802

[18] Thomas Fisher, M Sriram, and V Puhazhendhi. 2003. Beyond microcredit: putting development back into micro-finance. *Indian journal of agricultural economics* 58, 2 (2003), 283–284. http://search.proquest.com/docview/38547698/

[19] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Frst and Slow: Low-latency Video Processing Using Thousands of Tiny Threads. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 363–376.

[20] Google. [n. d.]. Cloud Functions. https://cloud.google.com/functions/.

[21] Google. [n. d.]. Firebase Firestore. https://firebase.google.com/products/firestore/.

[22] Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2019. Serverless Computing: One Step Forward, Two Steps Back. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. http://cidrdb.org/cidr2019/papers/p119-hellerstein-cidr19.pdf

[23] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes. 2018. A Look at the Design of Lua. *Commun. ACM* 61, 11 (Oct. 2018), 114–123. https://doi.org/10.1145/3186277

[24] Saied Kazemi. 2015. How did the Quake demo from DockerCon Work? (jul 2015). https://kubernetes.io/blog/2015/07/how-did-quake-demo-from-dockercon-work/.

[25] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 427–444.

[26] James Larisch, James Mickens, and Eddie Kohler. 2018. Alto: Lightweight VMs Using Virtualization-aware Managed Runtimes. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes*. ACM, 8.

[27] Anil Madhavapeddy and David J. Scott. 2014. Unikernels: The Rise of the Virtual Library Operating System. *Commun. ACM* 57, 1 (Jan. 2014), 61–69. https://doi.org/10.1145/2541883.2541895

[28] Microsoft. [n. d.]. Azure Functions. https://azure.microsoft.com/en-us/services/functions/.

[29] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. 2018. numpywren: Serverless Linear Algebra. *arXiv preprint arXiv:1810.09679* (2018).

[30] A. Strunk. 2012. Costs of Virtual Machine Live Migration: A Survey. In *2012 IEEE Eighth World Congress on Services*. 323–329. https://doi.org/10.1109/SERVICES.2012.23

[31] tass belgium. 2018. picoTCP. https://github.com/tass-belgium/picotcp.

[32] Chia-Che Tsai, Kumar Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald Porter. 2014. Cooperation and security isolation of library OSes for multi-process applications. In *Proceedings of the Ninth European Conference on computer systems (EuroSys '14)*. ACM, 1–14.

[33] The Firecracker virtual machine monitor. 2019. The Firecracker virtual machine monitor. https://lwn.net/Articles/775736/.

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