MEng Individual Project
Department Of Computing
Imperial College London

Refunction: Eliminating Serverless Cold Starts Through Container Reuse

---

Author: Oliver Stenbom

Supervisor: Dr Robert Chatley

June 2019

Abstract

The next generation of cloud hosting is known as *serverless* or *function-as-a-service* (Faas). The popularity of these easily deployable and infinitely scalable functions has skyrocketed since AWS announced their Lambda service in 2014. One primary concern regarding the performance of these functions is the effect of "cold starts". A cold start is the time it takes to boot a new container when the platform needs to increase its capacity for that function. We investigate the possibility of *restoring* function containers as an alternative to starting new containers. Our method focuses on Linux process primitives. We store and modify state such as raw memory and registers in order to reset the process to the way it was before the user's function was loaded. We discuss how to ensure *temporal isolation* in order to provide security guarantees in such a system. We find that it is possible to restore container processes in a variety of runtimes. Using this approach can decrease the effect of cold starts by up to 20x and increase the overall throughput of such systems.

Acknowledgements

I would like to thank my supervisor Dr Robert Chatley for his unfailing support and creative questions which always shed new light on my toughest problems. I would also like to thank Simon Shillaker for taking the time to share his intuition with me on everything from clusters to graphs. My discussions with both of you have been the highlights of my final year here and vital to the success of this project. Thank you.

Contents

Introduction 7

Background 9

The Need for Containers and the Genesis of Serverless 9

Serverless - the next frontier 9

Architecture of a FaaS Platform 10

OpenWhisk 11

State of Cold Starts 11

Recent Progress in Serverless Technology 12

Container Optimisations 12

Language Isolation 13

Unikernels 14

Firecracker 14

Project Approach 15

Container Essentials 15

Processes 15

Ptrace 16

Containers 17

Namespaces 17

Cgroups 17

Layered File System 18

Checkpoint & Restore 18

Checkpoint 19

Restore 19

Language runtimes 19

Refunction 21

The Worker: Restoring Live Processes 22

Preparation 22

File System 22

Spec 23

Communication 23

Function Loading & Request Handling 24

Runtimes 25

Isolation & Control 27

The Importance of Isolation 27

Refunction Isolation Approach 27

Container Control 28

Checkpoint & Restore 30

Registers 30
Memory 30
Area Size Changes 32
New Memory Areas 34
Removed Memory Areas 34
The File System 35
Other Restorable State 35

Library Loading 37
Towards Zero-time Library Loading 38

The Invoker: Scheduling the Worker 40

Methodology & Environment 42
Language 42
Libraries 42
Testing 42

Evaluation 44
Experimental Setup 44
Cold Starts 45
Low Throughput Cold Starts 45
Concurrent Request Cold Starts 47
Throughput Improvements 49
Measuring Restore Times 51
Type of Runtime 51
Libraries Used 51
Memory Usage 52
Memory Overhead of Storing Checkpoint State 53
Ptrace Resource Overhead 54

Conclusion 56

Bibliography 57

Introduction

The rise of virtualisation and cloud computing has made sharing computing resources ever easier and more lightweight. As our demand for the cloud increases, we increase the size of our data centres and purchase new devices to consume our data. The total cost of these services is around 2% of greenhouse gas emissions [1][2], on par with the aviation industry and set to multiply in the coming years. The efficient use of these resources is therefore a pressing concern.

The latest trend in cloud computing is known as *serverless*. Rather than renting a portion of a server via a container or VM on a monthly basis, serverless platforms promise billing by the 1/10th of a second and infinite scaling exactly according to demand with almost no configuration. These features promise reduced developer operations, a lower barrier to entry for the cloud and reduced running costs.

The properties of serverless systems come from its event-driven design. Upon receiving an "event" (ranging from a http request to a trigger from another function), a container is deployed on an available server to handle the request. The code needed to respond to the request is downloaded, run and the response sent back. Code that responds to an event is known as a *function*, which should be small, simple and perhaps stateless. The container where the function was deployed remains alive for a grace period (usually around half an hour) to speed up response times for subsequent requests. As a developer does not need to configure or run their server processes in such a system, the system is server-less.

Many suitable use cases have been found for this kind of event-driven system, such as image post-processing and notification control, but it is not yet considered suitable for all kinds of workload. Developers might choose not to use serverless functions either because of a lack of best practice or because of performance issues.

Since serverless is a fairly immature technology, standard "good practice" methods for debugging, deploying and keep state in serverless systems are not yet clear. While we expect that best practice will develop as the technology matures, inherent performance issues may hinder serverless from being considered suitable for general use. This report will mainly focus on one major performance problem for serverless containers - its startup latency, known as a "cold start".

A cold start is the time it takes to initialise a container in response to an increase in requests for a function. Depending on the system, cold starts are usually in the order of a second [3]. Much of this time is spent just starting the container itself, after which the runtime must be started and libraries loaded before the function can be run. In a system with hard latency requirements, these tail latencies make the use of serverless platforms out of the question [4].

This report aims to explore the possibility of reusing containers when running serverless functions so that different users can use the same container, one after another, with a

cleanup step in between. As opposed to typical container isolation (isolating containers from each other), *temporal* isolation must now be considered. This means that a user should not be able to modify the container in a way that will affect future runs. Various kinds of linux process inspection tools will be used to enforce temporal isolation, primarily ptrace & the `/proc` file system.

Checkpointing and restoring a process is something that has been done in the past [5]. Given that a container is a unix process attached to namespaces and cgroups, applying checkpoint/restore theory may allow containers to be safely transitioned back to their starting point after a function has run. Although checkpointing/restoring is otherwise a fairly heavyweight operation, the simple and often stateless nature of serverless functions make it possible to restore their processes in a small time frame.

A serverless system with a lower worst-case latency would benefit both developers and providers. First and foremost, it may make it possible for many workloads with stricter latency requirements to be considered suitable for serverless platforms. We also expect a much greater throughput to be possible, as fast restores replace costly evictions, potentially further reducing costs for developers. We would also eliminate the need for "keep warm" functions that are sometimes used by developers to stop their functions from going "cold".

To summarise, this project aims to:

- Develop a generic method of checkpointing/restoring a container while it runs
- Develop the ability to restore different language runtimes after running various functions
- Evaluate the performance and speed of this method in a larger serverless platform (in our case OpenWhisk)

Background

The Need for Containers and the Genesis of Serverless

Ever since the introduction of mainframes in the '50s, engineers have sought to make it as simple as possible to compute at scale. From time-sharing to the cloud, our demands in terms of power, storage and speed have skyrocketed. At the same time, developers striving to maximise their velocity expect it to be simpler than ever to deploy, maintain and even instantly scale their systems. These systems must also be completely secure and fully isolated from one another.

As a result of these strict requirements, code deployment has changed drastically. From the difficulty of maintaining physical servers, we saw the rise of virtualisation technology and virtual machines (VMs), which led to the creation of a publicly available cloud. There were, however, downsides to using VMs. It was still necessary to maintain a full operating system on each VM, and startup times of around a minute left much to be desired. Developments in the linux kernel and the layered file-system of Docker led to the popularisation of containers. This pushed deployment from the OS to the process/application level, paving the way for container orchestration technologies such as kubernetes [6].

Where does this leave us now? In a survey of 600 IT decision makers, 72% stated they were using or evaluating containers [7], Google starts 2 billion containers each week [8] and of Datadog's 10,000 customers, half deploy their containers with an orchestrator [9]. Containers have undoubtedly become a pervasive technology, yet developers are still faced with challenges when it comes to code deployment. The questions we ask are the same as they have been for decades - "How many do I need?", "How do I configure them?", and "How do I scale them as demand increases?". Although configuring and scaling containers is vastly easier than working out how to maintain as many operating systems, let alone servers, these questions remain non-trivial. Recent trends in sustainable software engineering involve splitting large codebases into microservices, as opposed to traditional monoliths [10], the result of this is that developers may have to begin answering scaling and configuration questions for hundreds of services, rather than just a small handful.

We may also observe a long-standing trend of low utilization of data-centre resources. Developers have a tendency to over-provision resources in preparation for unexpected increases in usage (spikes). Over-provisioning leads to a large proportion of on, but unused resources. A 2012 study of Google data centres revealed that while resource allocation was near its limit, utilization of CPU and memory struggled to reach 50% [11]. As data centres alone are responsible for around 0.3% of global greenhouse gas emissions [2], better utilization has become an important problem.

Serverless - the next frontier

Enter, then, the next wave of code deployment, coined "serverless". Named as such because the management of a server (or container), is taken away from the developer and automated

by a system. The core idea of a serverless platform is to deploy small stateless functions independently, that respond to *events*. Such a system is therefore also known as "Function-as-a-Service" (FaaS). These functions are ephemeral, run on-demand, and are billed only for actual execution time. In a serverless platform, almost all operational logic is delegated to the infrastructure, so scaling and management is completely automatic [12]. Event triggers include file updates, DNS events or other functions.

Due to their properties, serverless platforms promise greatly reduced operational and scaling costs. For example Adzic et al. present a 66 to 95% cost reduction in two industrial case studies [13], but these platforms also have drawbacks. Different cloud vendors define their functions differently, so migrating to serverless can cause *vendor lock-in*. There are no well-defined ways of *testing and debugging* serverless systems when deployed or locally, and these platforms can have very slow tail latencies. This report will focus on tail latency response times of serverless platforms, known as "cold starts".

Architecture of a FaaS Platform

The ability to scale serverless functions rapidly and infinitely comes from the design of such systems. In short, a serverless platform is a "controller" (entry point) connected to a database (containing all function code) and a fleet of servers (to run them on). When the controller receives an event, a new container is created to run the target function code on one of the worker servers, if such a container does not already exist. Containers have so far been the standard runtime used in serverless platforms due to their industry-proven stability and relatively fast startup times, although alternatives are beginning to surface [14]. We will demonstrate three use cases of serverless platforms to highlight how they work.

The initial case is when no container exists containing the target function code. This may be because the developer has just uploaded the function, or if the function has not been used in a while. Here, the controller selects a worker with available capacity to handle the request. The worker then starts a new container for the target language. This involves starting the container itself, loading the runtime and loading necessary dependencies. Starting a container takes a few hundred milliseconds, and then depending on the runtime and the function's dependencies, the total startup time can take a second or more. This is what is known as a cold start.

Luckily once a container has started it is common practice to leave loaded containers around for a while, normally around half an hour. This is known as a "warm" state. If the controller receives a request when there are warm containers, the request can be forwarded to the container and the function run immediately. Warm responses are typically fast.

If all warm containers are currently processing a request, the platform can not wait until a request completes (after all it could take an undefined amount of time). Instead, it must cold start a new container to deal with the request, thus increasing the number of available warm containers and as such the capacity of the system for that function. This is how scalable platforms can scale quickly and infinitely in response to demand, but we also note that cold starts must occur as the platform scales.

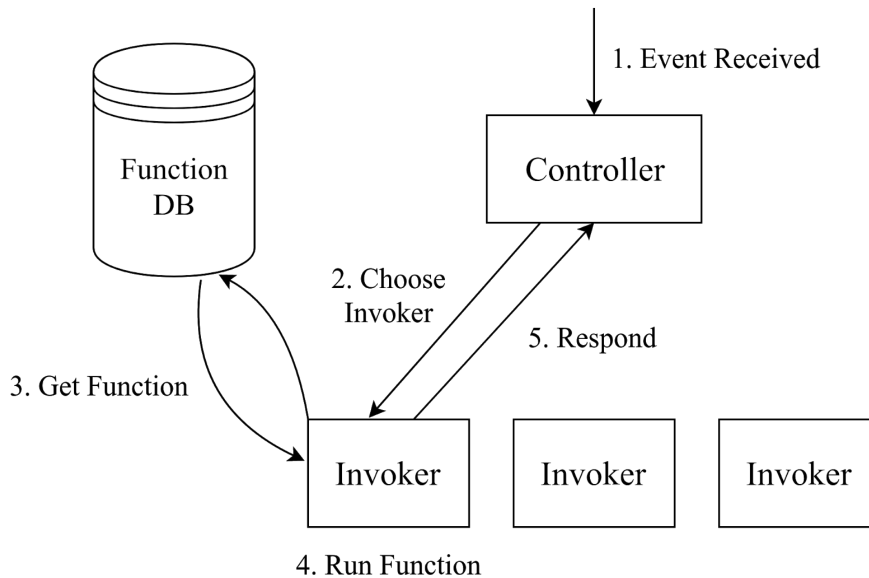

Figure 1: Simplified FaaS Architecture

OpenWhisk

OpenWhisk is the serverless platform of choice for this project. Although not the most popular, it is an apache foundation project and so open source and free to modify. OpenWhisk was started by and is used extensively by IBM in their bluemix platform. Amazon, Microsoft and Google, on the other hand, keep the code for their serverless platforms under wraps, so how they work can only be inferred by talks, articles and black box testing.

The OpenWhisk platform generally works as described in the general platform description above. The controller assigns requests to invokers, which it sends via a kafka messaging queue. The invoker fetches the function code from CouchDB if it is not already available, runs the code and returns the result again via kafka. As well as each invoker having a pool of containers in cold or warm states, OpenWhisk has support for "prewarm" containers, where it can preemptively start containers for a particular runtime.

Scheduling in OpenWhisk follows a "sharded load balancing" algorithm, where a hash of the function modulo no. invokers decides which invoker to start looking for capacity on. If the invoker has no capacity for the function, the rest of the invokers are checked in a deterministic fashion. The aim of this algorithm is to send the same function to the same invoker as much as possible in order to maximise warm invocations.

State of Cold Starts

In reality, the granularity of cold starts is more complex, depending on if it is the first time a request is sent to the provider, the virtual machine (worker), or the container. Different platforms also tend to handle cold starts differently, and approaches involving alternate runtimes such as unikernels should also be considered. Lloyd et al. demonstrate that the differences between traditional cold start states can cause up to a 15x change in performance

[15]. The performance of a cold start varies further based on the provider and runtime. In another recent study from Wang et al., AWS/python cold starts (the fastest combination) were recorded around 170ms and AWS/java cold starts were around 900ms, while the median Azure cold start time was 3600ms [3]. The variance in their observations was also very large.

Although interest in serverless platforms is large, with 46% of IT decision makers evaluating serverless [7], these performance insecurities are one of the reasons why serverless platforms are only used in specific scenarios at the moment. Such typical use cases include image post-processing [16] or entry level data processing [17]. Serverless platforms are currently not considered to be a feasible alternative for time-critical systems.

Cold starts are also detrimental for other reasons. For it to be feasible to consider using serverless at all, platforms typically use long container-warm times, around 30 minutes for AWS, but potentially an hour or more in Google and Azure cases [3]. This leaves resources idle for long periods of time, even though they may only have been used for hundreds of milliseconds. Developers also know that this is the case, and some use "keep warm" functions to call their functions before they become dormant. This makes cold starts destructive for developers, providers and the planet.

Recent Progress in Serverless Technology

We will now explore what current approaches are being taken to improve the performance of serverless platforms. These approaches can be largely placed in three categories: container optimisations, language isolation and unikernels. All approaches boil down to the choice of isolation method.

Container Optimisations

Containers were initially designed for long-running applications that should be isolated from other containers in every possible way. It is worth stopping to consider if the short-lived and often stateless nature of serverless functions requires all of the features of a container, and therefore how containers could be optimised specifically for serverless functions. Further, one should consider that simply starting a given isolation construct is only one step towards running a serverless function. In many cases, starting the language runtime and downloading/installing the relevant libraries may take hundreds of milliseconds before the execution of the function may begin.

Oakes et al. consider a complete perspective of cold start times in serverless platforms, examining and optimising both container startup and dependency loading times in the context of the python runtime [18]. They use three methods to speed up the start time of their containers. First, they avoid the use of slow to start namespaces (particularly the network namespace), which they argue are redundant in a serverless context. Then they *bind-mount* the containers file system, rather than using the more costly AUFS, which takes advantage of serverless functions read-only access to all but a small part of the file system. Finally, as a further optimization, they reuse cgroups, which they find to be more effective

than creating new cgroups for each container. This reduces their container cold start times by 2.8x relative to AWS Lambda.

Oakes SOCK system then makes extensive use of zygotes and package caching to improve complete cold start times. The usage of zygotes is a technique where one forks a process which already has certain dependencies loaded. By keeping a hierarchy of processes with different dependencies loaded readily available, most new requests will be able to fork a running process with the dependencies it needs, doing away with the need to load them at all. This, paired with a cache of commonly used packages led to a 45x speed improvement in loading python packages, taking only 20ms.

The main weakness of this approach is that it was only designed for the python runtime. While it would certainly be possible to create similar functionality for other dynamic languages like JavaScript, this is not a universally applicable approach, and it may be much harder to see similar speed improvements for all languages. They also speed up container creation by not using a network namespace, arguing that they provide no added security in the context of AWS Lambda. This approach may not be applicable to all deployments, and can not be considered a universal speed improvement either.

Although SOCK focused heavily on the high-level dynamic language python, this approach is still particularly interesting due to its applicability to existing serverless systems. If it were possible to significantly reduce the starting time of containers across many languages/runtimes, it may not be long until serverless platforms could be considered for time critical applications, and its usage could become more widespread.

Language Isolation

The reason containers are needed is because running arbitrary code in a process can have side effects that impact the whole machine. Normal processes can read and write to large parts of the file system and use up an almost arbitrary amount of resources. If many processes were to share a machine then, checks and balances would be required, which today are realised in the form of containers. The idea of language based isolation is that users do not necessarily need to run *arbitrary* code in their process, rather if developers could write their code in a language that cannot interfere with other processes by design, then there would be no need to isolate them at all.

Boucher et al. make use of the safety properties of the Rust language paired with some blocked system calls and limited access to the file system in order to cold start their serverless functions in *micro*seconds [19]. Each function runs in only a lightweight thread, isolated from the worker. The major drawback of this approach is that the functions must be written in an isolatable language (in this case, Rust), and dependent libraries must be trusted by a whitelist. The WebAssembly project aims to create a completely isolatable instruction format, and so any language that could be made to compile to WebAssembly could also be isolated [20]. While this appears to be possible for statically typed languages such as C++ or Rust, compiling dynamic languages like python or javascript will be much harder. Most AWS functions are written in Python (52%), followed by NodeJS (39%) [21], and although it is not yet possible to write serverless functions in Rust at any major providers, this language

selection disadvantage would make it difficult for developers wanting to migrate existing code to serverless or developers who do not know the selected language. This does not rule out the use of language isolation in serverless platforms completely, but limits its feasibility as a universal approach.

Unikernels

A normal linux distribution contains all sorts of libraries and binaries for general purpose use. Many of these debuggers, drivers and libraries are not needed in the normal execution of a specific task, especially in the case of running one function in one runtime, as in the case of serverless. Projects such as MirageOS [22], and others [23] have been built around creating minimal linux distributions known as *unikernels*. If these unikernels could be small enough in size and memory footprint, it may be possible to reduce the start time of VMs to be a feasible alternative to containers. By optimising the start flow of the Xen hypervisor, Manco et al. showed that small custom made VM images could be started in 2ms, comparable to the 1ms it takes for linux to *fork()*[24]. They modified the Xen daemon to "pre-warm" vms in the background and showed that their system was scalable to thousands of VMs.

While unikernels promise VM level at speed, they face major drawbacks. First, all libraries and runtimes must be ported to run in unikernels, as there is no "user-mode", although there have been significant efforts to create unikernels for popular languages. Secondly, unikernels, when run as VMs, have the same performance overheads normal VMs face. Finally, they are *not debuggable*.

One potential solution to these problems was presented by Williams et al., who successfully run their unikernels as *processes*. Although these unikernels used only 10 system calls, providing a similar level of isolation to VMs, they could make use of many of the native process performance benefits [25]. These unikernels could be started in milliseconds, but consistently outperformed traditional unikernel throughput.

Due to specialised and simple nature of serverless functions, using unikernels for speed and isolation is an appealing solution. However, the difficulty involved in testing unikernels may be its largest deterrent. In a survey of serverless developers, over half stated that a lack of testing/deployment tooling was a significant challenge for FaaS services [26].

Firecracker

In late 2018, Amazon announced their own bid to eliminate the cold start problem, named Firecracker. Firecracker is a lightweight virtual machine monitor that runs VMs using the kernel-based virtual machine api (KVM) [27]. Although VMs are traditionally slow to start, AWS's small (50,000 LOC) rust codebase provides very lightweight VMs that can start in around 12 ms (but always under 60ms) and with only 5MB of memory overhead per thread. Once the VM has started it takes 125ms further for the init process to start. Given that Amazon already use Firecracker in production, it has reduced the "container start" element of a serverless cold start, however, users must still wait for the start of their chosen runtime and libraries to load before their function can run.

Project Approach

Which approach, then, will dominate serverless platforms in the years to come? Language isolation and unikernels are both in their early stages of development. Language isolation may always remain language-biased in nature, and therefore perhaps not be a universal solution. WebAssembly, for instance, has a runtime and portable interface on its roadmap, but not in the foreseeable future [28]. Unikernels, while shown to work for simple use cases, have not yet been battle-tested in production, and it could be years before unikernel technology reaches a level of maturity where it can be trusted by industry.

Is it possible for containers to live up to the performance requirements of serverless? Some argue that linux containers can not achieve the desired throughput of serverless functions [29], so new approaches (such as unikernels) will be necessary. At the same time, the complexity of the Linux kernel has been growing exponentially over time [30], and so implementing new kernel constructs for serverless is becoming all the more difficult. This report aims to determine if existing container technology can be modified to create more performant platforms. The approach taken will be to make use of serverless functions unique requirements in terms of size, speed and access to state in order to *reuse* containers to run different functions. Process checkpoint/restore along with existing container isolation mechanisms will be the primary technologies used to achieve this goal.

In this project, we will only be targeting linux containers. Specifically, runc containers, the underlying container runtime used by docker which is the most widely used kind of container. In order to understand how one might go about safely reusing containers, we will first explain the layout of unix processes and containers, and dive into some background regarding checkpoint/restore technology.

Container Essentials

Processes

The unit of organisation for running anything on a unix system is a *process*. A process is a collection of data that describes the state and attributes of a running program. This data includes core information like its name, id, program counter and registers, but a process also consists of the files, pipes and mounts it owns, and the memory it has allocated. The *ps* command and the */proc/{pid}* directory are good starting points to understand the information a process contains. All processes are related, forming a tree hierarchy of processes. In other words, all processes are in some way a child of pid 1, which is the initial process.

The kernel's job is to schedule and run processes, and to stop them from excessively harming the system. Unix systems are organised into two "rings" according to privilege. Ring 0, is the kernel's domain (or kernel-space), which has total privileged access to the entire system. Normal processes should not have this level of control over the machine, and so run in ring 3, "user-space". Whenever a process would like access to a part of the system, say, to open a file, or allocate itself more memory, it must ask the kernel for permission.

Permission questions are expressed in the form of system calls (syscalls), and are the primary interface between a process and the kernel.

For the purposes of this project, we need to know how a user's code *can* mutate the state of the process in which it runs. For example, we might not want functions to be able to create children, which can be observed when a function calls the *fork* system call. We can use ptrace to guard this boundary, which will, for example, allow us to observe the processes state and prevent certain behaviour.

Ptrace

The *Ptrace* system call allows one process to almost arbitrarily control another. Ptrace is mainly used to create debugging tools, as it allows us to stop at system calls (eg. strace), set breakpoints and modify the processes state. Heavy debugging with ptrace is therefore associated with high overhead and performance degradation, and so would not normally be considered for production use. However, ptrace can be configured in different ways that cause more or less stops in the process and therefore more or less overhead.

One form of inter-process communication in linux is *signals*, which are a limited set of asynchronous messages that processes can send to one another. For some signals, the process can define what will happen when such a signal is received, such as cancelling a network download on the keyboard interrupt signal `SIGINT`. Other signals cannot be caught, such as `SIGKILL` which forcefully terminates the process, and `SIGSTOP` which stops it. Since they abruptly stop the process from executing, signals are not usually very heavily used in applications, they are rather used for exceptional behaviours, such as dropped connections or closed pipes.

Stopping when a signal is received is the basic ptrace configuration. When ptrace is enabled, the process will always enter "ptrace-stop" *before* a signal is delivered. While the process is in ptrace-stop, many commands that are useful to us can be run, such as examining the content of the processes registers. These will however only be needed when checkpointing/restoring, and not on a regular basis. We hope that the overhead of this basic ptrace configuration will be minimal, which we will explore in the evaluation.

Configuring the process to stop at every system call would certainly incur a performance hit from two context switches per system call and should be avoided [31]. Instead of causing consistent performance hits by constantly trying to stop certain behaviour at certain syscalls, we will take the approach of observing differences in state before and after the user's code runs. Ptrace can efficiently guard us against some behaviour, such as setting the `PTRACE_O_TRACEFORK` setting to stop the process when it tries to fork, and extremely damaging syscalls can be prevented by using `SECCOMP` (Linux secure computing) filters. Ultimately, we hope to efficiently observe state changes during restore.

This frees us to think about the state of a process as a collection of data, and how that data has changed over time, rather than trying to white/blacklist syscalls - an arduous and error-prone process.

Containers

The most important thing to know when coming to understand containers, is that a container *is just a process*. The container process has added measures to make it *appear* to the container that it has all privileges, complete control over the system, and sole access to all its resources, even though in reality, it is under the control of the host. These measures are also applied to all of the container's children, which gives the impression that a container is its own machine. The key difference to a VM is that since all containers run on a machine run as processes, they share the same kernel, whereas each VM has its own kernel.

Three important features are the key to creating this illusion. *Namespaces* control privileges, *cgroups* control access to the cpu and other resources, and a *layered file system* allows for fast, shared file systems between containers.

Namespaces

A namespace is the linux kernel abstraction that makes a global resource appear to be under the sole control of a process within it. It can provide this functionality in an isolated fashion to many processes [32]. This means, for example, that two processes can appear to have the same pid, or that processes can appear to have different hostnames. There are 7 namespaces.

Namespace	**Isolates**
User	User ID/Group ID spaces
PID	The process ID space
UTS	Hostname & domainname
Mount	File system mount points
IPC	Interprocess communication resources
Network	Network interfaces
Cgroup	The Cgroup root directory

These namespaces provide guiding boundaries for what state we must consider restoring between uses in this project, as we assume these primitives are strong enough to prevent modifications to the rest of the system. For example, we will have to ensure that no other processes are left running in the container after the function has finished, and check what new mount points may have been made.

Cgroups

While namespaces allow processes to be isolated from one another, they do not prevent the process from using, or excessively using, the systems resources. Cgroups are the Linux kernel feature that allow this kind of monitoring, implemented by a virtual file system at */sys/fs/cgroup*. Here, each resource (memory, cpu, etc.) has its own hierarchy of directories, representing the limits places on the processes within them. The cgroup interface also has a *freezer* which allows processes to be arbitrarily started and stopped without using signals.

Layered File System

For a container to appear to have complete access to its own system, it needs to have full access to its own file system. Complete file systems, though, are generally quite large, and copying an entirely new file system for each container would both be slow, and take up a considerable amount of space. Most containers will also use similar "base layers" in their file system, and large parts of the file system will never be written to. Containers therefore use *layered* file systems, where each change made to a file system is applied in a read-only fashion to a base-layer as a "commit". When a new container is made, a thin read-write layer is placed on top of the read-only layers, which the container works on. Any changes made to underlying layers are edited in a copy-on-write fashion. This approach allows many underlying layers of the file system to be shared between containers, regardless of their purpose, greatly increasing its performance, especially when scaling.

The changes to the filesystem must also be considered when reusing containers. Our serverless containers should only need read-access to the filesystem, potentially with write access to a "scratch space" mounted to /tmp. This way we should be able to reuse the complete file system between runs, taking care to clear or mount a new /tmp directory.

Checkpoint & Restore

The idea of stopping, checkpointing (creating a backup of) and restarting arbitrary processes has been around since Berkeley Lab's Checkpoint/Restore project started in 2002 [33]. Many of these projects required modifications to the kernel, or only supported processes with certain properties, until the introduction of Checkpoint & Restore in Userspace (CRIU) in 2012. CRIU has the ability to dump the complete state of a process to a file, and to start an identical process from just the information in that file, with memory, file descriptors and even its pid all in order.

Research has been done into live migrating containers using CRIU, which showed that a fairly heavyweight MySQL process could be checkpointed in 1 second and restored in 0.8 seconds [34]. Clearly if preparing a container for reuse would take 2 seconds, there would be little benefit in doing so over starting a new container. However, these C/R operations are slow because they output/input complete dump files containing all of a processes state, which may be hundreds of MB in size. In this project, it will not be necessary to read or save the *complete* state a process. Instead only the *difference* in state between the start and end of a running function will be considered. This is where the special properties of serverless functions are crucial, as the differences in process state when running small, fast serverless functions should be minimal. This should make it possible to live-restore a container efficiently.

We can see that the techniques used by CRIU in C/R are essential to this project, as they have dealt with examining and reproducing process state in minute detail. To provide an understanding of the key techniques used by CRIU, we will describe a typical process checkpoint and restore.

Checkpoint

When checkpointing a process, its entire process hierarchy of children must also be checkpointed. CRIU begins by recursively freezing (i.e. stopping via cgroups) all of these children and their threads. While doing so it also "attaches" to these processes via the ptrace interface. Ptrace allows a processes to arbitrarily observe and control another, and will be extensively used in this project.

Once CRIU has frozen and is in control of all child processes, it begins to read information about the processes. Much of this information can be read straight off the file system, especially from the */proc/{pid}* directory. In this project, we will be able to track differences in memory, file descriptors and more using information in this directory.

The rest of the information needed must be read by the process itself. CRIU injects *parasite code* [35] using ptrace into each process in order to do this, which saves information such as memory contents and credentials. All information is saved to dump files, the parasite code is removed and the processes can then resume. In our case, we will already be attached to the target container via ptrace, and so the worker should be able to read the rest of the process state through this interface.

Restore

When restoring, CRIU *fork()*s many times until it has a process with the desired pid. It then opens files, creates maps and sockets and restores other basic process information. The CRIU process will eventually morph into the target process, so some "outside" code must exist that will eventually unmap CRIU's memory, replacing it with the target processes' memory. For this purpose, a "restorer blob" is created. All memory mappings, timers, credentials and threads are finally restored by the blob, before restarting the process.

A lot of time during restore is spent creating a new process similar to the target process using expensive operations such as potentially thousands of forks to achieve the right pid. In our world, we already have a process that is *very* similar to what it was before the function ran and so also expect our restoring phase to be much more lightweight.

Language runtimes

Most Serverless platforms gain popularity partially due to their ease of use, partially due to what languages/technologies can be used with that system. In fact, some newer platforms such as OpenFaas pride themselves on allowing any kind of code that can be packaged as a docker container [36]. For the purpose of this project, we aim to show that the most popular languages used by serverless platforms today would benefit from and be compatible with a checkpoint/restore methodology.

For refunction to be able to use a given runtime, *dynamic code loading* is required. One of the main benefits of this system will be the ability to run functions without having to wait for the runtime to start - instead having a loaded runtime waiting to dynamically load and run functions. In dynamic languages such as python and js, such loading is easily done through

the `exec()` and `Function()/eval()` functions respectively. Java can dynamically load functions using the ClassLoader [37], Golang through plugins [38] and C has dynamic linking.

Refunction

The outcome of this project is *Refunction*. Refunction is an open-source go program that creates, monitors and restores live container processes for serverless functions [39]. The name stands for *re*usable *function* containers.

Refunction acts as a layer on top of containerd [40], the industry-standard container runtime manager used by Docker and kubernetes among others. Containerd, in turn, runs what it calls "tasks" which are running containers. Containerd can theoretically use any kind of container, but we choose the standard, well used and battle tested runc container [41].

Containerd is configured to create refunction's desired containers, like a python runtime on top of an alpine-linux distribution. Refunction then takes control of the container using ptrace, and manages the continued function lifecycle from there; inserting new functions, handling incoming requests and restoring the containers to their original state after a timeout. Once restored, the container is ready to safely run a completely different function.

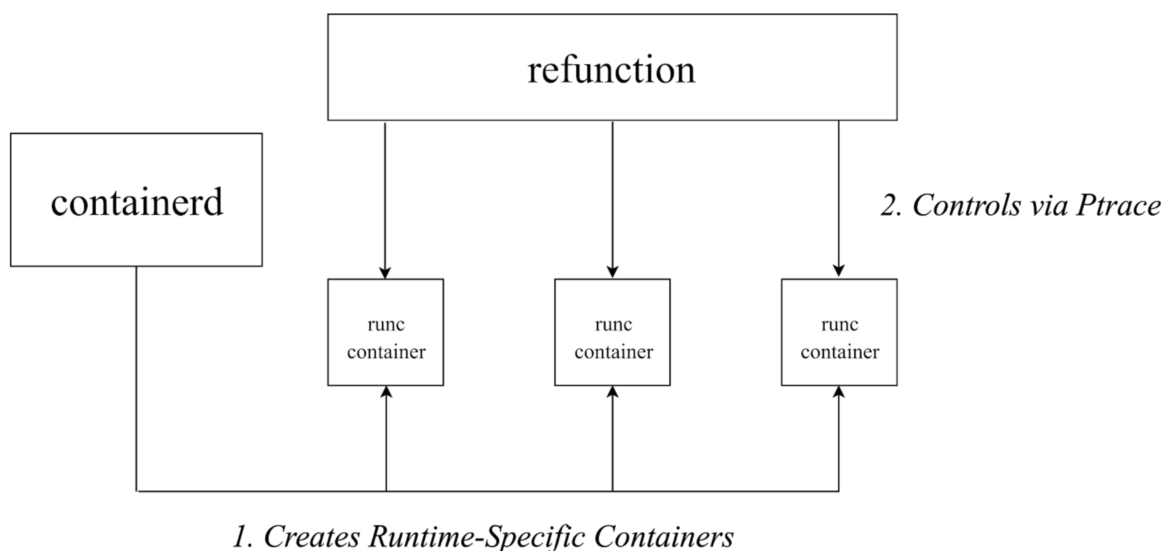

Figure 2: Refunction Overview

Refunction is intended to fit in to larger serverless systems as the "function running" component. For evaluation purposes, an "invoker" component has been built in order for it to run as part of Apache's OpenWhisk platform.

This project it built on the assumption that restoring a running process to the state it was in before it loaded the serverless function will be faster than starting up a new container, runtime and function. In other words that a container restore is faster than a cold start. The body of this report will explain how this restoring is done in a fast and secure fashion, and the evaluation will test whether the assumption holds under these conditions.

The *Worker*: Restoring Live Processes

The worker controls the complete lifecycle of a single container. It prepares, starts and controls the container.

Preparation

File System

Each worker will need a barebones file system containing essential system information and whatever is needed for the runtime to start. We try to keep this file system small and simple. A read-only file system can even be shared between containers, so all workers using the same runtime can use a single read-only filesystem. This makes it possible to run thousands of containers on a single machine very space-efficiently, which would be difficult with virtual machines. We create these file systems in *layers*.

Container filesystems are *layered* file systems, with each layer representing the difference (diff) to a filesystem on applying an operation (like `apt-get install nodejs`). Containerd creates and uses layers via *snapshots*. A snapshot represents the state of the filesystem at a point in time. It can either be *active* where changes can be made or *committed*, representing its final state. All snapshots have a parent, and so form chains to represent complete filesystems.

In this project, a container file system is made by creating snapshots from a runtime's saved docker image and then adding the custom function container code to a final layer on top. The `docker save` command creates a semi-oci [42] compliant tar archive containing a `manifest.json` file and a directory for each of the docker image layers. When the worker is used inside of the invoker component, these runtime archives are downloaded from an amazon s3 bucket and saved to the `/var/cache/refunction` directory when the invoker starts. The worker then turns these layers into containerd snapshots and creates a final layer containing the runtime-depent code that loads and runs user's functions. Snapshots are uniquely identified by an id, and shared between workers, so after the first worker has created its file system, other workers using the same file system do not have to repeat the same work.

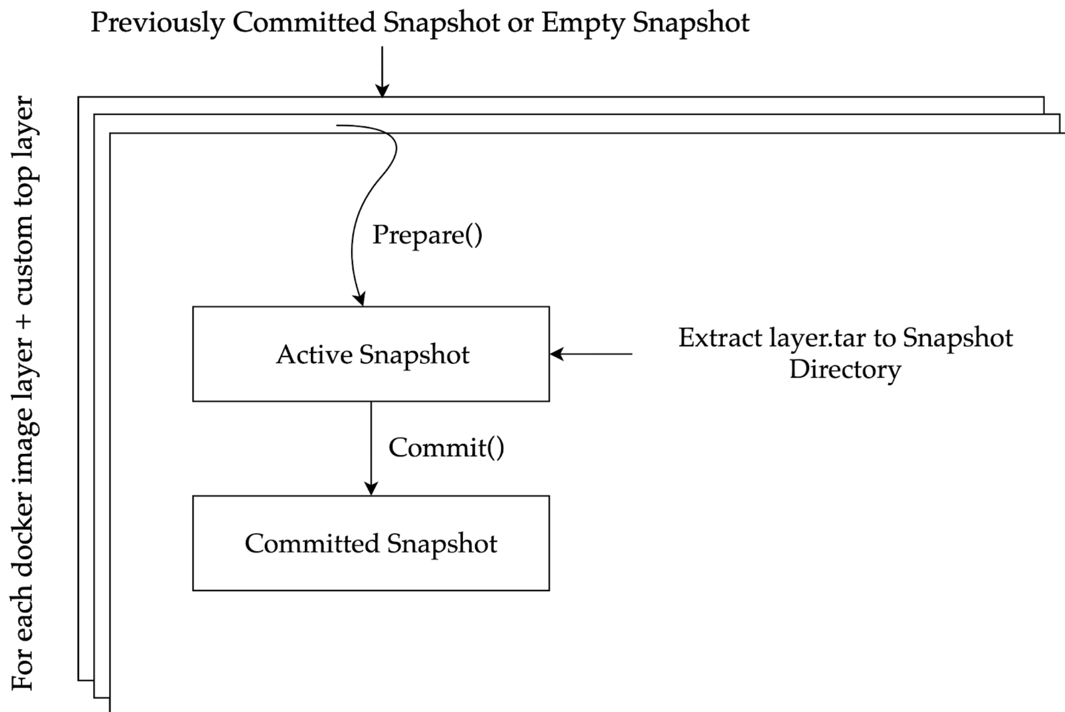

Figure 3: Layered Snapshots

Spec

Once the file system is prepared and the worker has a snapshot to create its container from, it needs a container *specification* describing the parameters for the container. Containers are highly configurable. Even choosing which namespaces to create for the container can be changed in the spec, so these parameters are vital to the security of the container.

Although there is a wide range of configuration parameters, we largely use the default linux container parameters defined by containerd [43]. These defaults provide us with our container basics - a root user in the normal namespaces with a few capability limits (importantly NoNewPrivileges to avoid escalation) and mounts for /proc, /dev, /sys and /run. Besides these defaults, we add a custom hook for bridge networking [52] and custom process arguments dictating what should be run by the container (for example the command "`python serverless-function.py`" should run when our python container starts). We also provide pipes to be used as the containers stdin, stdout and stderr - which we will use as our primary method of communication with the container.

Communication

Containers are designed to be completely isolated from the host - ideally, it should not even know that a host exists. How, then, should the container and the host talk to each other about which function should be run, and then about the requests to be sent to the function? We tried two approaches to communication, using network sockets and using stdin/out pipes.

Initially, communicating over the network seemed to be the logical approach. The container would open a socket listening to any incoming requests, and the host knowing the containers IP, could then send the container functions or requests - this is a fairly normal

thing for a container to do. However, we had some problems using this approach in a restoring context. When restoring the container, you would have to choose between using the same socket to connect with the host over many function runs or to open a new socket after each restore. Using the same socket over many runs is difficult because sockets are inherently stateful regarding data such as sequence numbers while restoring should make the process exactly the same as it was - including the variables in memory which store state about the socket. Starting a new socket on each function can also be problematic. When closing a unix socket, the operating system will leave it in `TIME_WAIT` state by default to prevent delayed messages from the sender being received by the next process to use that address. You could use a different port - which would require further communication about which port to use and it still leaves sockets lying around. Alternatively, it would be possible to do some dirty socket manipulation using linuxes' `TCP_REPAIR` option. This made using network sockets in this restoring context difficult.

Instead, a much simpler approach of communicating solely via json messages to the containers stdin/out pipes was used. Pipes are unidirectional fifo communication channels, and a standard way of communicating between processes. The pipes connected to a containers stdin/stdout are pipes just like any other, and the only way to directly connect pipes between the host and container. We therefore decided that these pipes could be used for message passing. All messages have the format `{"type": string, "data": JsonObject}` and messages are separated by newlines. We also allow the process to print normal data to standard out, such as error messages. These lines which can not be parsed as json objects can simply be logged for debugging purposes.

Function Loading & Request Handling

This project intends to eliminate as much of serverless cold starts as possible. There are three parts of a cold start. First, the container has to be created. It then takes some time for the runtime to start, and finally we must load all dependent libraries before the function is run. We must attempt to address all of these aspects. If we could restore the container to the state it was in before the runtime started, then would only avoid waiting for the container to be created. Although python can load in just a few milliseconds, other runtimes such as Java can take hundreds of milliseconds to start. That being the case, we also want to avoid waiting for the runtime to start. We will attempt to load function code and handle its requests from within the runtime, after it has started.

This means that the runtime must support dynamic code loading. This does not strictly limit us to using dynamic *languages*, however. Although JavaScript's built in eval function is well known, Java can dynamically load code with its `ClassLoader` [37], Golang has support through what it calls *plugins* [38], and there is even the *hint*[44] package which allows evaluation of strings in Haskell. While this requirement may exclude some languages, it still supports a wide array of the most popular languages for serverless.

For each new runtime to be supported, one must write the code that will dynamically load a function into the runtime and then call the function given a set of parameters. Given that our primary method of communication is via messages on stdin/out, we will send json objects

containing first the function code then the request parameters. The code in a runtime simply needs to act on these requests. An example of such code for the python runtime is shown below.

```python
def start_function_server():
    send_message("started")

    function_loaded = False
    while not function_loaded:
        try:
            # Parses a json message from stdin that contains
            # our python function
            function_string = receive_function_message()

            # Loads python function as string into runtime
            # Can also execute `import x` statements
            global main
            exec(function_string, globals())
            function_loaded = True
        except:
            send_message("function_load_failed")
            continue

    send_message("function_loaded")

    while True:
        request = receive_request_message()
        result = main(request)
        send_data("response", result)

    # Never finishes. Either killed or restored
```

Given a few helper functions to parse json and read from stdin, we can see that the barrier for adding a new runtime is fairly low. **Python 3.6, Node 8 and Java 13 have so far been added and shown to work in this system.**

Runtimes

We will now briefly describe how user code and libraries are loaded into each runtime, as it differs slightly depending on the runtime.

Python3 uses the `exec()` function as in the example above. This is a general purpose function that can interpret any python code. This makes it suitable for loading the user's `main()` function but it can also handle any `import x` statements or other initialisation code the user may want to run.

Node 8 currently uses the `Module._compile()` function, which is what is used under the hood when a module is `require`d by node. The user's function code must `export` whichever function they would like as their entry point under the name `handler`.

Java 13 uses a custom `ClassLoader` to load the user's class. In general, Java will lazily load all classes from ClassLoaders the first time they are called. We therefore need to define and register a new `ClassLoader` that will return the user's `Function` class when we call it. Our `ClassLoader` receives a base64 encoded string that represents the user's packaged jar file.

When the JVM asks our `ClassLoader` for a new `Function` class, we search the jar (which is essentially just a zip) for the compiled `Function.class` file, and return its contents. Any initialization can be done in the `Function` classes constructor. As for required packages, the `ClassLoader` can be configured to load any classes in the user's jar, so any libraries that the user may need can be packaged in the jar and dynamically loaded.

Isolation & Control

The Importance of Isolation

A container is an isolation construct that prevents a process from affecting other processes on a host. One could call this *spatial* isolation, as it describes that a container should not be affected by the things around it while it is running. In a world where we want to reuse the same container for different tenants, one must consider its *temporal* isolation. By this, we mean that a previously run function should not be able to affect future function runs, and a currently running function should not be able to deduce anything about previously run functions.

This is essential to the security of the container - a worst case scenario might be that a function manages to spawn a separate process inside the container that gathers data about later functions, or data may be corrupted in a way that makes future functions crash.

If we are going to reuse a container for multiple function executions, then it is important that we find a mechanism that allows us to modify the container's state to be exactly as it was before any external code was run. A daunting task, perhaps. "Missing something" may not be easily visible and could be fatal. In some ways, containers face a similar task where there is no "physical" barrier - only one in software. As with any software barrier, bugs in containers can crop up [45], but a large collective effort has brought the technology to a level where it is trusted by a large part of industry in production. This project *can not exhaustively deduce whether it is possible to provide complete temporal isolation* for containers. However, we *can* outline and tackle the main challenges in doing so, and show what benefits we might have from it. A much larger effort besides this project would be required in order to create trustworthy temporal isolation.

Refunction Isolation Approach

In the end, we believe that the success of this project comes down to ensuring cheap temporal isolation. In our case, that translates to being able to safely restore function containers quickly. For this, we will use a two-level approach. First, we aim to use low level process information to restore the primitives of the container. We look at pages of memory, the file system as a whole and all process registers to get a complete picture of which parts of the processes state has changed. Knowing that the actual bytes of the processes memory are the same is much more comforting than concerning ourselves with how the function might have been loaded in a particular runtime and the effects that it might have had.

Secondly, we accept that we do not have to be able to restore *all* of the processes state (at least initially). Being able to *detect* misuse of the container is a good enough backstop to ensure the integrity of the system. For example, many users may open sockets to fetch data from the network, and typically such sockets would be closed when the user's function is finished. If a malicious or faulty user attempts to leave a socket open, being able to see the

open socket and kill the container instead of loading a new function would keep the system secure.

Our approach is to restore the containers state if “nothing went wrong” in the execution of the user’s code and the user was not trying to be malicious. In the small portion of function runs where this is not the case, detecting (which is normally much easier than restoring) is key. We have decided that in this situation, killing the container is acceptable.

Container Control

In order to be able to restore a process, we need to know what the state of the process originally was, and what has changed since. As described in the background, ptrace offers a window into the state of a process. It has functions for stopping and starting the process, getting/setting registers, changing areas of the process memory, preventing certain system calls and much more. We will use these functions to help us record the state of the process before and after the user’s function runs. The same functions can then help us modify the process to again be in its original state.

In order to run these special ptrace functions, the parent (tracer) must *attach* itself to the child (tracee). In linux, a task or a “thread” of execution *is* a process. Each task/thread has its own stack, registers and `/proc` file system entries. It shares most of its memory with its parent, but is still considered as an independent process. Due to this, each task must be attached to individually, and all subsequent ptrace calls must then be made from the same task that the attach call was made from.

We would like for the “checkpoint” of the process (the point at which the original state is saved) to happen after the runtime has started, but before the function is loaded. This is because we are not troubled by any actions that the runtime may have to perform in order to start, such as spawning threads or sending internal signals - every user’s function expects these actions to occur before their function is run. To take this initial checkpoint at the right point in time implies that there should be some level of synchronization between the tracer and tracee as to when the tracer should assume control and the checkpoint should start. We attempted two methods to synchronize the parent and child. The initial attach implementation was a signal based ptrace dance.

When a process is brought under a tracers control using `PTRACE_ATTACH`, ptrace will pause the process and notify the tracer when certain actions happen. Depending on the *options* you have set, this might be when a certain syscall was called, but the tracer will always be notified when the tracee receives a signal. Synchronizing on signals was the first approach used. The tracee could notify the tracer that it is ready by sending a certain signal to itself, which the tracer would catch, and the tracee could be notified that the tracer has completed the checkpoint by injecting a certain kind of signal which the tracee would interpret as a signal to move on.

While this approach worked well for python because the python runtime does not make use of the unreserved `SIGUSR1` and `SIGUSR2` signals, it did not hold for other runtimes. Node uses `SIGUSR1` to start the debugger, and the JVM uses both signals internally.

Instead, we opted to use the simpler stdin/out pipes, as we did for loading the functions. On startup, the tracee simply writes a “started” message to its stdout, after which it blocks, waiting for messages on stdin regarding which function to load. When the tracer/parent receives the “started” message, it knows that the tracee is ready and already in a blocking state (waiting for a message on stdin). This is a suitable time for checkpointing. It can then safely attach to the container and checkpoint before sending a message with the function it should load.

In order to trace multithreaded processes, we have to do a bit more work to control each thread individually. For each task in the processes `/proc/pid/tasks` directory, we will start a goroutine that is locked to a thread using `runtime.LockOSThread()` (by the same tracer/same tracee rule). This goroutine can call `PTRACE_SEIZE` to attach to the process, and will then spend the rest of its life in an event loop waiting for commands from the worker to send to the tracee. The tracee can only be sent commands in ptrace-stop state, so the goroutine will forever `wait4()` the tracee to change its state. When stopped, the goroutine can be sent ptrace functions to run, like getting the values of the tasks registers or telling the tracee to continue after being stopped. All such actions are synchronized with the main worker using go channels.

Due to the same tracer/same tracee rule, we can not debug with strace at the same time as refunction is tracing the process. Particularly for this project, understanding what syscalls functions are running is interesting, so we have instead implemented an strace option that will stop at and print every syscall for debugging purposes.

Checkpoint & Restore

If a container is just a linux process tree with added isolation constructs, and a linux process is just a collection of information about the state of a running program, then checkpointing a container would mean taking a copy of this process state, and restoring would be editing the current state to match that of the checkpointed state. With a comprehensive understanding of process state, this should be possible.

A simplified model of a process has *registers*, *memory* (its virtual address space, managed by the kernel), and a *file system*. We will now discuss how each of these process elements can be restored.

Registers

Registers resident in the CPU are distinct from other process memory. Depending on the architecture, their count and types vary. In general, a process has a collection of general purpose registers (rax, rbx etc.) and a program counter. On many architectures special floating point registers and vector registers also exist. When in ptrace-stop (which can be triggered for example by signals), a tracer can run one of a few `PTRACE_GETREG` commands to retrieve the current values of these registers. In a future stop, equivalent `PTRACE_SETREG` commands exist to replace the current register values.

Since linux task "threads" share a lot of state with the main process, they can mostly be ignored during checkpoint/restore. However, registers are an exception. Each task has its own independent set of registers. This means that all tasks must be stopped at the same time during checkpoint or restore, and that `PTRACE_GET/SETREG` should be run on each task separately.

Memory

Think of a heavy runtime that needs to allocate itself a lot of memory to run, like a JVM with a large heap. If we were to naively restore this memory, we would take a complete copy of it before the function run, and "paste" it back into place once it was done. This poses two problems. If the process had allocated hundreds of megabytes of memory it would be slow and, in the case of our simple serverless functions, we expect that most of this memory will not change. We would like to find a more efficient way of deducing which memory needs to be restored.

Each linux process is given its own complete virtual address space for it to use. The process is not actually allocated the entire physical address space when it starts. Instead, the kernel keeps a mapping between physical and virtual memory locations on behalf of the process. When the process asks for access to a pointer not previously used, a "page fault" will occur, and the kernel will transparently allocate a new physical page to that virtual memory address before continuing. This is called paging, and the process has no knowledge that it happens.

The process utilises its entire virtual address space. At the lowest addresses is the text (code) of the program itself, followed by initialized and uninitialized static data segments. From here on the heap grows "up". At the top of the address range are any command line arguments and environment variables, and from here the stack grows "down". The address space is also used when files are mapped into memory.

`/proc/pid/maps` shows us an overview of the memory usage of the process. Shown below is an example of a running python process. It is important to notice the access level of the process to some areas of memory. The second entry (read-only python3.6 at `557602ea5000`), for example, is for initialized data (initialized static variables) - these are not ever expected to change, so even the process itself cannot write to them. We can also see various libraries needed by python here. Depending on how they are mapped into memory, the process may or may not need to write to them, but these areas of memory are all backed by a file on the file system.

You will also notice that there are "anonymous" mappings that have no path. These are areas of memory allocated to the process on calls to `mmap`.

low addresses

Region	Address range	Access	Path
text (code) & data	557602ca4000-557602ca5000	r-xp	/usr/local/bin/python3.6
	557602ea5000-557602ea6000	r--p	/usr/local/bin/python3.6
	557602ea6000-557602ea7000	rw-p	/usr/local/bin/python3.6
heap	557603f92000-5576040d0000	rw-p	[heap]
mapped files & malloc data	7f522ddb7000-7f522de77000	rw-p	
	7f522de77000-7f522e313000	r-xp	/usr/local/lib/libpython3.6m.so.1.0
	7f522e313000-7f522e319000	r--p	/usr/local/lib/libpython3.6m.so.1.0
	7f522e319000-7f522e37b000	rw-p	/usr/local/lib/libpython3.6m.so.1.0
	7f522e37b000-7f522e3ac000	rw-p	
	7f522e3ac000-7f522e439000	r-xp	/lib/ld-musl-x86_64.so.1
	7f522e439000-7f522e639000	rw-p	
	7f522e639000-7f522e63a000	r--p	/lib/ld-musl-x86_64.so.1
	7f522e63a000-7f522e63b000	rw-p	/lib/ld-musl-x86_64.so.1
	7f522e63b000-7f522e63e000	rw-p	
stack	7fff3bce8000-7fff3bd0a000	rw-p	[stack]

high addresses

Figure 4: Labelled `/proc/pid/maps` File of a Python Process

Knowing more about the processes memory, we realise that we will only have to take a copy of all *writable* areas of memory at checkpoint time. It is important to note that we need to store copies of these writable areas of memory elsewhere in memory. This incurs an extra running cost in the system which will be evaluated later. An interesting further research

question could be to consider whether this memory state could be shared between several containers on the same machine.

In restore, we would like to avoid completely rewriting all writable areas of memory if possible. Ideally, we would like to know which pages were actually written to during the running of the function, and only restore them. Luckily this functionality was built into the linux kernel by the CRIU project [46]. By writing a particular number to the `/proc/pid/clear_refs` file, the "soft-dirty" bit is cleared for all pages associated with the process. On every page access from then on, the kernel will set the soft-dirty bit in its page table entry. Therefore once the function has finished running, only the pages with their soft-dirty bits set will need restoring. The soft-dirty bits can be read from each pages entry in `/proc/pid/pagemap`, and the `/proc/pid/mem` file gives us raw access to the processes memory to read to/write from.

Unfortunately, memory restoring is still not as simple as copy/pasting pages of memory that were used by the process. This will work in basic cases - for simple network functions or functions that mostly use cpu, but if a user's code is very memory intensive, the story gets a little more complex.

We have, up until this point, looked at restoring a group of memory areas in a processes `/proc/pid/maps` file. We made the assumption that the size of these maps remained the same, and that the number of entries in the maps file remain the same. Let us explore what happens when these assumptions do not hold.

Area Size Changes

Most interesting code needs to use memory. From a programming perspective, when we need more memory we will normally ask `malloc()` for some, and say that we are done with it by calling `free()`. `malloc` is just another library which will try to allocate you some memory on the heap it already has and then uses `brk` and `mmap` to allocate more space.

We mentioned earlier that the heap grows "down", but how long for? We already know that a process is not allocated the entire physical memory space when it starts, so there must be a defined end. The end of the heap is called the program break. The program break is actually initially set to the first address after the uninitialized data segment, so initially the program *has no heap*. By calling the `brk` system call, we can change the position of the program break, which effectively allocates more or less memory to the process. As system calls are relatively expensive, `malloc()` will allocate itself large chunks of memory with `brk` and then allocate its memory to you internally. This means that there will normally be some space left on the heap. For us, this means that simpler functions will not need to grow the heap/call `brk`, so the area size will not change.

When a user's code uses more than the heap available, calls to `brk` are inevitable, and the size of this area of memory will change. What should we do with this memory when restoring the process? Leave it? Let us assume that the memory may be used to store sensitive credit rating information on customers. It should not be possible for the next user's function that runs to simply read that information straight from memory. This leaves us

with two options. Zero the memory and leave the program break where it is, or change the break back to where it was.

Zeroing the memory is a brutish solution, but will do the job. However, it leaves some lingering questions. Will the restored program know that its heap size has magically changed and be able to use it? Over time, will this not lead to containers being allocated more memory than they need, simply because an earlier user needed it?

Restoring the program break back to where it was is certainly the cleaner solution, but is more difficult to do. The `brk` system call would have to be run from within the execution context of the traced process - there is no `pbrk` call that changes the program break of another process. Suddenly the container in which the user runs its function needs to run some special restoring code. Let us explore how this might work.

Imagine we simply placed any special restore code we need after the user's function runs. For example a snippet of python code that calls the `brk` system call to deallocate memory. Unfortunately, we would never be able to ensure that this code actually ran the way we intended it to. After all, the user controls the memory and execution of the process while the function runs, it could remove or modify our code. Instead, we will have to insert our restoring code into the process after the function has run.

CRIU's approach to running code in the tracees execution context is to inject a *parasite* [35]. A parasite is a blob of position independent code that can execute outside of the processes normal stack, and remove itself when it has completed running, leaving the container in the same state as it was before injection (minus changes caused by the parasite code itself). Parasites are powerful, but complex, they have to be compiled into assembly instructions for a particular machine, and must communicate with our tracer over a socket to receive command parameters.

The way parasite blobs are inserted into the user's address space may offer us a way to avoid complete parasite injection. The parasite must be placed somewhere, like an area of memory created by an `mmap` syscall. This syscall must also be run from the processes execution context. In order to do this `compel`, CRIU's parasite injection library, finds the first executable area of memory, and changes the instruction to be a syscall instruction using `PTRACE_PEEK/POKEDATA`. After properly setting the registers for the relevant syscall (in this case `mmap`), the process is allowed to run the syscall, then stopped on its return. The instruction and registers can now be replaced to what it was, returning the process back to its original state. This method of syscall injection is enough to let us run arbitrary syscalls inside the processes execution context.

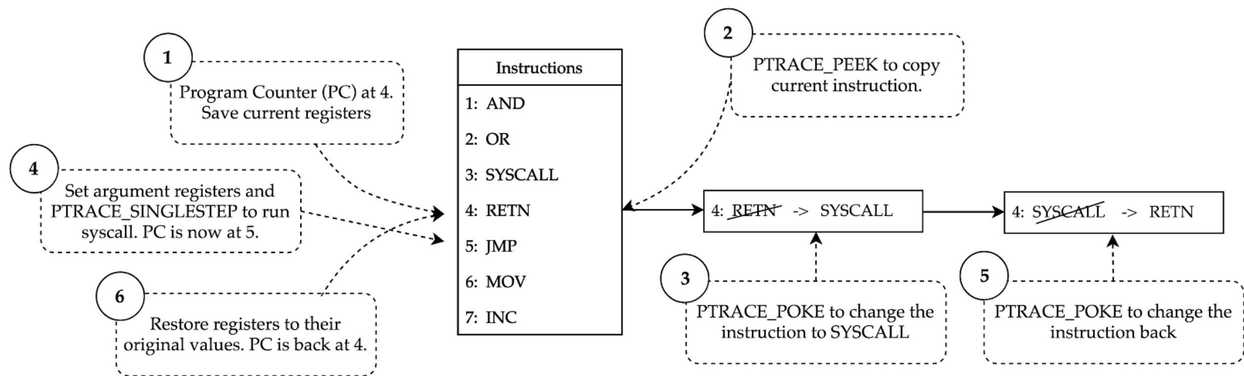

Figure 5: Running a System Call Inside the Execution Context of the Tracee

In our case, our tracee will be a stopped process currently pointing to an instruction in memory. We simply replace the current instruction with `SYSCALL`, and the `PTRACE_SINGLESTEP` to run only that instruction. After changing the registers and instruction back, the process is as it was.

We use this method of syscall injection in order to reset the programs breakpoint to where it was. Since setting the program break to a value lower than what it was means that you let the kernel reclaim that memory, this has the added benefit of removing all newly allocated memory to the process. When the next function needs to allocate memory to itself using `brk`, the kernel will provide the process with fresh underlying pages.

We can also extend the use of this method to `mremap` any other areas of memory where the size has changed.

New Memory Areas

Not only the size of memory areas can change, but also how many memory areas are available to the process. This can happen if the user's function needs to allocate a large area of memory (`malloc` would `mmap` instead of `brk`), or if the user maps a file into memory. These are legitimate use cases, so we should be able to handle this too.

Luckily, noticing new memory areas are as simple as re-parsing the `/proc/pid/maps` file. Unmapping memory can be done with the `munmap` syscall. Since we now have a method of calling arbitrary syscalls, unmapping the newly allocated areas is trivial.

Removed Memory Areas

This will be a much less common case for normal use cases. The areas of memory created when a runtime is started are usually required during the entire operation of the runtime, such as standard libraries etc. We will assume that the removal of areas is caused by unintended or malicious use. Although we could create new mappings for these areas and fill them with our saved copies, detection and termination will suffice for now.

Finally, we also perform many other small checks on each memory area, such as checking that their inode and device numbers remain the same. We will again assume that any differences in this data is uncommon and cause for termination. This leaves us in a position

where we have a good overview of the processes address space and any changes that could have been made by the user.

The File System

File systems can be extremely large and intricate graphs of data. If the user was given complete control of the file system, exhaustively calculating what changes might have been made could be far too expensive. Many current serverless platforms have read only file systems with a "scratch space" located at `/tmp`. We should be able to enforce isolation between functions under this premise with minimal added restore time.

First, we know that any of the file system that is read-only could not have been changed by the process on disk. This is enforceable even in containers that appear to have root access using mount namespaces. When using a mount namespace, the container file system will be mounted as read-only in the host mount table. The user inside the container will be in its own namespace and see its file system mounted at `/`. If the user, for example, tries to remount `/` as read-write, it would not be allowed by the true owner - the mount table in the host space.

Users can still load read-only files into memory, which are modifiable. Here there are two scenarios. One case is where the file was loaded into memory before the function ran, in which case refunction would know of its existence and have memory change tracking on, making it possible to restore. The other case is when the function loaded the file into memory after starting, in which case the new memory mapped file would be noticeable in `/proc/maps`. In this case, the tracer would either have to unmap/obliterate the file before the next function would run, or abandon the restore. Most importantly these changes are easily noticeable.

Finally, we must reason about the state in the writable `/tmp` directory. Each new function must be given a clean `/tmp` directory, so the previous directory either needs to be wiped or a new one must appear in its place. AWS Lambda tmp directories are given up to 512MB of space in `/tmp` [47]. Actually removing 512MB of data from disk would slow up the restore time. One alternative solution would be to mount a new `/tmp` directory on top of the current one. This would be much faster, ensure a clean directory, and the host could empty the old `/tmp` directory at a later point in time.

Therefore if container file systems can remain read-only and if we can safely and quickly provide a new `/tmp` directory to a container during restore, we can ensure temporal isolation in refunction container file systems.

Other Restorable State

Many or perhaps most users serverless functions will want to use the network. One example may be a function that takes a postcode as input and queries several weather apis for the weather at that location. Such network connections will usually be made with a socket, which should be properly closed during restore as not to affect the next function. If the

socket is closed by the user (rather than the remote), the socket will enter a `TIME_WAIT` state in order to ensure that the remote receives an acknowledgement of the connection termination request. This ensures that the next user to use the socket can not receive any messages from previous connections. This state can be problematic because `TIME_WAIT` can be many seconds long. CRIU deals with this problem by placing the socket into `TCP_REPAIR` mode where the socket can be terminated regardless of its state [48].

There are other small pieces of state that should be restored in a fully safe restoring system. These include: signals that the process may have in its queue, timers that may have been set, new mounts that may have been made and lingering or modified file descriptors. All of these pieces of state are observable in when the process is in a stopped state, and given that the CRIU project can restore them, we expect them to be equally restorable in this case.

Library Loading

We have previously postulated that in order to completely eliminate cold starts, the container start, the runtime start and then all library loading must be considered. Having tackled the first two, we should address the third. Serverless functions stand on the shoulders of giants. While the functions themselves may be under 100 lines long, they may need networking or data processing libraries that are thousands of lines long and need to keep their own internal state. These libraries can often play a large part in the time it will take a function to start.

Since each function has different dependencies, there will never be a one-size-fits-all solution here. Instead, we will aim to improve the library loading performance for many common functions, while making sure it is possible for functions with special dependencies to be able to use them.

For clarity's sake, let us generically classify a dependency. All libraries are somehow *loaded* into the runtime before the function that depends on it runs. Usually, this means that a collection of files is existent in the file system are loaded into the memory of the runtime, and some initialization code might be run.

Our suggested baseline approach would be to have the most popular libraries in each language already available in the containers image. As a provider, one would have to choose a whitelist of libraries in each supported runtime in a few selected versions. These libraries could either be already loaded into the runtime before the user function loads or simply on disk to avoid the transfer cost. Pre-loading libraries on behalf of the user should be done cautiously as it is possible for packages to interfere with each other. At the very least it should be clear to the developer which packages will be already loaded to avoid unwanted surprises. This option might still be considered acceptable, for example, it may be worthwhile to import some common python system libraries like `math` and `json`. Storing popular libraries on disk is a cheap way to speed up the loading times for many cases, especially given that function containers on the same machine can share filesystems.

At the final level, there should be a mechanism to load the required libraries into the runtime after it has started but before the user's function is loaded. As with the function code itself, this data can be transferred over our stdin pipe. Once the container has received the raw bytes of the libraries to be loaded, they somehow need to be interpreted by the runtime.

Normally this data would exist as files on disk that the runtimes import machinery loads the library from. Given that our file system is read-only, saving these files on disk would both be costly and very difficult. Once you realise that the first step any runtimes import machinery takes when presented with a new dependency is to open and read the data of all relevant files, we can see a potential solution to this problem. With some more in-depth knowledge of

the runtime, we should be able to skip the "read file from disk" step, and directly pass the runtime the bytes that those files would contain.

We have already used this trick once. The way we loaded our java class into the JVM was by using a `ClassLoader`. This allowed us to pass the raw bytes of our compiled class file directly to the runtime. Java is a particularly nice example to use here, as it lazily loads all classes. Before the user's function is loaded, we can send the same `ClassLoader` all of the class files that the function relies on. Then when the function is eventually run and the new dependency is stumbled upon, Java will ask all available `ClassLoader`s if they have a copy of that class available. Luckily it will!

Any runtime that can dynamically load code the same way our user's function code has to be loaded is also capable of dynamically loading libraries, which at their core are also just code. We think that this approach will be sufficient in the case of less common dependencies that must be loaded on demand.

Towards Zero-time Library Loading

How might we completely eliminate library loading times if we could? The user's dependencies would all have to be loaded into the runtime before starting. This means that all the dependencies a user could ever need would have to be loaded into every container. At the end of the day this is impossible because there is no way we can trust all packages in any runtime and we have no idea how these packages might interact with each other, let alone the amount of memory needed for this.

As we stated earlier, there is no one size fits all approach, but there are certainly common types of serverless function. Consider the problem statement for this project, "is it possible to reuse the process state of a container for several function runs?". After a dependency has been loaded, but before the user's function is loaded, the process is in a state where any user's function that requires this dependency could be run. Can we reuse the processes state caught at this point in time? This project has taught us that reusing process state certainly is possible.

By importing a library, a developer places their trust into it. They commit to the state of the system when that library is loaded. Every user that imports the same version of that library signs the same contract. If we can supply the exact same state of the system as the user committed to, "sharing" this library between different users functions could be deemed safe. Usually, a developer places their trust in several libraries. To share several libraries between users they would either have to want the same ones, or we should devise a minimal subset and return to a point in time where only those libraries were loaded.

At this point we are faced with an issue that we have considered before. Previously we considered whether or not we had to store all of the processes memory on a checkpoint and deduced that we only wanted to store what the user could *change*. Using `clear_refs`, we could then deduce what pages actually changed after then function ran. We can apply the same technique here. Rather than storing a completely new checkpoint, we can store the

difference in state between before and after the library was loaded. We can call this incremental checkpointing.

Incremental checkpointing could provide a way for "kinds" of functions to exist in a serverless platform. Function containers deployed on containers with many of its dependencies already loaded would start much faster. On the other hand sending a function to a container with no dependencies in common would cause the container to have to "roll back". The key question here would be what the cost of rolling back is compared to the cost of loading the library in the first place.

The key to organically minimise rollbacks is for the controller to alter its scheduling. The aim would be to schedule a function to an invoker that already has containers with many of the libraries it needs loaded. In other words, you would want to schedule functions with the similar dependencies to the same invokers. Luckily OpenWhisk's scheduling is currently focused on reducing cold starts - with that much less of a problem with refunction, scheduling attention can be focused elsewhere.

For some workloads this could be a deal-breaking improvement. Think of serverless functions that serve predictions based on pre-trained machine learning models. Avoiding to have to load large machine learning libraries such as TensorFlow might make this use case much more appealing.

This approaches strength comes in numbers, if enough people were to be using the same libraries as you, the probability of you not having to take the time hit of loading them on startup becomes small. If you are the only person using an obscure library, you should still expect to have to load it.

Due to the resource constraints of a single person project, incremental checkpointing has not yet been implemented. However, all of the mechanisms by which it could work have been implemented, which leads us to believe that it is possible.

The *Invoker*: Scheduling the Worker

To understand the impact that restored containers can have on serverless platforms, we should fit the refunction workers into a larger serverless system. If our *worker* is a single managed container, we need a component to manage several workers on a machine. In OpenWhisk, the component that schedules containers is called the *invoker*. In general, there is one invoker per machine where functions will run, whose job is to receive requests from the controller, potentially pull function code from a database and decide how best to run the function.

The OpenWhisk invoker is a well contained component that only interacts with the controller (the entry point) via kafka, and needs access to the database where functions are stored. We built a new *refunciton invoker* component for this project, that manages restorable containers and replaces OpenWhisk's current invoker component. The refunction invoker is written in golang and is part of the same open source repository used for the worker. It uses go-kivik to communicate with the couch database and kafka-go [49] to attach to the relevant kafka queues. Kafka-go had to be modified to speed up the read polling loop which has subsequently been contributed back to kafka-go.

The invoker makes the best use of its available resources. For both OpenWhisk's invoker and our modified one, requests received from the controller will be scheduled in available slots until there are none left. A "slot" will approximately describe a fraction of resources available on the machine (like a 256MB container with a share of CPU) and should not be overprovisioned. Slots can then become free in two ways. Either enough time elapses without receiving another request (the so called "keep-warm" period) or the invoker begins to evict functions, which must happen when the invoker is overloaded. Normally a slot becoming free means shutting down a container. Using the refunction worker, we instead restore the worker. This means that we can achieve a much higher throughput using our new invoker as the restore time should be much shorter than the time it takes to shut down one container in order to start a new one for a different function.

The way functions are scheduled when our refunction invoker is used is both one of the main strengths and weaknesses of this project. While a normal OpenWhisk invoker *may* have some "pre-warm" containers with a ready runtime, all refunction invoker containers must be pre-allocated and "warmed". For example, in the normal OpenWhisk flow, if an invoker receives a java request, but only has python containers pre-warmed, the invoker would start a new java container to run the function code in. Our aim is to never let such a cold start happen. Refunction invokers instead always have a fixed pool of started containers for a particular runtime. This means that requests *must* be scheduled to invokers that have containers running in the appropriate runtime. A python request must only be sent to an invoker that has python containers in its pool.

This trait makes serverless platforms better by eliminating the need to wait for runtimes to start but introduces a new scheduling problem for operators. There now has to be enough

capacity for the functions of each individual runtime in the system rather than the simpler problem of ensuring the cluster has enough capacity for the sum of all functions. Although we will later see how refunctions greatly increased throughput will partially mitigate this problem, one would still have to introduce a more sophisticated algorithm to change the allocation of runtimes according to demand in a successfully run a diverse serverless platform. We do not address this problem as a part of this project, instead assuming a fixed set of available containers on each runtime.

Methodology & Environment

When dealing with low-level systems code, knowing that your code works as intended can be difficult and frustrating. Many actions may be very hard to view by human inspection like bits set on a particular line in a `/proc/` file. Some actions like a change in state of a process may happen too quickly to notice. Systems code can also have a tendency to be flakey - it might work on one run but not on the next. On top of this, when the code is experimental, it may be necessary to rethink and overhaul sections in the space of a few weeks. Ensuring that all of your previous built functionality still runs as intended can quickly become a very difficult or impossible task.

All of this makes the methodology and environment for developing systems code particularly important.

Language

Golang was chosen as the language of choice for this project. One important reason for this is that the underlying container systems, containerd and runc, are both built in golang and as such have all necessary bindings for running containers. A more important reason for choosing golang is that it is a very powerful statically-typed language that makes it easy to interact with low-level parts of the system in a standard way. For example, much of the ptrace interface that this project heavily relies on comes as part of the standard library.

Libraries

It is worth noting why containerd/runc was the chosen container runtime. Containerd is really just a daemon that runs any kind of container under the hood, and so could run kata containers, or Amazon's firecracker vms. It was chosen because it (although a very young project) is already the de facto standard in industry for running Open Container Initiative (OCI) [42] compliant containers. It is trusted by docker, kubernetes, cloud foundry etc. Runc is the now open-source component of docker that actually spawns containers. It has been around since 2015 and is well trusted and extensively used and scrutinised by industry [50].

Testing

This project was built in a behaviour driven fashion primarily using integration tests. In a behaviour driven world, you write tests describing how you think your application should behave, and then write code that makes those tests pass. Integration tests are well suited to systems code since the code primarily speaks to the kernel to change the state of various parts of the system, rather than speaking to other components. Creating tests that check how the system was affected by particular function calls then makes sense. Most of these operations are very fast, so run quickly despite not being unit tests. BDD tests should also help readers understand the effects your code has without having to deep dive into the codebase. For example, the code below demonstrates how a function container might be restored.

```
It("can restore and change a python function", func() {
        Expect(worker.Activate()).To(Succeed())

        function := "def main(req):\n  print(req)\n  return req"
        Expect(worker.SendFunction(function)).To(Succeed())

        request := "jsonstring"
        response, err := worker.SendRequest(request)
        Expect(err).NotTo(HaveOccurred())
        Expect(response).To(Equal(request))

        Expect(worker.Restore()).To(Succeed())

        function = "def main(req):\n  print(req)\n  return 'unrelated'"
        Expect(worker.SendFunction(function)).To(Succeed())

        request = "anotherstring"
        response, err = worker.SendRequest(request)
        Expect(err).NotTo(HaveOccurred())
        Expect(response).To(Equal("unrelated"))
 })
```

This testing suite made it possible to discover and fix tricky flakes such as errors caused by commands being called from the wrong thread, and it gave the confidence to rewrite the entire container activation sequence at a late stage of the project without introducing new bugs. This goes to show how valuable good software engineering practice can be even in experimental research projects.

The entire testing suite is also run in parallel and in random order by a continuous integration tool on every commit pushed to GitHub, as an extra net to catch bugs or flakes.

Evaluation

We have slowly been outlining an alternate approach to running serverless functions. Ultimately, our aim has been to eliminate cold starts almost entirely from serverless systems. The approach we have taken has certainly increased the complexity involved in running serverless containers, so we now aim to evaluate what the cost of this added complexity is.

We have two burdens of proof due to our checkpoint/restore approach. Firstly, we have had to show that function containers can be *safely* restored, which we have thus far named temporal isolation. Although we have not produced an exhaustive proof of this, we have attempted to convince you that we can provide temporal isolation through our implementation. We believe in the strength of our low-level approach. Our implementation is not complete due to the limitations of a one-person project, but even so, we hope that our reasoning about the primitives of linux processes has been enough to persuade you of the feasibility of this approach.

Our second burden of proof is to show that there is little or no change in the performance of serverless functions by using this approach. We have lightly touched on the potential performance impact ptrace can have on processes and so have avoided excessive use of it during the running of the function. We should explore if these earlier assumptions hold. The other performance impact our approach can have is the overhead of storing the checkpoint states of all running containers in memory, which we should also explore.

Besides these burdens of proof, we would also like to display the strengths of restoring functions. We predict three improvements. First, we expect cold start times to be greatly reduced. Secondly, we predict that there will be less latency spikes as the function concurrency increases. Finally, we anticipate that our system will be able to achieve a markedly better throughput, particularly when dealing with diverse sets of functions.

We will now explore these burdens of proof and system benefits by using our refunction invoker within an OpenWhisk deployment.

Experimental Setup

Our OpenWhisk is deployed to a cluster of 6 machines running Ubuntu 18.04. Each machine has 16GB of memory, an Intel Xeon processor with 4 3.1GHz cores, and a 7200rpm 500GB hard drive. We use another separate such machine to run our tests on. All of these machines are located in the same datacenter (or even rack), connected by the same switch.

We use Kubernetes [51] to deploy the "normal" OpenWhisk components. That is, all components except for our special refunction invokers. Our refunction invokers require root ptrace capabilities and extensive access to the `/proc` filesystem. Although it may be possible to enable these capabilities in order to run refunction invokers inside containers, it simply was not a priority in this project.

Our OpenWhisk kubernetes deployment only varies from the default setup in a few ways. We use our custom docker images for the invoker and controller. For the controller we want to change the language of OpenWhisk's "health action" which the controller schedules to check that invokers are healthy. For the invoker we want to change the default keep-warm time, which we will discuss later.

We also change the affinity of the cluster to force certain components onto certain machines. We do this to spread our resources as much as possible, and to minimise interference. Therefore the main OpenWhisk components: the kafa queue, the couch database and the controller are on three separate machines. The 3 invokers we use also have a machine to themselves.

On the same machines that the OpenWhisk invokers are deployed to, we deploy our refunction invokers. This allows us to switch between OpenWhisk invokers and refunction invokers by removing the "invoker" label from the machines in kubernetes (which deschedules the OpenWhisk invokers) and instead starting the refunction invokers.

The Invokers are configured to have space for 8 containers each. With our default memory limit of 256MB this only costs us 2GB, but we also do not want an overloaded cpu to impact our performance, so we stick with the lower container number. This leaves us with 24 container slots in our system.

Cold Starts

Cold starts occur when there is no deployed container available to respond to a request. This will happen either if the function has not been run in a while (and been decommissioned) or if the number of concurrent requests increases. Let us explore the effect function restoring has in these two cases.

Low Throughput Cold Starts

In order to save resources, serverless platforms will have a grace time during which they will leave your container running even if no requests are received. We call this the keep-warm time, which is normally configured to be half an hour or more. If the throughput of the function is low enough, the functions container will constantly be decommissioned and every invocation will result in a cold start. As the throughput increases beyond a certain threshold, the container will not be decommissioned so every invocation will be warm.

We therefore tested the effect of cold starts at low throughputs. In order to speed up the time taken for each test, we changed the keep-warm time in OpenWhisk and refunction invokers to be 20 seconds. With this keep-warm time, we expect throughputs below 3 invocations per minute (one every 20 seconds) to always incur cold starts. We collect 50 measurements at each step by invoking the `wsk` cli with no-op functions in java and python. We use a no-op function because we only want to measure the function round-trip time and so the time it takes to run the function is inconsequential.

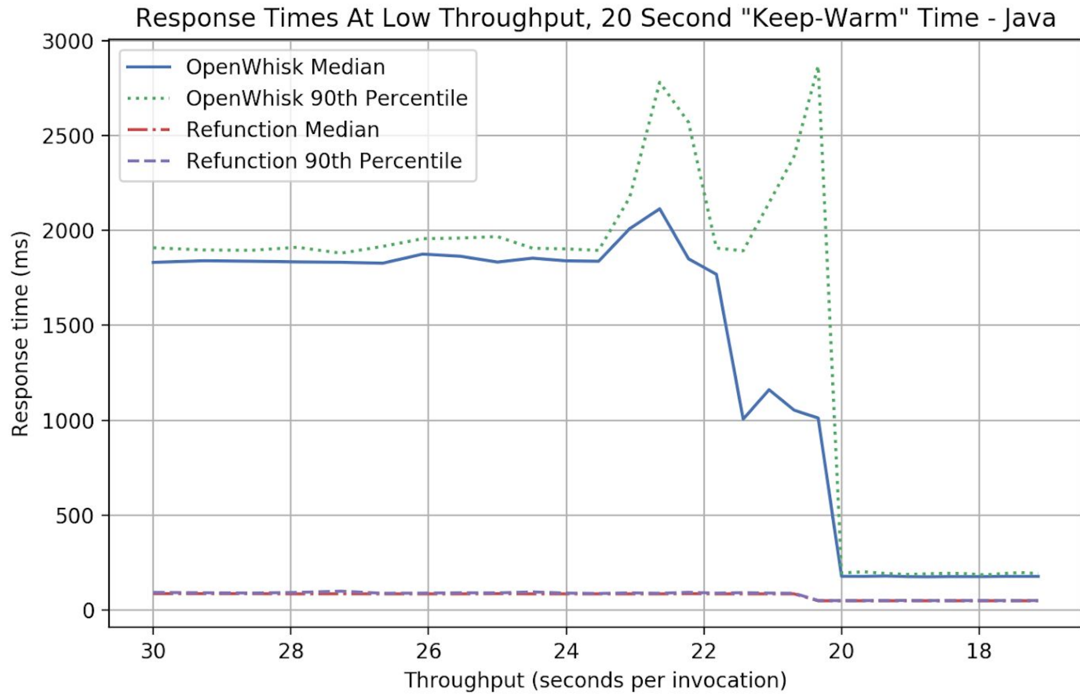

Figure 6: Java Low Throughput Cold Starts

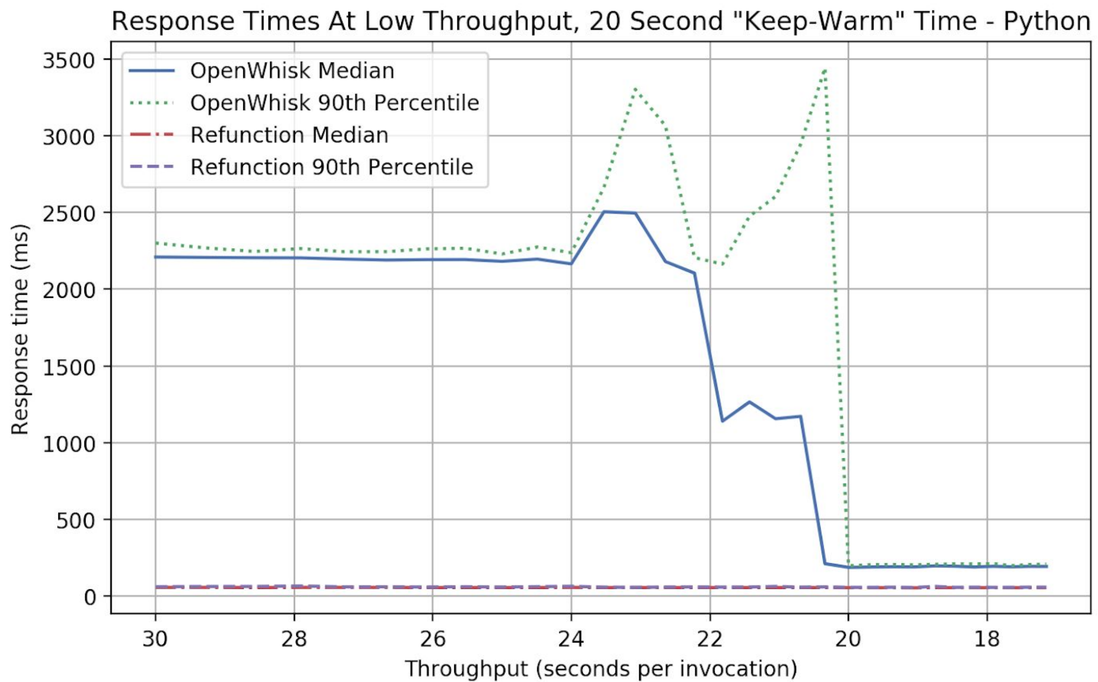

Figure 7: Python Low Throughput Cold Starts

The results are clear, we have decreased the cost of a cold start by an order of magnitude. The median and 90th percentile results for refunction are so close that they appear overlapping on the graphs, the actual difference between the two is around 5ms.

Some notes on these results. Firstly we notice that just at the boundary of the 20 second keep-warm time, OpenWhisk can handle cold starts even worse than normal. This is because

it is possible for OpenWhisk's invoker to schedule an incoming request to a container slot that is still being decommissioned, which can incur extra wait time.

We note that our refunction invoker also performs better than OpenWhisk's when the containers are warm. This is because OpenWhisk pauses containers between each invocation in order to save resources. Unpausing containers when receiving a request takes around 80ms, which explains our latency difference.

Serverless platforms pride themselves on making efficient usage of available compute resources. Keep-warm times are costly for serverless platforms. Currently, keep-warm times have to be fairly high in serverless platforms because their customers would not stand for frequent performance spikes. The higher the keep-warm time, the happier the customer, but the more resources that are left idle. Reflecting on these results, we consider the possibility of reducing the keep-warm time in serverless platforms using function restoring. A performance hit of 30ms between warm and cold functions could make keep-warm times in the order of seconds acceptable.

We have to be careful when considering the above argument, however. Here we have demonstrated the cold start time of a no-op function. Most serverless functions will have some dependencies that have to be loaded before the function can run. All such dependencies would add a constant value to both OpenWhisk and refunction start times.

We should also remind ourselves that this speed of cold start is only possible when the system has available capacity for it. In a serverless system using refunction this is more difficult than it was before because containers for a particular runtime must be available, rather than just capacity on machines to start new containers. We will shortly describe how running out of capacity is a much less dire situation to be in than it was before, but this added scheduling complexity should be noted.

Concurrent Request Cold Starts

cold starts are not just a problem if your serverless function is used very infrequently. At any point in time when the number of started function containers is not enough to deal with the amount of incoming requests, the system will have to boot a new container to serve your request, causing a cold start. This will most commonly happen when your function needs to scale according to demand.

Imagine a service that suddenly has to deal with a spike in traffic. As the spike begins, the required concurrency will steadily increase. This scenario is approximately equivalent to comparing the response times of a function as it scales from zero to peak concurrency. In our setup this means scaling from 0 to 24 concurrent requests (we have 24 “slots”). In order to simulate the system being “overloaded” we will slowly ramp up from 0 to 24 concurrent threads. A ramp up time of 24 seconds will cause a new thread to start every second, which will call a python function that sleeps for one second. The second long sleep forces the deployed containers to be almost always occupied. This forces the serverless system to steadily deal with one more concurrent requests.

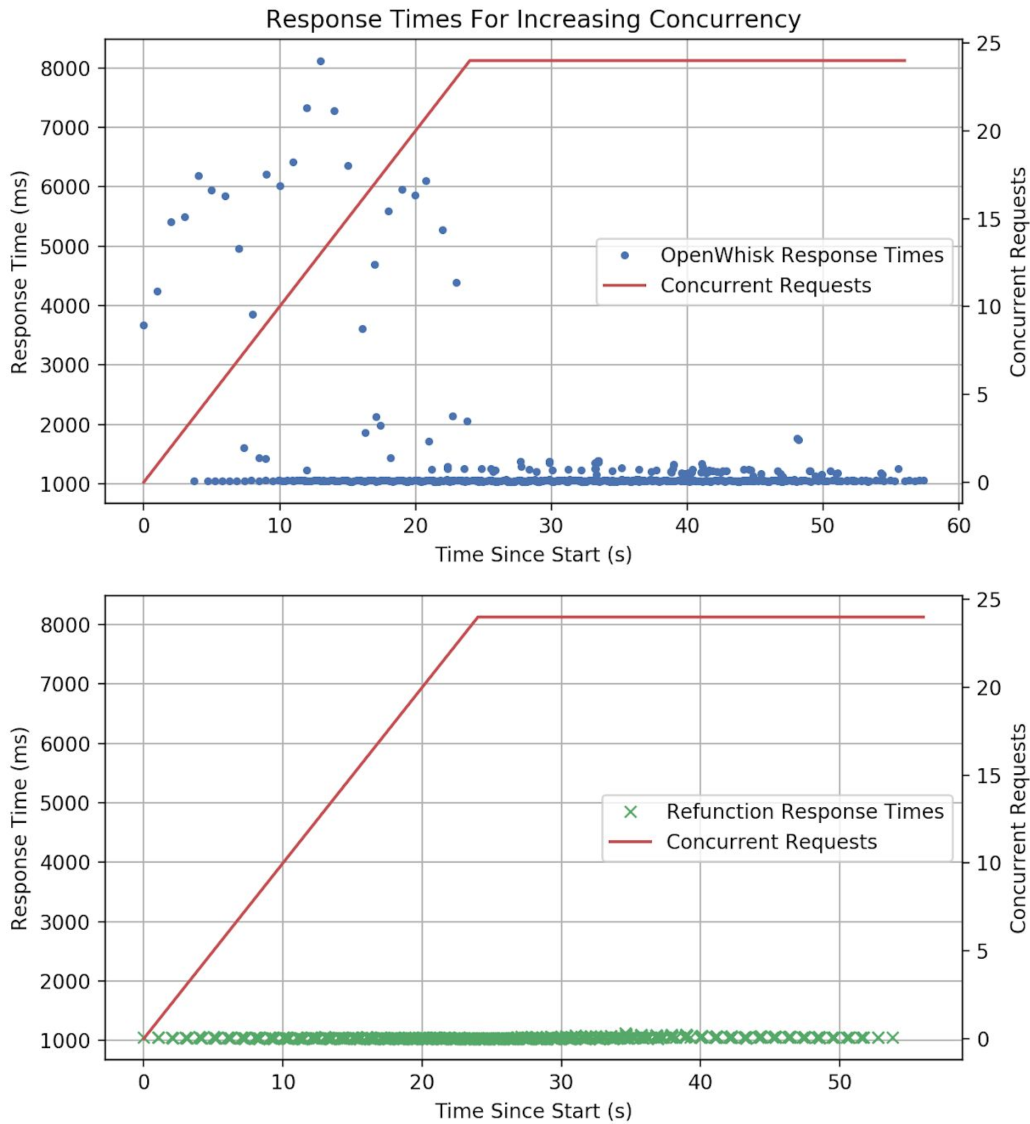

Figure 8: sleep(1) Response Times at Increasing Concurrency

The results show us that scaling functions also cause cold starts. We can see that OpenWhisk can really struggle in this scenario as the scheduler piles more requests onto the same invoker until it reaches its capacity. Since starting a container takes more than a second, container cold starts begin to pile up. Starting several containers in parallel clearly slows everything down. Some resources like network namespaces even have to start sequentially, so the starting containers may wait for global locks. Refunction manages this scaling smoothly, where the cost of loading another copy of the function into memory is small.

We see that as the number of concurrent requests evens out, the average response time for OpenWhisk decreases dramatically as it now has enough containers deployed to respond to every request.

These results should, similarly to our low-throughput tests, be understood in our no-op function context. Given that we load almost no libraries (in this case only pythons `time`), the

function load time is minimal. If the function had heavy dependencies, their loading times would be added to both refunction and OpenWhisk cold start times.

Throughput Improvements

Our wins in terms of reduced cold start times offer serious improvements for serverless platforms, but we think that function restoring can provide more than these edge-case gains. Given that our restore time is much faster than the time it takes to shutdown/startup a container, the overall throughput of serverless systems using refunction should improve.

In a simple scenario of a single function deployed to a serverless platform we would generally expect OpenWhisk and refunction to have the same response times (after a while) if the request rate stays the same. Both would have the same number of deployed containers, which would be enough to handle all incoming requests. Only the underlying hardware and the resources allocated to each container would affect its speed. However, no installations of serverless platforms will handle single functions. Providers and larger organisations will install a serverless platform in order to be used by a wide range of functions. Serverless functions are in general of a much smaller and simpler nature and users are expected to want to deploy many more of them than they would of say, full applications.

We want to measure how platforms perform when dealing with many different kinds of requests. We will make the same number of calls to OpenWhisk while randomly choosing which function to run from an increasingly large set size. The first test will call the same function 1000 times, the second will randomly choose one of two functions 1000 times, then one from three 1000 times and so on. We will call the size of the set the *saturation* of the system, where 100% saturation would be 24 different functions due to our 24 available slots for containers.

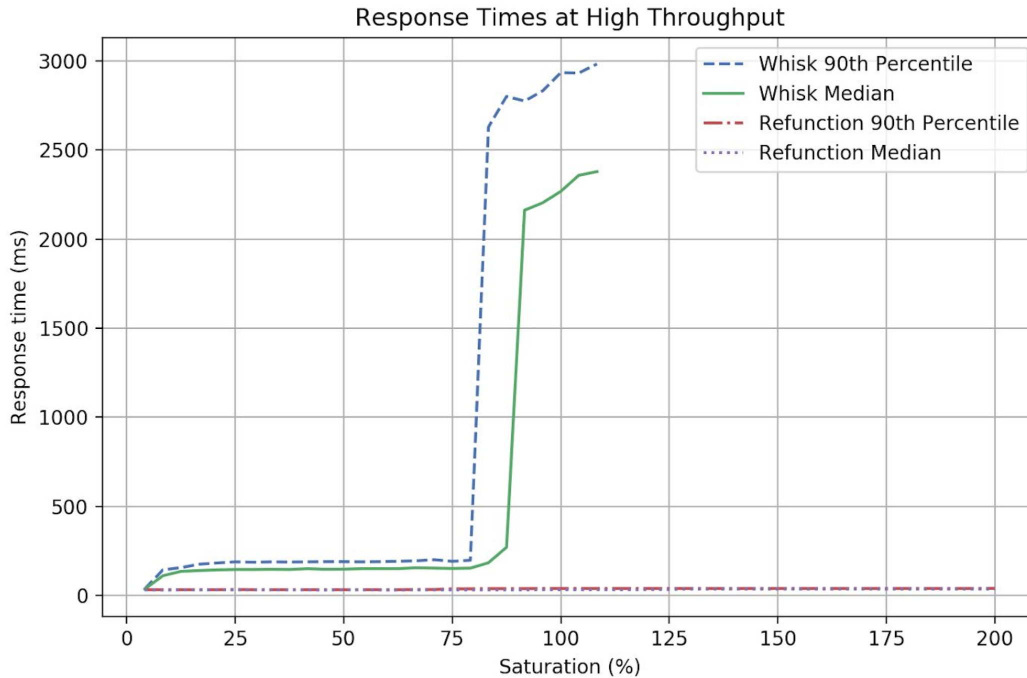

Figure 9: Response Times With Varied Saturation - One Thread

As expected, both systems perform as well when dealing with a single function. With lower saturations, both can provide a stable, constant throughput. As OpenWhisk reaches a critical point, however, it no longer has enough capacity for all of the available functions. When all of OpenWhisk's container slots are filled with deployed functions and it receives a request for a function not yet deployed, it is forced to evict a container and start a new one to handle the request. This is why we see the response times for OpenWhisk skyrocketing to more than that of a cold start as it reaches 100% saturation. Refunction, on the other hand, can happily perform a forceful eviction by means of a restore when it needs to, which in this case takes only a few milliseconds.

This is a simple experiment with only a single thread performing all requests sequentially. When faced with many concurrent requests, OpenWhisk would perform even worse since starting containers in parallel increases their boot time. We must also realise that refunctions restore times will not always be this fast - python is the fastest runtime to restore, and we made the assumption that the functions did not use many large libraries which would have increased the restore time further. Regardless of this, we can conclude that refunction performs better with diverse workloads and in “overload” scenarios.

Why is it useful to increase the system's throughput? For two reasons. Serverless platforms are already popular in part because of their cheap and granular billing model. Costs are calculated to the nearest fraction of a second, and several million requests can cost tens of dollars. What we can see is that a system that can reach a higher throughput can utilise its available resources better. This means that the service can be offered even cheaper for users and that providers might not have to over-provision resources to the same extent *in case of* a spike in traffic.

Measuring Restore Times

When discussing the maximum throughput achievable by refunction, we mentioned that it will depend on the types of functions being run, and therefore the time it will take to restore them. This could depend on three factors. The kind of runtime used, the size and number of libraries loaded, and the amount of memory the function uses while running. Let us explore in what way these factors will affect the restore time.

Type of Runtime

The restore times of our three types of runtime were measured with a noop function over 100 runs.

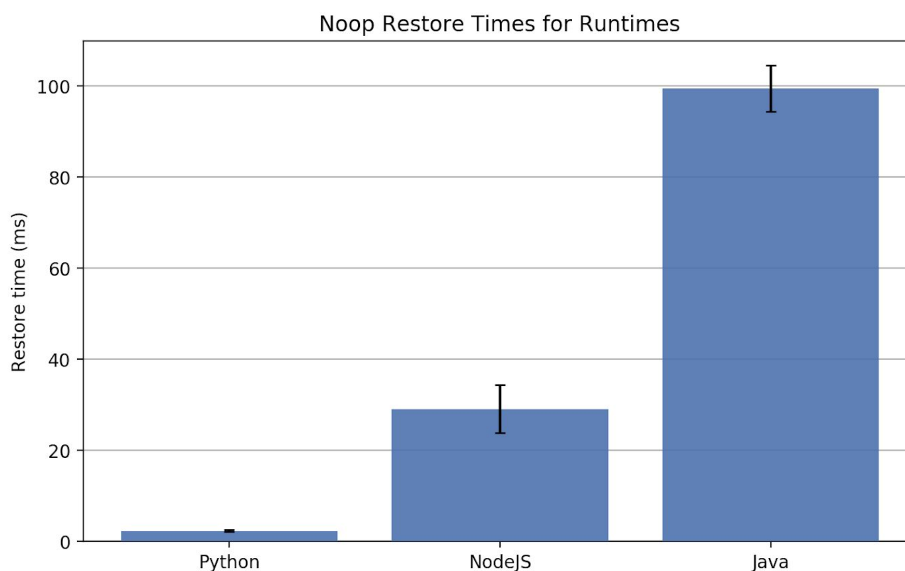

Figure 10: Mean Restore Times for Different Runtimes

The kind of runtime has a large impact on the time it takes to restore the container. This depends on how "heavy" the runtime is in terms of how many threads are needed, how much state the runtime has and how much the runtime needs to use that state. For example, Java seems to need to touch many pages of its memory while running functions, so much of the time it takes to restore java is changing the memory pages back to how they were.

Libraries Used

Every imported library will undoubtedly change the state of the system. To test the impact this would have on restore times, we tested the time it took to restore a few common python libraries over 100 restores. The associated function loaded with the library was just a no-op in order to only measure the restore time of the library. We also measured the size of these libraries when loaded into memory by recording the amount of memory used by the process before and after the library was loaded.

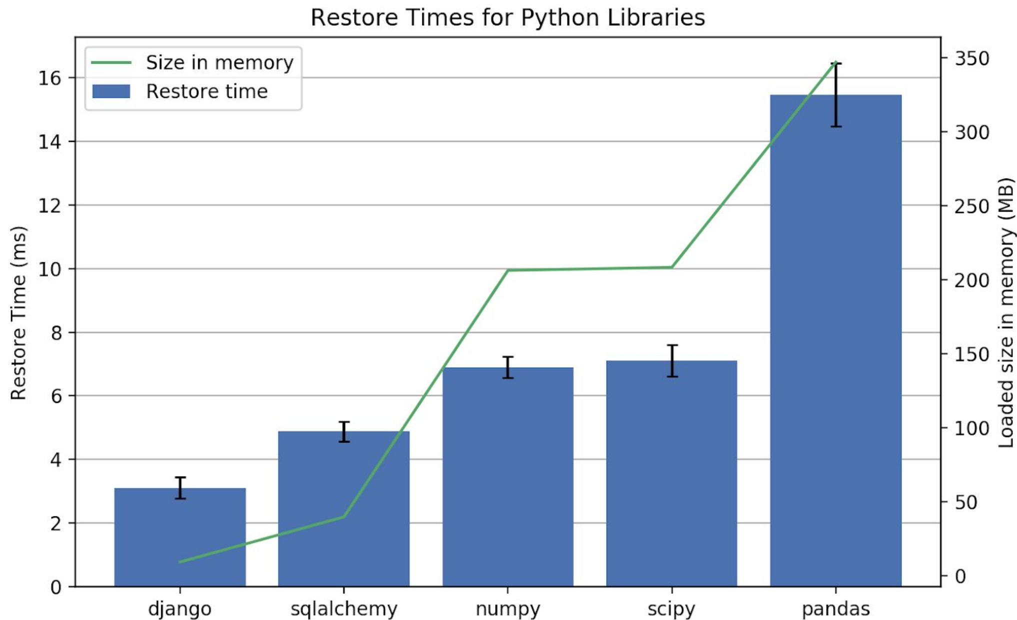

Figure 11: Mean Restore Times and Loaded Sizes of Common Python Libraries

Here we can see that the added time it takes to restore a function with imports is roughly correlated to the loaded size of the library in memory. We deliberately say *roughly* because this added restore time will again be correlated to the number of pages that have been touched by the library which therefore have to be restored along with the time it will take to unmap new memory regions. While the size of a library in memory will indicate the effect of these impacts, the actual impact will vary.

Memory Usage

To measure the impact the memory usage of a function has on restoring, we recorded the restore time when running a python function that allocates x MB of data, where x is the argument passed to the function. The function is restored and reloaded between each invocation.

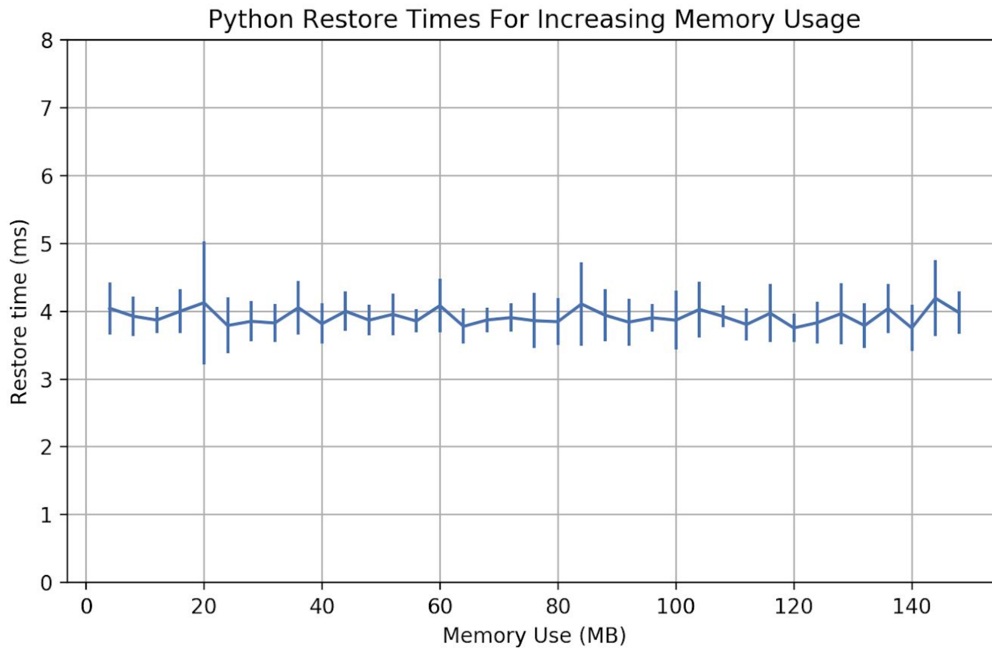

Figure 12: Mean Restore Times for Memory Intensive Functions

We include standard deviation bars to display the spread of restore times. As the memory usage increases, the restore time remains constant. This is because restoring a function with an increased heap size is the cost of calling `sbrk` to restore the program break. Any new pages that the process needed on the heap for its use are reclaimed by the kernel, which has a constant-time cost.

Memory Overhead of Storing Checkpoint State

The speed of our restore comes at a cost, and that cost is storing a copy of the processes writable state in memory. We store these memory areas as raw bytes that are later used to restore those pages that were modified.

On the same system with the same underlying container image, this size will remain constant. We expect the type and version of runtime to be the only factor influencing the checkpoint size. We display the checkpoint sizes of our three runtimes below.

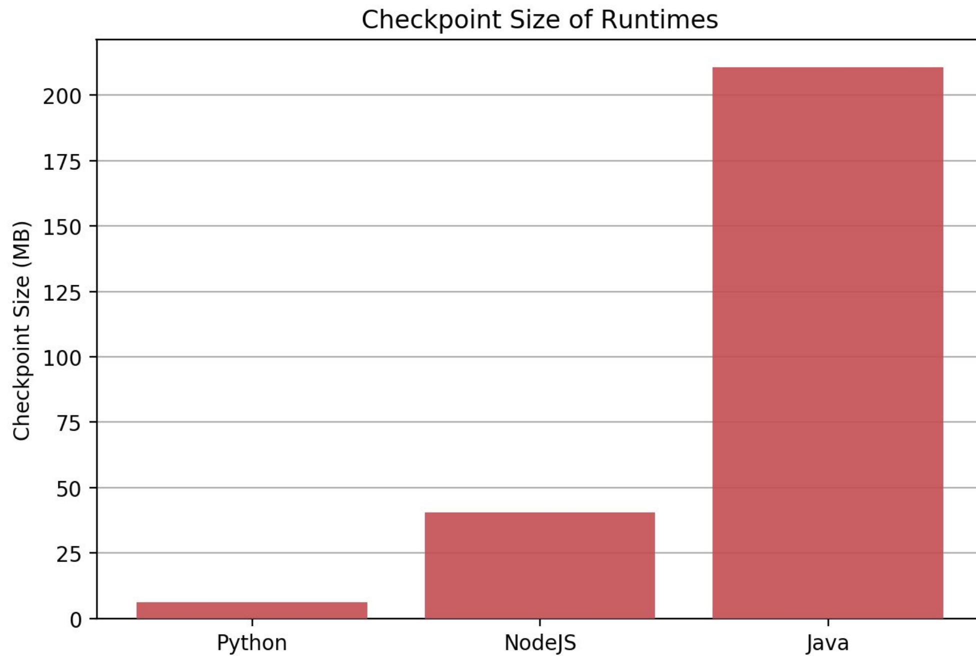

Figure 13: Size of Runtime Checkpoints in Memory

The size of a checkpoint varies considerably between runtimes. At the lower end of the spectrum the cost of the 6 MB checkpoint for python will certainly not have an impact on the system, while a 210 MB java checkpoint for a container whose memory limit is 256 or even 128 MB can be costly.

Currently, the same memory checkpoint will be used over many function restores, so the checkpoint cost is therefore not per user, but divided by all the users that use that container. This is one reason why storing large checkpoints can be motivated. We also note that memory is cheap, while sharing processor cores may be more expensive. In our earlier concurrency experiment, we saw how the startup time for containers could increase fourfold because of the strain put on the system by container starts. We should play off the cost paying for a few extra GBs of memory to store checkpoint state vs the strain placed on the system by starting containers afresh.

This problem could be alleviated in the future. Through byte-wise comparisons of checkpoint states, we found that 98.8% of java checkpoint states are the same. This leads us to believe that either much of a checkpoint could be shared between several containers, or the process could be forced to use the same memory layout by using other restore techniques used by CRIU. This could be an interesting area of future research.

Ptrace Resource Overhead

Ptrace is known to incur overhead when used. In particular, it would require extra computation time at its boundaries. When protecting the syscall boundary using `PTRACE_SYSCALL`, ptrace can slow down syscalls by an order of magnitude. We have to ensure that ptrace does not incur overhead when we do not have syscall tracing enabled.

To test this we call the syscall `getpid()` 10 million times, both with the process being attached, and when it is not. We use `getpid()` as a our syscall because we want to measure the time that it takes to switch between user and kernel execution contexts, rather than the time it takes for the syscall itself to run. `getpid()` which reads a single variable is then a suitable choice.

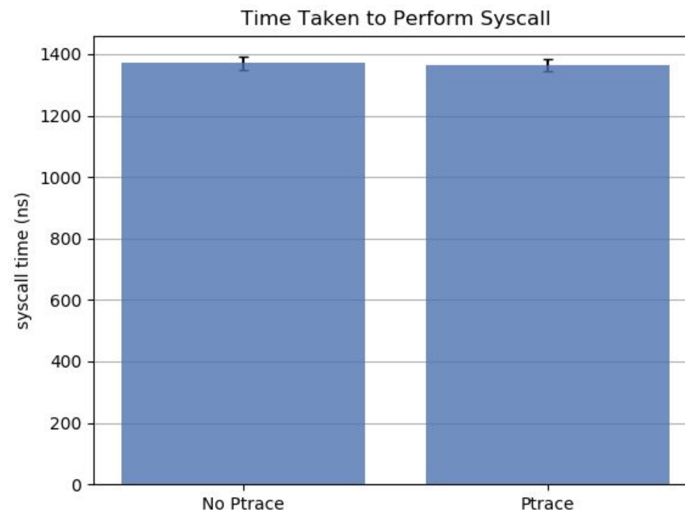

Figure 14: Nanoseconds to Perform `getpid()` With and Without Ptrace

Luckily this shows us that ptrace is not slowing down our program when we do not have the extra syscall option enabled. The only other time ptrace will have to work while the process is running is when signals are being sent to the process, which in most cases is rare. Therefore, since we only have the tracee attached while the user's code runs without any options enabled, we can conclude that we have no ptrace overhead.

Conclusion

We have presented *refunction*, an alternative approach to deploying serverless functions that reuses containers. Our approach addresses all constituent parts of a cold start: container start, runtime start and library load. Our *restore* method can alter the state of a processes' registers, memory and file system to the way they were in before any user's code was run. We are able to restore Python, NodeJS and Java functions in 5-100ms, 20x faster than OpenWhisk's 2 second cold start time. We outline a simple, generic way to support more languages. Using dynamically loaded functions paired with restores also increases the overall throughput of serverless systems that have diverse workloads, up to 20x. We have laid the foundations for how this method can be regarded as secure, by observing and manipulating the primitives of the container process.

Although checkpoint/restore projects such as CRIU have been in the works for a long time, applying these concepts to live processes has been completely virgin territory. Dealing with raw process memory can certainly result in obscure and frustrating behaviour. The key ingredients to success were a meticulous behaviour driven integration test approach paired with in-depth knowledge of the structure of linux processes. In general, I have found that all cryptic behaviour becomes entirely logical with enough inspection, although sometimes this clarity can only be achieved through sheer determination.

This project demonstrates that a container reuse solution to the cold start problem is feasible and beneficial, but there is much more work to be done before it can be considered for production use. Some potential areas of **future work** outlined in this report include:

- Restoring lingering areas of process state: signals, timers, mounts, file descriptors.
- The development of *incremental checkpoints* to be taken between each library load. This would allow functions with the same imports to avoid loading those libraries on cold start. A new scheduling method that optimises function placement by their imports would be required.
- Sharing checkpoint state between containers, or forcing processes to have the same memory layout, given that 98% of Java checkpoints are the same.
- Expanded reliability testing for the current runtimes Python, NodeJS & Java as well as introducing further languages.

Bibliography

[1] A. Vaughn, "How viral cat videos are warming the planet", The Guardian, 2019 [Online]. Available: https://www.theguardian.com/environment/2015/sep/25/server-data-centre-emissions-air-travel-web-google-facebook-greenhouse-gas. [Accessed: 14- Jun- 2019]

[2] Jones, N, "How to stop data centres from gobbling up the world's electricity", Nature, 2018. [Online]. Available: https://www.nature.com/articles/d41586-018-06610-y [Accessed: 26-Jul-2019]

[3] Wang, L., Li, M., Zhang, Y., Ristenpart, T., & Swift, M. (2018). Peeking Behind the Curtains of Serverless Platforms. In 2018 USENIX Annual Technical Conference (pp. 133–146). Boston, MA: USENIX Association. Retrieved from https://www.usenix.org/conference/atc18/presentation/wang-liang

[4] Dean, J., & Barroso, L. (2013). The tail at scale. Communications of the ACM (Vol. 56, pp. 74–80). https://doi.org/10.1145/2408776.2408794

[5] Checkpoint/Restore In Userspace (CRIU), 2019. [Online]. Available: https://criu.org/Main_Page [Accessed: 10-Jun-2019]

[6] van Eyk, E., Toader, L., Talluri, S., Versluis, L., Uță, A., & Iosup, A. (2018). Serverless is More: From PaaS to Present Cloud Computing. IEEE Internet Computing, 22(5), 8–17. https://doi.org/10.1109/MIC.2018.053681358

[7] Where PaaS, Containers and Serverless Stand in a Multi-Platform World. [Online]. Available: https://www.cloudfoundry.org/multi-platform-trend-report-2018/ [Accessed: 7-Jan-2019]

[8] Containers at Google. [Online]. Available: https://cloud.google.com/containers/ [Accessed: 7-Jan-2019]

[9] 8 Surprising Facts About Real Docker Adoption. [Online]. Available: https://www.datadoghq.com/docker-adoption/ [Accessed: 7-Jan-2019]

[10] Newman, S. (2015). Building microservices.

[11] Reiss, C., Tumanov, A., Ganger, G., Katz, R., & Kozuch, M. (n.d.). Heterogeneity and dynamicity of clouds at scale (pp. 1–13). ACM. https://doi.org/10.1145/2391229.2391236

[12] van Eyk, E., Iosup, A., Seif, S., & Thömmes, M. (n.d.). The SPEC cloud group's research vision on FaaS and serverless architectures (pp. 1–4). ACM. https://doi.org/10.1145/3154847.3154848

[13] Adzic, G., & Chatley, R. (n.d.). Serverless computing: economic and architectural impact (pp. 884–889). ACM. https://doi.org/10.1145/3106237.3117767

[14] Firecracker Microvm. [Online]. Available: https://firecracker-microvm.github.io/ [Accessed: 10-Jun-2019]

[15] Lloyd, W., Ramesh, S., Chinthalapati, S., Ly, L., & Pallickara, S. (2018). Serverless Computing: An Investigation of Factors Influencing Microservice Performance. In 2018 IEEE International Conference on Cloud Engineering (IC2E) (pp. 159–169). https://doi.org/10.1109/IC2E.2018.00039

[16] Liston, B, "Resize Images on the Fly with Amazon S3, AWS Lambda, and Amazon API Gateway". [Online]. Available: https://aws.amazon.com/blogs/compute/resize-images-on-the-fly-with-amazon-s3-aws-lambda-and-amazon-api-gateway/ [Accessed: 14-Jun-2019]

[17] Jonas, E., Pu, Q., Venkataraman, S., Stoica, I., & Recht, B. (n.d.). Occupy the cloud (pp. 445–451). ACM. https://doi.org/10.1145/3127479.3128601

[18] Oakes, E., Yang, L., Zhou, D., Houck, K., Harter, T., Arpaci-Dusseau, A., & Arpaci-Dusseau, R. (2018). SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In 2018 USENIX Annual Technical Conference (pp. 57–70). Boston, MA: USENIX Association. Retrieved from https://www.usenix.org/conference/atc18/presentation/oakes

[19] Boucher, S., Kalia, A., Andersen, D. G., & Kaminsky, M. (2018). Putting the Micro Back in Microservice. In 2018 USENIX Annual Technical Conference (USENIX ATC 18) (pp. 645–650). Boston, MA: USENIX Association. Retrieved from https://www.usenix.org/conference/atc18/presentation/boucher

[20] Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., … Bastien, J. F. (2017). Bringing the web up to speed with WebAssembly. ACM SIGPLAN Notices, 52(6), 185–200. https://doi.org/10.1145/3140587.3062363

[21] AWS Lambda in Production: State of Serverless Report 2017. (n.d.) (Vol. 2019). [Online]. Available: https://blog.newrelic.com/product-news/aws-lambda-state-of-serverless/

[22] Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., … Crowcroft, J. (n.d.). Unikernels: Library Operating Systems for the Cloud (pp. 461–472). ACM. https://doi.org/10.1145/2451116.2451167

[23] Kantee, A., & Cormack, J. (2014). Rump Kernels. https://doi.org/10.5446/32618

[24] Manco, F., Lupu, C., Schmidt, F., Mendes, J., Kuenzer, S., Sati, S., … Huici, F. (n.d.). My VM is Lighter (and Safer) than your Container (pp. 218–233). ACM. https://doi.org/10.1145/3132747.3132763

[25] Williams, D., Koller, R., Lucina, M., & Prakash, N. (2018). Unikernels As Processes. In Proceedings of the ACM Symposium on Cloud Computing (pp. 199–211). Carlsbad, CA, USA: ACM. https://doi.org/10.1145/3267809.3267845

[26] Leitner, P., Wittern, E., Spillner, J., & Hummer, W. (2019). A mixed-method empirical study of Function-as-a-Service software development in industrial practice. The Journal of Systems & Software, 149, 340–359. https://doi.org/10.1016/j.jss.2018.12.013

[27] Desai, A. (n.d.). The Firecracker virtual machine monitor - LWN.net. [Online]. Available: https://lwn.net/Articles/775736/ [Accessed: 14-Jun-2019]

[28] WebAssembly's post-MVP future: A cartoon skill tree. [Online]. Available: https://hacks.mozilla.org/2018/10/webassemblys-post-mvp-future/ [Accessed: 9-Jan-2019]

[29] Koller, R., & Williams, D. (n.d.). Will Serverless End the Dominance of Linux in the Cloud? (pp. 169–173). ACM. https://doi.org/10.1145/3102980.3103008

[30] Israeli, A., & Feitelson, D. G. (2010). The Linux kernel as a case study in software evolution. The Journal of Systems & Software, 83(3), 485–501. https://doi.org/10.1016/j.jss.2009.09.042

[31] Zinke, J. (2009). System call tracing overhead. [Online]. Available: http://www.linux-kongress.org/2009/slides/system_call_tracing_overhead_joerg_zinke.pdf

[32] Namespaces - Man7. [Online]. Available: http://man7.org/linux/man-pages/man7/namespaces.7.html [Accessed: 11-Jan-2019]

[33] Duell, J. (2005). The design and implementation of Berkeley Lab's linux checkpoint/restart. [Online]. Available: https://www.openaire.eu/search/publication?articleId=od______325::9106a463d7a7515aabe31bb91b67f77d

[34] Nadgowda, S., Suneja, S., Bila, N., & Isci, C. (n.d.). Voyager: Complete Container State Migration (pp. 2137–2142). IEEE. https://doi.org/10.1109/ICDCS.2017.91

[35] Gorcunov, C. Parasite Code. [Online]. Available: https://criu.org/Parasite_code [Accessed: 30-May-2019]

[36] OpenFaas Watchdog. [Online]. Available: https://github.com/openfaas/faas/tree/master/watchdog [Accessed: 30-May-2019]

[37] Java ClassLoader - Oracle Documentation. [Online]. Available: https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/lang/ClassLoader.html [Accessed: 30-May-2019]

[38] Plugin Package - The Go Programming Language. [Online]. Available: https://golang.org/pkg/plugin/ [Accessed: 30-May-2019]

[39] Refunction Github. [Online]. Available: https://github.com/ostenbom/refunction [Accessed: 14-Jun-2019]

[40] Containerd. [Online]. Available: https://containerd.io [Accessed: 9-May-2019]

[41] runc. [Online]. Available: https://github.com/opencontainers/runc [Accessed: 9-May-2019]

[42] Open Container Initiative. [Online]. Available: https://www.opencontainers.org/ [Accessed: 29-May-2019]

[43] Containerd Unix OCI Spec Default. [Online]. Available: https://github.com/containerd/containerd/blob/0e7a3c9e513da1f1dda163d5872a974a4db07d02/oci/spec.go#L132 [Accessed: 29-May-2019]

[44] hint: Runtime Haskell interpreter. [Online]. Available: http://hackage.haskell.org/package/hint [Accessed: 30-May-2019]

[45] RunC CVE-2019-5736. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2019-5736 [Accessed: 12-Jun-2019]

[46] Emelyanov, P. (n.d.). mm: Ability to monitor task memory changes (v3). [Online]. Available: https://lwn.net/Articles/546966/ [Accessed: 29-May-2019]

[47] AWS Lambda Limits. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/limits.html [Accessed: 30-May-2019]

[48] Corbet, J. (2012). TCP connection repair. [Online]. Available: https://lwn.net/Articles/495304/ [Accessed: 14-Jun-2019]

[49] SegmentIO/Kafka-Go - GitHub. [Online]. Available: https://github.com/segmentio/kafka-go [Accessed: 30-May-2019]

[50] Hykes, S. (2015). Introducing runC: a lightweight universal container runtime. [Online]. Available: https://blog.docker.com/2015/06/runc/ [Accessed: 29-May-2019]

[51] Kubernetes - Container Orchestration. [Online]. Available: https://kubernetes.io/ [Accessed: 12-Jun-2019]

[52] Genuinetools/Netns - GitHub. [Online]. Available:
https://github.com/genuinetools/netns [Accessed: 12-Jun-2019]