Opportunities for Optimizing the Container Runtime

Adam Hall
College of Computing
Georgia Institute of Technology
Atlanta, Georgia
ach@gatech.edu

Umakishore Ramachandran
College of Computing
Georgia Institute of Technology
Atlanta, Georgia
rama@gatech.edu

***Abstract*—Container-based virtualization provides lightweight mechanisms for process isolation and resource control that are essential for maintaining a high degree of multi-tenancy in Function-as-a-Service (FaaS) platforms, where compute functions are instantiated on-demand and exist only as long as their execution is active. This model is especially advantageous for Edge computing environments, where hardware resources are limited due to physical space constraints. Despite their many advantages, state-of-the-art container runtimes still suffer from startup delays of several hundred milliseconds. This delay adversely impacts user experience for existing human-in-the-loop applications and quickly erodes the low latency response times required by emerging machine-in-the-loop IoT and Edge computing applications utilizing FaaS. In turn, it causes developers of these applications to employ unsanctioned workarounds that artificially extend the lifetime of their functions, resulting in wasted platform resources. In this paper, we provide an exploration of the cause of this startup delay and insight on how container-based virtualization might be made more efficient for FaaS scenarios at the Edge. Our results show that a small number of container startup operations account for the majority of cold start time, that several of these operations have room for improvement, and that startup time is largely bound by the underlying operating system mechanisms that are the building blocks for containers. We draw on our detailed analysis to provide guidance toward developing a container runtime for Edge computing environments and demonstrate how making a few key improvements to the container creation process can lead to a 20% reduction in cold start time.**

***Index Terms*—Containers, Edge Computing, Runtime, Serverless, FaaS**

I. Introduction

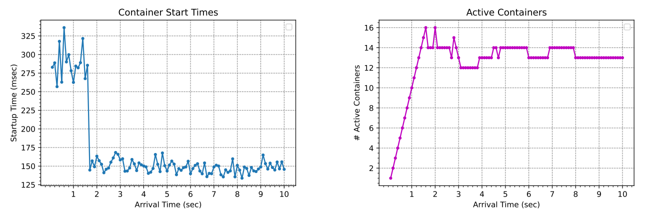
In the near future, relocating or extending computational resources from the Cloud toward the Edge of the network will be necessary to support next-generation applications (such as Augmented Reality, Video Analytics, and Self-Driving Cars) requiring high data throughput and low latency response times. Serving these applications from the Cloud alone is impractical due to its centralized nature, which introduces high latency (because of the physical distance between clients and data centers) and constrained bandwidth (because of a limited number of backhaul links relative to a large number of client devices). However, physical space constraints in the last mile mean Edge computing hardware is necessarily limited relative to its Cloud counterparts. Despite this limitation, the Edge must support a large number of different applications serving various clients. To this end, it is important to ensure application runtimes utilize hardware resources as efficiently as possible to achieve a high degree of multi-tenancy for supporting many different application services simultaneously.

One method for efficiently hosting different applications in the same environment is *container-based virtualization*. With this method, each process related to a service runs within a separate namespace of the host operating system (OS). This namespace gives the appearance of a full OS to the processes, complete with a unique view of and limit on which resources can be consumed. Unlike fully virtualized machines, which each require an entirely separate copy of the OS and all its components, containers exist within the same host OS and share the same kernel, so their operation is considerably more lightweight. This distinction is especially important in an Edge computing environment, where the processes of different application services must be separated for security and performance reasons but must be done so with minimal resource cost. Several application orchestration and Function-as-a-Service (FaaS) platforms targeting Edge computing environments [1] [2] [3] rely on containers for this reason.

In FaaS platforms, applications are represented as independent component parts known as functions. These single-purpose functions exist on-demand, in that they are instantiated when called, execute for a brief period of time, and shut down until needed again. Each function is isolated within its own short-lived container. In recent years, FaaS has proved a boon to Cloud providers and their customers, enabling significant cost and resource savings. Similarly, the scale-to-zero[1] nature of FaaS platforms further extends the utility of containers for Edge computing environments, allowing even greater degrees of multi-tenancy through more efficient utilization of limited hardware resources.

Despite the advantages containers provide, they still suffer from an issue known as the *cold-start problem*, requiring several hundred milliseconds or more to complete their start up operations. This means that any applications they host cannot begin serving clients until container startup operations complete. Applications that rely on Edge computing expect rapid response times, so any advantage achieved by placing computational resources within close proximity to clients can be quickly eroded if the services supporting these applications

[1]"Scale-to-zero" describes the ability of a FaaS platform to reduce a function to zero replicas when idle and more replicas as needed.



(a) (b)

Fig. 1: Container startup times for a function with 100 ms average inter-arrival time and 1 second average execution time. The first 16 function invocations result in a cold start, after which time running containers can be recycled to handle new requests (left graph). After approximately 3 seconds, the workload requires no more than 14 warm containers, leaving 2 containers idle (right graph).

take too long to start. This problem is further exacerbated by the limited hardware available to Edge nodes, which may necessitate recycling containers during periods of application inactivity to free up resources.

Figure 1 demonstrates the impact of cold start on a FaaS platform handling a workload of 10 requests per second (RPS) over a 10 second period. In this scenario, a new container needs to be spawned every 100 ms to service incoming requests. With an average execution time of 1 second and cold start of 250 ms, at least 1.25 seconds elapse before the first container can be recycled to service additional requests. During the ramp up period the platform must spawn 16 containers to meet demand, but afterwards a maximum of 14 containers are needed to service incoming requests. As a result, 2 containers remain idle for the remainder of the test. Since many FaaS platforms keep containers warm in anticipation of serving future requests faster, these extra containers consume additional memory even when unused.

Application platforms targeting the Edge focus on optimizing orchestration and communications costs, but still rely on containers as-is under the hood. The common method used to reduce container cold starts involves keeping containers active for some period following invocation (e.g., OpenWhisk [4] uses a 10 minute window before turning off an idle container). While this method greatly reduces cold start, it comes at a high resource cost. Other methods for reducing or eliminating cold start exist, such as pre-warming containers [5] [6] and checkpoint/restore [7], but these methods also trade already limited resources for performance gains and are thus unsuitable for resource constrained Edge computing platforms. To achieve low latency and high efficiency, the Edge needs its own optimized container runtime that addresses the root causes of the cold start problem. As we later demonstrate, several opportunities for container runtime optimization exist and can be leveraged to achieve this goal.

In this paper, we deconstruct a container's startup routine by instrumenting and analyzing a container runtime. We use this analysis to determine the operations involved in creating a container environment, costs associated with each operation, and operations that can be streamlined to reduce container startup time. Our work makes the following contributions:

1) We describe our methodology for analyzing a container runtime, including techniques to add instrumentation for benchmarking an application that spans multiple processes in different namespaces.
2) We provide fine-grained measurements of the container startup procedure, including a list of operations that contribute significantly to cold start time.
3) We provide a detailed analysis of the functions that impact the startup time of containers, pinpointing some key issues that are candidates for improvement as a way to inform the creation of a more efficient container runtime for resource constrained Edge environments.
4) We demonstrate that opportunity exists for optimizing the container runtime by implementing proof-of-concept improvements that achieve a 20% reduction in cold start time without requiring significant changes to the runtime's functionality.

The remainder of this paper is laid out as follows. In Section II we provide brief background information on how container-based virtualization technology works. In Section III, we discuss challenges to deconstructing a container's startup time and our solution approach to meeting these challenges. In Section IV, we provide a description of the container runtime aspects that were measured and our results from these measurements. In Section V, we provide a detailed analysis of our measurement results, highlighting the operations that most impact container startup. In Section VI, we discuss some proof-of-concept container runtime improvements that result in a reduction in cold start time. In Section VII, we demonstrate the benefit of our optimized runtime on container cold start when faced with realistic workloads from a production FaaS platform. And in Sections VIII and IX we discuss works related to our own, the impact of our findings, and provide overall conclusions drawn from this research.

II. Background

In this section we discuss low-level details of how containers work to lay the framework for later explorations into container startup operations. For the reader already familiar with container technology, this section can be skipped.

Container-based virtualization provides a way to segment one or more processes into different logical views of their underlying host OS and give each process a unique allocation of system resources. Under Linux, This segmentation is made possible by two components: *namespaces* [8] and *cgroups* [9] (also known as *control groups*)[2]. Namespaces provide a logical division of OS resources, such as the process tree or filesystem. For example, one process may have a unique filesystem root directory layout or network routing table that other processes

[2]Although different methods exist for container-based virtualization, for the purposes of this paper we limit our discussion to the implementation of containers within Linux.

do not share. Linux provides eight different namespaces: Cgroup, IPC, Network, Mount, PID, Time, User, and UTS. The Cgroups kernel feature provides a way to limit the amount of system resources that processes can consume. For example, processes in a container may be limited to a subset of their system's virtual memory space while processes outside the container may have access to the entire virtual memory space. Multiple namespaces and cgroups may exist within a host OS, and each namespace or cgroup may have one or more processes tied to it.

Container creation is grounded in the Unix fork/exec paradigm and typically begins when an active process makes a *clone()* [10] system call. This system call allows the calling process (the parent) to spawn a new process (the child) with arguments specifying how namespaces will be assigned to the newly cloned process. For example, if *clone()* is called with the *CLONE_NEWNS* flag set, the child process will be started in a new `mount` namespace that is initialized with a copy of the parent's `mount` namespace. By default, the child process inherits most of the properties of its parent. After the child process has spawned, it can be further configured to customize properties like hostname or filesystem layout, security capabilities that limit privileged system operations, and cgroups that limit the amount of resources it can consume. A diagram illustrating the relationship of a namespace to its parent/host operating system can be seen in Figure 2.

After the child process has been configured to create an isolated container environment, it uses an *exec()* system call [11] to effectively replace itself with the container's init program. The init program in turn spawns other applications, providing a facsimile of a real operating system inside the container. Since every new process inherits its parent's namespaces and capabilities, every new application spawned by init also runs in the same container. Typically a container runtime serves as the initial parent and child processes creating the container, and the init program is specified by the container creator.

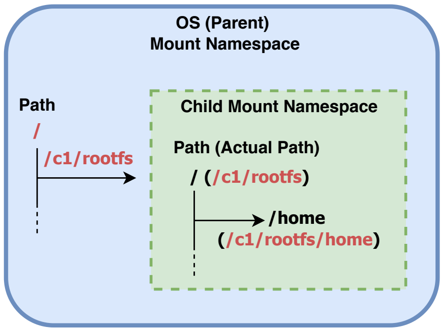


Fig. 2: The relationship between a child's mount namespace and its parent/host. The parent directory '/c1/rootfs' is mounted as the root directory '/' of the child. Any directory created in '/' on the child is actually stored in '/c1/rootfs' on the host. For example, although the child sees a directory at '/home', this directory is actually stored at '/c1/rootfs/home' on the parent.

III. Dissecting Container Cold Start

Deconstructing the startup operations of a container is more challenging than it initially seems, due to both the nature of the container creation model and a lack of available tooling features to provide enough insight into the inner workings of its operations. In this section we describe these challenges and our solution approach to dissecting the container startup procedure.

A. Inherent Challenges

The nature of the container creation model poses two main challenges to understanding the inner workings of container creation:

1) **Multiple processes running simultaneously**: During its execution, a container runtime clones itself to a new child process and these two processes execute in parallel to complete container setup. Each of these processes executes in a different namespace and with different security capabilities. Failing to properly account for simultaneous operations can lead to false positive timing results (in the form of a double counting problem) if two functions executing simultaneously are both counted as contributing to the cold start time.
2) **Limitations of existing tools**: A standard approach to measurement is to invoke common performance analysis and profiling tools like *perf* [12] and *strace* [13]. While some tools provide part of the functionality needed, in our experience none are sufficient for container analysis for two reasons. First, attempting to use these tools to analyze a container runtime results in partial or missing output. Popular analysis tools are designed to work with a single view of the system, but during container creation the runtime and its child provide multiple views (i.e., namespaces) of the system. For example, after the container runtime clones itself its child process will execute in a new PID namespace with reduced security capabilities. Both of these changes restrict the information off-the-shelf analysis tools can gather due to either limited information availability or lack of permissions. Second, the information recorded by these tools does not provide deep enough insight into container startup operations. Performing a thorough analysis requires information such as how long each individual function takes to execute, how and why it was invoked, and where in the code base that function is defined. For these reasons we found that off-the-shelf profiling tools were inadequate for exploring the container startup procedure.

To further clarify these challenges, we provide a high-level diagram of the container startup procedure in Figure 3. This procedure begins with the execution of a container runtime binary (1) which first performs preliminary operations such as loading a container's configuration and setting up security policies. After these initial operations, the runtime process clones itself (2) into a new child process and waits for the child process to load and return its process identifier (3). The child process performs its own initialization procedures while communicating with the parent process to coordinate operations such as the setup of cgroups (4) and namespaces (5). During this period the parent and child may be occasionally blocked while waiting for each other to complete different operations (as indicated by the yellow circles along each timeline). The operations performed by the child process occur in a separate namespace from the parent process. When startup operations have completed (6), the child process begins execution of the container's init program (7) and the container is ready to run.

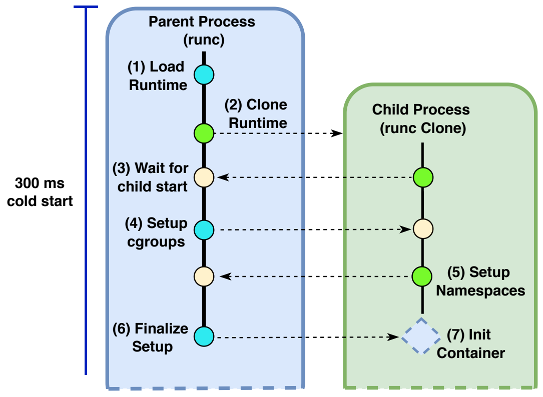


Fig. 3: A high-level illustration of the container startup procedure. Container startup consists of two processes (parent and child) which work simultaneously in different namespaces to create an environment to isolate the container init process.

As previously mentioned, a methodology for timing operations must account for the interplay between runtime parent and child processes to provide a thorough analysis of the operations that contribute to cold start. For example, if *functionA()* in the parent calls *functionB()* in the child, its recorded execution time will include the time for its own operations plus the execution time for *functionB()*. Not accounting for this situation falsely attributes execution time to different startup operations and inflates the overall measured startup time. In the next section, we describe how we address these challenges to achieve accurate logging and timing of container startup operations.

B. Methodology

Containers are created through a combination of system calls and kernel functions, and these features are commonly accessed via a standard container library like *libcontainer* [14]. The *runc* container runtime [15] implements *libcontainer* and is used by popular platforms such as Docker and Kubernetes for container creation. For the purposes of this paper, we added instrumentation to *runc* that records each function called during the container creation process. Although our instrumentation records container creation from the perspective of *runc*, we note that our observations generalize across different container runtimes and our findings apply equally for any container creation scenario.

The *runc* code base is written in the Go programming language and consists of just over 1,000 functions. To instrument these functions, we created a new package within *runc* with a function, *TimeC()*, that records function start and stop times along with each function's caller and location within the *runc* source tree. We leverage two features of Go to gather accurate function timing information. The first feature is Go's *defer* statement [16], that ensures a block of code within a function will not be executed until its surrounding function returns. All arguments to a function called by *defer* are evaluated immediately. This means if we place a *defer* call to *TimeC()* at the beginning of each instrumented function with an argument that records the current time, the argument will be evaluated as soon as the surrounding function starts, effectively providing us with the start time of that function. The second feature is Go's built-in *runtime* package [17] that provides operations allowing a program to interact with the Go runtime system. We use these operations to extract information about the instrumented function.

When an instrumented function is executed, the *defer* statement is evaluated immediately and records the current time as the function *start* time. When the instrumented function returns, *defer* calls *TimeC()* with information about the instrumented function and its parent (calling) function, such as the names of the functions and the filename, line number, and package in which they appear in the *runc* source tree. The *TimeC()* function immediately records the current time as the *end* time of the surrounding function and calculates the elapsed time to determine the function execution time in nanoseconds. Statistics for the function are then converted to a string of comma-separated values and stored in an in-memory log that is written to disk after the container has started.

If an instrumented function calls other functions within *runc*, its recorded execution time will also encompass the execution time of those called functions. This situation leads to false positive timing information. As a solution, during the log parsing phase the execution times of a parent's children are subtracted from the parent's recorded execution time, yielding a normalized result.

IV. Measurement Results

In this section, we discuss the results from measuring container execution times with our instrumented container runtime.

A. Container Startup Time

We chose the Ubuntu 18.04 container image from Docker Hub for use in our measurements. These measurements were taken on a machine running a 4-core Intel i5-1035G1 with 8 GB of RAM. To eliminate additional overhead, our container was configured to execute a single script that prints a timestamp upon initialization and then exits. Figure 4 demonstrates the distribution of cold start times for 100 container creations spaced 100 ms apart using an unmodified version of the *runc* container runtime. We recorded an average container startup time of 259 ms, but note that this time can be highly variable, increasing to as much as 329 ms in our experiment.

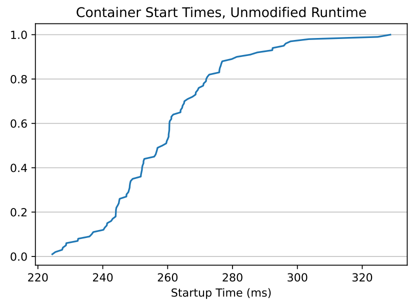


Fig. 4: Distribution of 100 container startup times.

Our instrumented version of *runc* was used to measure the execution times taken by the functions called during each container's startup to shutdown period. We recorded measurements of execution times for an average of 3900 function calls per container creation. These execution times were recorded with nanosecond precision. To gauge the accuracy of our instrumentation technique, we compared the execution times of both an unmodified and instrumented version of *runc* over 100 container startup events. Since the difference between results reported by both versions was 1% on average, we consider this validation that our instrumentation technique provides an accurate representation of the time taken to execute functions during container startup.

B. Container Startup Flow

Our instrumentation records the time that a function is called and its parent (calling) function, allowing us to recreate the operational flow that makes up a container startup process. A basic version of this flow is illustrated in Figure 5. We discovered that the basic flow of startup operations is almost identical among different container types, in that the same functions or types of functions are called in almost the same order across container startups. Typically, a *runc* parent process begins by loading and validating a container's configuration and then checking for and initializing IntelRDT (Resource Director Technology) [18], if available. The parent process then creates a copy of itself - a *runc* child process - and specifies which new namespaces should be created for the copy. Security profiles for AppArmor [19] and seccomp [20] are then loaded, Cgroups are applied to the child to limit its resource utilization, and the child's namespaces are configured to isolate it from the rest of the system. At this point the child process is running in a fully contained environment. To complete container setup, the child process executes the entry point application specified by the container config, effectively replacing itself with the container init process. After container execution completes, *runc* reclaims control from the init process to destroy and clean up container resources.

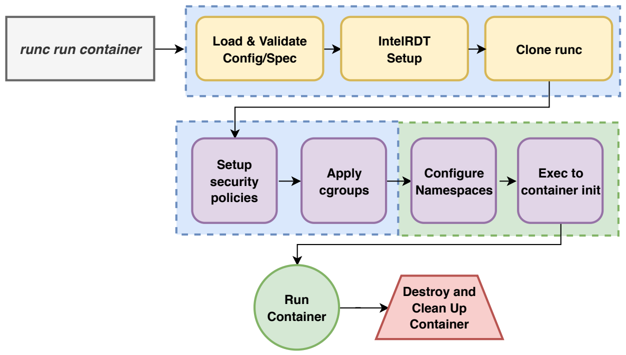


Fig. 5: The basic flow of operations that occur during a container's startup process.

C. Component Parts of Container Startup

To better describe container startup operations, we group functions that contribute to cold start and have a similar purpose into one of several categories. Our category placements are based on both the layout of the *runc* source code and on the trace data generated by our instrumented version of that code. We use the following categories to describe container startup operations:

- **Security**. Functions in this category are concerned with the setup and maintenance of security operations that restrict what a container may do. Examples include functions from the *AppArmor* and *seccomp* implementations used by *libcontainer*.
- **Mounts**. Functions in this category perform operations related to the mount namespace, such as establishing a unique filesystem layout for the container.
- **Platform Specific**. Functions in this category perform operations associated with platform-specific features, such as IntelRDT.
- **Cgroups**. Functions in this category perform operations related to the creation and application of Linux control groups. These operations are used to restrict the amount of resources of the system that a container can use.
- **Bootstrap**. Functions in this category are responsible for setting up the initial container parent process before it is cloned.
- **Sync**. Functions in this category handle operations related to synchronizing between parent and child processes during container creation, such as serializing/deserializing data and maintaining communications channels via pipes.
- **Configuration**. Functions in this category load, interpret, and apply configuration directives, such as those found in the OCI runtime specification.

D. Quantifying Component Parts of Container Startup

To provide a more concrete view of how each category contributes to a container's start time, Table I demonstrates execution times and function counts of these categories as averaged across the 100 container startup operations described previously. As we discuss in the next section, the cause of the cold start problem in containers appears to be related to just a few types of operations in each category. In Figure 6 we demonstrate the magnitude of these dominant operations on the cold start time of a container.

Category	Exec Time	Functions	Calls
Security	42 ms	11	975
Mounts	99 ms	19	164
Platform	20 ms	9	20
Cgroups	38 ms	84	1,577
Bootstrap	43 ms	99	309
Sync	7 ms	22	35
Configuration	10 ms	43	230

TABLE I: Execution times, number of unique functions, and number of function calls for categories contributing to cold start.

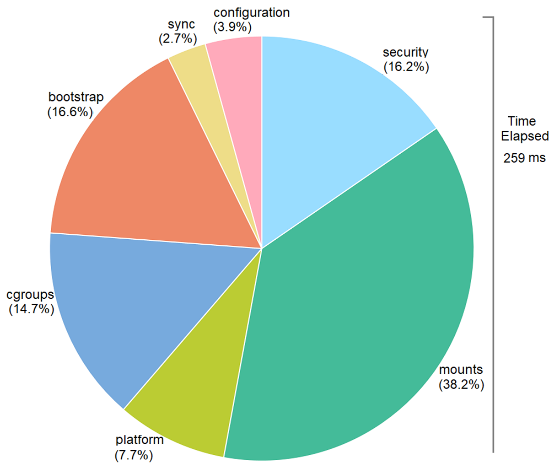


Fig. 6: Average execution time percentages for each category as it contributes to cold start.

V. Analysis of Measurement Results

According to the data gathered for our analysis, five categories have a small number of significantly expensive operations that contribute to the slow startup of containers: *Security*, *Mounts*, *Platform Specific*, *Bootstrap*, and *Cgroups*. In the following subsections we perform a breakdown of each operation type and explore components that may be the cause of slowdowns, followed by an exploration of the miscellaneous supporting functions that are used by the runtime, and suggest how these components could be optimized for the FaaS model. We conclude the section with a discussion of the container shutdown procedure.

A. Security

The *Security* category includes functions that implement security restrictions for container processes via the `seccomp` Linux kernel facility. The *libseccomp* library [21] is used to interface with these mechanisms and restrict the OS system calls (syscalls) that can be made by processes in the container. Allowing a container process to make syscalls is essential for certain operations, as some functionality is only available outside of userspace. At the same time, denying a process the ability to make some syscalls is necessary to provide a secure execution environment that is isolated from the host OS and other container processes. This tradeoff resulted in the need for security facilities such as `seccomp` to provide strong isolation at the cost of some performance penalty. The category contains 11 unique functions and 975 function calls. Of these 11 functions, 2 of them account for almost 16% of the cold start time of a container: *seccomp.InitSeccomp()* and *seccomp.matchCall()*.

The *InitSeccomp()* function creates and configures a new seccomp filter context [22], into which the rules for each syscall defined in a container's configuration will be added. It then iterates through those rules and adds them one at a time by calling *matchCall()*. After all rules have been added, the filter context is loaded into the kernel via *libseccomp*'s *Load()* function [23]. This initial setup takes 16 ms to complete.

Each syscall is represented by a common name and a numeric value that is specific to the kernel/architecture on which the container is being run, and has an action associated with it that dictates how `seccomp` will react when the syscall is used by a process. These two properties are resolved by *matchCall()* before they can be used by the filter context. First, the syscall name listed in the container config is converted to the syscall number used by the kernel. Next, the filter action is resolved from its name to a libseccomp object. Finally, both syscall number and action are added to the filter.

Our measurements show that *matchCall()* only takes 80 μs to execute on average, but the number of times it is called is directly related to the number of syscall policies in a container's configuration. The configuration for a typical container has 316 entries for syscalls, representing 67% of the 470 syscalls available in the Linux 4.19 kernel [24]. For a standard container, adding syscall rules to the filter takes 25 ms.

In the context of a FaaS platform, we can envision two approaches for reducing the overhead of security operations on the cold start time:

1) By reducing the number of syscalls filtered to a minimum, less calls to *matchCall()* are needed. This in turn speeds up the container creation process as well as reduces the attack surface by limiting unnecessary privileged functionality. One drawback to this approach is that platform operators or FaaS application developers may not have the knowledge or time required to determine what syscalls a container and its processes actually need. On the other hand, the behavior of FaaS applications is similar (i.e., short-lived, single-purpose functions) and so there may be opportunity for platforms to dictate a limited number of syscalls that make sense for the functions they allow their users to run.

2) Optimization of the `libseccomp` library and/or the `seccomp` kernel facility merits further investigation. As noted previously, rules are added to a filter one at a time via multiple function calls, so the ability to batch this operation into a single call could provide some cost savings. Similarly, although each invocation of *matchCall()* is relatively fast, in aggregate the number of invocations required adds significant slowdown to the startup process and if streamlined should result in significant time cost savings.

B. Mounts

The *Mounts* category consists of 19 functions called 164 times, and its operations are the largest contributor to cold start time. Two functions, *libcontainer.maskPath()* and *libcontainer.pivotRoot()*, account for 90 ms of time spent starting a container.

The *maskPath()* function configures paths on the host filesystem that cannot be virtualized within the container. For example, certain parts of the `/proc` filesystem [25] contain access to things like the system memory or low-level device drivers and do not support isolation via namespaces. Instead, these filesystem paths must be isolated by rebinding via either a bind mount to `/dev/null` for special files or a read-only `tmpfs` mount for special directories. Under the hood, this operation relies on the *mount()* syscall. In our experiments, *maskPath()* was called 9 times. Its execution contributed 61 ms to container startup time.

The *pivotRoot()* function acts as a wrapper for the *pivot_root()* syscall. It is responsible for switching the root filesystem of a container from that of the host (typically located at `/`, and referred to as `oldroot`) to the container's root path - the location on the host that holds all the containers files (referred to as `newroot`), similar to what is illustrated in Figure 2. Its operation also relies heavily on system calls, calling *open()* on both these paths, performing a change of directory to `newroot` via *fchdir()*, and calling *pivot_root()* on this location. After `newroot` is set as the container's root filesystem location, *pivot_root()* unmounts `oldroot` by changing to its location via another *fchdir()* call, re-mounting it as a recursive bind mount via a *mount()* call, and finally detaching it via an *unmount()* call. Finally, it completes its operation by calling *chdir("/")*, placing the container process at the top of the new root filesystem hierarchy. Changing to the new container root using this method added 29 ms to startup time in our experiments.

Both *maskPath()* and *pivotRoot()* rely almost exclusively on system calls to complete their operations, and do not incorporate much custom logic otherwise. They utilize Go's built-in `unix` package [26] to access these low-level OS primitives. This property suggests that opportunity for optimization may exist in either the `unix` package or within the Linux kernel itself. Although such exploration is outside the scope of this work, given the impact of these operations on the container further investigation is warranted.

C. Platform Specific

Functions in the *Platform Specific* category are responsible for performing operations related to Intel's Resource Director Technology [18] that allows for fine-grained resource allocation and monitoring for hardware such as cache and memory bandwidth. Of these nine functions, all but one of them require under 0.5 ms to complete, contributing a total of approximately 1 ms to the category's execution time. The remainder of this time is dominated by the *parseCpuInfoFile()* function that takes approximately 9.5 ms to complete each execution.

The *parseCpuInfoFile()* reads and parses the file at `/proc/cpuinfo` to determine if the flags `cat_l3` and `mba` exist, indicating CPU and kernel support for IntelRDT sub-features. If either of these flags are present, the container runtime performs additional actions to apply any IntelRDT configuration directives. This function is called twice: once by the parent process and once by the child process. These two calls contribute 19 ms to the cold start time.

The IntelRDT functions are executed regardless of whether or not the container is running on Intel-based hardware and whether or not configuration directives for them exist in a container's config. The short-lived application functions that execute on a FaaS platform are not likely to benefit from IntelRDT's presence, especially given the overhead it introduces. Making the execution of these functions an optional feature (e.g., via a container config directive) could introduce a cost savings to container startup time, regardless of the underlying platform.

D. Cgroups

Functions in the *Cgroups* category perform operations such as gathering preliminary information for setting a container's resource limits, locating the Cgroup file system path, creating and applying Cgroups, and verifying that a Cgroup's creation and/or application was successful. Although there are many function calls, most execute in under 1 ms and their aggregate execution contributes approximately 15 ms to startup time. The remainder of this setup cost is incurred by the executions of 4 functions, amounting to approximately 23 ms of the Cgroup category's execution time. These functions include:

- **parseCgroupFromReader()**, that gathers the Cgroup configurations that exist for a given process ID (pid) by reading the file located at `/proc/<pid>/cgroup`. It returns these entries to the caller in the form of a key-value map.
- **ParseCgroupFile()**, that opens a Cgroup file at a given path (typically `/proc/<pid>/cgroup`) and then calls *parseCgroupFromReader()* to parse that file.
- **findCgroupMountpointAndRootFromReader()**, that attempts to locate the mount point of a Cgroup virtual file system by scanning through the file `/proc/self/mountinfo`, that contains information about mount points in a process's mount namespace.
- **FindCgroupMountpointAndRoot()**, that is responsible for the first step toward locating the mount point for the Cgroup virtual file system. It begins by performing a file

open on `/proc/self/mountinfo`. It then calls *findCgroupMountpointAndRootFromReader()* to parse this file. Its execution takes approximately 5.1 ms.

The most expensive operations associated with Cgroup creation and management are related to locating and parsing the files serving as interfaces to Cgroup subsystems. Although cgroups are interfaced via a virtual file system [27], accessing this file system still incurs the same overhead as a real filesystem due to the cost of marshaling and unmarshaling data between kernelspace and userspace. As a potential optimization, FaaS platforms might employ different discrete resource tiers, create cgroups representing these tiers once, and reuse them. For example, if the platform supports assigning functions 128 MB, 256 MB, or 512 MB of RAM, three different memory cgroups could be created when the platform starts and associated functions could be assigned to the appropriate tier when invoked.

E. Bootstrap, Sync, & Configuration

The remaining three categories are similar to each other in that they provide operations essential to container startup but do not contain operations that stand out as candidates for deeper analysis. The **Bootstrap** category consists of operations that create the initial container runtime parent process and manage communications with its child process. Two of its functions, *newContainerInit()* and *getChildPid()*, contribute the majority of execution time and are responsible for establishing a shared socket and named pipe that enable communication between parent and child. The **Sync** category operations are responsible for supporting communications between the container runtime parent and child processes during startup. For example, during container setup the parent must place the child in various cgroups to limit resource utilization, but can only do so after the child has communicated its PID back to the parent. Expensive functions in this category are associated with process signal handling and parsing JSON-encoded messages sent between parent and child. The **Configuration** category contains operations related to the loading, parsing, and validation of the container config.

F. Shutdown

Our instrumentation is comprehensive in that it captures timings for container runtime functions during the entire lifecycle of the container (i.e., from the time a container is created to when it is destroyed). We initially suspected that shutdown would contribute a negligible amount of time to the overall results, but found that it takes 52 ms on average. Several operations are required to shut down a container cleanly, such as removing its Cgroups/namespaces. Although none of these operations directly impact the cold start of the container, we note that this finding is still of interest to efforts in making containers more efficient in FaaS or Edge platforms. The need to frequently start and stop containers in these platforms means that a reduction in shutdown latency could decrease the time spent waiting for resources to become available when the platform is overloaded. Research has shown that mount and IPC namespace cleanup in particular is expensive, due to the way the Linux kernel handles synchronized access to shared data [5]. Improvements to this area would entail a deeper exploration of the kernel.

VI. Improvements Based on Measurement Results

Two approaches exist for reducing the execution times of operations contributing to cold start. The first approach involves modifying the underlying kernel mechanisms that are used by the container runtime. If we optimize beyond the runtime (i.e., by modifying the Linux kernel) we can achieve more generic benefits to container creation. However, this effort involves a deep dive into the Linux kernel and has the potential to disrupt its entire ecosystem. An alternative approach is to implement optimizations to the runtime itself. This approach is more tractable in the immediate term and, as we later demonstrate, can yield considerable reductions to cold start times.

From the results of our analysis we can identify three key mechanisms for optimizing runtime operations: (1) exploiting parallelism of operations where possible, (2) pre-loading or caching information to avoid redundant tasks, and (3) eliminating functionality that is unnecessary for our use case (e.g., features that add to cold start but are not used in a FaaS platform). In the following sections, we discuss proof-of-concept improvements made to a container runtime using each of these mechanisms.

A. Exploiting Parallelism

In recording the container creation process, we noted that one of two requirements exist for the execution of startup operations: (1) the operation must complete before other startup operations may begin or (2) the operation must complete before container initialization is finalized. In the latter case, operations that block the creation process when not strictly necessary present an opportunity to reduce cold start time by performing their executions in parallel. For example, in the *Mounts* category the *pivotRoot()* function uses the *unmount()* syscall to unmount the system root from the container's namespace after it is no longer in use. Waiting for this operation to complete call blocks the startup process, but can be performed in a non-blocking way for several reasons:

1) Unmount occurs after pivoting the cloned container runtime process to the `newroot` namespace, causing subsequent startup operations of the process to operate within this context.
2) The call is made with the *MNT_DETACH* flag [28], which immediately disconnects a filesystem and makes it unavailable for new accesses, thereby making it ineligible for use by subsequent operations in the container startup procedure.
3) Since the system root is not directly available to new processes, the unmount operation is performed as a precaution to remove any dangling references which could allow a malicious process in the container to gain privileged access to the true system root. No processes

will be spawned in the container until its startup procedure completes, meaning we only need to ensure that the true system root is unmounted before this occurs.

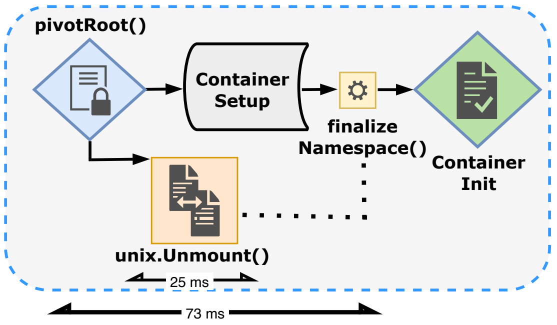


Fig. 7: Optimization of the Mounts pivotRoot() procedure. A new thread is created to execute unix.Unmount() in parallel, overlapping the 73 ms time period between pivotRoot() and finalizeNamespace().

With these points in mind, we modified the source code of *runc*'s *pivotRoot()* function to run the system root unmount in a non-blocking manner, which we illustrate in Figure 7. Our optimization leverages two features of the Go programming language: Goroutines [29] (functions that run concurrently with other functions) and WaitGroups [30] (functions that wait for goroutines to finish). Our modification creates a global WaitGroup within our timing instrumentation code, encapsulates *pivotRoot()*'s original *unix.Unmount()* call within a new goroutine, and adds this goroutine to the global WaitGroup. To ensure the unmount operation completes before the container starts, we cause the WaitGroup to perform a blocking wait within the *finalizeNamespace()* function, which is called by *runC* near the end of the container creation process. On average, a 73 ms period is elapsed between the call to *pivotRoot()* and the call to *finalizeNamespace()*, providing adequate time for the 25 ms unmount operation to complete without blocking.

The same principle also applies to *maskPath()* operations, which rely on *mount()* system calls to hide privileged filesystem paths that cannot be isolated via namespaces. We applied the same optimization technique used with *pivotRoot()* to *maskPath()*, allowing otherwise blocking rebinding calls to complete in parallel outside the critical path of container startup. These parallel calls leverage the same global WaitGroup to ensure that all masking operations have completed before control is given to the container init process.

B. Pre-Loading / Caching

Most directives defined in a container's configuration do not change between FaaS invocations, but still require the same initialization during every container startup. For example, in the *Security* category the *InitSeccomp()* function must resolve every named system call in the config to a numeric kernel-specific representation before it can load them into the seccomp filter. These numeric representations only change if the underlying OS kernel or hardware architecture changes, which is unlikely to occur on any regular basis. This property provides an opportunity to perform the name-to-number translation once, cache the results, and avoid redundant work during container startup.

To implement this improvement, we modified *runc* as follows:

- Pre-parsed the configuration of each container to extract syscall names and translate them to their numeric equivalents
- Added a new function to `init_linux.go`, *addSyscallNumbers()*, that takes as input a list of system call numbers and a seccomp filter and adds rules for each number to the filter using the seccomp library's *AddRule()* function
- Removed the loop inside *InitSeccomp()* that previously used *matchCall()* to resolve each system call and replaced it with a single call to our *addSyscallNumbers()* function

C. Eliminating Unnecessary Functionality

Because *runc* was created for a wide array of use cases, it includes functionality which adds to startup time but may be unnecessary in certain scenarios. For example, our analysis of IntelRDT shows that its most expensive function, *parseCpuInfoFile()*, is called regardless of whether or not RDT is utilized on a system. Short-lived application functions on a FaaS platform are not likely to benefit from IntelRDT and thus it makes sense to disable this feature in favor of a reduced time to start. To demonstrate this improvement, we modified the *runc* source code to disable IntelRDT.

D. Summary of Improvements

Overall, our three proof-of-concept improvements reduce the average cold start of a container by 20%. Figure 8 demonstrates the effect of these optimizations on 100 container startups over a 10 second period. In addition to reducing the average time to start a container, our optimizations also reduce the variability among container startup times. Figure 9 demonstrates the distribution of startup times before optimization (standard deviation of 19.6 ms) and after optimization (standard deviation of 7 ms). Reducing variability is especially important in latency critical applications, such as those envisioned for the future of Edge computing. For example, in a latency critical application that distributes work across multiple Edge nodes the application's response time is dominated by the response time of the slowest node.

VII. Evaluation

To show the impact of our proof-of-concept improvements in a real-world scenario, we evaluated three serverless functions configured with execution times and invocation traces from the Microsoft Azure Public Dataset [31]. We selected three functions from this set with high, medium, and low interarrival times to understand the effects of cold start when serving mixed workloads. Table II describes these functions and their configurations. All functions were invoked from cold

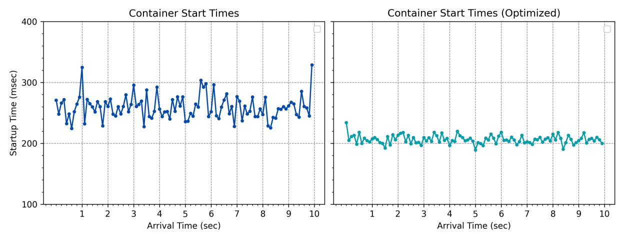


Fig. 8: A before and after comparison of container startup times over a 10 second period using an unmodified version of runc (left) and a version that implements or three proof-of-concept optimizations (right). Our optimizations result in a 20% reduction to container startup time.

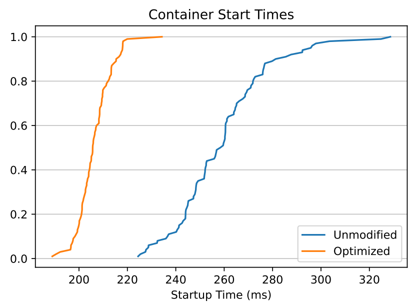


Fig. 9: Distributions of container cold start times for an unmodified version of runc and a version using our proof-of-concept optimizations. In addition to reducing average container startup time, these optimizations also reduce variability in startup times.

start during a 1 minute test period to demonstrate the worst-case scenario (i.e., no recycling of already warm containers). Results from this experiment are shown in Figure 10.

Function	Interarrival Time	Execution Time
5e0db5fd7898	500 ms	629 ms
f9dd89047700	385 ms	306 ms
c8c0dc2ebb78	88 ms	545 ms

TABLE II: Interarrival and execution times for our 3 example functions.

Similar to our standalone experiments, here we see that cold start time for function invocations are reduced by 22% on average with just three optimizations to the container runtime. As the interarrival time decreases (i.e., as the requests per second increase), the amount of interference in the system increases as container setup operations attempt to modify shared data structures in the operating system simultaneously. Optimizing blocking operations allows the container setup to continue doing useful work while this interference is resolved in the background. Figure 11 demonstrates the distributions in cold start times among these functions, showing a reduction in variability when our proof-of-concept optimizations are used.

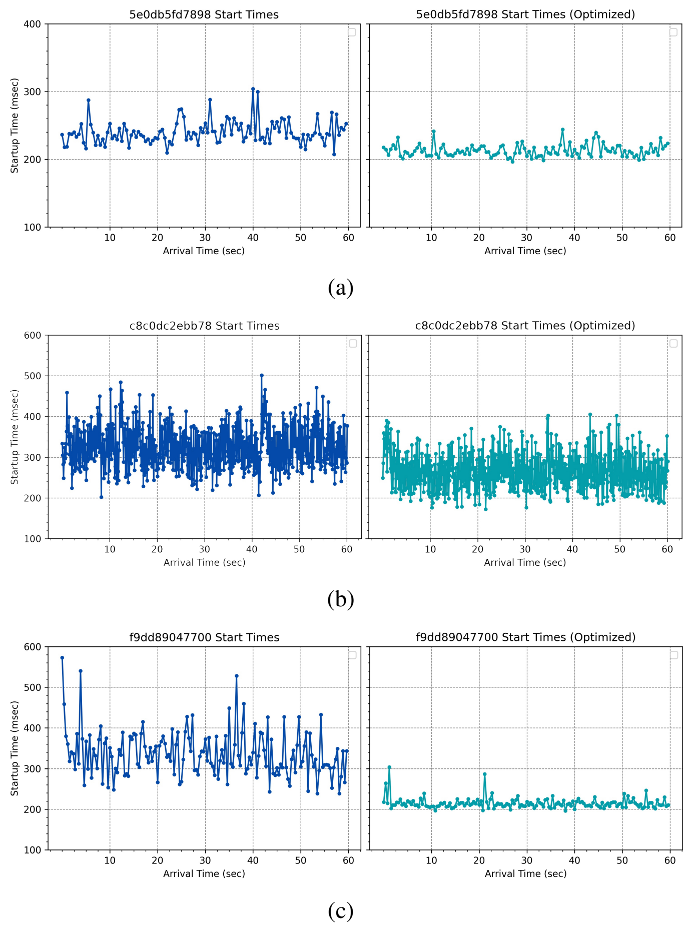
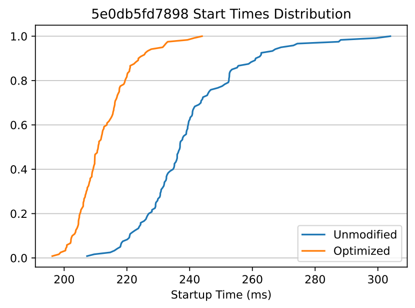


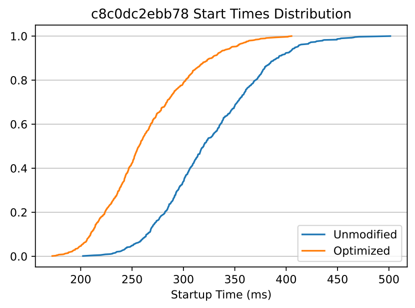
Fig. 10: Comparison of startup times for our 3 example functions when running a standard runtime (left figures) and a runtime with our proof-of-concept optimizations (right figures). Our optimizations yield an average of 22% reduction to cold start in these tests.

VIII. Related Work

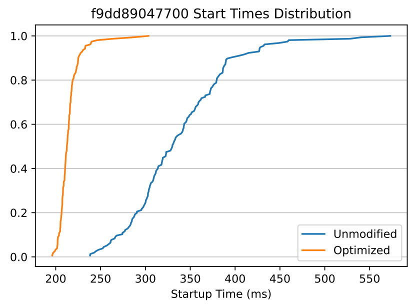
The primary goal of this paper is a thorough exploration of container startup operations to understand the cold start problem. To our knowledge, this work is the first instrumentation and evaluation of the component parts of a container runtime. However, over the past few years several similar research efforts have explored the overheads associated with container-based virtualization. Most recently, Young, et al. [32] conducted an evaluation on the performance of *gVisor*, the security-oriented container engine developed by Google and used to back their FaaS platform. In this study they deconstructed *gVisor* and provided an analysis of its inner workings, including overhead introduced by its sandboxing and security features. The *gVisor* runtime is similar to *runc* in that it isolates processes within a single OS and implements the OCI standard, but provides a much stronger security model by strictly brokering system calls and I/O via its Sentry and Gofer services running in userspace. The authors of this study focus on high level measurement of *gVisor*'s impact



(a)



(b)



(c)

Fig. 11: Distribution of startup times for our 3 example workloads when using a standard runtime vs. an optimized runtime. Variability and startup time are reduced for each workload evaluated.

on the speed of container startup, system calls, memory, networking, and storage, whereas our work focuses on low level measurement of the individual component functions used to create containers. Similar to our findings, the *gVisor* study shows that securing processes running in a container incurs significant overhead. Oakes, et al. [5] provide a brief exploration of the performance of some individual Linux OS primitives that enable containers, and use this to motivate and inform the design of a new container system for FaaS. Their evaluation stresses each primitive individually and shows how they behave when faced with large workloads. The work we do in this paper operates one level above this exploration, measuring the individual functions of a container runtime that operate with these primitives and how that operation in concert contributes to the container startup time.

Our motivation for understanding the cold start problem is to open the door to future research for enabling more efficient application isolation in FaaS platforms for Edge computing environments. Several alternative methods for application isolation have been proposed recently. As mentioned earlier in this section, *gVisor* [33] was developed as a way to provide container-like operation with an enhanced security model. In 2017, Manco, et al. introduced LightVM, a streamlined version of the Xen hypervisor that leverages unikernels to achieve fast boot times [34]. More recently, Agache, et al. presented Firecracker, a virtual machine monitor built to run MicroVMs and custom tailored to provide fast startup on serverless platforms without adversely affecting the capabilities available to end-users in existing solutions [35]. These methods show promising results toward improving process or application isolation, but also introduce trade-offs to achieve their goals, meaning there is no one-size-fits-all solution. For example, unikernels achieve very fast boot times by stripping their software stack to a bare minimum, but do so at the expense of reduced flexibility, in turn preventing them from supporting more robust applications. Because different use cases necessitate different approaches, containers remain a viable contender in this space and can be made more competitive through further improvements.

IX. Discussion and Concluding Remarks

Recent trends in low-latency, high-throughput applications such as AR/VR coupled with advances in connectivity such as 5G paint a picture of a near future where Edge computing is essential to providing computational offload with fast enough turnaround time to be useful. To meet this coming demand, it is important that we devise ways to provide execution environments that are very nimble. The research presented in this paper is a step in that direction.

Improving container-based virtualization for faster startup can be a boon to Edge computing, but is unlikely to be the only solution. Because of the dynamic nature of the Edge, and because it presents the opportunity to develop application types that have not yet been envisioned, it may be necessary to combine approaches to make a hybrid FaaS runtime which adapts to meet the needs of the application being served. For example, the hybrid model might leverage a programming language framework such as WebAssembly (as proposed by Hall, et al. [36]) or a microkernel (as proposed by Ren, et al. [37]) for simple, short-running applications; containers for moderately complex applications with longer execution times; and virtual machines for applications with long execution times that require strong isolation.

Opportunity exists to create a more nimble execution environment for the Edge. The measurement results we provide lay the foundation for determining how to improve container-based virtualization. Our analysis and proof-of-concept implementations to *runc* demonstrate the potential for creating a container runtime that provides fast enough startup to preserve the low-latency advantages of the Edge while keeping resource overhead low. These general ideas are transferable to other areas of related research which can yield new runtime paradigms for the unique applications of Edge computing.

X. Acknowledgements

We thank our anonymous reviewers for their insightful feedback and suggestions, which substantially improved the content of this paper. We would also like to thank fellow students in our lab for their careful reading and critique of earlier drafts of our manuscript. This work was funded in part by NSF CNS-2008368, Cisco, and a gift from Microsoft Corp.

References

[1] KubeEdge Project. (2022) KubeEdge. [Online]. Available: https://kubeedge.io

[2] K3s Project. (2022) K3s: Lightweight Kubernetes. [Online]. Available: https://k3s.io

[3] OpenFaas Ltd. (2022) OpenFaaS - Serverless Functions Made Simple. [Online]. Available: https://www.openfaas.com

[4] Apache Software Foundation. (2022) Apache OpenWhisk Open Source Serverless Cloud Platform. [Online]. Available: https://openwhisk.apache.org/

[5] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Sock: Rapid task provisioning with serverless-optimized containers," in *USENIX ATC'18*, 2018.

[6] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "Sand: towards high-performance serverless computing," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2018.

[7] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, analysis, and optimization of serverless function snapshots," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 559–572.

[8] Linux Manpages. (2022) namespaces - overview of Linux namespaces. [Online]. Available: http://man7.org/linux/man-pages/man7/namespaces.7.html

[9] ——. (2022) cgroups - Linux control groups. [Online]. Available: http://man7.org/linux/man-pages/man7/cgroups.7.html

[10] ——. (2022) clone - create a child process. [Online]. Available: http://man7.org/linux/man-pages/man2/clone.2.html

[11] ——. (2022) exec(3) Linux manual page. [Online]. Available: https://man7.org/linux/man-pages/man3/exec.3.html

[12] kernel.org. (2022) perf: Linux profiling with performance counters. [Online]. Available: https://perf.wiki.kernel.org/index.php

[13] Linux Manpages. (2022) strace - trace system calls and signals. [Online]. Available: http://man7.org/linux/man-pages/man1/strace.1.html

[14] Open Container Initiative. (2022) libcontainer. [Online]. Available: https://github.com/opencontainers/runc/tree/main/libcontainer

[15] ——. (2022) runc. [Online]. Available: https://github.com/opencontainers/runc

[16] Go Programming Language Maintainers. (2022) Defer statements. [Online]. Available: https://golang.org/ref/specDefer_statements

[17] ——. (2022) Package runtime. [Online]. Available: https://pkg.go.dev/runtime

[18] Intel Corporation, "Intel Resource Directory Technology (Intel RDT)," 2022. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html

[19] AppArmor Project, "The AppArmor user space development project," 2022. [Online]. Available: https://gitlab.com/apparmor/apparmor

[20] Linux Manpages, "seccomp - operate on Secure Computing state of the process," 2022. [Online]. Available: http://man7.org/linux/man-pages/man2/seccomp.2.html

[21] The libseccomp Project. (2022) libseccomp. [Online]. Available: https://github.com/seccomp/libseccomp-golang

[22] GoDoc Project. (2022) package seccomp, func NewFilter. [Online]. Available: https://godoc.org/github.com/seccomp/libseccomp-golangNewFilter

[23] The GoDoc Project, "GoDoc - package seccomp, func (*ScompFilter) Load," https://godoc.org/github.com/seccomp/libseccomp-golangScmpFilter.Load, 2020.

[24] Linux Manpages. (2022) syscalls - Linux system calls. [Online]. Available: http://man7.org/linux/man-pages/man2/syscalls.2.html

[25] ——, "proc - process information pseudo-filesystem," http://man7.org/linux/man-pages/man5/proc.5.html, 2020.

[26] The GoDoc Project, "GoDoc - package unix," https://godoc.org/golang.org/x/sys/unix, 2020.

[27] Linux Kernel Maintainers. (2022) cgroups. [Online]. Available: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt

[28] Linux Manpages. (2022) umount(2) Linux manual page. [Online]. Available: https://man7.org/linux/man-pages/man2/umount.2.html

[29] C. Doxsey, "An Introduction to Programming in Go / Concurrency," https://www.golang-book.com/books/intro/10, 2020.

[30] G. P. L. Maintainers, "sync - The Go Programming Language," https://golang.org/pkg/sync/WaitGroup, 2020.

[31] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 205–218.

[32] E. G. Young, P. Zhu, T. Caraza-Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "The true cost of containing: A gvisor case study," in *11th USENIX HotCloud 19*, 2019.

[33] The gVisor Authors, "gVisor," https://gvisor.dev, 2020.

[34] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici, "My vm is lighter (and safer) than your container," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 218–233.

[35] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020, pp. 419–434.

[36] A. Hall and U. Ramachandran, "An execution model for serverless functions at the edge," in *Proceedings of the International Conference on Internet of Things Design and Implementation*, 2019, pp. 225–236.

[37] Y. Ren, G. Liu, V. Nitu, W. Shao, R. Kennedy, G. Parmer, T. Wood, and A. Tchana, "Fine-grained isolation for scalable, dynamic, multi-tenant edge clouds," in *USENIX ATC 20*, 2020, pp. 927–942.