CNTR: Lightweight OS Containers

Jörg Thalheim, Pramod Bhatotia
University of Edinburgh

Pedro Fonseca
University of Washington

Baris Kasikci
University of Michigan

Abstract

Container-based virtualization has become the de-facto standard for deploying applications in data centers. However, deployed containers frequently include a wide-range of tools (e.g., debuggers) that are not required for applications in the common use-case, but they are included for rare occasions such as in-production debugging. As a consequence, containers are significantly larger than necessary for the common case, thus increasing the build and deployment time.

CNTR[1] provides the *performance* benefits of lightweight containers and the *functionality* of large containers by splitting the traditional container image into two parts: the "fat" image — containing the tools, and the "slim" image — containing the main application. At run-time, CNTR allows the user to efficiently deploy the "slim" image and then expand it with additional tools, when and if necessary, by dynamically attaching the "fat" image.

To achieve this, CNTR transparently combines the two container images using a new nested namespace, without any modification to the application, the container manager, or the operating system. We have implemented CNTR in Rust, using FUSE, and incorporated a range of optimizations. CNTR supports the full Linux filesystem API, and it is compatible with all container implementations (i.e., Docker, rkt, LXC, systemd-nspawn). Through extensive evaluation, we show that CNTR incurs reasonable performance overhead while reducing, on average, by 66.6% the image size of the Top-50 images available on Docker Hub.

1 Introduction

Containers offer an appealing, lightweight alternative to VM-based virtualization (e.g., KVM, VMware, Xen) that relies on process-based virtualization. Linux, for instance, provides the `cgroups` and `namespaces` mechanisms that enable strong performance and security isolation between containers [24]. Lightweight virtualization is fundamental to achieve high efficiency in virtualized datacenters and enables important use-cases, namely just-in-time deployment of applications. Moreover, containers significantly reduce operational costs through higher consolidation density and power minimization, especially in multi-tenant environments. Because of all these advantages, it is no surprise that containers have seen wide-spread adoption by industry, in many cases replacing altogether traditional virtualization solutions [17].

Despite being lightweight, deployed containers often include a wide-range of tools such as shells, editors, coreutils, and package managers. These additional tools are usually not required for the application's core function — the common operational use-case — but they are included for management, manual inspection, profiling, and debugging purposes [64]. In practice, this significantly *increases container size* and, in turn, translates into slower container deployment and inefficient datacenter resource usage (network bandwidth, CPU, RAM and disk). Furthermore, larger images degrade container deployment time [52, 44]. For instance, previous work reported that downloading container images account for 92% of the deployment time [52]. Moreover, a larger code base directly affects the reliability of applications in datacenters [50].

Given the impact of using large containers, users are discouraged from including additional tools that would otherwise simplify the process of debugging, deploying, and managing containers. To mitigate this problem, Docker has recently adopted smaller run-times but, unfortunately, these efforts come at the expense of compatibility problems and have limited benefits [13].

To quantify the practical impact of additional tools on the container image size, we employed Docker Slim [11] on the 50 most popular container images available on the Docker Hub repository [10]. Docker Slim uses a combination of static and dynamic analyses to generate smaller-sized container images, in which, only files needed by the core application are included in the final image. The results of this experiment (see Figure 5) are

[1] Read it as "center".

encouraging: we observed that by excluding unnecessary files from typical containers it is possible to reduce the container size, on average, by 66.6%. Similarly, others have found that a only small subset (6.4%) of the container images is read in the common case [53].

CNTR addresses this problem[2] by building lightweight containers that still remain fully functional, even in uncommon use-cases (e.g., debugging and profiling). CNTR enables users to deploy the application and its dependencies, while the additional tools required for other use-cases are supported by expanding the container "on-demand", during runtime (Figure 1 (a)). More specifically, CNTR splits the traditional container image into two parts: the "fat" image containing the rarely used tools and the "slim" image containing the core application and its dependencies.

During runtime, CNTR allows the user of a container to efficiently deploy the "slim" image and then expand it with additional tools, when and if necessary, by dynamically attaching the "fat" image. As an alternative to using a "fat" image, CNTR allows tools from the *host* to run inside the container. The design of CNTR simultaneously preserves the performance benefits of lightweight containers and provides support for additional functionality required by different application workflows.

The key idea behind our approach is to create a new *nested namespace* inside the application container (i.e., "slim container"), which provides access to the resources in the "fat" container, or the host, through a FUSE filesystem interface. CNTR uses the FUSE system to combine the filesystems of two images without any modification to the application, the container implementation, or the operating system. CNTR selectively redirects the filesystem requests between the mount namespace of the container (i.e., what applications within the container observe and access) and the "fat" container image or the host, based on the filesystem request path. Importantly, CNTR supports the full Linux filesystem API and all container implementations (i.e., Docker, rkt, LXC, systemd-nspawn).

We evaluated CNTR across three key dimensions: (1) *functional completeness* – CNTR passes 90 out of 94 (95.74%) `xfstests` filesystem regression tests [14] supporting applications such as SQLite, Postgres, and Apache; (2) *performance* – CNTR incurs reasonable overheads for the `Phoronix` filesystem benchmark suite [18], and the proposed optimizations significantly improve the overall performance; and lastly, (3) *effectiveness* – CNTR's approach on average results in a 66.6% reduction of image size for the Top-50 images available on Docker hub [10]. We have made publicly available the CNTR implementation along with the experimental setup [6].

[2]Note that Docker Slim [11] does not solve the problem; it simply identifies the files not required by the application, and excludes them from the container, but it does not allow users to access those files at run-time.

2 Background and Motivation

2.1 Container-Based Virtualization

Containers consist of a lightweight, process-level form of virtualization that is widely used and has become a cornerstone technology for datacenters and cloud computing providers. In fact, all major cloud computing providers (e.g., Amazon [2], Google [16] and Microsoft [4]) offer Containers as a Service (CaaS).

Container-based virtualization often relies on three key components: (1) the OS mechanism that enforces the process-level isolation (e.g., the Linux `cgroups` [41] and `namespaces` [40] mechanisms), (2) the application packaging system and runtime (e.g., Docker [9], Rkt [38]), and (3) the orchestration manager that deploys, distributes and manages containers across machines (e.g., Docker Swarm [12], Kubernetes [22]). Together, these components enable users to quickly deploy services across machines, with strong performance and security isolation guarantees, and with low-overheads.

Unlike VM-based virtualization, containers do not include a guest kernel and thus have often smaller memory footprint than traditional VMs. Containers have important advantages over VMs for both users and data centers:

1. **Faster deployment.** Containers are transferred and deployed faster from the registry [44].
2. **Lower resource usage.** Containers consume fewer resources and incur less performance overhead [62].
3. **Lower build times.** Containers with fewer binaries and data can be rebuilt faster [64].

Unfortunately, containers in practice are still unnecessarily large because users are forced to decide which auxiliary tools (e.g. debugging, profiling, etc.) should be included in containers at *packaging-time*. In essence, users are currently forced to strike a balance between lightweight containers and functional containers, and end up with containers that are neither as light nor as functional as desirable.

2.2 Traditional Approaches to Minimize Containers

The container-size problem has been a significant source of concern to users and developers. Unfortunately, existing solutions are neither practical nor efficient.

An approach that has gained traction, and has been adopted by Docker, consists of packing containers using smaller base distributions when building the container runtime. For instance, most of Docker's containers are now based on the Alpine Linux distribution [13], resulting in smaller containers than traditional distributions. Alpine Linux uses the `musl` library, instead of `libc`, and bundles `busybox` , instead of `coreutils` — these differences enable a smaller container runtime but at the expense of compatibility problems caused by runtime differences. Further, the set of tools included is still restricted and fundamentally does not help users when less common auxiliary tools are required (e.g., custom debugging tools).

The second approach to reduce the size of containers relies on union filesystems (e.g., UnionFS [60]). Docker, for instance, enables users to create their containers on top of commonly-used base images. Because such base images are expected to be shared across different containers (and already deployed in the machines), deploying the container only requires sending the diff between the base image and the final image. However, in practice, users still end up with multiple base images due to the use of different base image distributions across different containers.

Another approach that has been proposed relies on the use of *unikernels* [57, 58], a single-address-space image constructed from a *library OS* [61, 49, 65]. By removing layers of abstraction (e.g., processes) from the OS, the unikernel approach can be leveraged to build very small virtual machines—this technique has been considered as containerization because of its low overhead, even though it relies on VM-based virtualization. However, unikernels require additional auxiliary tools to be statically linked into the application image; thus, it leads to the same problem.

2.3 Background: Container Internals

The container abstraction is implemented by a userspace container run-time, such as Docker [9], rkt [38] or LXC [37]. The kernel is only required to implement a set of per-process isolation mechanisms, which are inherited by child processes. This mechanism is in turn leveraged by container run-times to implement the actual container abstraction. For instance, applications in different containers are isolated and have all their resources bundled through their own filesystem tree. Crucially, the kernel allows the partitioning of system resources, for a given process, with very low performance overhead thus enabling efficient process-based virtualization.

The Linux operating system achieves isolation through an abstraction called namespaces. Namespaces are modular and are applied to individual processes inherited by child processes. There are seven namespaces to limit the scope what a process can access (e.g., filesystem mountpoints, network interfaces, or process IDs[40]).

During the container startup, by default, namespaces of the host are unshared. Hence, processes inside the container only see files from their filesystem image (see Figure 1 (a)) or additional volumes, that have been statically added during setup. New mounts on the host are not propagated to the container since by default, the container runtime will mount all mount points as private.

2.4 Use-cases of CNTR

We envision three major use cases for CNTR that cover three different debugging/management scenarios:

Container to container debugging in production. CNTR enables the isolation of debugging and administration tools in *debugging containers* and allows application containers to use debugging containers on-demand. Consequently, application containers become leaner, and the isolation of debugging/administration tools from applications allows users to have a more consistent debugging experience. Rather than relying on disparate tools in different containers, CNTR allows using a single debugging container to serve many application containers.

Host to container debugging. CNTR allows developers to use the debugging environments (e.g., IDEs) in their host machines to debug containers that do not have these environments installed. These IDEs can sometimes take several gigabytes of disk space and might be not even compatible with the distribution of the container image is based on. Another benefit of using CNTR in this context is that development environments and settings can be also efficiently shared across different containers.

Container to host administration and debugging. Container-oriented Linux distributions such as CoreOS [8] or RancherOS [30] do not provide a package manager and users need to extend these systems by installing containers even for basic system services. CNTR allows a user of a privileged container to access the root filesystem of the host operating system. Consequently, administrators can keep tools installed in a debug container while keeping the host operating system's filesystem lean.

3 Design

In this section, we present the detailed design of CNTR.

3.1 System Overview

Design goals. CNTR has the following design goals:

- *Generality:* CNTR should support a wide-range of workflows for seamless management and problem diagnosis (e.g., debugging, tracing, profiling).
- *Transparency:* CNTR should support these workflows without modifying the application, the container manager, or the operating system. Further, we want to be compatible with all container implementations.
- *Efficiency:* Lastly, CNTR should incur low performance overheads with the split-container approach.

Basic design. CNTR is composed of two main components (see Figure 1 (a)): a nested namespace, and the CNTRFS filesystem. In particular, CNTR combines slim and fat containers by creating a new *nested namespace* to merge the namespaces of two containers (see Figure 1 (b)). The nested namespace allows CNTR to selectively break the isolation between the two containers by transparently redirecting the requests based on the accessed path. CNTR achieves this redirection using the CNTRFS filesystem. CNTRFS is mounted as the root filesystem (`/`), and the application filesystem is remounted to another path (`/var/lib/cntr`) in the nested namespace. CNTRFS implements a filesystem in userspace (FUSE), where the CNTRFS server handles the requests for auxiliary tools installed on the fat container (or on the host).

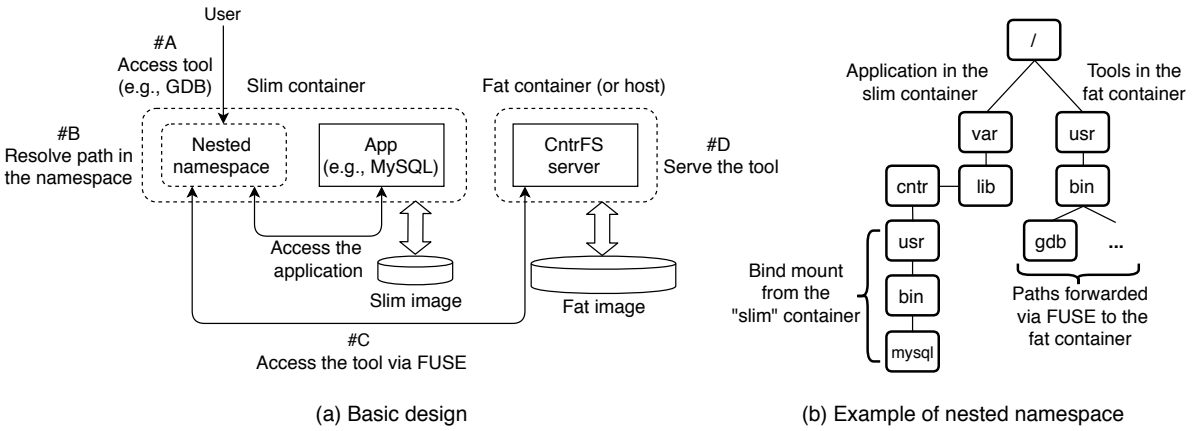

Figure 1: Overview of CNTR

At a high-level, CNTR connects with the CNTRFS server via the generic FUSE kernel driver. The kernel driver simply acts as a proxy between processes accessing CNTRFS, through Linux VFS, and the CNTRFS server running in userspace. The CNTRFS server can be in a different mount namespace than the nested namespace, therefore, CNTR establishes a proxy between two mount namespaces through a request/response protocol. This allows a process that has all its files stored in the fat container (or the host) to run within the mount namespace of the slim container.

Cntr workflow. CNTR is easy to use. The user simply needs to specify the name of the "slim" container and, in case the tools are in another container, the name of the "fat" container. CNTR exposes a shell to the user that has access to the resources of the application container as well as the resources forwarded from the fat container.

Figure 1 (a) explains the workflow of CNTR when a user requests to access a tool from the slim container (#A): CNTR transparently resolves the requested path for the tool in the nested namespace (#B). Figure 1 (b) shows an example of CNTR's nested namespace, where the requested tool (e.g., gdb) is residing in the fat container. After resolving the path, CNTR redirects the request via FUSE to the fat container (#C). Lastly, CNTRFS serves the requested tool via the FUSE interface (#D). Behind the scenes, CNTR executes the following steps:

1. *Resolve container name to process ID and get container context.* CNTR resolves the name of the underlying container process IDs and then queries the kernel to get the complete execution context of the container (container namespaces, environment variables, capabilities, ...).
2. *Launch the* CNTRFS *server.* CNTR launches the CNTRFS server. CNTR launches the server either directly on the host or inside the specified "fat" container containing the tools image, depending on the settings that the user specified.
3. *Initialize the tools namespace.* Subsequently, CNTR attaches itself to the application container by setting up a nested mount namespace within the namespace of the application container. CNTR then assigns a forked process to the new namespace. Inside the new namespace, the CNTR process proceeds to mount CNTRFS, providing access to files that are normally out of the scope of the application container.
4. *Initiate an interactive shell.* Based on the configuration files within the debug container or on the host, CNTR executes an interactive shell, within the nested namespace, that the user can interact with. CNTR forwards its input/output to the user terminal (on the host). From the shell, or through the tools it launches, the user can then access the application filesystem under `/var/lib/cntr` and the tools filesystem in `/`. Importantly, tools have the same view on system resources as the application (e.g., `/proc`, `ptrace`). Furthermore, to enable the use of graphical applications, CNTR forwards Unix sockets from the host/debug container.

3.2 Design Details

This section explains the design details of CNTR.

3.2.1 Step #1: Resolve Container Name and Obtain Container Context

Because the kernel has no concept of a container name or ID, CNTR starts by resolving the container name, as defined by the used container manager, to the process IDs running inside the container. CNTR leverages wrappers based on the container management command line tools to achieve this translation and currently, it supports Docker, LXC, rkt, and systemd-nspawn.

After identifying the process IDs of the container, CNTR gathers OS-level information about the container namespace. CNTR reads this information by inspecting the `/proc` filesystem of the main process within the container.

This information enables CNTR to create processes inside the container in a *transparent* and *portable* way.

In particular, CNTR gathers information about the container namespaces, cgroups (resource usage limits), mandatory access control (e.g., AppArmor [26] and SELinux [19] options), user ID mapping, group ID mapping, capabilities (fine-grained control over super-user permissions), and process environment options. Additionally, CNTR could also read the seccomp options, but this would require non-standard kernel compile-time options and generally has limited value because seccomp options have significant overlap with the capability options. CNTR reads the environment variables because they are heavily used in containers for configuration and service discovery [36].

Before attaching to the container, in addition, to gather the information about the container context, the CNTR process opens the FUSE control socket (`/dev/fuse`). This file descriptor is required to mount the CNTRFS filesystem, after attaching to the container.

3.2.2 Step #2: Launch the CNTRFS Server

The CNTRFS is executed either directly on the host or inside the "fat" container, depending on the option specified by the user (i.e., the location of the tools). In the host case the CNTRFS server simply runs like a normal host process.

In case the user wants to use tools from the "fat" container, the CNTRFS process forks and attaches itself to the "fat" container. Attaching to the "fat" container is implemented by calling the `setns()` system call, thereby assigning the child process to the container namespace that was collected in the previous step.

After initialization, the CNTRFS server waits for a signal from the nested namespace (Step #3) before it starts reading and serving the FUSE requests (reading before an unmounted FUSE filesystem would otherwise return an error). The FUSE requests then will be read from the `/dev/fuse` file descriptor and redirected to the filesystem of the server namespace (i.e., host or fat container).

3.2.3 Step #3: Initialize the Tools Namespace

CNTR initializes the tool namespace by first attaching to the container specified by the user—the CNTR process forks and the child process assigns itself to the cgroup, by appropriately setting the `/sys/` option, and namespace of the container, using the `setns()` system call.

After attaching itself to the container, CNTR creates a new nested namespace, and marks all mountpoints as private so that further mount events (regarding the nested namespace) are not propagated back to the container namespace. Subsequently, CNTR creates a new filesystem hierarchy for the nested namespace, mounting the CNTRFS in a temporary mountpoint (`TMP/`).

Within the nested namespace, the child process mounts CNTRFS, at `TMP/`, and signals the parent process (running outside of the container) to start serving requests. Signalling between the parent and child CNTR processes is implemented through a shared Unix socket.

Within the nested namespace, the child process remounts all pre-existing mountpoints, from the application container, by moving them from `/` to `TMP/var/lib/cntr`. Note that the application container is not affected by this since all mountpoints are marked as private.

In addition, CNTR also mounts special container-specific files from the application over files from the tools or host (using bind mount [42]). The special files include the pseudo filesystems `procfs` (`/proc`), ensuring the tools can access the container application, and devtmpfs (`/dev`), containing block and character devices that have been made visible to our container. Furthermore, we bind mount a set of configuration files from the application container into the temporary directory (e.g., `/etc/passwd`, and `/etc/hostname`).

Once the new filesystem hierarchy has been created in the temporary directory, CNTR atomically executes a `chroot` turning the temporary directory (`TMP/`) into the new root directory (`/`).

To conclude the container attachment and preserve the container isolation guarantees, CNTR updates the remaining properties of the nested namespace: (1) CNTR drops the capabilities by applying the AppArmor/SELinux profile and (2) CNTR applies all the environment variables that were read from the container process; with the exception of `PATH` – the `PATH` is instead inherited from the debug container since it is often required by the tools.

3.2.4 Step #4: Start Interactive Shell

Lastly, CNTR launches an interactive shell within the nested namespace, enabling users to execute the tools. CNTR forwards the shell I/O using a pseudo-TTY, and supports graphical interface using Unix sockets forwarding.

Shell I/O. Interactive shells perform I/O through standard file descriptors (i.e., stdin, stdout, and stderr file descriptors) that generally refer to terminal devices. For isolation and security reasons, CNTR prevents leaking the terminal file descriptors of the host to a container by leveraging pseudo-TTYs – the pseudo-TTY acts as a proxy between the interactive shell and the user terminal device.

Unix socket forwarding. CNTR forwards connections to Unix sockets, e.g., the X11 server socket and the D-Bus daemon running on the host. Unix sockets are also visible as files in our FUSE. However, since our FUSE has inode numbers that are different from the underlying filesystem, the kernel does not associate them with open sockets in the system. Therefore, we implemented a socket proxy that runs an efficient event loop based on epoll. It uses the splice syscall to move data between clients in the application container and servers listening on Unix sockets in the debug container/host.

3.3 Optimizations

We experienced performance slowdown in CNTRFS when we measured the performance using the Phoronix benchmark suite [18] (§5.2). Therefore, we incorporated the following performance optimizations in CNTR.

Caching: Read and writeback caches. The major performance improvement gain was by allowing the FUSE kernel module to cache data returned from the read requests as well as setting up a writeback buffer for the writes. CNTR avoids automatic cache invalidation when a file is opened by setting the `FOPEN_KEEP_CACHE` flag. Without this flag the cache cannot be effectively shared across different processes. To allow the FUSE kernel module to batch smaller write requests, we also enable the writeback cache by specifying the `FUSE_WRITEBACK_CACHE` flag at the mount setup time. This optimization sacrifices write consistency for performance by delaying the sync operation. However, we show that it still performs correctly according to the POSIX semantics in our regression experiments (see § 5.1).

Multithreading. Since the I/O operations can block, we optimized the CNTRFS implementation to use multiple threads. In particular, CNTR spawns independent threads to read from the CNTRFS file descriptor independently to avoid contentions while processing the I/O requests.

Batching. In addition to caching, we also batch operations to reduce the number of context switches. In particular, we apply the batching optimization in three places: (a) pending inode lookups, (b) forget requests, and (c) concurrent read requests.

Firstly, we allow concurrent inode lookups by applying `FUSE_PARALLEL_DIROPS` option on mount. Secondly, the operating system sends `forget` requests, when inodes can be freed up by CNTRFS. The kernel can batch a forget intent for multiple inodes into a single request. In CNTR we have also implemented this request type. Lastly, we set `FUSE_ASYNC_READ` to allow the kernel to batch multiple concurrent read requests at once to improve the responsiveness of read operations.

Splicing: Read and write. Previous work suggested the use of splice reads and writes to improve the performance of FUSE [66]. The idea behind splice operation is to avoid copying data from and to userspace. CNTR uses splice for read operations. Therefore, the FUSE userspace process moves data from the source file descriptor into a kernel pipe buffer and then to the destination file descriptor with the help of the `splice` syscall. Since splice does not actually copy the data but instead remaps references in the kernel, it reduces the overhead.

We also implemented a splice write optimization. In particular, we use a pipe as a temporary storage, where the data is part of the request, and the data is not read from a file descriptor. However, FUSE does not allow to read the request header into userspace without reading the attached data. Therefore, CNTR has to move the whole request to a kernel pipe first in order to be able to read the request header separately. After parsing the header it can move the remaining data to its designated file descriptor using the splice operation. However, this introduces an additional context switch, and slowdowns all FUSE operations since it is not possible to know in advance if the next request will be a write request. Therefore, we decided not to enable this optimization by default.

4 Implementation

To ensure portability and maintainability, we decided not to rely on container-specific APIs, since they change quite often. Instead, we built our system to be as generic as possible by leveraging more stable operating system interfaces. Our system implementation supports all major container types: Docker, LXC, systemd-nspawn and rkt. CNTR's implementation resolves container names to process ids. Process ids are handled in an implementation-specific way. On average, we changed only 70 LoCs for each container implementation to add such container-specific support for CNTR.

At a high-level, our system implementation consists of the following four components:

- *Container engine* (1549 LoC) analyzes the container that a user wants to attach to. The container engine also creates a nested mount namespace, where it starts the interactive shell.
- CNTRFS (1481 LoC) to serve the files from the fat container. We implemented CNTRFS based on Rust-FUSE [33]. We extended Rust-FUSE to be able to mount across mount namespaces and without a dedicated FUSE mount executable.
- A *pseudo TTY* (221 LoC) to connect the shell input/output with the user terminal.
- A *socket proxy* (400 LoC) to forward the Unix socket connection between the fat (or the host) and slim containers for supporting X11 applications.

All core system components of CNTR were implemented in Rust (total 3651 LoC). To simplify deployment, we do not depend on any non-Rust libraries. In this way, we can compile CNTR as a $\sim$1.2MB single self-contained static executable by linking against musl-libc [23]. This design is imperative to ensure that CNTR can run on container-optimized Linux distributions, such as CoreOS [8] or RancherOS [30], that do not have a package manager to install additional libraries.

Since CNTR makes heavy use of low-level filesystem system calls, we have also extended the Rust ecosystem with additional 46 system calls to support the complete Linux filesystem API. In particular, we extended the nix Rust library [34], a library wrapper around the Linux/`POSIX` API. The changes are available in our fork [29].

5 Evaluation

In this section, we present the experimental evaluation of CNTR. Our evaluation answers the following questions.

1. Is the implementation complete and correct? (§5.1)
2. What are the performance overheads and how effective are the proposed optimizations? (§5.2)
3. How effective is the approach to reducing container image sizes? (§5.3)

5.1 Completeness and Correctness

We first evaluate the completeness and correctness claim of the CNTR implementation. The primary goal is to evaluate whether CNTR implements the same features (completeness) as required by the underlying filesystem, and it follows the same `POSIX` semantics (correctness).

Benchmark: xfstests regression test suite. For this experiment, we used the `xfstests` [14] filesystem regression test suite. The `xfstests` suite was originally designed for the XFS filesystem, but it is now widely used for testing all of Linux's major filesystems. It is regularly used for quality assurance before applying changes to the filesystem code in the Linux kernel. `xfstests` contains tests suites to ensure correct behavior of all filesystem related system calls and their edge cases. It also includes crash scenarios and stress tests to verify if the filesystem correctly behaves under load. Further, it contains many tests for bugs reported in the past.

Methodology. We extended `xfstests` to support mounting CNTRFS. For running tests, we mounted CNTRFS on top of `tmpfs`, an in-memory filesystem. We run all tests in the generic group once.

Experimental results. `xfstests` consists of 94 unit tests that can be grouped into the following major categories: *auto*, *quick*, *aio*, *prealloc*, *ioctl*, and *dangerous*.

Overall, CNTR passed 90 out of 94 (95.74%) unit tests in `xfstests`. Four tests failed due minor implementation details that we currently do not support. Specifically, these four unit tests were automatically skipped by `xfstests` because they expected our filesystem to be backed by a block device or expected some missing features in the underlying `tmpfs` filesystem, e.g. copy-on-write ioctl. We next explain the reasons for the failed four test cases:

1. **Test #375** failed since `SETGID` bits were not cleared in `chmod` when the owner is not in the owning group of the access control list. This would require manual parsing and interpreting ACLs in CNTR. In our implementation, we delegate `POSIX` ACLs to the underlying filesystem by using `setfsuid/setfsguid` on inode creation.
2. **Test #228** failed since we do not enforce the per-process file size limits (`RLIMIT_FSIZE`). As replay file operations and `RLIMIT_FSIZE` of the caller is not set or enforced in CNTRFS, this has no effect.
3. **Test #391** failed since we currently do not support the direct I/O flag in `open` calls. The support for direct I/O and `mmap` in FUSE is mutually exclusive. We chose `mmap` here, since we need it to execute processes. In practice, this is not a problem because not all docker drivers support this feature, including the popular filesystems such as `overlayfs` and `zfs`.
4. **Test #426** failed since our inodes are not exportable. In Linux, a process can get inode references from filesystems by the `name_to_handle_at` system call. However, our inodes are not persisted and are dynamically requested and destroyed by the operating system. If the operating system no longer uses them, they become invalid. Many container implementations block this system call as it has security implications.

To summarize, the aforementioned failed test cases are specific to our current state of the implementation, and they should not affect most real-world applications. As such, these features are not required according to the `POSIX` standard, but, they are Linux-specific implementation details.

5.2 Performance Overheads and Optimizations

We next report the performance overheads for CNTR's split-containers approach (§5.2.1), detailed experimental results (§5.2.2), and effectiveness of the proposed optimizations (§5.2.3).

Experimental testbed. To evaluate a realistic environment for container deployments [3], we evaluated the performance benchmarks using `m4.xlarge` virtual machine instances on Amazon EC2. The machine type has two cores of Intel Xeon E5-2686 CPU (4 hardware threads) assigned and 16GB RAM. The Linux kernel version was 4.14.13. For storage, we used a 100GB EBS volume of type GP2 formatted with `ext4` filesystem mounted with default options. GP2 is an SSD-backed storage and attached via a dedicated network to the VM.

Benchmark: Phoronix suite. For the performance measurement, we used the disk benchmarks [39] from the `Phoronix` suite [18]. `Phoronix` is a meta benchmark that has a wide range of common filesystem benchmarks, applications, and realistic workloads. We compiled the benchmarks with GCC 6.4 and CNTR with Rust 1.23.0.

Methodology. For the performance comparison, we ran the benchmark suite once on the native filesystem (the baseline measurement) and compared the performance when we access the same filesystem through CNTRFS. The `Phoronix` benchmark suite runs each benchmark at least three times and automatically adds additional trials if the variance is too high. To compute the relative overheads with respect to the baseline, we computed the ratio between the native filesystem access and CNTRFS (*native*/*cntr*) for benchmarks where higher values are better (e.g. throughput), and the inverse ratio

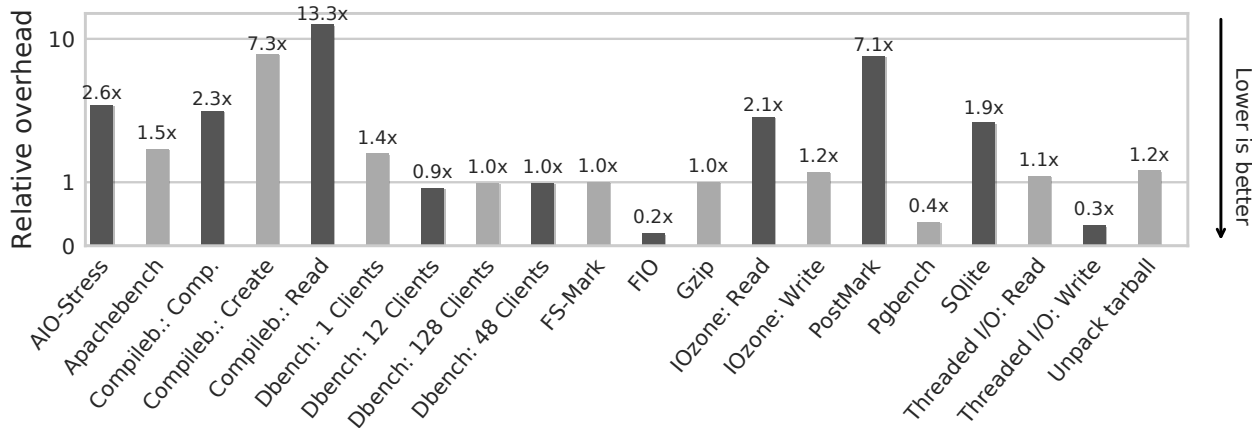

Figure 2: Relative performance overheads of CNTR for the `Phoronix` suite. The absolute values for each benchmark is available online on the openbenchmark platform [31].

(*cntr*/*native*), where lower values are better (e.g. time required to complete the benchmark).

5.2.1 Performance Overheads

We first present the summarized results for the entire benchmark suite. Thereafter, we present a detailed analysis of each benchmark individually (§5.2.2).

Summary of the results. Figure 2 shows the relative performance overheads for all benchmarks in the `Phoronix` test suite. We have made the absolute numbers available for each benchmark on the openbenchmark platform [31].

Our experiment shows that 13 out of 20 (65%) benchmarks incur moderate overheads below 1.5× compared to the native case. In particular, three benchmarks showed significantly higher overheads, including `compilebench-create` (7.3×) and `compilebench-read` (13.3×) and the `postmark` benchmark (7.1×). Lastly, we also had three benchmarks, where CNTRFS was faster than the native baseline execution: `FIO` (0.2×), `PostgreSQL Bench` (0.4×) and the write workload of `Threaded I/O` (0.3×).

To summarize, the results show the strengths and weaknesses of CNTRFS for different applications and under different workloads. At a high-level, we found that the performance of inode lookups and the double buffering in the page cache are the main performance bottlenecks in our design (much like they are for FUSE). Overall, the performance overhead of CNTR is reasonable. Importantly, note that while reporting performance numbers, we resort to the worst-case scenario for CNTR, where the "slim" application container aggressively uses the "fat" container to run an I/O-intensive benchmark suite. However, we must emphasize the primary goal of CNTR: to support auxiliary tools in uncommon operational use-cases, such as debugging or manual inspection, which are not dominated by high I/O-intensive workloads.

5.2.2 Detailed Experimental Results

We next detail the results for each benchmark.

AIO-Stress. `AIO-Stress` submits 2GB of asynchronous write requests. In theory, CNTRFS supports asynchronous requests, but only when the filesystem operates in the direct I/O mode. However, the direct I/O mode in CNTRFS restricts the `mmap` system call, which is required by executables. Therefore, all requests are, in fact, processed synchronously resulting in 2.6× slowdown.

Apache Web server. The `Apache Web server` benchmark issues 100K `http` requests for test files (average size of 3KB), where we noticed a slowdown of up to 1.5×. However, the bottleneck was not due to serving the actual content, but due to writing of the webserver access log, which triggers small writes (< 100 bytes) for each request. These small requests trigger lookups in CNTRFS of the extended attributes `security.capabilities`, since the kernel currently neither caches such attributes nor it provides an option for caching them.

Compilebench. `Compilebench` simulates different stages in the compilation process of the Linux kernel. There are three variants of the benchmark: (a) the `compile` stage compiles a kernel module, (b) the `read tree` stage reads a source tree recursively, and lastly, (c) the `initial creation` stage simulates a tarball unpack. In our experiments, `Compilebench` has the highest overhead of all benchmarks with the `read tree` stage being the slowest (13.4×). This is due to the fact that inode lookups in CNTRFS are slower compared to the native filesystem: for every lookup, we need one `open()` system call to get a file handle to the inode, followed by a `stat()` system call to check if we already have lookup-ed this inode in a different path due hardlinks. Usually, after the first lookup, this information can be cached in the kernel, but in this benchmark for every run, a different source tree with many files are read. The slowdown of

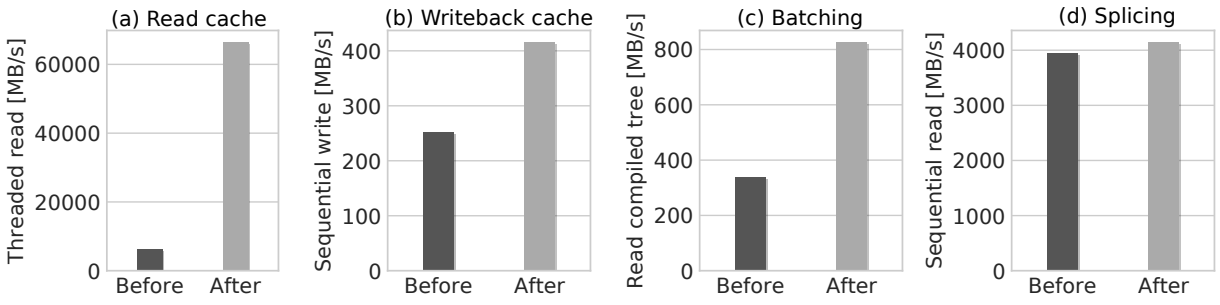

Figure 3: Effectiveness of optimizations
(a) `Threaded I/O` bench - Read (b) `IOZone` - Sequential write with 4GB (record size 4KB)
(c) `Compilebench` - Read (d) `IOZone` - Sequential read 4GB (record size 4KB)

lookups for the other two variants, namely the `compile stage` (2.3×) and `initial create` (7.3×) is lower, since they are shadowed by write operations.

Dbench. `Dbench` simulates a file server workload, and it also simulates clients reading files and directories with increasing concurrency. In this benchmark, we noticed that with increasing number of clients, CNTRFS is able to cache directories and file contents in the kernel. Therefore, CNTRFS does not incur performance overhead over the native baseline.

FS-Mark. `FS-Mark` sequentially creates 1000 1MB files. Since the write requests are reasonably large (16 KB per `write` call) and the workload is mostly disk bound. Therefore, there is no difference between CNTRFS and `ext4`.

FIO benchmark. The `FIO` benchmark profiles a fileserver and measures the read/write bandwidth, where it issues 80% random reads and 20% random writes for 4GB data with an average blocksize of 140KB. For this benchmark, CNTRFS outperforms the native filesystem by a factor of 4× since the writeback cache leads to fewer and larger writes to the disk compared to the underlying filesystem.

Gzip benchmark. The `Gzip` benchmark reads a 2GB file containing only zeros and writes the compressed version of it back to the disk. Even though the file is highly compressible, `gzip` compresses the file slower than the data access in CNTRFS or `ext4`. Therefore, there was no significant performance difference between CNTR and the native version.

IOZone benchmark. `IOZone` performs sequential writes followed by sequential reads of a blocksize of 4KB. For the write requests, as in the `apache` benchmark, CNTR incurs low overhead (1.2×) due to extended attribute lookup overheads. Whereas, for the sequential read request, both filesystems (underlying native filesystem and CNTRFS) can mostly serve the request from the page cache. For smaller read sizes (4GB) the read throughput is comparable for both CNTRFS and `ext4` filesystems because the data fits in the page cache. However, a larger workload (8GB) no longer fits into the page cache of CNTRFS and degrades the throughput significantly.

Postmark mailserver benchmark. `Postmark` simulates a mail server that randomly reads, appends, creates or deletes small files. In this benchmark, we observed higher overhead (7.1×) for CNTR. In this case, inode lookups in CNTRFS dominated over the actual I/O because the files were deleted even before they were sync-ed to the disk.

PGBench – PostgreSQL Database Server. `PGBench` is based on the `PostgreSQL` database server. It simulates both read and writes under normal database load. Like `FIO`, CNTRFS was faster in this benchmark also, since `PGBench` flushes the writeback buffer less often.

SQLite benchmark. The `SQlite` benchmark measures the time needed to insert 1000 rows in a SQL table. We observed a reasonable overhead (1.9×) for CNTR, since each insertion is followed by a filesystem `sync`, which means that we cannot make efficient use of our disk cache.

Threaded I/O benchmark. The `Threaded I/O` benchmark separately measures the throughput of multiple concurrent readers and writers to a 64MB file. We observed good performance for reads (1.1×) and even better performance for writes (0.3×). This is due to the fact that the reads can be mostly served from the page cache, and for the writes, our writeback buffer in the kernel holds the data longer than the underlying filesystem.

Linux Tarball workload. The `Linux tarball` workload unpacks the kernel source code tree from a compressed tarball. This workload is similar to the `create` stage of the `compilebench` benchmark. However, since the source is read from a single tarball instead of copying an already unpacked directory, there are fewer lookups performed in CNTRFS. Therefore, we incur relatively lower overhead (1.2×) even though many small files are created in the unpacking process.

5.2.3 Effectiveness of Optimizations

We next evaluate the effectiveness of the proposed optimizations in CNTR (as described in §3.3).

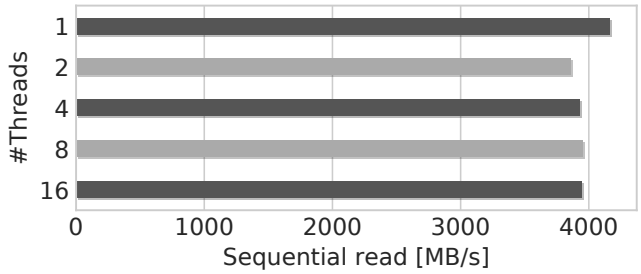

Figure 4: Multithreading optimization with IOZone: Sequential read 500MB/4KB record size with increasing number of CntrFS threads.

Read cache. The goal of this optimization is to allow the kernel to cache pages across multiple processes. Figure 3 (a) shows the effectiveness of the proposed optimization for `FOPEN_KEEP_CACHE`: we observed 10× higher throughput with `FOPEN_KEEP_CACHE` for concurrent reads with 4 threads for the `Threaded I/O` benchmark.

Writeback cache. The writeback optimization was designed to reduce the amount of write requests by maintaining a kernel-based write cache. Figure 3 (b) shows the effectiveness of the optimization: Cntr can achieve 65% more write throughput with the writeback cache enabled compared to the native I/O performance for sequential writes for the `IOZone` benchmark.

Multithreading. We made CntrFS multi-threaded to improve responsiveness when the filesystem operations block. While threads improve the responsiveness, their presence hurts throughput as measured in Figure 4 (up to 8% for sequential read in `IOZone`). However, we still require multithreading to cope with blocking filesystem operations.

Batching. To improve the directory and inode lookups, we batched requests to kernel by specifying the `PARALLEL_DIROPS` flag. We observed a speedup of 2.5× in the `compilebench` read benchmark with this optimization (Figure 3 (c)).

Splice read. Instead of copying memory into userspace, we move the file content with the `splice()` syscall in the kernel to achieve zero-copy I/O. Unfortunately, we did not notice any significant performance improvement with the splice read optimization. For instance, the sequential read throughput in `IOZone` improved slightly by just 5% as shown in Figure 3 (d).

5.3 Effectiveness of Cntr

To evaluate the effectiveness of Cntr's approach to reducing the image sizes, we used a tool called Docker Slim [11].

Docker Slim applies static and dynamic analyses to build a smaller-sized container image that only contains the files that are actually required by the application. Under the hood, Docker Slim records all files that have been accessed during a container run in an efficient way using the `fanotify` kernel module.

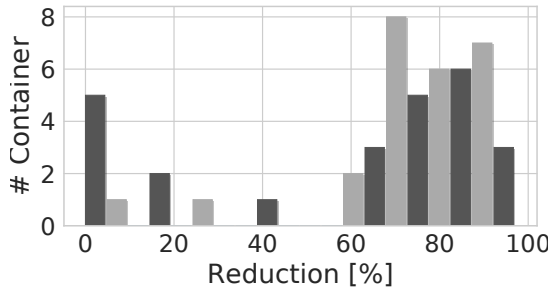

Figure 5: Reduction of container size after applying docker-slim on Top-50 Docker Hub images.

For our analysis, we extended Docker Slim to support container links, which are extensively used for service discovery and it is available as a fork [28].

Dataset: Docker Hub container images. For our analysis, we chose the Top-50 popular official container images hosted on Docker Hub [10]. These images are maintained by Docker and contain commonly used applications such as web servers, databases and web applications. For each image, Docker provides different variants of Linux distributions as the base image. We used the variant set to be `default` as specified by the developer.

Note that Docker Hub also hosts container images that are not meant to be used directly for deploying applications, but they are meant to be used as base images to build applications (such as language SDKs or Linux distributions). Since Cntr targets concrete containerized applications, we did not include such base images in our evaluation.

Methodology. For our analysis, we instrumented the Docker container with Docker Slim and manually ran the application so it would load all the required files. Thereafter, we build new smaller containers using Docker Slim. These new smaller images are equivalent to containers that developers could have created when having access to Cntr. We envision the developers will be using a combination of Cntr and tools such as Docker Slim to create smaller container images. Lastly, we tested to validate that the smaller containers still provide the same functionality.

Experimental results. On average, we could reduce the size by 66.6% for the Top-50 Docker images. Figure 5 depicts the histogram plot showcasing percentage of container size that could be removed in this process. For over 75% of all containers, the reduction in size was between 60% and 97%. Beside the applications, these containers are packaged with common used command line auxiliary tools, such as coreutils, shells, and package managers. For only 6 out of 50 (12%) containers, the reduction was below 10%. We inspected these 6 images and found out they contain only single executables written in Go and a few configuration files.

6 Related Work

In this section, we survey the related work in the space of lightweight virtualization.

Lambda functions. Since the introduction of AWS Lambda [1], all major cloud computing providers offer serverless computing, including Google Cloud Functions [15], Microsoft Azure Functions [5], IBM OpenWhisk [20]. Moreover, there exists a research implementation called Open Lambda [55]. In particular, serverless computing offers a small language runtime rather than the full-blown container image. Unfortunately, lambdas offer limited or no support for interactive debugging or profiling purposes [63] because the clients have no access to the lambda's container or container-management system. In contrast, the goal of the CNTR is to aim for lightweight containers, in the same spirit of lambda functions, but to also provide an on-demand mechanism for auxiliary tools for debugging, profiling, etc. As a future work, we plan to support auxiliary tools for lambda functions [43] using CNTR.

Microkernels. The microkernel architecture [54, 46, 56] shares a lot of commonalities with the CNTR architecture, where the applications/services are horizontally partitioned and the communication happens via the interprocess communication (IPC) mechanism. In CNTR, the application container obtains additional service by communicating with the "fat" container via IPC using CNTRFS.

Containers. Recently, there has been a lot of interest in reducing the size of containers, but still allowing access to the rich set of auxiliary tools. For instance, Toolbox [35] in CoreOS [7] allows to bind the mount of the host filesystem in a container to administrate or debug the host system with installing the tools inside the container. In contrast to Toolbox, CNTR allows bidirectional access with the debug container. Likewise, `nsenter` [27] allows entering into existing container namespaces, and spawning a process into a new set of namespaces. However, `nsenter` only covers namespaces, and it does not provide the rich set of filesystem APIs as provided by CNTR. Lastly, Slacker [53] proposed an opportunistic model to pull images from registries to reduce the startup times. In particular, Slacker can skip downloading files that are never requested by the filesystem. Interestingly, one could also use Slacker to add auxiliary tools such as gdb to the container in an "on-demand" fashion. However, Slacker could support additional auxiliary tools to a container, but these tools would be only downloaded to the container host, if the container is started by the user. Furthermore, Slacker also has a longer build time and greater storage requirements in the registry. In contrast, CNTR offers a generic lightweight model for the additional auxiliary tools.

Virtual machines. Virtual machines [25, 47, 51] provide stronger isolation compared to containers by running applications and the OS as a single unit. On the downside, full-fledged VMs are not scalable and resource-efficient [62]. To strike a balance between the advantages of containers and virtual machines, Intel Clear Containers (or Kata Containers) [21] and SCONE [45] offer stronger security properties for containers by leveraging Intel VT and Intel SGX, respectively. Likewise, LightVM [59] uses unikernel and optimized Xen to offer lightweight VMs. In a similar vein, CNTR allows creating lightweight containers, which are extensively used in the data center environment.

Unikernels and Library OSes. Unikernels [57, 58] leverage library OSes [61, 49, 65, 48] to selectively include only those OS components required to make an application work in a single address space. Unikernels use a fraction of the resources required compared to full, multipurpose operating systems. However, Unikernels also face a similar challenge as containers — If Unikernels need additional auxiliary tools, they must be statically linked in the final image as part of the library OS. Moreover, unikernel approach is orthogonal since it targets the kernel overhead, whereas CNTR targets the tools overhead.

7 Conclusion

We presented CNTR, a system for building and deploying lightweight OS containers. CNTR partitions existing containers into two parts: "slim" (application) and "fat" (additional tools). CNTR efficiently enables the application container to dynamically expand with additional tools in an on-demand fashion at runtime. Further, CNTR enables a set of new development workflows with containers:

- When testing the configuration changes, instead of rebuilding containers from scratch, the developers can use their favorite editor to edit files in place and reload the service.
- Debugging tools no longer have to be manually installed in the application container, but can be placed in separate debug images for debugging or profiling in production.

To the best of our knowledge, CNTR is the first generic and complete system that allows attaching to container and inheriting all its sandbox properties. We have used CNTR to debug existing container engines [32]. In our evaluation, we have extensively tested the completeness, performance, and effectiveness properties of CNTR. We plan to further extend our evaluation to include the nested container design.

Software availability. We have made CNTR along with the complete experimental setup publicly available [6].

Acknowledgments. We thank our shepherd Swaminathan Sundararaman and the anonymous reviewers for their helpful comments. The work is supported in part by the Alan Turing Institute and an Amazon Web Services Education Grant.

References

[1] Amazon AWS Lambdas. https://aws.amazon.com/lambda/.

[2] Amazon Elastic Container Service (ECS). https://aws.amazon.com/ecs/.

[3] Amazon's documentation on EBS volume types. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EBSVolumeTypes.html.

[4] Azure Container Service (AKS). https://azure.microsoft.com/en-gb/services/container-service/.

[5] Azure Functions. https://azure.microsoft.com/en-gb/services/functions/.

[6] Cntr homepage. https://github.com/Mic92/cntr.

[7] Container optimized Linux distribution. https://coreos.com/.

[8] CoreOS. https://coreos.com/.

[9] Docker. https://www.docker.com/.

[10] Docker repositories. https://hub.docker.com/explore/.

[11] Docker Slim. https://github.com/docker-slim/docker-slim.

[12] Docker Swarm. https://www.docker.com/products/docker-swarm.

[13] Docker switch to Alpine Linux. https://news.ycombinator.com/item?id=11000827.

[14] File system regression test on linux implemented for all major filesystems. https://kernel.googlesource.com/pub/scm/fs/ext2/xfstests-bld/+/HEAD/Documentation/what-is-xfstests.md.

[15] Google Cloud Functions. https://cloud.google.com/functions/.

[16] Google Compute Cloud Containers. https://cloud.google.com/compute/docs/containers/.

[17] Google: 'EVERYTHING at Google runs in a container'. https://www.theregister.co.uk/2014/05/23/google_containerization_two_billion/.

[18] Homepage of Phoronix test suite. https://www.phoronix-test-suite.com/.

[19] Homepage of SELinux. https://selinuxproject.org/page/Main_Page.

[20] IBM OpenWhisk. https://www.ibm.com/cloud/functions.

[21] Intel Clear Containers. https://clearlinux.org/containers.

[22] Kubernetes. https://kubernetes.io/.

[23] Lightweight standard libc implementation. https://www.musl-libc.org/.

[24] Linux Containers. https://linuxcontainers.org/.

[25] Linux Kernel Virtual Machine (KVM). https://www.linux-kvm.org/page/Main_Page.

[26] Manual of AppArmor. http://manpages.ubuntu.com/manpages/xenial/en/man7/apparmor.7.html.

[27] nsenter. https://github.com/jpetazzo/nsenter.

[28] Our fork of Docker Slim used for evaluation. https://github.com/Mic92/docker-slim/tree/cntr-eval.

[29] Our fork the nix rust library. https://github.com/Mic92/cntr-nix.

[30] RancherOS. https://rancher.com/rancher-os/.

[31] Raw benchmark report generated by phoronix test suite. https://openbenchmarking.org/result/1802024-AL-CNTREVALU05.

[32] Root cause analysis in unprivileged nspawn container with cntr. https://github.com/systemd/systemd/issues/6244#issuecomment-356029742.

[33] Rust library for filesystems in userspace. https://github.com/zargony/rust-fuse.

[34] Rust library that wraps around the Linux/Posix API. https://github.com/nix-rust/nix.

[35] Toolbox. https://github.com/coreos/toolbox.

[36] Twelve-Factor App. https://12factor.net/config.

[37] Website of the lxc container engine. https://linuxcontainers.org/.

[38] Website of the rkt container engine. `https://coreos.com/rkt/`.

[39] Wiki page for the Phoronix disk test suite. `https://openbenchmarking.org/suite/pts/disk`.

[40] *namespaces(7) Linux User's Manual*, July 2016.

[41] *cgroups(7) Linux User's Manual*, September 2017.

[42] *mount(8) Linux User's Manual*, September 2017.

[43] 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.

[44] A. Anwar, M. Mohamed, V. Tarasov, M. Littley, L. Rupprecht, Y. Cheng, N. Zhao, D. Skourtis, A. S. Warke, H. Ludwig, D. Hildebrand, and A. R. Butt. Improving Docker Registry Design based on Production Workload Analysis. In *16th USENIX Conference on File and Storage Technologies (FAST)*, 2018.

[45] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. SCONE: Secure linux containers with intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[46] N. Asmussen, M. Völp, B. Nöthen, H. Härtig, and G. Fettweis. M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[47] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[48] A. Baumann, D. Lee, P. Fonseca, L. Glendenning, J. R. Lorch, B. Bond, R. Olinsky, and G. C. Hunt. Composing OS Extensions Safely and Efficiently with Bascule. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*, 2013.

[49] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe User-level Access to Privileged CPU Features. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[50] P. Bhatotia, A. Wieder, R. Rodrigues, F. Junqueira, and B. Reed. Reliable Data-center Scale Computations. In *Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware (LADIS)*, 2010.

[51] R. J. Creasy. The Origin of the VM/370 Time-sharing System. *IBM J. Res. Dev.*, 1981.

[52] L. Du, R. Y. Tianyu Wo, and C. Hu. Cider: A Rapid Docker Container Deployment System through Sharing Network Storage. In *Proceedings of the 19th International Conference on High Performance Computing and Communications (HPCC)*, 2017.

[53] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Slacker: Fast Distribution with Lazy Docker Containers. In *14th USENIX Conference on File and Storage Technologies (FAST)*, 2016.

[54] G. Heiser and K. Elphinstone. L4 Microkernels: The Lessons from 20 Years of Research and Deployment. *ACM Transaction of Computer Systems (TOCS)*, 2016.

[55] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Serverless Computation with OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2016.

[56] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP)*, 2009.

[57] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[58] A. Madhavapeddy and D. J. Scott. Unikernels: The Rise of the Virtual Library Operating System. *Communication of ACM (CACM)*, 2014.

[59] 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 (SOSP)*, 2017.

[60] D. P. Quigley, J. Sipek, C. P. Wright, and E. Zadok. UnionFS: User- and Community-oriented Development of a Unification Filesystem. In *Proceedings of the 2006 Linux Symposium (OLS)*, 2006.

[61] D. Schatzberg, J. Cadden, H. Dong, O. Krieger, and J. Appavoo. EbbRT: A Framework for Building Per-Application Library Operating Systems. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[62] P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay. Containers and Virtual Machines at Scale: A Comparative Study. In *Proceedings of the 17th International Middleware Conference (Middleware)*, 2016.

[63] J. Thalheim, P. Bhatotia, and C. Fetzer. INSPECTOR: Data Provenance Using Intel Processor Trace (PT). In *IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, 2016.

[64] J. Thalheim, A. Rodrigues, I. E. Akkus, R. C. Pramod Bhatotia, B. Viswanath, L. Jiao, and C. Fetzer. Sieve: Actionable Insights from Monitored Metrics in Distributed Systems. In *Proceedings of Middleware Conference (Middleware)*, 2017.

[65] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter. Cooperation and Security Isolation of Library OSes for Multi-process Applications. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)*, 2014.

[66] B. K. R. Vangoor, V. Tarasov, and E. Zadok. To FUSE or Not to FUSE: Performance of User-Space File Systems. In *15th USENIX Conference on File and Storage Technologies (FAST)*, 2017.