Slacker: Fast Distribution with Lazy Docker Containers

Tyler Harter, Brandon Salmon[†], Rose Liu[†],
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

University of Wisconsin, Madison   [†] Tintri

Abstract

Containerized applications are becoming increasingly popular, but unfortunately, current container-deployment methods are very slow. We develop a new container benchmark, HelloBench, to evaluate the startup times of 57 different containerized applications. We use HelloBench to analyze workloads in detail, studying the block I/O patterns exhibited during startup and compressibility of container images. Our analysis shows that pulling packages accounts for 76% of container start time, but only 6.4% of that data is read. We use this and other findings to guide the design of Slacker, a new Docker storage driver optimized for fast container startup. Slacker is based on centralized storage that is shared between all Docker workers and registries. Workers quickly provision container storage using backend clones and minimize startup latency by lazily fetching container data. Slacker speeds up the median container development cycle by 20× and deployment cycle by 5×.

1 Introduction

Isolation is a highly desirable property in cloud computing and other multi-tenant platforms [8, 14, 27, 22, 24, 34, 38, 40, 42, 49]. Without isolation, users (who are often paying customers) must tolerate unpredictable performance, crashes, and privacy violations.

Hypervisors, or virtual machine monitors (VMMs), have traditionally been used to provide isolation for applications [12, 14, 43]. Each application is deployed in its own virtual machine, with its own environment and resources. Unfortunately, hypervisors need to interpose on various privileged operations (*e.g.*, page-table lookups [7, 12]) and use roundabout techniques to infer resource usage (*e.g.*, ballooning [43]). The result is that hypervisors are heavyweight, with slow boot times [50] as well as run-time overheads [7, 12].

Containers, as driven by the popularity of Docker [25], have recently emerged as a lightweight alternative to hypervisor-based virtualization. Within a container, all process resources are virtualized by the operating system, including network ports and file-system mount points. Containers are essentially just processes that enjoy virtualization of all resources, not just CPU and memory; as such, there is no intrinsic reason container startup should be slower than normal process startup.

Unfortunately, as we will show, starting containers is much slower in practice due to file-system provisioning bottlenecks. Whereas initialization of network, compute, and memory resources is relatively fast and simple (*e.g.*, zeroing memory pages), a containerized application requires a fully initialized file system, containing application binaries, a complete Linux distribution, and package dependencies. Deploying a container in a Docker or Google Borg [41] cluster typically involves significant copying and installation overheads. A recent study of Google Borg revealed: *"[task startup latency] is highly variable, with the median typically about 25 s. Package installation takes about 80% of the total: one of the known bottlenecks is contention for the local disk where packages are written"* [41].

If startup time can be improved, a number of opportunities arise: applications can scale instantly to handle flash-crowd events [13], cluster schedulers can frequently rebalance nodes at low cost [17, 41], software upgrades can be rapidly deployed when a security flaw or critical bug is fixed [30], and developers can interactively build and test distributed applications [31].

We take a two-pronged approach to solving the container-startup problem. First, we develop a new open-source Docker benchmark, HelloBench, that carefully exercises container startup. HelloBench is based on 57 different container workloads and measures the time from when deployment begins until a container is ready to start doing useful work (*e.g.*, servicing web requests). We use HelloBench and static analysis to characterize Docker images and I/O patterns. Among other findings, our analysis shows that (1) copying package data accounts for 76% of container startup time, (2) only 6.4% of the copied data is actually needed for containers to begin useful work, and (3) simple block-deduplication across images achieves better compression rates than gzip compression of individual images.

Second, we use our findings to build Slacker, a new Docker storage driver that achieves fast container distribution by utilizing specialized storage-system support at multiple layers of the stack. Specifically, Slacker uses the snapshot and clone capabilities of our backend storage server (a Tintri VMstore [6]) to dramatically reduce the cost of common Docker operations. Rather than prepropagate whole container images, Slacker lazily pulls image data as necessary, drastically reducing network I/O. Slacker also utilizes modifications we make to the Linux kernel in order to improve cache sharing.

The result of using these techniques is a massive improvement in the performance of common Docker operations; image pushes become 153× faster and pulls become 72× faster. Common Docker use cases involving these operations greatly benefit. For example, Slacker achieves a 5× median speedup for container deployment cycles and a 20× speedup for development cycles.

We also build MultiMake, a new container-based build tool that showcases the benefits of Slacker's fast startup. MultiMake produces 16 different binaries from the same source code, using different containerized GCC releases. With Slacker, MultiMake experiences a 10× speedup.

The rest of this paper is organized as follows. First, we describe the existing Docker framework (§2). Next, we introduce HelloBench (§3), which we use to analyze Docker workload characteristics (§4). We use these findings to guide our design of Slacker (§5). Finally, we evaluate Slacker (§6), present MultiMake (§7), discuss related work (§8), and conclude (§9).

2 Docker Background

We now describe Docker's framework (§2.1), storage interface (§2.2), and default storage driver (§2.3).

2.1 Version Control for Containers

While Linux has always used virtualization to isolate memory, cgroups [37] (Linux's container implementation) virtualizes a broader range of resources by providing six new namespaces, for file-system mount points, IPC queues, networking, host names, process IDs, and user IDs [19]. Linux cgroups were first released in 2007, but widespread container use is a more recent phenomenon, coinciding with the availability of new container management tools such as Docker (released in 2013). With Docker, a single command such as "`docker run -it ubuntu bash`" will pull Ubuntu packages from the Internet, initialize a file system with a fresh Ubuntu installation, perform the necessary cgroup setup, and return an interactive bash session in the environment.

This example command has several parts. First, "ubuntu" is the name of an *image*. Images are read-only copies of file-system data, and typically contain application binaries, a Linux distribution, and other packages needed by the application. Bundling applications in Docker images is convenient because the distributor can select a specific set of packages (and their versions) that will be used wherever the application is run. Second, "run" is an operation to perform on an image; the run operation creates an initialized root file system based on the image to use for a new *container*. Other operations include "push" (for publishing new images) and "pull" (for fetching published images from a central location); an image is automatically pulled if the user attempts to run a non-local image. Third, "bash" is the program to start within the container; the user may specify any executable in the given image.

Docker manages image data much the same way traditional version-control systems manage code. This model is suitable for two reasons. First, there may be different branches of the same image (*e.g.*, "ubuntu:latest" or "ubuntu:12.04"). Second, images naturally build upon one another. For example, the Ruby-on-Rails image builds on the Rails image, which in turn builds on the Debian image. Each of these images represent a new *commit* over a previous commit; there may be additional commits that are not tagged as runnable images. When a container executes, it starts from a committed image, but files may be modified; in version-control parlance, these modifications are referred to as unstaged changes. The Docker "commit" operation turns a container and its modifications into a new read-only image. In Docker, a *layer* refers to either the data of a commit or to the unstaged changes of a container.

Docker worker machines run a local Docker *daemon*. New containers and images may be created on a specific worker by sending commands to its local daemon. Image sharing is accomplished via centralized *registries* that typically run on machines in the same cluster as the Docker workers. Images may be published with a push from a daemon to a registry, and images may be deployed by executing pulls on a number of daemons in the cluster. Only the layers not already available on the receiving end are transferred. Layers are represented as gzip-compressed tar files over the network and on the registry machines. Representation on daemon machines is determined by a pluggable storage driver.

2.2 Storage Driver Interface

Docker containers access storage in two ways. First, users may mount directories on the host within a container. For example, a user running a containerized compiler may mount her source directory within the container so that the compiler can read the code files and produce binaries in the host directory. Second, containers need access to the Docker layers used to represent the application binaries and libraries. Docker presents a view of this application data via a mount point that the container uses as its root file system. Container storage and mounting is managed by a Docker storage driver; different drivers may choose to represent layer data in different ways. The methods a driver must implement are shown in Table 1 (some uninteresting functions and arguments are not shown). All the functions take a string "id" argument that identifies the layer being manipulated.

The `Get` function requests that the driver mount the layer and return a path to the mount point. The mount point returned should contain a view of not only the "id"

Method	Description
Get(id)=dir	mount "id" layer file system, return mount point
Put(id)	unmount "id" layer file system
Create(parent, id)	logically copy "parent" layer to "id" layer
Diff(parent, id)=tar	return compressed tar of changes in "id" layer
ApplyDiff(id, tar)	apply changes in tar to "id" layer

Table 1: **Docker Driver API.**

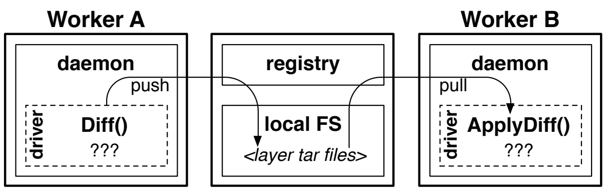

Figure 1: **Diff and ApplyDiff.** *Worker A is using `Diff` to package local layers as compressed tars for a push. B is using `ApplyDiff` to convert the tars back to the local format. Local representation varies depending on the driver, as indicated by the question marks.*

layer, but of all its ancestors (*e.g.*, files in the parent layer of the "id" layer should be seen during a directory walk of the mount point). `Put` unmounts a layer. `Create` copies from a parent layer to create a new layer. If the parent is `NULL`, the new layer should be empty. Docker calls `Create` to (1) provision file systems for new containers, and (2) allocate layers to store data from a pull.

`Diff` and `ApplyDiff` are used during Docker push and pull operations respectively, as shown in Figure 1. When Docker is pushing a layer, `Diff` converts the layer from the local representation to a compressed tar file containing the files of the layer. `ApplyDiff` does the opposite: given a tar file and a local layer it decompresses the tar file over the existing layer.

Figure 2 shows the driver calls that are made when a four-layer image (*e.g.*, ubuntu) is run for the first time. Four layers are created during the image pull; two more are created for the container itself. Layers A-D represent the image. The `Create` for A takes a `NULL` parent, so A is initially empty. The subsequent `ApplyDiff` call, however, tells the driver to add the files from the pulled tar to A. Layers B-D are each populated with two steps: a copy from the parent (via `Create`), and the addition of files from the tar (via `ApplyDiff`). After step 8, the pull is complete, and Docker is ready to create a container. It first creates a read-only layer E-init, to which it adds a few small initialization files, and then it creates E, the file system the container will use as its root.

2.3 AUFS Driver Implementation

The AUFS storage driver is a common default for Docker distributions. This driver is based on the AUFS file system (Another Union File System). Union file systems do not store data directly on disk, but rather use another file system (*e.g.*, ext4) as underlying storage.

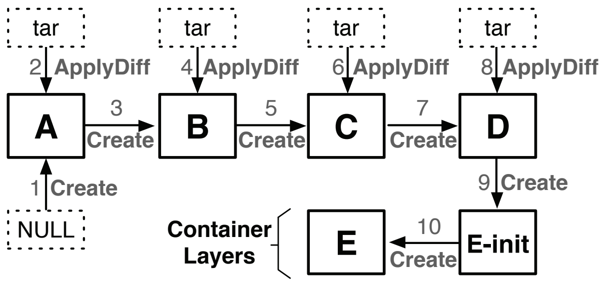

Figure 2: **Cold Run Example.** *The driver calls that are made when a four-layer image is pulled and run are shown. Each arrow represents a call (`Create` or `ApplyDiff`), and the nodes to which an arrow connects indicate arguments to the call. Thick-bordered boxes represent layers. Integers indicate the order in which functions are called.*

A union mount point provides a view of multiple directories in the underlying file system. AUFS is mounted with a list of directory paths in the underlying file system. During path resolution, AUFS iterates through the list of directories; the first directory to contain the path being resolved is chosen, and the inode from that directory is used. AUFS supports special *whiteout* files to make it appear that certain files in lower layers have been deleted; this technique is analogous to deletion markers in other layered systems (*e.g.*, LSM databases [29]). AUFS also supports COW (copy-on-write) at file granularity; upon write, files in lower layers are copied to the top layer before the write is allowed to proceed.

The AUFS driver takes advantage the AUFS file system's layering and copy-on-write capabilities while also accessing the file system underlying AUFS directly. The driver creates a new directory in the underlying file system for each layer it stores. An `ApplyDiff` simple untars the archived files into the layer's directory. Upon a `Get` call, the driver uses AUFS to create a unioned view of a layer and its ancestors. The driver uses AUFS's COW to efficiently copy layer data when `Create` is called. Unfortunately, as we will see, COW at file granularity has some performance problems (§4.3).

3 HelloBench

We present HelloBench, a new benchmark designed to exercise container startup. HelloBench directly executes Docker commands, so pushes, pulls, and runs can be measured independently. The benchmark consists of two parts: (1) a collection of container images and (2) a test harness for executing simple tasks in said containers. The images were the latest available from the Docker Hub library [3] as of June 1, 2015. HelloBench consists of 57 images of the 72 available at the time. We selected images that were runnable with minimal configuration and do not depend on other containers. For example, WordPress is not included because a WordPress container depends on a separate MySQL container.

Linux Distro:	alpine, busybox, centos, cirros, crux, debian, fedora, mageia, opensuse, oraclelinux, ubuntu, ubuntu-debootstrap, ubuntu-upstart
Database:	cassandra, crate, elasticsearch, mariadb, mongo, mysql, percona, postgres, redis, rethinkdb
Language:	clojure, gcc, golang, haskell, hylang, java, jruby, julia, mono, perl, php, pypy, python, r-base, rakudo-star, ruby, thrift
Web Server:	glassfish, httpd, jetty, nginx, php-zendserver, tomcat
Web Framework:	django, iojs, node, rails
Other:	drupal, ghost, hello-world, jenkins, rabbitmq, registry, sonarqube

Table 2: **HelloBench Workloads.** *HelloBench runs 57 different container images pulled from the Docker Hub.*

Table 2 lists the images used by HelloBench. We divide the images into six broad categories as shown. Some classifications are somewhat subjective; for example, the Django image contains a web server, but most would probably consider it a web framework.

The HelloBench harness measures startup time by either running the simplest possible task in the container or waiting until the container reports readiness. For the language containers, the task typically involves compiling or interpreting a simple "hello world" program in the applicable language. The Linux distro images execute a very simple shell command, typically "`echo hello`". For long-running servers (particularly databases and web servers), HelloBench measures the time until the container writes an "up and ready" (or similar) message to standard out. For particularly quiet servers, an exposed port is polled until there is a response.

HelloBench images each consist of many layers, some of which are shared between containers. Figure 3 shows the relationships between layers. Across the 57 images, there are 550 nodes and 19 roots. In some cases, a tagged image serves as a base for other tagged images (*e.g.*, "ruby" is a base for "rails"). Only one image consists of a single layer: "alpine", a particularly lightweight Linux distribution. Application images are often based on non-latest Linux distribution images (*e.g.*, older versions of Debian); that is why multiple images will often share a common base that is not a solid black circle.

In order to evaluate how representative HelloBench is of commonly used images, we counted the number of pulls to every Docker Hub library image [3] on January 15, 2015 (7 months after the original HelloBench images were pulled). During this time, the library grew from 72 to 94 images. Figure 4 shows pulls to the 94 images, broken down by HelloBench category. HelloBench is representative of popular images, accounting for 86% of all pulls. Most pulls are to Linux distribution bases (*e.g.*, BusyBox and Ubuntu). Databases (*e.g.*, Redis and MySQL) and web servers (*e.g.*, nginx) are also popular.

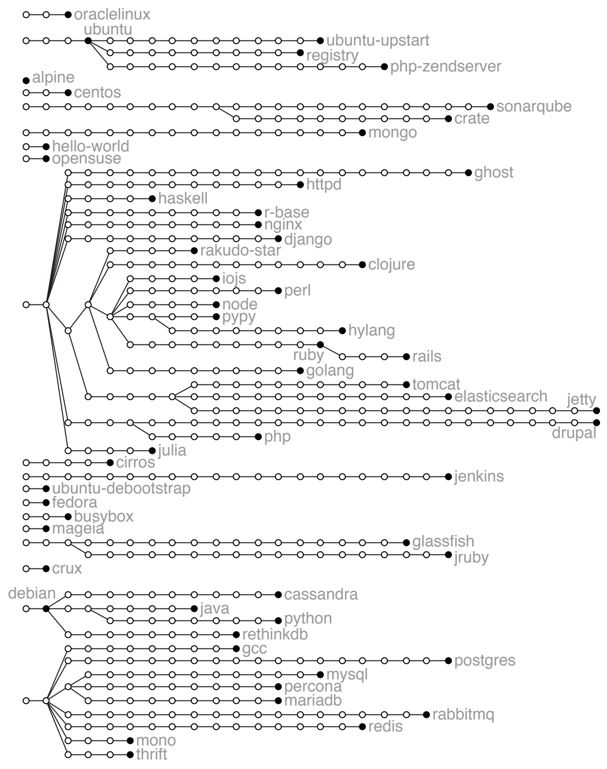

Figure 3: **HelloBench Hierarchy.** *Each circle represents a layer. Filled circles represent layers tagged as runnable images. Deeper layers are to the left.*

4 Workload Analysis

In this section, we analyze the behavior and performance of the HelloBench workloads, asking four questions: how large are the container images, and how much of that data is necessary for execution (§4.1)? How long does it take to push, pull, and run the images (§4.2)? How is image data distributed across layers, and what are the performance implications (§4.3)? And how similar are access patterns across different runs (§4.4)?

All performance measurements are taken from a virtual machine running on an PowerEdge R720 host with 2 GHz Xeon CPUs (E5-2620). The VM is provided 8 GB of RAM, 4 CPU cores, and a virtual disk backed by a Tintri T620 [1]. The server and VMstore had no other load during the experiments.

4.1 Image Data

We begin our analysis by studying the HelloBench images pulled from the Docker Hub. For each image, we take three measurements: its compressed size, uncompressed size, and the number of bytes read from the image when HelloBench executes. We measure reads by running the workloads over a block device traced with `blktrace` [11]. Figure 5 shows a CDF of these three numbers. We observe that only 20 MB of data is read on median, but the median image is 117 MB compressed and 329 MB uncompressed.

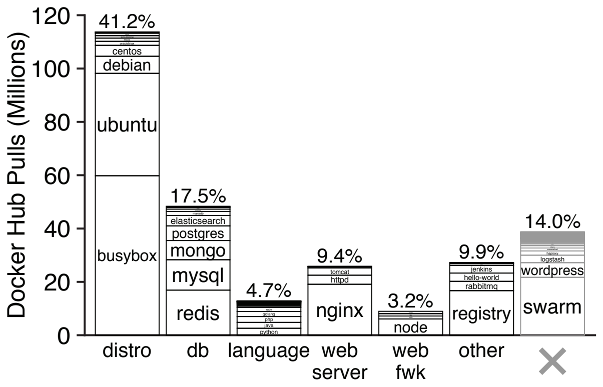

Figure 4: **Docker Hub Pulls.** *Each bar represents the number of pulls to the Docker Hub library, broken down by category and image. The far-right gray bar represents pulls to images in the library that are not run by HelloBench.*

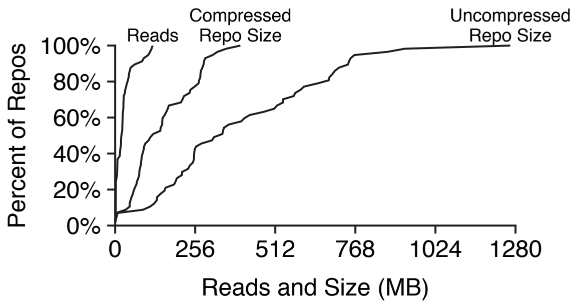

Figure 5: **Data Sizes (CDF).** *Distributions are shown for the number of reads in the HelloBench workloads and for the uncompressed and compressed sizes of the HelloBench images.*

We break down the read and size numbers by category in Figure 6. The largest relative waste is for distro workloads (30× and 85× for compressed and uncompressed respectively), but the absolute waste is also smallest for this category. Absolute waste is highest for the language and web framework categories. Across all images, only 27 MB is read on average; the average uncompressed image is 15× larger, indicating only 6.4% of image data is needed for container startup.

Although Docker images are much smaller when compressed as gzip archives, this format is not suitable for running containers that need to modify data. Thus, workers typically store data uncompressed, which means that compression reduces network I/O but not disk I/O. Deduplication is a simple alternative to compression that is suitable for updates. We scan HelloBench images for redundancy between blocks of files to compute the effectiveness of deduplication. Figure 7 compares gzip compression rates to deduplication, at both file and block (4 KB) granularity. Bars represent rates over single images. Whereas gzip achieves rates between 2.3 and 2.7, deduplication does poorly on a per-image basis. Deduplication across all images, however, yields rates of 2.6 (file granularity) and 2.8 (block granularity).

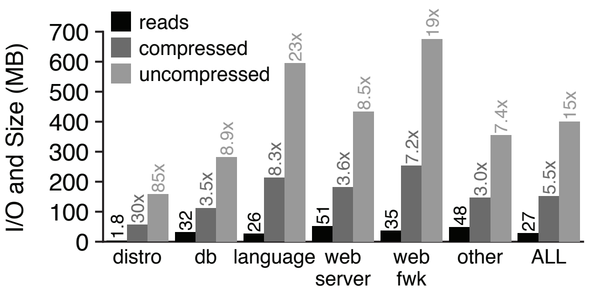

Figure 6: **Data Sizes (By Category).** *Averages are shown for each category. The size bars are labeled with amplification factors, indicating the amount of transferred data relative to the amount of useful data (i.e., the data read).*

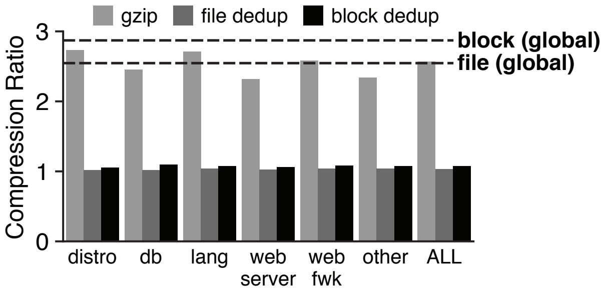

Figure 7: **Compression and Deduplication Rates.** *The y-axis represents the ratio of the size of the raw data to the size of the compressed or deduplicated data. The bars represent per-image rates. The lines represent rates of global deduplication across the set of all images.*

Implications: the amount of data read during execution is much smaller than the total image size, either compressed or uncompressed. Image data is sent over the network compressed, then read and written to local storage uncompressed, so overheads are high for both network and disk. One way to decrease overheads would be to build leaner images with fewer installed packages. Alternatively, image data could be lazily pulled as a container needs it. We also saw that global block-based deduplication is an efficient way to represent image data, even compared to gzip compression.

4.2 Operation Performance

Once built, containerized applications are often deployed as follows: the developer *pushes* the application image once to a central registry, a number of workers *pull* the image, and each worker *runs* the application. We measure the latency of these operations with HelloBench, reporting CDFs in Figure 8. Median times for push, pull, and run are 61, 16, and 0.97 seconds respectively.

Figure 9 breaks down operation times by workload category. The pattern holds in general: runs are fast while pushes and pulls are slow. Runs are fastest for the distro and language categories (0.36 and 1.9 seconds re-

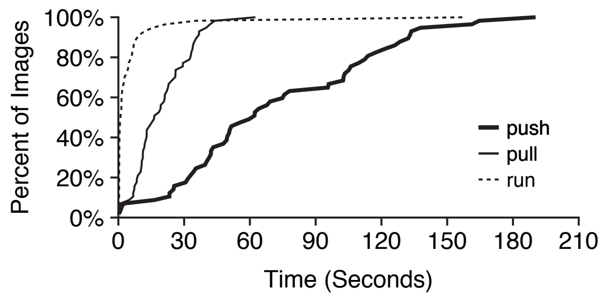

Figure 8: **Operation Performance (CDF).** *A distribution of push, pull, and run times for HelloBench are shown for Docker with the AUFS storage driver.*

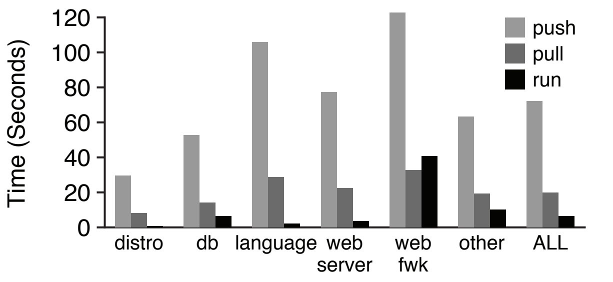

Figure 9: **Operation Performance (By Category).** *Averages are shown for each category.*

spectively). The average times for push, pull, and run are 72, 20, and 6.1 seconds respectively. Thus, 76% of startup time will be spent on pull when starting a new image hosted on a remote registry.

As pushes and pulls are slowest, we want to know whether these operations are merely high latency, or whether they are also costly in a way that limits throughput even if multiple operations run concurrently. To study scalability, we concurrently push and pull varying numbers of artificial images of varying sizes. Each image contains a single randomly generated file. We use artificial images rather than HelloBench images in order to create different equally-sized images. Figure 10 shows that the total time scales roughly linearly with the number of images and image size. Thus, pushes and pulls are not only high-latency, they consume network and disk resources, limiting scalability.

Implications: container startup time is dominated by pulls; 76% of the time spent on a new deployment will be spent on the pull. Publishing images with push will be painfully slow for programmers who are iteratively developing their application, though this is likely a less frequent case than multi-deployment of an already published image. Most push work is done by the storage driver's `Diff` function, and most pull work is done by the `ApplyDiff` function (§2.2). Optimizing these driver functions would improve distribution performance.

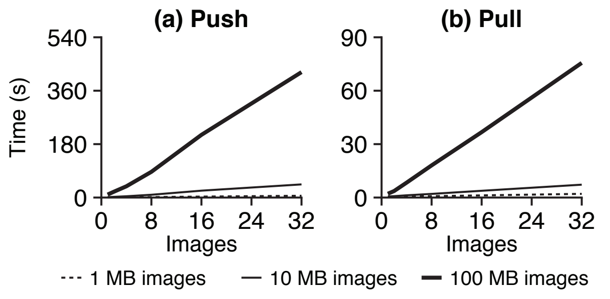

Figure 10: **Operation Scalability.** *A varying number of artificial images (x-axis), each containing a random file of a given size, are pushed or pulled simultaneously. The time until all operations are complete is reported (y-axis).*

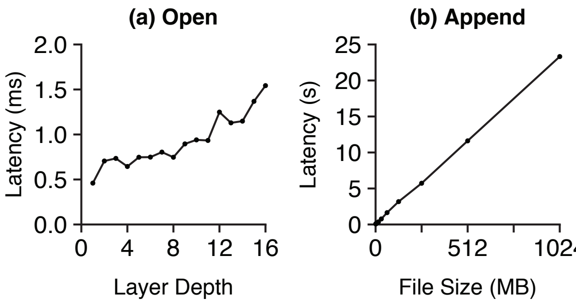

Figure 11: **AUFS Performance.** *Left: the latency of the `open` system call is shown as a function of the layer depth of the file. Right: the latency of a one-byte append is shown as a function of the size of the file that receives the write.*

4.3 Layers

Image data is typically split across a number of layers. The AUFS driver composes the layers of an image at runtime to provide a container a complete view of the file system. In this section, we study the performance implications of layering and the distribution of data across layers. We start by looking at two performance problems (Figure 11) to which layered file systems are prone: lookups to deep layers and small writes to non-top layers.

First, we create (and compose with AUFS) 16 layers, each containing 1K empty files. Then, with a cold cache, we randomly open 10 files from each layer, measuring the open latency. Figure 11a shows the result (an average over 100 runs): there is a strong correlation between layer depth and latency. Second, we create two layers, the bottom of which contains large files of varying sizes. We measure the latency of appending one byte to a file stored in the bottom layer. As shown by Figure 11b, the latency of small writes correspond to the file size (not the write size), as AUFS does COW at file granularity. Before a file is modified, it is copied to the topmost layer, so writing one byte can take over 20 seconds. Fortunately, small writes to lower layers induce a one-time cost per container; subsequent writes will be faster because the large file will have been copied to the top layer.

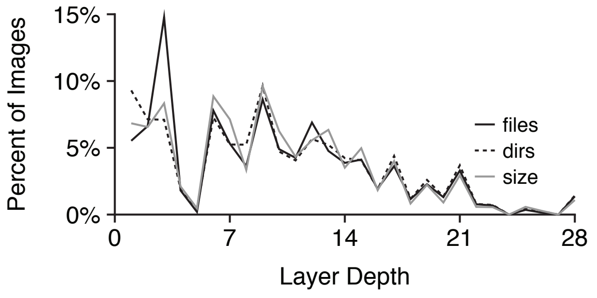

Figure 12: **Data Depth.** *The lines show mass distribution of data across image layers in terms of number of files, number of directories, and bytes of data in files.*

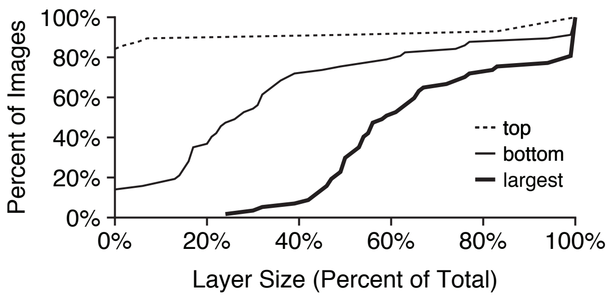

Figure 13: **Layer Size (CDF).** *The size of a given layer is measured relative to the total image size (x-axis), and the distribution of relative sizes is shown. The plot considers the topmost layer, bottommost layer, and whichever layer happens to be largest. All measurements are in terms of file bytes.*

Having considered how layer depth corresponds with performance, we now ask, *how deep is data typically stored for the HelloBench images?* Figure 12 shows the percentage of total data (in terms of number of files, number of directories, and size in bytes) at each depth level. The three metrics roughly correspond. Some data is as deep as level 28, but mass is more concentrated to the left. Over half the bytes are at depth of at least nine.

We now consider the variance in how data is distributed across layers, measuring, for each image, what portion (in terms of bytes) is stored in the topmost layer, bottommost layer, and whatever layer is largest. Figure 13 shows the distribution: for 79% of images, the topmost layer contains 0% of the image data. In contrast, 27% of the data resides in the bottommost layer in the median case. A majority of the data typically resides in a single layer.

Implications: for layered file systems, data stored in deeper layers is slower to access. Unfortunately, Docker images tend to be deep, with at least half of file data at depth nine or greater. Flattening layers is one technique to avoid these performance problems; however, flattening could potentially require additional copying and void the other COW benefits that layered file systems provide.

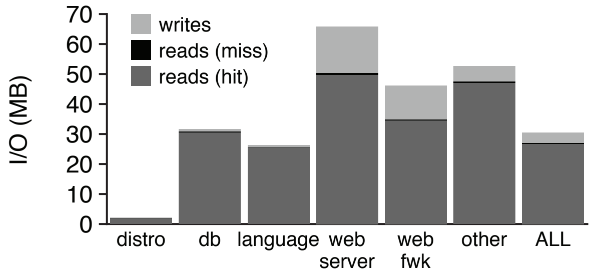

Figure 14: **Repeated I/O.** *The bars represent total I/O done for the average container workload in each category. Bar sections indicate read/write ratios. Reads that could have potentially been serviced by a cache populated by previous container execution are dark gray.*

4.4 Caching

We now consider the case where the same worker runs the same image more than once. In particular, we want to know whether I/O from the first execution can be used to prepopulate a cache to avoid I/O on subsequent runs. Towards this end, we run every HelloBench workload twice consecutively, collecting block traces each time. We compute the portion of reads during the second run that could potentially benefit from cache state populated by reads during the first run.

Figure 14 shows the reads and writes for the second run. Reads are broken into hits and misses. For a given block, only the first read is counted (we want to study the workload itself, not the characteristics of the specific cache beneath which we collected the traces). Across all workloads, the read/write ratio is 88/12. For distro, database, and language workloads, the workload consists almost completely of reads. Of the reads, 99% could potentially be serviced by cached data from previous runs.

Implications: The same data is often read during different runs of the same image, suggesting cache sharing will be useful when the same image is executed on the same machine many times. In large clusters with many containerized applications, repeated executions will be unlikely unless container placement is highly restricted. Also, other goals (*e.g.*, load balancing and fault isolation) may make colocation uncommon. However, repeated executions are likely common for containerized utility programs (*e.g.*, python or gcc) and for applications running in small clusters. Our results suggest these latter scenarios would benefit from cache sharing.

5 Slacker

In this section, we describe Slacker, a new Docker storage driver. Our design is based on our analysis of container workloads and five goals: (1) make pushes and pulls very fast, (2) introduce no slowdown for long-running containers, (3) reuse existing storage systems whenever possible, (4) utilize the powerful primitives

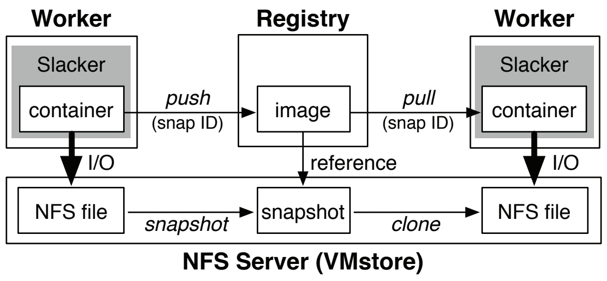

Figure 15: **Slacker Architecture.** *Most of our work was in the gray boxes, the Slacker storage plugin. Workers and registries represent containers and images as files and snapshots respectively on a shared Tintri VMstore server.*

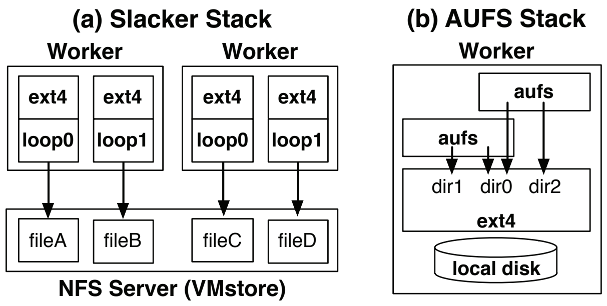

Figure 16: **Driver Stacks.** *Slacker uses one ext4 file system per container. AUFS containers share one ext4 instance.*

provided by a modern storage server, and (5) make no changes to the Docker registry or daemon except in the storage-driver plugin (§2.2).

Figure 15 illustrates the architecture of a Docker cluster running Slacker. The design is based on centralized NFS storage, shared between all Docker daemons and registries. Most of the data in a container is not needed to execute the container, so Docker workers only fetch data lazily from shared storage as needed. For NFS storage, we use a Tintri VMstore server [6]. Docker images are represented by VMstore's read-only snapshots. Registries are no longer used as hosts for layer data, and are instead used only as name servers that associate image metadata with corresponding snapshots. Pushes and pulls no longer involve large network transfers; instead, these operations simply share snapshot IDs. Slacker uses VMstore `snapshot` to convert a container into a shareable image and `clone` to provision container storage based on a snapshot ID pulled from the registry. Internally, VMstore uses block-level COW to implement `snapshot` and `clone` efficiently.

Slacker's design is based on our analysis of container workloads; in particular, the following four design subsections (§5.1 to §5.4) correspond to the previous four analysis subsections (§4.1 to §4.4). We conclude by discussing possible modifications to the Docker framework itself that would provide better support for nontraditional storage drivers such as Slacker (§5.5).

5.1 Storage Layers

Our analysis revealed that only 6.4% of the data transferred by a pull is actually needed before a container can begin useful work (§4.1). In order to avoid wasting I/O on unused data, Slacker stores all container data on an NFS server (a Tintri VMstore) shared by all workers; workers lazily fetch only the data that is needed. Figure 16a illustrates the design: storage for each container is represented as a single NFS file. Linux loopbacks (§5.4) are used to treat each NFS file as a virtual block device, which can be mounted and unmounted as a root file system for a running container. Slacker formats each NFS file as an ext4 file system.

Figure 16b compares the Slacker stack with the AUFS stack. Although both use ext4 (or some other local file system) as a key layer, there are three important differences. First, ext4 is backed by a network disk in Slacker, but by a local disk with AUFS. Thus, Slacker can lazily fetch data over the network, while AUFS must copy all data to the local disk before container startup.

Second, AUFS does COW above ext4 at the file level and is thus susceptible to the performance problems faced by layered file systems (§4.3). In contrast, Slacker layers are effectively flattened at the file level. However, Slacker still benefits from COW by utilizing block-level COW implemented within VMstore (§5.2). Furthermore, VMstore deduplicates identical blocks internally, providing further space savings between containers running on different Docker workers.

Third, AUFS uses different directories of a single ext4 instance as storage for containers, whereas Slacker backs each container by a different ext4 instance. This difference presents an interesting tradeoff because each ext4 instance has its own journal. With AUFS, all containers will share the same journal, providing greater efficiency. However, journal sharing is known to cause priority inversion that undermines QoS guarantees [48], an important feature of multi-tenant platforms such as Docker. Internal fragmentation [10, Ch. 17] is another potential problem when NFS storage is divided into many small, non-full ext4 instances. Fortunately, VMstore files are sparse, so Slacker does not suffer from this issue.

5.2 VMstore Integration

Earlier, we found that Docker pushes and pulls are quite slow compared to runs (§4.2). Runs are fast because storage for a new container is initialized from an image using the COW functionality provided by AUFS. In contrast, push and pull are slow with traditional drivers because they require copying large layers between different machines, so AUFS's COW functionality is not usable. Unlike other Docker drivers, Slacker is built on shared storage, so it is conceptually possible to do COW sharing between daemons and registries.

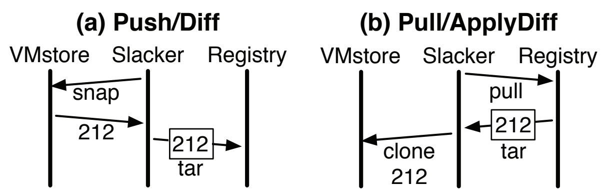

Figure 17: **Push/Pull Timelines.** *Slacker implements `Diff` and `ApplyDiff` with `snapshot` and `clone` operations.*

Fortunately, VMstore extends its basic NFS interface with an auxiliary REST-based API that, among other things, includes two related COW functions, `snapshot` and `clone`. The `snapshot` call creates a read-only snapshot of an NFS file, and `clone` creates an NFS file from a snapshot. Snapshots do not appear in the NFS namespace, but do have unique IDs. File-level snapshot and clone are powerful primitives that have been used to build more efficient journaling, deduplication, and other common storage operations [46]. In Slacker, we use `snapshot` and `clone` to implement `Diff` and `ApplyDiff` respectively. These driver functions are respectively called by Docker push and pull operations (§2.2).

Figure 17a shows how a daemon running Slacker interacts with a VMstore and Docker registry upon push. Slacker asks VMstore to create a snapshot of the NFS file that represents the layer. VMstore takes the snapshot, and returns a snapshot ID (about 50 bytes), in this case "212". Slacker embeds the ID in a compressed tar file and sends it to the registry. Slacker embeds the ID in a tar for backwards compatibility: an unmodified registry expects to receive a tar file. A pull, shown in Figure 17b, is essentially the inverse. Slacker receives a snapshot ID from the registry, from which it can clone NFS files for container storage. Slacker's implementation is fast because (a) layer data is never compressed or uncompressed, and (b) layer data never leaves the VMstore, so only metadata is sent over the network.

The names "`Diff`" and "`ApplyDiff`" are slight misnomers given Slacker's implementation. In particular, `Diff(A, B)` is supposed to return a delta from which another daemon, which already has A, could reconstruct B. With Slacker, layers are effectively flattened at the namespace level. Thus, instead of returning a delta, `Diff(A, B)` returns a reference from which another worker could obtain a clone of B, with or without A.

Slacker is partially compatible with other daemons running non-Slacker drivers. When Slacker pulls a tar, it peeks at the first few bytes of the streamed tar before processing it. If the tar contains layer files (instead of an embedded snapshot), Slacker falls back to simply decompressing instead cloning. Thus, Slacker can pull images that were pushed by other drivers, albeit slowly. Other drivers, however, will not be able to pull Slacker images, because they will not know how to process the snapshot ID embedded in the tar file.

5.3 Optimizing Snapshot and Clone

Images often consist of many layers, with over half the HelloBench data being at a depth of at least nine (§4.3). Block-level COW has inherent performance advantages over file-level COW for such data, as traversing block-mapping indices (which may be flattened) is simpler than iterating over the directories of an underlying file system.

However, deeply-layered images still pose a challenge for Slacker. As discussed (§5.2), Slacker layers are flattened, so mounting any one layer will provide a complete view of a file system that could be used by a container. Unfortunately, the Docker framework has no notion of flattened layers. When Docker pulls an image, it fetches all the layers, passing each to the driver with `ApplyDiff`. For Slacker, the topmost layer alone is sufficient. For 28-layer images (*e.g.*, jetty), the extra clones are costly.

One of our goals was to work within the existing Docker framework, so instead of modifying the framework to eliminate the unnecessary driver calls, we optimize them with *lazy cloning*. We found that the primary cost of a pull is not the network transfer of the snapshot tar files, but the VMstore clone. Although clones take a fraction of a second, performing 28 of them negatively impacts latency. Thus, instead of representing every layer as an NFS file, Slacker (when possible) represents them with a piece of local metadata that records a snapshot ID. `ApplyDiff` simply sets this metadata instead of immediately cloning. If at some point Docker calls `Get` on that layer, Slacker will at that point perform a real `clone` before the mount.

We also use the snapshot-ID metadata for *snapshot caching*. In particular, Slacker implements `Create`, which makes a logical copy of a layer (§2.2) with a snapshot immediately followed by a clone (§5.2). If many containers are created from the same image, `Create` will be called many times on the same layer. Instead of doing a snapshot for each `Create`, Slacker only does it the first time, reusing the snapshot ID subsequent times. The snapshot cache for a layer is invalidated if the layer is mounted (once mounted, the layer could change, making the snapshot outdated).

The combination of snapshot caching and lazy cloning can make `Create` very efficient. In particular, copying from a layer A to layer B may only involve copying from A's snapshot cache entry to B's snapshot cache entry, with no special calls to VMstore. In Figure 2 from the background section (§2.2), we showed the 10 `Create` and `ApplyDiff` calls that occur for the pull and run of a simple four-layer image. Without lazy caching and snapshot caching, Slacker would need to perform 6 snapshots (one for each `Create`) and 10 clones (one for each `Create` or `ApplyDiff`). With our optimizations, Slacker only needs to do one snapshot and two clones. In step 9, `Create` does a lazy clone, but Docker calls Get on the

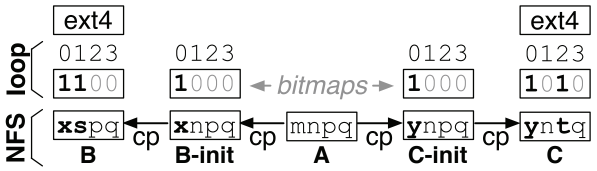

Figure 18: **Loopback Bitmaps.** *Containers B and C are started from the same image, A. Bitmaps track differences.*

E-init layer, so a real clone must be performed. For step 10, `Create` must do both a snapshot and clone to produce and mount layer E as the root for a new container.

5.4 Linux Kernel Modifications

Our analysis showed that multiple containers started from the same image tend to read the same data, suggesting cache sharing could be useful (§4.4). One advantage of the AUFS driver is that COW is done above an underlying file system. This means that different containers may warm and utilize the same cache state in that underlying file system. Slacker does COW within VMstore, beneath the level of the local file system. This means that two NFS files may be clones (with a few modifications) of the same snapshot, but cache state will not be shared, because the NFS protocol is not built around the concept of COW sharing. Cache deduplication could help save cache space, but this would not prevent the initial I/O. It would not be possible for deduplication to realize two blocks are identical until both are transferred over the network from the VMstore. In this section, we describe our technique to achieve sharing in the Linux page cache at the level of NFS files.

In order to achieve client-side cache sharing between NFS files, we modify the layer immediately above the NFS client (*i.e.*, the loopback module) to add awareness of VMstore snapshots and clones. In particular, we use bitmaps to track differences between similar NFS files. All writes to NFS files are via the loopback module, so the loopback module can automatically update the bitmaps to record new changes. Snapshots and clones are initiated by the Slacker driver, so we extend the loopback API so that Slacker can notify the module of COW relationships between files.

Figure 18 illustrates the technique with a simple example: two containers, B and C, are started from the same image, A. When starting the containers, Docker first creates two init layers (B-init and C-init) from the base (A). Docker creates a few small init files in these layers. Note that the "m" is modified to an "x" and "y" in the init layers, and that the zeroth bits are flipped to "1" to mark the change. Docker the creates the topmost container layers, B and C from B-init and C-init. Slacker uses the new loopback API to copy the B-init and C-init bitmaps to B and C respectively. As shown, the B and C bitmaps accumulate more mutations as the containers run and write data. Docker does not explicitly differentiate init layers from other layers as part of the API, but Slacker can infer layer type because Docker happens to use an "-init" suffix for the names of init layers.

Now suppose that container B reads block 3. The loopback module sees an unmodified "0" bit at position 3, indicating block 3 is the same in files B and A. Thus, the loopback module sends the read to A instead of B, thus populating A's cache state. Now suppose C reads block 3. Block 3 of C is also unmodified, so the read is again redirected to A. Now, C can benefit from the cache state of A, which B populated with its earlier read.

Of course, for blocks where B and C differ from A, it is important for correctness that reads are not redirected. Suppose B reads block 1 and then C reads from block 1. In this case, B's read will not populate the cache since B's data differs from A. Similarly, suppose B reads block 2 and then C reads from block 2. In this case, C's read will not utilize the cache since C's data differs from A.

5.5 Docker Framework Discussion

One our goals was to make no changes to the Docker registry or daemon, except within the pluggable storage driver. Although the storage-driver interface is quite simple, it proved sufficient for our needs. There are, however, a few changes to the Docker framework that would have enabled a more elegant Slacker implementation. First, it would be useful for compatibility between drivers if the registry could represent different layer formats (§5.2). Currently, if a non-Slacker layer pulls a layer pushed by Slacker, it will fail in an unfriendly way. Format tracking could provide a friendly error message, or, ideally, enable hooks for automatic format conversion. Second, it would be useful to add the notion of flattened layers. In particular, if a driver could inform the framework that a layer is flat, Docker would not need to fetch ancestor layers upon a pull. This would eliminate our need for lazy cloning and snapshot caching (§5.3). Third, it would be convenient if the framework explicitly identified init layers so Slacker would not need to rely on layer names as a hint (§5.4).

6 Evaluation

We use the same hardware for evaluation as we did for our analysis (§4). For a fair comparison, we also use the same VMstore for Slacker storage that we used for the virtual disk of the VM running the AUFS experiments.

6.1 HelloBench Workloads

Earlier, we saw that with HelloBench, push and pull times dominate while run times are very short (Figure 9). We repeat that experiment with Slacker, presenting the new results alongside the AUFS results in Figure 19. On average, the push phase is 153× faster and the pull phase is 72× faster, but the run phase is 17% slower (the AUFS pull phase warms the cache for the run phase).

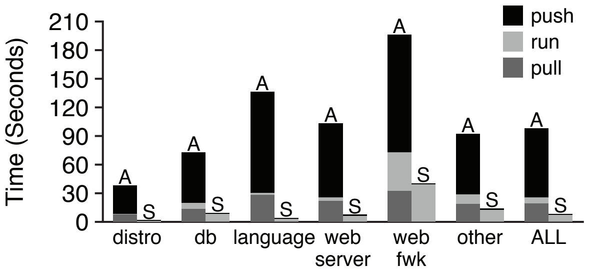

Figure 19: **AUFS vs. Slacker (Hello).** *Average push, run, and pull times are shown for each category. Bars are labeled with an "A" for AUFS or "S" for Slacker.*

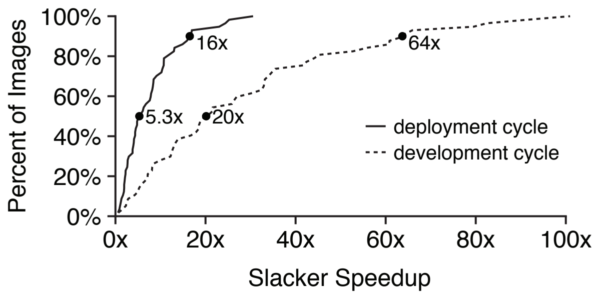

Figure 20: **Slacker Speedup.** *The ratio of AUFS-driver time to Slacker time is measured, and a CDF shown across HelloBench workloads. Median and 90th-percentile speedups are marked for the development cycle (push, pull, and run), and the deployment cycle (just pull and run).*

Different Docker operations are utilized in different scenarios. One use case is the *development cycle*: after each change to code, a developer pushes the application to a registry, pulls it to multiple worker nodes, and then runs it on the nodes. Another is the *deployment cycle*: an infrequently-modified application is hosted by a registry, but occasional load bursts or rebalancing require a pull and run on new workers. Figure 20 shows Slacker's speedup relative to AUFS for these two cases. For the median workload, Slacker improves startup by 5.3× and 20× for the deployment and development cycles respectively. Speedups are highly variable: nearly all workloads see at least modest improvement, but 10% of workloads improve by at least 16× and 64× for deployment and development respectively.

6.2 Long-Running Performance

In Figure 19, we saw that while pushes and pulls are much faster with Slacker, runs are slower. This is expected, as runs start before any data is transferred, and binary data is only lazily transferred as needed. We now run several long-running container experiments; our goal is to show that once AUFS is done pulling all image data and Slacker is done lazily loading hot image data, AUFS and Slacker have equivalent performance.

For our evaluation, we select two databases and two web servers. For all experiments, we execute for five minutes, measuring operations per second. Each experiment starts with a pull. We evaluate the PostgreSQL database using pgbench, which is "loosely based on TPC-B" [5]. We evaluate Redis, an in-memory database, using a custom benchmark that gets, sets, and updates keys with equal frequency. We evaluate the Apache web server, using the `wrk` [4] benchmark to repeatedly fetch a static page. Finally, we evaluate io.js, a JavaScript-based web server similar to node.js, using the `wrk` benchmark to repeatedly fetch a dynamic page.

Figure 21a shows the results. AUFS and Slacker usually provide roughly equivalent performance, though Slacker is somewhat faster for Apache. Although the drivers are similar with regard to long-term performance, Figure 21b shows Slacker containers start processing requests 3-19× sooner than AUFS.

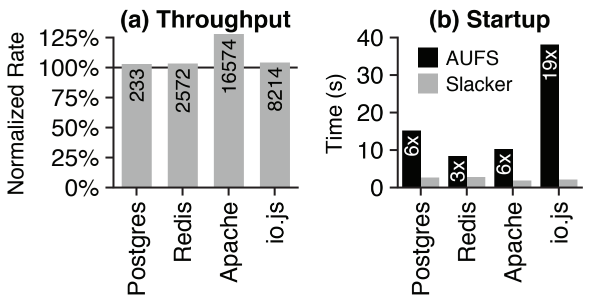

Figure 21: **Long-Running Workloads.** *Left: the ratio of Slacker's to AUFS's throughput is shown; startup time is included in the average. Bars are labeled with Slacker's average operations/second. Right: startup delay is shown.*

6.3 Caching

We have shown that Slacker provides much faster startup times relative to AUFS (when a pull is required) and equivalent long-term performance. One scenario where Slacker is at a disadvantage is when the same short-running workload is run many times on the same machine. For AUFS, the first run will be slow (as a pull is required), but subsequent runs will be fast because the image data will be stored locally. Moreover, COW is done locally, so multiple containers running from the same start image will benefit from a shared RAM cache.

Slacker, on the other hand, relies on the Tintri VMstore to do COW on the server side. This design enables rapid distribution, but one downside is that NFS clients are not naturally aware of redundancies between files without our kernel changes. We compare our modified loopback driver (§5.4) to AUFS as a means of sharing cache state. To do so, we run each HelloBench workload twice, measuring the latency of the second run (after the first has warmed the cache). We compare AUFS to Slacker, with and without kernel modifications.

Figure 22 shows a CDF of run times for all the workloads with the three systems (note: these numbers were

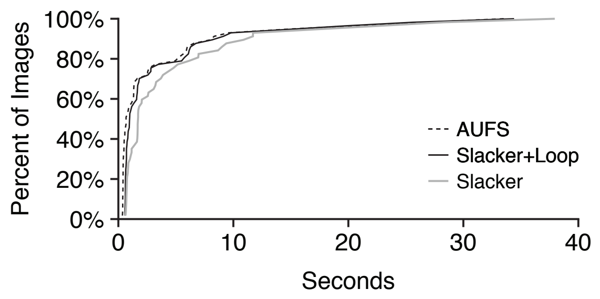

Figure 22: **Second Run Time (CDF).** *A distribution of run times are shown for the AUFS driver and for Slacker, both with and without use of the modified loopback driver.*

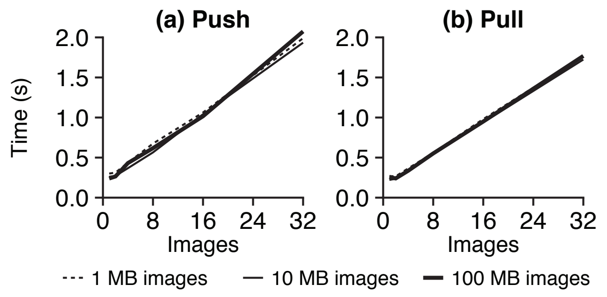

Figure 23: **Operation Scalability.** *A varying number of artificial images (x-axis), each containing a random file of a given size, are pushed or pulled simultaneously. The time until all operations are complete is reported (y-axis).*

collected with a VM running on a ProLiant DL360p Gen8). Although AUFS is still fastest (with median runs of 0.67 seconds), the kernel modifications significantly speed up Slacker. The median run time of Slacker alone is 1.71 seconds; with kernel modifications to the loopback module it is 0.97 seconds. Although Slacker avoids unnecessary network I/O, the AUFS driver can directly cache ext4 file data, whereas Slacker caches blocks beneath ext4, which likely introduces some overhead.

6.4 Scalability

Earlier (§4.2), we saw that AUFS scales poorly for pushes and pulls with regard to image size and the number of images being manipulated concurrently. We repeat our earlier experiment (Figure 10) with Slacker, again creating synthetic images and pushing or pulling varying numbers of these concurrently.

Figure 23 shows the results: image size no longer matters as it does for AUFS. Total time still correlates with the number of images being processed simultaneously, but the absolute times are much better; even with 32 images, push and pull times are at most about two seconds. It is also worth noting that push times are similar to pull times for Slacker, whereas pushes were much more expensive for AUFS. This is because AUFS uses compression for its large data transfers, and compression is typically more costly than decompression.

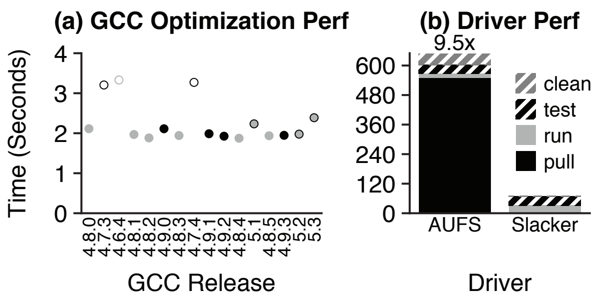

Figure 24: **GCC Version Testing.** *Left: run time of a C program doing vector arithmetic. Each point represents performance under a different GCC release, from 4.8.0 (Mar '13) to 5.3 (Dec '15). Releases in the same series have a common style (e.g., 4.8-series releases are solid gray). Right: performance of MultiMake is shown for both drivers. Time is broken into pulling the image, running the image (compiling), testing the binaries, and deleting the images from the local daemon.*

7 Case Study: MultiMake

When starting Dropbox, Drew Houston (co-founder and CEO) found that building a widely-deployed client involved a lot of "grungy operating-systems work" to make the code compatible with the idiosyncrasies of various platforms [18]. For example, some bugs would only manifest with the Swedish version of Windows XP Service Pack 3, whereas other very similar deployments (including the Norwegian version) would be unaffected. One way to avoid some of these bugs is to broadly test software in many different environments. Several companies provide containerized integration-testing services [33, 39], including for fast testing of web applications against dozens of releases of of Chrome, Firefox, Internet Explorer, and other browsers [36]. Of course, the breadth of such testing is limited by the speed at which different test environments can be provisioned.

We demonstrate the usefulness of fast container provisioning for testing with a new tool, MultiMake. Running MultiMake on a source directory builds 16 different versions of the target binary using the last 16 GCC releases. Each compiler is represented by a Docker image hosted by a central registry. Comparing binaries has many uses. For example, certain security checks are known to be optimized away by certain compiler releases [44]. MultiMake enables developers to evaluate the robustness of such checks across GCC versions.

Another use for MultiMake is to evaluate the performance of code snippets against different GCC versions, which employ different optimizations. As an example, we use MultiMake on a simple C program that does 20M vector arithmetic operations, as follows:

```
for (int i=0; i<256; i++) {
  a[i] = b[i] + c[i] * 3;
}
```

Figure 24a shows the result: most recent GCC releases optimize the vector operations well, but the but the code generated by the 4.6- and 4.7-series compilers takes about 50% longer to execute. GCC 4.8.0 produces fast code, even though it was released before some of the slower 4.6 and 4.7 releases, so some optimizations were clearly not backported. Figure 24b shows that collecting this data is 9.5× faster with Slacker (68 seconds) than with the AUFS driver (646 seconds), as most of the time is spent pulling with AUFS. Although all the GCC images have a common Debian base (which must only be pulled once), the GCC installations represent most of the data, which AUFS pulls every time. Cleanup is another operation that is more expensive for AUFS than Slacker. Deleting a layer in AUFS involves deleting thousands of small ext4 files, whereas deleting a layer in Slacker involves deleting one large NFS file.

The ability to rapidly run different versions of code could benefit other tools beyond MultiMake. For example, `git bisect` finds the commit that introduced a bug by doing a binary search over a range of commits [23]. Alongside container-based automated build systems [35], a bisect tool integrated with Slacker could very quickly search over a large number of commits.

8 Related Work

Work optimizing the multi-deployment of disk images is similar to ours, as the ext4-formatted NFS files used by Slacker resemble virtual-disk images. Hibler *et al*. [16] built Frisbee, a system that optimizes differential image updates by using techniques based on file-system awareness (*e.g*., Frisbee does not consider blocks that are unused by the file system). Wartel *et al*. [45] compare multiple methods of lazily distributing virtual-machine images from a central repository (much like a Docker registry). Nicolae *et al*. [28] studied image deployment and found *"prepropagation is an expensive step, especially since only a small part of the initial VM is actually accessed."* They further built a distributed file system for hosting virtual machine images that supports lazy propagation of VM data. Zhe *et al*. [50] built Twinkle, a cloud-based platform for web applications that is designed to handle *"flash crowd events."* Unfortunately, virtual-machines tend to be heavyweight, as they note: *"virtual device creation can take a few seconds."*

Various cluster management tools provide container scheduling, including Kubernetes [2], Google's Borg [41], Facebook's Tupperware [26], Twitter's Aurora [21], and Apache Mesos [17]. Slacker is complementary to these systems; fast deployment gives cluster managers more flexibility, enabling cheap migration and fine-tuned load balancing.

A number of techniques bear resemblance to our strategy for sharing cache state and reducing redundant I/O. VMware ESX server [43] and Linux KSM [9] (Kernel Same-page Merging) both scan and deduplicate memory. While this technique saves cache space, it does not prevent initial I/O. Xingbo *et al*. [47] also observed the problem where reads to multiple nearly identical files cause avoidable I/O. They modified btrfs to index cache pages by disk location, thus servicing some block reads issued by btrfs with the page cache. Sapuntzakis *et al*. [32] use dirty bitmaps for VM images to identify a subset of the virtual-disk image blocks that must be transferred during migration. Lagar-Cavilla *et al*. [20] built a "VM fork" function that rapidly creates many clones of a running VM. Data needed by one clone is multicast to all the clones as a means of prefetch. Slacker would likely benefit from similar prefetching.

9 Conclusions

Fast startup has applications for scalable web services, integration testing, and interactive development of distributed applications. Slacker fills a gap between two solutions. Containers are inherently lightweight, but current management systems such as Docker and Borg are very slow at distributing images. In contrast, virtual machines are inherently heavyweight, but multi-deployment of virtual machine images has been thoroughly studied and optimized. Slacker provides highly efficient deployment for containers, borrowing ideas from VM image-management, such as lazy propagation, as well as introducing new Docker-specific optimizations, such as lazy cloning. With these techniques, Slacker speeds up the typical deployment cycle by 5× and development cycle by 20×. HelloBench and a snapshot [15] of the images we use for our experiments in this paper are available online: `https://github.com/Tintri/hello-bench`

10 Acknowledgements

We thank the anonymous reviewers and Atul Adya (our shepherd) for their tremendous feedback, as well as members of our research group at UW-Madison and coworkers at Tintri for their thoughts and comments on this work at various stages. We especially thank Zev Weiss, John Schmitt, Sean Chen, and Kieran Harty for their design suggestions and help with experiments.

This material was supported by funding from NSF grants CNS-1218405, CNS-1319405, CNS-1419199, and CNS-1421033, as well as generous donations from EMC, Facebook, Google, Huawei, Microsoft, NetApp, Samsung, Tintri, Veritas, Seagate, and VMware as part of the WISDOM research institute sponsorship. Tyler Harter is supported by Tintri and an NSF Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF or other institutions.

References

[1] Tintri VMstore(tm) T600 Series. http://www.tintri.com/sites/default/files/field/pdf/document/t600-datasheet_0.pdf, 2013.

[2] Kubernetes. http://kubernetes.io, August 2014.

[3] Docker Hub. https://hub.docker.com/u/library/, 2015.

[4] Modern HTTP Benchmarking Tool. https://github.com/wg/wrk/, 2015.

[5] pgbench. http://www.postgresql.org/docs/devel/static/pgbench.html, September 2015.

[6] Tintri Operating System. https://www.tintri.com/sites/default/files/field/pdf/whitepapers/tintri-os-datasheet-150701t10072.pdf, 2015.

[7] Keith Adams and Ole Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*, Seattle, Washington, March 2008.

[8] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. End-to-end Performance Isolation Through Virtual Datacenters.

[9] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using KSM. In *Proceedings of the linux symposium*, pages 19–28, 2009.

[10] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.91 edition, May 2015.

[11] Jens Axboe, Alan D. Brunelle, and Nathan Scott. blktrace(8) - Linux man page. http://linux.die.net/man/8/blktrace, 2006.

[12] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 143–156, Saint-Malo, France, October 1997.

[13] Jeremy Elson and Jon Howell. Handling Flash Crowds from Your Garage. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 171–184, Berkeley, CA, USA, 2008. USENIX Association.

[14] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. Enforcing Performance Isolation Across Virtual Machines in Xen. In *Proceedings of the ACM/IFIP/USENIX 7th International Middleware Conference (Middleware'2006)*, Melbourne, Australia, Nov 2006.

[15] Tyler Harter. HelloBench. http://research.cs.wisc.edu/adsl/Software/hello-bench/, 2015.

[16] Mike Hibler, Leigh Stoller, Jay Lepreau, Robert Ricci, and Chad Barb. Fast, Scalable Disk Imaging with Frisbee. In *USENIX Annual Technical Conference, General Track*, pages 283–296, 2003.

[17] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*, volume 11, pages 22–22, 2011.

[18] Drew Houston. https://www.youtube.com/watch?t=1278&v=NZINmtuTSu0, 2014.

[19] Michael Kerrisk and Eric W. Biederman. namespaces(7) - overview of Linux namespaces. http://man7.org/linux/man-pages/man7/namespaces.7.html, 2013.

[20] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M Rumble, Eyal De Lara, Michael Brudno, and Mahadev Satyanarayanan. SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 1–12. ACM, 2009.

[21] Dave Lester. All about Apache Aurora. https://blog.twitter.com/2015/all-about-apache-aurora, 2015.

[22] Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Physical Disentanglement in a Container-Based File System. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[23] Git Manpages. git-bisect(1) Manual Page. https://www.kernel.org/pub/software/scm/git/docs/git-bisect.html, 2015.

[24] Arif Merchant, Mustafa Uysal, Pradeep Padala, Xiaoyun Zhu, Sharad Singhal, and Kang Shin. Maestro: quality-of-service in large disk arrays.

[25] Dirk Merkel. Docker: lightweight Linux containers for consistent development and deployment. Linux Journal, Issue 239, March 2014.

[26] Aravind Narayanan. Tupperware: Containerized Deployment at Facebook. http://www.slideshare.net/Docker/aravindnarayanan-facebook140613153626phpapp02-37588997, 2014.

[27] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-Clouds: Managing Performance Interference Effects for QoS-Aware Clouds.

[28] Bogdan Nicolae, John Bresnahan, Kate Keahey, and Gabriel Antoniu. Going back and forth: Efficient multideployment and multisnapshotting on clouds. In *Proceedings of the 20th international symposium on High performance distributed computing*, pages 147–158. ACM, 2011.

[29] Patrick ONeil, Edward Cheng, Dieter Gawlick, and Elizabeth ONeil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33(4):351–385, 1996.

[30] John Pescatore. Nimda Worm Shows You Can't Always Patch Fast Enough. https://www.gartner.com/doc/340962, September 2001.

[31] David Saff and Michael D Ernst. An Experimental Evaluation of Continuous Testing During Development. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 76–85. ACM, 2004.

[32] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the Migration of Virtual Computers. *SIGOPS Oper. Syst. Rev.*, 36(SI):377–390, December 2002.

[33] Sunil Shah. Integration Testing with Mesos, Chronos and Docker. http://mesosphere.com/blog/2015/03/26/integration-testing-with-mesos-chronos-docker/, 2015.

[34] David Shue, Michael J. Freedman, and Anees Shaikh. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI '12)*, Hollywood, California, October 2012.

[35] Matt Soldo. Upgraded Autobuild System on Docker Hub. http://blog.docker.com/2015/11/upgraded-autobuild-docker-hub/, 2015.

[36] spoon.net. Containerized Selenium Testing. https://blog.spoon.net/running-a-selenium-grid-using-containers/, 2015.

[37] The Linux Community. LXC – Linux Containers, 2014.

[38] E. Thereska, A. Rowstron, H. Ballani, G. O'Shea, T. Karagiannis, T. Talpey, R. Black, and T. Zhu. IOFlow: A Software-Defined Storage Architecture. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Farmington, Pennsylvania, November 2013.

[39] Paul van der Ende. Fast and Easy Integration Testing with Docker and Overcast. http://blog.xebia.com/2014/10/13/fast-and-easy-integration-testing-with-docker-and-overcast/, 2014.

[40] Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Performance Isolation: Sharing and Isolation in Shared-memory Multiprocessors. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, pages 181–192, San Jose, California, October 1998.

[41] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[42] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger. Argon: Performance Insulation for Shared Storage Servers. In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST '07)*, San Jose, California, February 2007.

[43] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[44] Xi Wang, Nickolai Zeldovich, M Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 260–275. ACM, 2013.

[45] Romain Wartel, Tony Cass, Belmiro Moreira, Ewan Roche, Manuel Guijarro, Sebastien Goasguen, and Ulrich Schwickerath. Image distribution mechanisms in large scale cloud providers. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 112–117. IEEE, 2010.

[46] Zev Weiss, Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. ANViL: Advanced Virtualization for Modern Non-Volatile Memory Devices. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*, Santa Clara, CA, February 2015.

[47] Xingbo Wu, Wenguang Wang, and Song Jiang. TotalCOW: Unleash the Power of Copy-On-Write for Thin-provisioned Containers. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, APSys '15, pages 15:1–15:7, New York, NY, USA, 2015. ACM.

[48] Suli Yang, Tyler Harter, Nishant Agrawal, Salini Selvaraj Kowsalya, Anand Krishnamurthy, Samer Al-Kiswany, Rini T. Kaushik, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Split-level I/O Scheduling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 474–489, New York, NY, USA, 2015. ACM.

[49] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. CPI2: CPU Performance Isolation for Shared Compute Clusters.

[50] Jun Zhu, Zhefu Jiang, and Zhen Xiao. Twinkle: A Fast Resource Provisioning Mechanism for Internet Services. In *INFOCOM, 2011 Proceedings IEEE*, pages 802–810. IEEE, 2011.