Container Management as Emerging Workload for Operating Systems

Tatsushi Inagaki Yohei Ueda Moriyoshi Ohara
IBM Research – Tokyo
IBM Japan, Ltd.
{e29253,yohei,ohara}@jp.ibm.com

***Abstract*—Operating-system-level virtualization is becoming increasingly important for server applications since it provides containers as a foundation of the emerging microservice architecture, which enables agile application development, deployment, and operation – the essential characteristics in modern cloud-based services. Agility in the microservice architecture heavily depends on fast management operations for containers, such as create, start, and stop. Since containers rely on administrative kernel services provided by the host operating system, the microservice architecture can be considered as a new workload for an operating system as it stresses those services differently from traditional workloads.**

We studied the scalability of container management operations for Docker, one of the most popular container management systems, from two aspects: core and container scalability, which indicate how much the number of processor cores and number of containers affect container management performance, respectively. We propose a hierarchical analysis approach to identify scalability bottlenecks where we analyze multiple layers of a software stack from the top to bottom layer. Our analysis reveals that core scalability has bottlenecks at a virtualization layer for storage and network devices, and that container scalability has bottlenecks at various components that inquire mount points. While those bottlenecks exist in a daemon process of Docker, the root causes are a couple of interfaces of the underlying kernel. This implies the operating system has room for improvement to more efficiently host emerging microservice applications.

I. Introduction

The recent evolution of container technology [1], [2], [3], [4], [5], that is, operating-system-level (OS-level) virtualization, accelerates continuous development, deployment, and operation of large-scale web services. A common design practice is to implement a service as a set of microservices [6]. Each service is developed as an *image* that contains a complete but minimum set of files required to run the service and is deployed as a *container*, which is a virtual guest OS instance running on the kernel of the host OS. This design improves the portability, isolation, and robustness of a microservice by confining all the dependencies to dynamically linked libraries and configuration files inside its image.

While the management operations of containers, such as create, start, stop, commit, and remove of containers, is much faster than that of traditional virtual machines (VMs) [7], its performance is still important to accelerate the following activities for agile development, deployment, and operation practices:

1) **Test-driven development.** A container management system such as Docker [2] builds a container image by running a sequence of containers to execute each command in a build script called *Dockerfile*. While this design accelerates the building of an image by caching intermediate images, it incurs an overhead of container management operations at the first build. Faster container management operations enables a developer to be more productive since he or she more frequently builds an image and runs a container to test the application manually or automatically.
2) **Continuous integration and deployment.** When a developer adds a new feature, or fixes a bug, of a microservice, the change will be immediately integrated into a shared code repository, and will be eventually deployed to the production environment, if the updated microservice passes an appropriate test suite. Faster container management operations accelerate the building of an image to detect a build break, pushing it into an image registry, testing it in a container, and deploying it as a container to staging and production environments.
3) **Autoscaling.** Popular cloud computing services provide autoscaling, where the amount of computing resources is automatically adjusted according to the frequency of service requests or utilization of computing resources [8], [9], [10], [11]. Adding new computing resources to an application hosted on containers leads to the deployment of new containers. A special case of this scenario is a failover or migration of microservices running as a container.

We studied the scalability of two frequently used container management operations, *building* an image and *running* a container, of Docker, one of the most popular container management systems. It adopts a client-server architecture, in which a Docker daemon as a server manages all the containers associated with it and a client sends a request to the daemon to manage a container as necessary. We identified the bottlenecks in the operations, typically a system call that limits the concurrent execution of multiple management operations, by analyzing multiple layers in a software stack hierarchically. In particular, we focused on *vertical* scalability, which is on a single Docker daemon running on a single host OS with multiple processor cores (i.e. scale up). Luzzardi demonstrated

978-1-5090-3896-1/16/$31.00 ©2016 IEEE

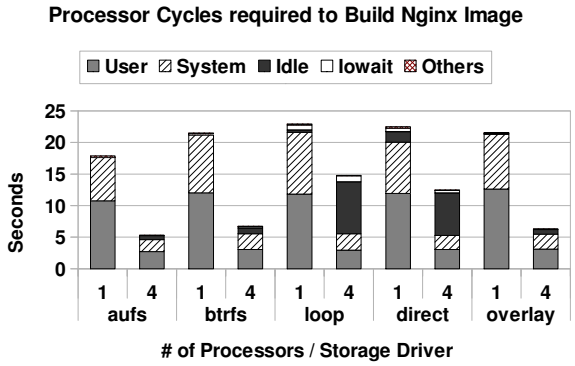

Fig. 1. Breakdown of processor cycles to build Nginx image when building eight images in single Docker daemon running on one and four processor cores with different types of storage drivers. The processor cycles are shown in seconds required to build one image per processor core, which corresponds to the elapsed time of entire operation. Measurements were conducted in the environment described in Appendix A.

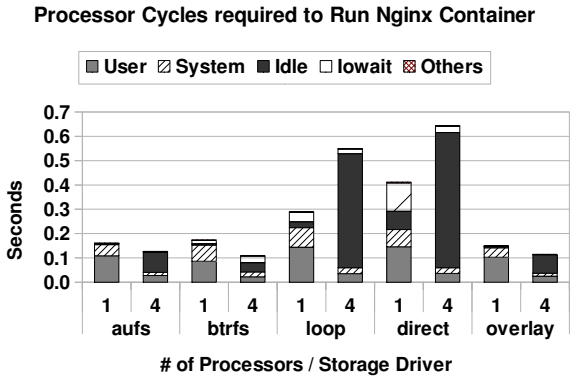

Fig. 2. Breakdown of processor cycles to run Nginx container when running sixty-four containers in single Docker daemon on one and four processor cores with different types of storage drivers. The processor cycles are shown in seconds spent to run one container per processor core, which correspond to the elapsed time of the entire operation.

massive *horizontal* scalability, which is on multiple Docker daemons running on multiple host OSs (i.e. scale out) [12]. The high vertical scalability allows each node to host a large number of containers by taking advantage of a large number of cores in a modern processor, which leads to high cost performance combined with the already demonstrated high horizontal scalability. We also studied the effect of *storage drivers* of Docker on scalability. A storage driver implements a copy-on-write strategy among container images and a root filesystem of containers to accelerate container management operations. The choice of a storage driver has a large impact on the performance of container management [13]. Another benefit of vertical scalability is disk-space efficiency. The higher vertical scalability becomes, the more containers can share a set of images associated with a Docker daemon. In Section II, we briefly introduce the architecture of a Docker daemon and its software stack.

We first investigated *core scalability*. We would like to answer the questions "How much can we accelerate container management operations by making extra processor cores available to a single Docker daemon?" and "If the operations do not scale, why?" Figures 1 and 2 show the processor cycles required to build an image of a container and to run a container from the built image. The image is built to run Nginx [14], which is widely used as a web server, reverse proxy server, and load balancer in web applications. Note that the numbers in Figures 1 and 2 correspond to the elapsed time since they also include idle cycles. With ideal core scalability, the number of processor cycles to build one image or run one container on four processors should be a quarter of those on a single processor. However, our experimental results show that building an image does not scale with the `devicemapper` storage driver operating in either the `loop-lvm` or `direct-lvm` mode, and running a container does not scale well with any storage driver. In Section III, we analyze idle processor cycles in a Docker daemon to identify the bottlenecks to these operations. The daemon is implemented in the Go [15] programming language which provides lightweight user-level threads called *goroutines*. Our hierarchical analysis allows us to identify bottlenecks when a number of goroutines are idle. For building images, the `devicemapper` storage driver has bottlenecks to remove virtual block devices. For running containers, all storage drivers have a bottleneck to allocate a root filesystem of a container and/or in the port mapper that connects a container to the network.

We then investigated *container scalability*. We would like to answer the questions, "How much can we consolidate multiple containers into a single Docker daemon without extra overhead?" and "If overhead exists, why?" Figure 3 shows the processor cycles required to run a container when we change the total number of containers run in a single Docker daemon. This container runs a shell command of the BusyBox [16] tool, which links popular command-line tools into a single binary executable. With ideal container scalability, the number of processor cycles to run one container should remain constant and be irrelevant to the total number of containers. However, our experimental results show that the number of processor cycles to run 1 of 1,024 containers is more than twice that to run 1 of 64 containers with any storage driver. This means running a new container becomes slower when a large number of containers are already running. In Section IV, we identify a bottleneck in the Docker daemon by analyzing busy processor cycles. While the daemon has a flat execution profile, which the hottest procedure consumes only a few percent of the total number of processor cycles, our hierarchical analysis again allows us to identify that the root cause is parsing overhead of `mountinfo`, which is a virtual file to provide information of mount points, and the size of which increases proportionally to the number of active containers.

In contrast to the related work described in Section V, our contributions in this paper are as follows:

1) **Proposing hierarchical bottleneck-analysis approach.** We argue that a novel approach of hierarchically analyz-

ing idle and busy processor cycles allows us to identify bottlenecks for core and container scalability, even when the target application uses a number of idle threads and has a flat execution profile.

2) **Identifying importance of the container management performance.** The emerging use of container technology makes the administrative performance of the OS more important to improve productivity of developers and responsiveness of cloud services. The performance of these interfaces had not been sufficiently investigated.
3) **Analyzing vertical scalability of container management and identifying bottlenecks.** While horizontal scalability of container management has been extensively studied, vertical scalability and where bottlenecks exist have not. Identifying these bottlenecks will be useful for developers of container management systems and of OSs to efficiently use many processor cores in a modern processor.

Our conclusion, discussed in detail in Section VI, is that container management is a new workload that stresses the administrative performance of the host OS. While the bottlenecks identified in Sections III and IV are in a Docker daemon, they have origins in system calls, virtual filesystems, and other subsystems of the underlying OS for resource management. Previously, human operators used these administrative interfaces daily or hourly, but the emerging microservice architecture exercises them in milliseconds to build responsive web services.

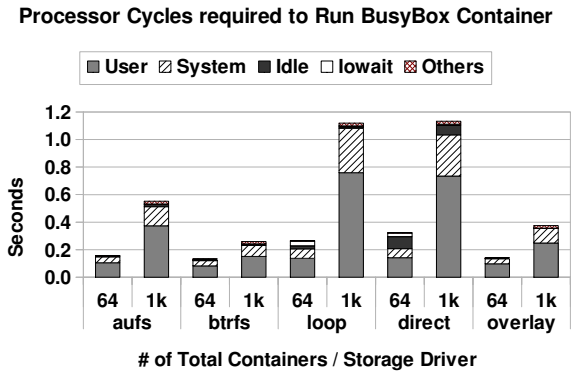

Fig. 3. Breakdown of number of processor cycles to run BusyBox container when running 64 and 1,024 containers simultaneously in single Docker daemon on 1 processor core. The processor cycles are shown in seconds required to run one container, which corresponds to the elapsed time divided by the number of total containers.

II. Docker Architecture

We briefly describe the architecture of Docker [2] and related features of the Linux kernel [17] for easy understanding of the succeeding Sections III and IV.

Docker uses a client-server architecture to implement OS-level virtualization. A client connects to a daemon via the application programming interface (API) in the style of Representational State Transfer (REST) over the Hypertext Transfer Protocol (HTTP). Two common operations that a client can do are as follows:

1) **Build** an *image* according to a given *Dockerfile*.
2) **Run** a *container* based on an image.

A container is a unit of OS-level virtualization and has

1) an initial process, which is isolated by *namespaces* and whose resources are managed by *control groups* (cgroups),
2) an isolated root filesystem, which is a copy-on-write snapshot of an image, and
3) virtual devices such as networks and terminals.

A Dockerfile contains a sequence of shell script commands to create files in an image. A unique feature of Docker as a container management system is that an image is also built as a snapshot from its parent image. That is, Docker runs each line of a Dockerfile as a container based on a parent image built by the previous line and commits the updated snapshot as a new child image of the parent image. The benefits of this implementation are that it can

1) accelerate repeated building of variations in the same image with an updated Dockerfile by caching intermediate images,
2) save disk spaces to store images by sharing common layers among multiple images, and
3) save memory spaces to store containers by sharing file caches among multiple containers based on the same image.

The downside is the overhead to implement the copy-on-write strategy, which is incurred at the runtime and management time of a container. The latter is our focus in this paper.

A. Storage Drivers

A *storage driver* (also known as a "graph driver") is a pluggable component of a Docker daemon that implements a copy-on-write strategy using various file systems and subsystems of the Linux OS. The daemon determines the storage driver when it starts and continues using the same driver while it is running. Images and containers cannot share snapshots across different storage drivers. The storage drivers we investigated in this study are `aufs`, `btrfs`, `devicemapper`, and `overlay`.

1) Aufs: The `aufs` storage driver uses the advanced multi-layered unification filesystem (AUFS) [18] on the Linux filesystem. The AUFS provides a *union mount* operation, which can stack multiple directories (also called as "branches") on the base file system into one virtual directory. In the context of Docker, the root file system of a container consists of a union-mounted directory with a writable empty branch at the top, and read-only branches from the base image below. When a container opens a file to read, AUFS searches for the file from the top to the bottom branches. When a container opens a file to write, AUFS searches for the file and *copies* it *up* into a writable branch. The copied file in the writable branch hides the later accesses to the original file in a lower branch.

2) Btrfs: The `btrfs` storage driver uses the Linux B-Tree filesystem (Btrfs) [19], which is a copy-on-write Linux filesystem. The Btrfs provides a *subvolume*, which is an independent root directory within a Btrfs or another subvolume. A *snapshot* of a subvolume is a subvolume that looks like a full copy of the source subvolume, but the actual data are not copied until the snapshot is updated. The `btrfs` storage driver implements the base layer of an image as a subvolume, and a child layer as a snapshot of its parent layer. The root file system of a container is a snapshot of the base image.

3) Devicemapper: The `devicemapper` storage driver uses the Device Mapper [20] framework of the Linux kernel, which can create a virtual block device based on another block device. The Device Mapper framework provides features *thin-provisioning*, which allocates a block on-demand when it is written, and *snapshot*, which can create a virtual block device as a snapshot of another block device. Given a logical volume having the thin-provisioning feature as a thin pool, `devicemapper` creates a *base device* from the pool and formats the device with a filesystem such as ext4 or xfs. Then a layer of an image or a container is created as a snapshot of its parent layer or the base image. When a container writes into a file, the Device Mapper framework allocates new blocks into the device for the root filesystem of the container and copies the corresponding blocks from the ancestor devices.

The `devicemapper` storage driver has two different modes regarding how to allocate the thin pool:

- The `loop-lvm` mode allocates a sparse file, which is a file allocated with a given size, but not yet written, and creates a thin pool on the sparse file as a *loop device*, which is a file accessed as a virtual block device.
- The `direct-lvm` mode uses a given thin pool which is allocated on a physical block device.

4) Overlay: The `overlay` storage driver uses the OverlayFS, which is another union file system of the Linux file system. The major difference from AUFS is that OverlayFS stacks two directories, an *upperdir* and *lowerdir*, and provides a single *merged* directory. In the context of Docker, the root file system of a container is a merged directory whose upperdir is a writable directory and lowerdir is a read-only directory, where the directories and files from the base image are hard-linked.

B. Network Driver

The network driver of a Docker daemon connects a container to the network. While Docker provides a pluggable network driver, we describe the default `bridge` network, which we used for our experiments in this study.

The `bridge` network allocates an Ethernet bridge with a set of private Internet Protocol (IP) addresses of the host OS. Each container has its own network namespace, is connected to the bridge via a Virtual Ethernet pair, and obtains a private IP address from the bridge. Thus, the containers connected to the same `bridge` network can communicate with each other using their private IP addresses.

For communication with the outside world, the `bridge` network uses the Netfilter framework which provides packet filtering and network address and port translation (NAPT). The `bridge` network uses the `iptables` command to configure the NAPT rules on the host OS.

For *hairpinning*, which is communication between containers using an external IP address and a port, the `bridge` network spawns a dedicated process called *userland proxy* for each container.

C. Hierarchy

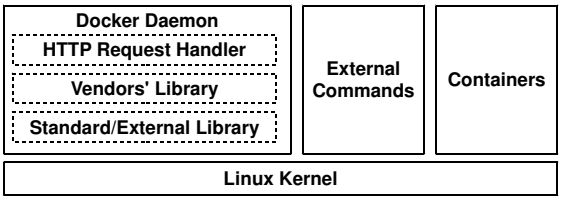

Fig. 4. Hierarchy of software stack around Docker daemon.

Figure 4 shows the hierarchy of the software stack inside and outside the Docker daemon. Since the daemon receives a request from a client via the REST API, the topmost layer contains HTTP request handlers to execute Docker commands. The next layer contains a standard runtime library from the Go programming language and external libraries, such as `libdevmapper`, to handle the Device Mapper framework. External commands, such as `iptables`, are also used. The lowest layer is the Linux kernel, including subsystems such as AUFS, Btrfs, Device Mapper, OverlayFS, Ethernet bridge, and Netfilter.

III. Core Scalability

Building images with `devicemapper` and running containers with any storage driver do not scale according to the number of processor cores, as shown in Figures 1 and 2. The breakdown in the processor cycles shows that the primary reason is idle cycles.

It is not trivial to identify why processors are idle and what is the bottleneck resource, because we cannot charge idle cycles directly to any executable instruction by using a profiling tool. Altman et al. [21] demonstrated that categorizing idle threads in a hub application can characterize performance problems of server applications due to idle cycles, and can infer their root cause. Since their primary target is an application server compliant to Java Platform, Enterprise Edition (Java EE) [22], they analyzed idle Java threads. In our case, these threads correspond to goroutines.

A problem is that goroutines are frequently spawned in a typical application written in the Go programming language, and most of the goroutines are idle. While a Java thread is typically implemented as a native thread, a goroutine is a user-level thread implemented by the runtime library of Go. It is spawned by both the runtime library and application programs for various purposes such as concurrent input and output

operations, and event processing. In our examples, the image-building scenario in Figure 1 spawns up to 150 goroutines with about 30 different contexts, and the container running scenario in Figure 2 spawns up to 1,000 goroutines with about 30 different contexts, and almost all of them are idle.

A. Hierarchical Analysis of Idle Cycles

On a typical client-server application, we can rephrase the question, "Why does the application not scale according to the number of processor cores?" into another question, "Why is the concurrency among the original requests from the clients lost after receiving requests and before using processor cores?" It is likely because a worker thread is blocked at synchronization with another thread to hold a software or hardware resource exclusively. This leads to the following approach to identify bottleneck resources based on a layered performance model [23].

1) Given a thread dump, identify the source code of the target application, and the topmost layer in the software stack of the application.
2) Identify the set of threads that handle incoming requests at the topmost layer.
3) Classify each thread into the following three subsets:
   a) **Running** to process a request from a client,
   b) **Blocked** by another thread at the same stack layer, or
   c) **Waiting** for completion of a service from a lower stack layer.
4) If the majority of the threads are *blocked*, the corresponding resources guarded by mutual exclusion are the bottleneck resources.
5) If the majority of the threads are *waiting* for services from a lower layer, identify a new set of threads that are handling the services at the layer. Go down into that layer and repeat from Step 3 above.

In our case, the topmost layer for Step 2 is an HTTP request handler in a Docker daemon, which is implemented as the method `ServeHTTP` of the interface `net/http.Handler`. The Docker daemon accepts a request from a client as an HTTP request, as shown in Figure 4.

A goroutine blocks other goroutines by acquiring a mutual exclusion lock (mutex). This is implemented as the method `Lock` of the structure `sync.Mutex`. At Step 3b, we can identify whether a goroutine is blocked by detecting this method in the stack dump of the goroutine.

A goroutine also waits for the completion of a service from a lower layer. This is implemented as a receive operation from another goroutine. At Step 3c, we can identify whether a goroutine is waiting for another goroutine by inspecting a receive operation at the top of the stack dump of the goroutine.

At Step 5, we need to identify a goroutine that *will* send a message into the corresponding channel upon a service completion. Since this information is not available in only a stack dump, we analyze the source code to identify which channel is used by the receive operation, which statement will send a value to the channel, and which goroutine in the stack dump will execute the send statement.

B. Bottlenecks to Building Images on Multiple Cores

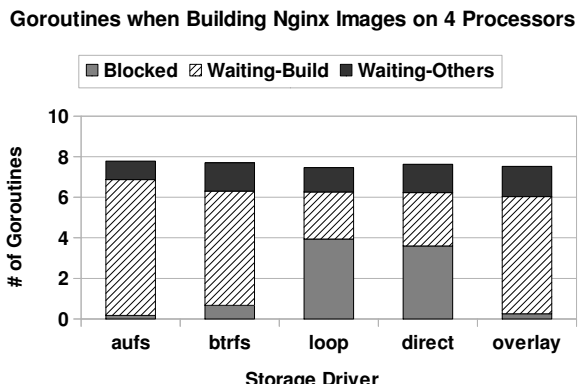

Fig. 5. Mean number of goroutines which are serving HTTP requests in daemon when building eight Nginx images concurrently in one daemon running on four processor cores. The component "Blocked" represents goroutines blocked with `sync.Mutex.Lock` to acquire a mutex. The component "Waiting-Build" represents goroutines that are waiting for a lower layer running a build script command in a Dockerfile to build an image.

According to the approach in Section III-A, we can now investigate why building Nginx images in Figure 1 does not scale with `devicemapper`. Figure 5 shows the mean number of goroutines serving HTTP requests in a daemon when building eight Nginx images concurrently in one daemon running on four processor cores. We can see that:

1) The majority of the goroutines are *waiting* for a build script command in a Dockerfile with `aufs`, `btrfs`, and `overlay`, but with the two modes `loop-lvm` and `direct-lvm` of `devicemapper`, only two out of eight goroutines are waiting for a build script command. That is, only two images can be built concurrently on `devicemapper`, while more images can on other storage drivers.
2) About half of the goroutines are *blocked* with `devicemaper`.

The first observation explains that `devicemapper` scales less than other storage drivers. The second observation explains why the difference in the first observation occurred.

We can then identify the bottleneck resources by investigating what mutex each blocked goroutine is acquiring. Figure 6 shows a breakdown of the blocked goroutines in Figure 5 by the structure guarded by a mutex. With `devicemapper`, the two dominant bottleneck resources are as follows:

1) The mutex `mountL` of the structure `layerStore`, which manages layers of images and containers, and is a singleton object in the Docker daemon. This mutex is used to serialize creation and removal of the topmost read-write layer of the container's root filesystem.
2) The mutex of the structure `DeviceSet`, which manages the list of virtual devices, and is a singleton object

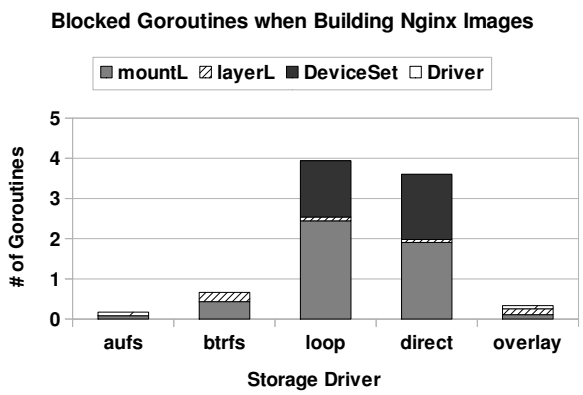

Fig. 6. Mean number of blocked goroutines serving HTTP requests in daemon when building eight Nginx images concurrently in one daemon running on four processor cores. Each component represents a mutex that a blocked goroutine is acquiring. The components "mountL" and "layerL" are two mutexes of the structure `layerStore` which manages layers of images and containers. The component "DeviceSet" is the mutex of the structure `DeviceSet`, which manages the set of virtual devices of `devicemapper`. The component "Driver" is the mutex of the storage driver.

in `devicemapper`. This mutex serializes all invocations to the Device Mapper subsystem via the library `libdevmapper`, which is not thread-safe.

When a goroutine acquires both mutexes, the ordering to avoid deadlock is `mountL` first and `DeviceSet` second. Thus, the primary bottleneck can be `DeviceSet`.

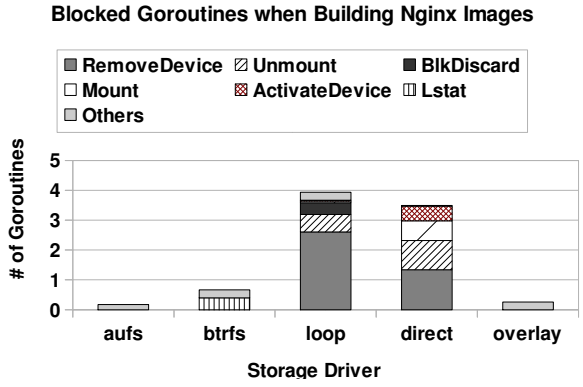

Fig. 7. Mean number of blocked goroutines serving HTTP requests in daemon when building eight Nginx images concurrently in one daemon running on four processor cores. The components represent blocking operations in the critical section. The components "RemoveDevice" and "ActivateDevice" are corresponding function calls of `libdevmapper`, which are implemented as `ioctl` system calls. The components "Unmount" and "Mount" are `umount` and `mount` system calls, respectively. The component "BlkDiscard" is another `ioctl` system call to discard a block device. The component "Lstat" is the system call `lstat`, which is invoked during traversal of file paths by the function `path/filepath.Walk`.

We can further identify the root cause of the bottlenecks by investigating what the *blocking* goroutine is doing while it locks the corresponding mutex in its critical section. This is less obvious than identifying *blocked* goroutines because the methods `Lock` and `Unlock` are just library calls. Fortunately, the scope of mutex and that of callers of the function that acquire the mutex are usually nested within each other since a common programming practice in Go is locking a mutex once in a function and unlocking it by the next `defer` statement, which is executed at an exit of the function. Thus, the caller function of the method `Lock` is typically on the stack of the blocking goroutine while it is in the critical section. Figure 7 shows the breakdown of blocked goroutines in Figure 5 by the operation the blocking goroutine is doing. With `devicemapper`, the structure `DeviceSet` is the primary bottleneck resource since the system calls `mount`, `umount`, and `ioctl` for activating and removing mapping of a virtual device and for discarding a block device are done while locking `DeviceSet`.

In summary, concurrent building of images does not scale with `devicemapper` due to the bottleneck to serialize creation, removal, mounting, and unmounting of virtual devices for the container's root filesystems.

C. Bottlenecks to Running Containers on Multiple Cores

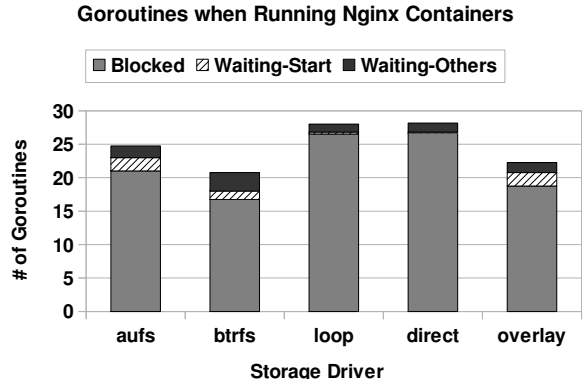

Fig. 8. Mean number of goroutines serving HTTP requests in daemon when running thirty-two Nginx containers concurrently in one daemon running on four processor cores. The component "Blocked" represents goroutines blocked with `sync.Mutex.Lock`. The component "Waiting-Start" represents goroutines waiting for an Nginx daemon that had started in a container.

The approach in Section III-A also allows us to explain why running Nginx containers in Figure 2 does not scale with any storage drivers. Figure 8 shows the mean number of goroutines serving HTTP requests in the daemon when running thirty-two Nginx containers in one daemon running on four processor cores. We can see that the number of goroutines that are actually starting Nginx daemon in each container is less than two with any storage driver. This is the direct reason that concurrent running of Nginx containers does not scale.

We can then identify bottleneck resources by investigating mutexes that the blocked goroutines are acquiring. Figure 9 shows a breakdown of the blocked goroutines by the structure guarded by a mutex. The two dominant bottleneck resources are as follows:

1) The mutex `mountL` of the structure `layerStore`, which was also a bottleneck resource for building images in Section III-B, for `aufs` and the two modes

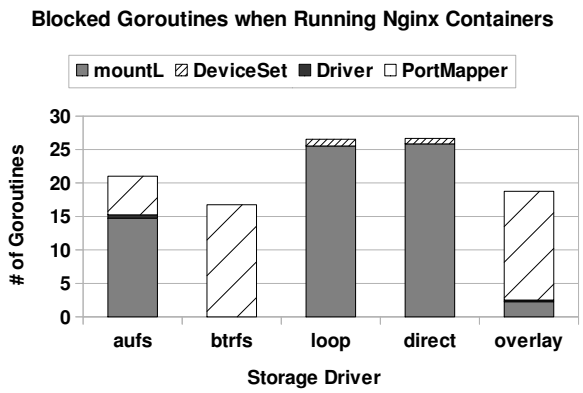

Fig. 9. Mean number of goroutines serving HTTP requests in daemon when running thirty-two Nginx containers concurrently in one daemon running on four processor cores. The component "mountL" represents a mutex of the structure `layerStore`, which manages layers of images and containers. The component "DeviceSet" represents the mutex of the structure `DeviceSet`, which manages the set of virtual devices of `devicemapper`. The component "Driver" represents the mutex of the storage driver. The component "PortMapper" is the mutex of the structure `PortMapper`, which manages network connection to the default `bridge` network for containers.

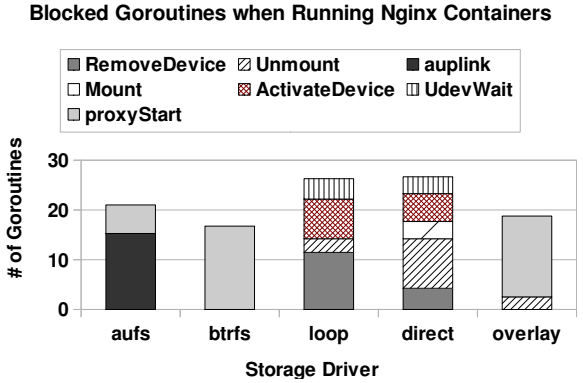

Fig. 10. Mean number of blocked goroutines serving HTTP requests when running thirty-two Nginx containers concurrently in one daemon running on four processor cores. The components represent blocking operations in the critical section. The components "RemoveDevice", "ActivateDevice", "Mount" and "Unmount" are the same as those in Figure 7. The component "auplink" represents a goroutine waiting for completion of an external script `auplink`, which makes copied-up hard-links permanent. The component "UdevWait" represents a call to `libdevmapper` to synchronize with the `udev` daemon. The component "proxyStart" represents a goroutine waiting for a userland proxy that had started.

`loop-lvm` and `direct-lvm` of `devicemapper`, and

2) The mutex of the structure `PortMapper`, which manages the connection to the default bridge network for containers, for `btrfs` and `overlay`.

Since the network driver is independent of the storage driver, these two mutexes do not have dependency. The performance of running containers with `btrfs` or `overlay` is faster than that with `aufs` or `devicemapper`. Thus, when `mountL` of `layerStore` is not a bottleneck, `PortMapper` becomes the next bottleneck.

We can now identify the root causes of the bottlenecks by identifying what the blocking goroutine is doing. Figure 10 shows the breakdown of the blocked goroutines in Figure 8 by the operation that the blocking goroutine is doing. We can see the operations in the critical sections are as follows:

1) Creation and removal of virtual devices with `devicemapper`, as we discussed in Section III-B,
2) Running an external process to clean up copied-up hard links with `aufs`, and
3) Running a proxy process for port forwarding with `btrfs` and `overlay`.

In summary, concurrent running of Nginx containers does not scale due to the bottlenecks in `aufs` and `devicemapper`, and the next bottleneck in the network driver to serialize running of a proxy process for a container.

IV. Container Scalability

Running many containers does not scale, as shown in Figure 3. This time, the direct reason is the increase in the number of busy cycles from 64 to 1,024 containers.

It is also challenging to analyze and optimize a workload that has a flat execution profile, and container management is such a

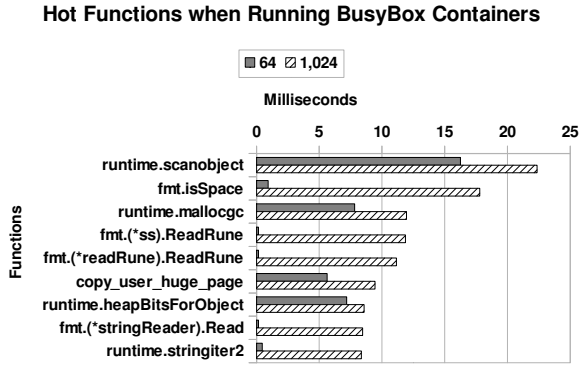

Fig. 11. Ten hottest functions in Docker daemon with `aufs` running on one processor core when running multiple BusyBox containers concurrently. The numbers are those of busy processor cycles required in each function to run one container. The bars are sorted in descending order of time spent with the 1,024 containers.

workload. Figure 11 shows the ten hottest functions in a Docker daemon with `aufs` when running multiple BusyBox containers concurrently. The numbers are those of busy processor cycles (in milliseconds) required in each function to run one container. The hottest function `runtime.scanobject` requires 15 and 6% of the total number of processor busy cycles of the Docker daemon, with 64 and 1,024 containers, respectively. Most of these functions are in the standard library of Go. We can see that the functions for text processing become quite hot with 1,024 containers. Thus, these functions are "bottlenecks" to container scalability, that is, the direct reasons running one container out of many slows down when we increase the total number of containers.

Unfortunately, optimizing each function separately in a flat

execution profile is usually not cost-effective. To obtain insight for more effective optimization, we need to identify the root cause to answer the question, "Why do these functions become hot with many containers?"

A. Hierarchical Analysis of Busy Cycles

We can identify the root cause of the slowdown by applying another top-down approach to analyze the difference in busy cycles of two execution profiles with different total number of containers. The observation regarding Figure 11 suggests that when we increase the total number of containers from 64 to 1,024, it is more likely that the library functions are more frequently invoked, rather than spending longer cycles at each invocation. This leads to the following approach to identify the root cause:

1) Given two execution profiles with different problem sizes, identify the source code of the target application and the topmost layer in the software stack of the application.
2) For each function at the given software stack in the two execution profiles, calculate busy cycles per unit size of the problem.
3) If the same function is significantly slower in one execution profile with a larger problem size, it is likely that the function relates to the root cause, since the identified function is that at the highest software stack.
4) If the busy cycles per unit problem size for all functions in that stack are equivalent between the two profiles, go down into the next stack and repeat from Step 2.

In our case, the topmost layer of the software stack includes the HTTP request handlers, as described in Figure 4. The functions in this layer have the package names starting with `github.com`.

B. Bottlenecks to Running Many Containers

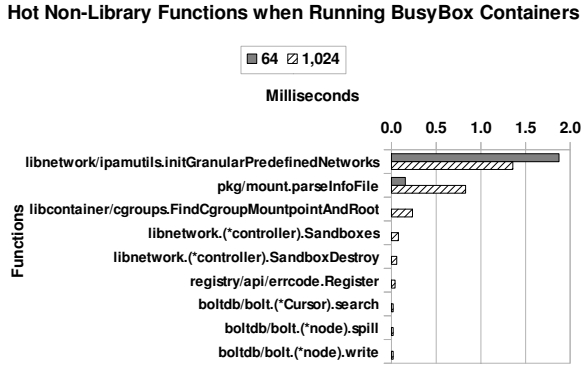

Fig. 12. Top ten hot functions of packages `github.com` in Docker daemon with `aufs` running on one processor core when running multiple BusyBox containers concurrently. The numbers represent those of busy cycles required in each function to run one container. The bars are sorted in descending order of time spent with 1,024 containers.

According to the approach in Section IV-A, we can now explain why the number of busy cycles to run one BusyBox container increases when the total number of containers increases. Figure 12 shows the ten hottest functions in the packages starting with `github.com`. While the first function `initGranularPredefinedNetworks` is hot with both of 64 and 1,024 containers, the second `parseInfoFile` and third `FindCgroupMountPointAndRoot` are hot only with 1,024 containers. The hotness and common functionality of these two functions gives us the insight that parsing information of mount points can be the root cause.

The functions with the package names starting with `github.com` consume 1.1 and 0.6% of the total number of busy cycles of the daemon when running 64 and 1,024 containers, respectively. While their direct contribution is small, their difference between two execution profiles gives us the insight that is not directly available from the difference in the hot functions in all profiles.

C. Root Cause: Mountinfo

While this bottleneck has been identified and fixed in the latest versions of Docker [24], for completeness of discussion, we describe how the bottleneck originates from the administrative interface of the current Linux kernel.

The first problem is that the complete information of mount points is only available in the pseudo text file `mountinfo` on the Process Filesystem. The questions asked are as follows:

1) Where is the mount point of a specific subsystem, such as cgroups and Security-Enhanced Linux (SELinux), which is mounted as a filesystem?,
2) What is the sharing relationship between the global mount namespace and mount namespace of a container?

Since the Linux kernel does not yet provide convenient system calls to answer these questions, we need to parse `mountinfo` sequentially.

The second problem is that the size of `mountinfo` increases proportionally to the number of live containers. Each live container usually appends the following three lines at the tail of `mountinfo`:

1) The mount point of the root filesystem,
2) A pseudo filesystem "nsfs" to represent the network namespace, and
3) A `tmpfs` filesystem to implement the inter-process communication (IPC) namespace.

The `btrfs` storage driver does not append the first item since each subvolume is not mounted. While the mount point of those subsystems can be usually found near the beginning of `mountinfo`, searching information of a container's mount points, searching for a disabled subsystem, or parsing all the lines as a list of objects, incurs cost of parsing text, creating objects, and collecting garbage proportional to the number of live containers.

D. Eliminating Redundant Parsing

We evaluated how container scalability can be improved according to the insight obtained from the bottleneck analysis in Section IV-B.

As described in Section IV-C, the mount point information is used for various purposes and there is no way to obtain

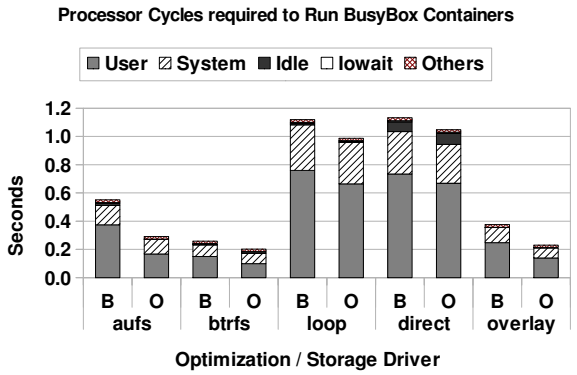

Fig. 13. Processor cycles to run BusyBox container when running 1,024 containers simultaneously in single Docker daemon running on one processor core. The bars labeled "O" represent processor cycles with a Docker daemon optimized to reduce redundant parsing of `mountinfo` according to the approach in Section IV-D. The bars labeled "B" represent the baseline numbers for comparison, which are shown in Figure 3.

one other than parsing `mountinfo`. Thus, according to the fix [24], we modified the client components in the Docker daemon so that they do not parse unused lines in `mountinfo`. More specifically, in the optimized daemon:

- The `aufs` storage driver checks whether the root file system is mounted by checking the filesystem type.
- The library to handle cgroups searches a mount point only for available subsystems.
- The library to handle SELinux quits parsing after it detects the mount point.
- The sharing status between the global mount namespace and mount namespace of a container is checked only after the system call `pivot_root` had failed.

Figure 13 shows a comparison of the processor cycles to run one out of 1,024 BusyBox containers in a single daemon on a single processor core. Reductions in the total number of cycles from the baseline version to the optimized version were 47 and 39% for `aufs` and `overlay`, respectively. Reductions with `btrfs` and `devicemapper` were smaller since they also have different bottlenecks for container scalability.

We also observed that the processor cycles required by the daemon `systemd` also significantly increased with 1,024 containers. With `devicemapper`, it consumed almost 50% of the total busy cycles on the host Linux. Since the `systemd` daemon also monitors the current status of the mount points, this observation agrees with our analysis that the root cause originates from the Linux kernel.

V. Related Work

Prior performance studies on the container technologies [25], [26], [27] mainly focused on showing that containers have smaller overhead than VMs when a single host machine serves multiple workloads:

- Felter et al. [25] compared the performance of Docker containers against the Kernel-based Virtual Machine (KVM).
- Che et al. [26] analyzed three virtualization technologies, i.e., Open Virtuozzo (OpenVZ) using the container technology, Xen using para-virtualization, and the KVM using full virtualization.
- Xavier et al. [27] also analyzed the performance of multiple container technologies, i.e., Linux-VServer, OpenVZ, and Linux Containers (LXC), against the KVM.

However, none of the above analyze bottlenecks in container management.

The performance of VM management has been studied in the context of cloud computing [28], [29], [30]. The primary overhead of *provisioning* a VM on a cloud environment is the throughput of disk and network operations since an image for a VM typically contains the entire files installed on an OS. While block-based copy-on-write technologies are used to reduce the size of an image, they do not achieve the level of reduction comparable to the file-based copy-on-write technologies used by container management systems such as Docker. Another overhead in provisioning is the time to boot a VM, which is much larger than the time to initialize a container.

Our analysis of bottlenecks, in particular of idle cycles for core scalability, is an extension of the approach by Altman et al. [21]. The primary difference is that our analysis is hierarchical, based on a layered performance model [23]. Our target application, a Docker daemon written in Go, uses much larger numbers of idle workers than their target application, a Java EE application server, since a goroutine is a user-level thread but a Java thread is typically a native thread. With a number of irrelevant idle threads, our hierarchical analysis is useful to identify bottlenecks and to obtain insights into their root cause.

VI. Conclusion

Container management is a new workload for OSs since it intensively stresses their administrative interface. The bottlenecks in container management with Docker we identified originate from the performance of administrative operations such as mounting and unmounting a filesystem, creating and removing a virtual block device, inquiring the mount point of a subsystem, and inquiring an attribute of a mount point. We identified them by applying our novel approach to analyze multiple layers of a software stack hierarchically, which includes a Docker daemon implemented as goroutines, which are different from native threads assumed with a current bottleneck analysis tool. We validated our analysis results by applying them to improve the container scalability for Docker.

As Felter et al. [25] summarize in their related work section, OS-level virtualization has a long history, and it is not new. The new point is its usage as a unit to build responsive microservices for agile and continuous development, deployment, and operation. Operating systems have room for improvements to more efficiently host those emerging workloads.

References

[1] "Linux containers." https://linuxcontainers.org/.
[2] Docker, Inc., "Docker Docs." https://docs.docker.com/.

[3] Salesforce.com, "Dynos and the Dyno Manager." https://devcenter.heroku.com/articles/dynos.
[4] Cloud Foundry Foundation, "Warden." https://docs.cloudfoundry.org/concepts/architecture/warden.html.
[5] "Rocket - App Container runtime." https://rocket.readthedocs.org/.
[6] J. Lewis and M. Fowler, "Microservices." http://martinfowler.com/articles/microservices.html.
[7] Ali Hussain, "Performance of Docker vs VMs." http://www.slideshare.net/Flux7Labs/performance-of-docker-vs-vms. presented at CloudOpen North America, August 21st, 2014.
[8] Amazon Web Services, Inc., "Auto Scaling." http://aws.amazon.com/autoscaling/.
[9] Microsoft, "Autoscaling Guidance." https://msdn.microsoft.com/en-us/library/dn589774.aspx.
[10] Google, "Autoscaling Groups of Instances." https://cloud.google.com/compute/docs/autoscaler/.
[11] SoftLayer Technologies, Inc., "Auto Scale." http://knowledgelayer.softlayer.com/topic/auto-scale.
[12] A. Luzzardi, "Scale Testing Docker Swarm to 30,000 Containers." https://blog.docker.com/2015/11/scale-testing-docker-swarm-30000-containers/.
[13] J. Eder, "Comprehensive Overview of Storage Scalability in Docker." http://developerblog.redhat.com/2014/09/30/overview-storage-scalability-docker/.
[14] Nginx, Inc., "nginx news." http://nginx.org/.
[15] "The Go Programming Language." https://golang.org/.
[16] E. Andersen, "BusyBox." https://www.busybox.net/.
[17] The Linux Kernel Organization, Inc., "The Linux Kernel Archives." https://www.kernel.org/.
[18] J. R. Okajima, " Aufs4 – advanced multi layered unification filesystem version 4.x." http://aufs.sourceforge.net/.
[19] "Btrfs." https://btrfs.wiki.kernel.org.
[20] "Device-mapper Resource Page." https://www.sourceware.org/dm/.
[21] E. Altman, M. Arnold, S. Fink, and N. Mitchell, "Performance Analysis of Idle Programs," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pp. 739–753, 2010. doi:10.1145/1869459.1869519 .
[22] Oracle, "Java EE at a Glance." http://www.oracle.com/technetwork/java/javaee/index.html.
[23] J. A. Rolia and K. C. Sevcik, "The method of layers," *IEEE Transactions on Software Engineering*, vol. 21, no. 8, pp. 689–700, 1995. doi:10.1109/32.403785 .
[24] T. Inagaki, "Running a container becomes slower when a large number of containers are alive #20851." https://github.com/docker/docker/issues/20851.
[25] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, pp. 171–172, March 2015. doi:10.1109/ISPASS.2015.7095802 .
[26] J. Che, Y. Yu, C. Shi, and W. Lin, "A synthetical performance evaluation of openvz, xen and kvm," in *Services Computing Conference (APSCC), 2010 IEEE Asia-Pacific*, pp. 587–594, Dec 2010. doi:10.1109/APSCC.2010.83 .
[27] M. Xavier, M. Neves, F. Rossi, T. Ferreto, T. Lange, and C. De Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pp. 233–240, Feb 2013. doi:10.1109/PDP.2013.41 .
[28] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, A. Klepchukov, S. Patil, O. Fox, and D. Patterson, "Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0," 2008.
[29] C. Binnig, D. Kossmann, T. Kraska, and S. Loesing, "How is the weather tomorrow?: Towards a benchmark for the cloud," in *Proceedings of the Second International Workshop on Testing Database Systems*, DBTest '09, (New York, NY, USA), pp. 9:1–9:6, ACM, 2009. doi:10.1145/1594156.1594168 .
[30] Y. Ueda and T. Nakatani, "Performance variations of two open-source cloud platforms," in *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pp. 1–10, Dec 2010. doi:10.1109/IISWC.2010.5650280 .

Appendix A
Experimental Environment

We now describe the measurement environment for the experiments discussed in this paper.

A. Machine

The machine is IBM 2827 zEnterprise EC12 Model HA1 with 16-GB memory. The disk storage is IBM 2421 DS8870 Model 961, connected via four 8-Gb/s Fibre Channel Protocol cards. The network card is OSA-Express5S 1000BASE-T Ethernet Copper.

B. Operating System

The Linux kernel is a custom build of the stable kernel 4.4.4, which is extended with AUFS 4.4 and configured with Red Hat Enterprise Linux (RHEL) 7.2. The userland tools are from RHEL 7.2.

C. Docker

Docker Engine is version 1.10.3 compiled with a port of the Go programming language to the IBM LinuxONE. The graph volume for the storage driver is allocated on a 20-GB direct access storage device.

D. Images

The Dockerfile used to build the Nginx image is a simplified version of that used to build the official image. To build on the s390x architecture, we modified the Dockerfile to install only the main package from the Debian repository instead of the Nginx repository. The BusyBox image is `s390x/busybox` on DockerHub.

When building an image, the base image is already available in the Docker daemon. Similarly, when running a container, the corresponding image is available in the daemon.

E. Measurement

The time spent to execute Docker commands was calculated from the difference in the processor cycles in `/proc/stat` from the start to the end of measurement. The stack dump of goroutines are generated using the `pprof` profiler of the Go programming language. When we calculate the mean number of goroutines, we periodically dump the stacks of all goroutines in the daemon. The busy cycles required in each function were measured using the timer mode of the `oprofile` profiling tool.

Trademarks