FastBuild: Accelerating Docker Image Building for Efficient Development and Deployment of Container

Conference Paper · May 2019

CITATIONS
0

READS
92

4 authors, including:

Song Wu
Huazhong University of Science and Technology
190 PUBLICATIONS **1,626** CITATIONS
SEE PROFILE

Hai Jin
Huazhong University of Science and Technology
901 PUBLICATIONS **6,784** CITATIONS
SEE PROFILE

FastBuild: Accelerating Docker Image Building for Efficient Development and Deployment of Container

Zhuo Huang[1], Song Wu[1]*, Song Jiang[2] and Hai Jin[1]
[1]*National Engineering Research Center for Big Data Technology and System*
[1]*Services Computing Technology and System Lab, Cluster and Grid Computing Lab*
[1]*School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China*
[2]*Department of Computer Science and Engineering, University of Texas at Arlington, Texas 76019, U.S*
[1] {*huangzhuo, wusong, hjin*}*@hust.edu.cn* [2]*song.jiang@uta.edu*

***Abstract*—Docker containers have been increasingly adopted on various computing platforms to provide a lightweight virtualized execution environment. Compared to virtual machines, this technology can often reduce the launch time from a few minutes to less than 10 seconds, assuming the Docker image has been locally available. However, Docker images are highly customizable, and are mostly built at run time from a remote base image by running instructions in a script (the Dockerfile). During the instruction execution, a large number of input files may have to be retrieved via the Internet. The image building may be an iterative process as one may need to repeatedly modify the Dockerfile until a desired image composition is received. In the process, each file required by an instruction has to be remotely retrieved, even if it has been recently downloaded. This can make the process of building, and launching a container unexpectedly slow.**

To address the issue, we propose a technique, named FastBuild, that maintains a local file cache to minimize the expensive file downloading. By non-intrusively intercepting remote file requests, and supplying files locally, FastBuild enables file caching in a manner transparent to image building. To further accelerate the image building, FastBuild overlaps operations of instructions' execution, and writing intermediate image layers to the disk. We have implemented FastBuild. Experiments with images and Dockerfiles obtained from Docker Hub show that our system can improve building speed by up to 10 times, reduce downloaded data by 72%.

***Index Terms*—Docker, container, Image building**

I. INTRODUCTION

Container, such as Docker [16], has become increasingly popular. It is a lightweight kernel virtualization technology as an alternative to traditional virtual machines [1], [2], [13], [14], [19]–[22], [32]. The process of launching a container entails creation of an isolated process space, allocation of required resources, and mounting image data. Because Docker container shares the system kernel with the host machine, its initialization of namespace for virtualization and Cgroup for resource restriction is fast [1], [3], [6], [10], [18], [19]. As a result, launching a container instance can be expected to take only a few seconds, and even as low as less than one second, much faster than that for virtual machines, which may take several minutes. This makes quick deployment of a large number of container instances possible. Such efficiency is especially desired in a DevOps environment where frequent deployments occur. In the meantime, DevOps also requires repeated adjustments and testing of container images. An image is often iteratively re-built before and after a container is launched from it. To support flexible customization of images, Docker adopts a mechanism to build images at the client side. An image can be built by running a Dockerfile, a script containing instructions for building an image, from a base image that provides basic and highly customizable execution environment. Base images are collected at Docker Hub, the official container image registry, and are available for downloading.

Therefore, to truly enable fast launching of Docker containers Docker must allow efficient building of user-customized images. However, even after a base image has been downloaded, image building can be much slower than expected. When instructions in the Dockerfile are executed at a local server, often a large number of input files are required, and these files have to be retrieved from remote servers on the Internet. For example, running instructions in a Dockerfile named "`docker-dev`" available in the Docker Hub to build a Docker image from the base image named "`ubuntu:14.04`" requires retrieval of 486 remote files with a total size of 1.9GB, which takes 283 seconds, or 77% of the entire image building time as measured in our experiments (The experiment setup will be detailed in Section §(II)). Apparently, in many cases when images need to be built before their deployment, remote retrieval of input files is a major source of inefficiency in the image building process. With wide adoption of Dockerfiles for on-site creation of images for deployment and need of iterative image development in the DevOps environment, image building can be on the critical path of launching a Docker instance, and it is important to minimize the expensive file retrieval operations. If the image building can be accelerated, there will be significant benefits: containers can be upgraded instantly [11], [12], [26], [28], [30], image updates can be completed quickly when there is a security issue [3], [14], [35], and users can easily build containerized applications [14], [25], [29], [34].

In this paper we will show that during building of images

there exist a large number of re-used input files, which warrants use of local buffer to cache recently accessed files to avoid unnecessary remote file retrievals. While this approach, named FastBuild, has its substantial potential to improve efficiency of image building, there is a significant challenge in its design. As a new feature of containers, represented by Docker, FastBuild does not run within Docker's instance, while execution of Dockerfile's instructions are within the instance, which generates requests for input files. As Docker does not generate the file requests, it cannot make the critical knowledge on what input files are requested available to FastBuild. Instead, FastBuild has to non-intrusively obtain it for caching files and redirecting requests to the cache should hits are detected.

In this paper we make three major contributions.

1) We extensively study how frequently input files are re-accessed in the building of Docker images and reveal opportunity of maintaining a local file cache for accelerating the process.
2) We propose and design FastBuild, a file caching function seamlessly integrated in Docker to transparently intercept and redirect requests for input files to minimize remote file access. FastBuild ensures that the buffer will supply only up-to-date files. To further accelerate image building, FastBuild overlaps the image writing operation and instruction execution and enables local building of base images.
3) We prototype FastBuild in Docker 17.12 and extensively evaluate its impact on speed of Docker image building. The experiment results show that almost 72% of the duplicated data can be avoided for downloading, improving image building speed by 3.1-7.0 with a high network bandwidth, up to 10.6 times with a low network bandwidth.

The rest of this paper is organized as follows. Section II elaborates the background and motivation of building images for Docker. A solution is in Section III. Section IV presents the design challenge. Section V evaluates the FastBuild, designed based on those findings. Section VI discusses the related work. And Conclusions are in Section VII.

II. BACKGROUND AND MOTIVATION

A. Building of Docker Image

To launch a Docker container instance, a user needs a locally available Docker image. Docker users often use the official Docker container image registry, Docker Hub [15], to obtain and share images. There are two manners for an image to be presented and shared at the hub. One is in the form of a set of fully-built ready-to-use image files. A Docker image is composed of multiple layers by using union mounting technology [33], and each file corresponds to a layer. A image layer file is produced when a user completes his operations in a container instance, and saves the instance by using the "`commit`" command.

The other is in the form of "recipe" and "ingredients". The recipe is named Dockerfile, which is a text file with lines of instructions. Docker provides 17 commands in Docker v17.12 [16] to create various instructions with selected arguments. When one customized Docker image is built, the instructions are executed in container instances line by line. Execution of each line of instruction(s) produces a branch (or a directory) in the instance's overlaid file system, such as Unionfs [33]. Each corresponds to a layer in the container's image. In particular, in the layered image structure, the later generated layers are laid on top of earlier generated layers. And their data, in the form of files in one of instance's directories, overlaps corresponding data in the lower layers. The layer at the bottom is named base image layer, also presented in the Docker Hub as a tar file. In addition to the recipe, there are ingredients to build an image. They consists of the base image and input files. During execution of an instruction in a container instance, a substantial number of input files may be requested. For successfully building of an image, the files have to be available and up-to-date. Current practice is to specify these files' locations at some well-maintained servers on the Internet. This can significantly slow up an image's building time.

The second manner is the most popular way to obtain and share Dockers images as it clearly discloses an image's composition, makes sure the most up-to-date images are generated at user sites, and allows users to conveniently customize an image with easy adjustments, such as editing environment variables or instruction arguments. By this design Docker's containers can be both lightweight and highly configurable. In fact, this is Docker's officially recommended way of obtaining images from its Docker Hub [15].

B. Reducing Remote File Access

While both ingredients are remotely retrieved, it can take a long time period to make image locally available and its container instance ready for use. Image provisioning can be a bottleneck for starting a Docker container. This certainly compromises one of container technology's major advantage, which is fast launching of Docker container instances. Google Borg [17] reveals that file-system provisioning time takes about 80% of the total container deploying time, of which the median is approximately 25 seconds. To ameliorate the issue, Docker has allowed reuse of image layers, as long as the layer to be built is exactly the same as an image layer that is locally available. The reusable layer is identified by knowing that their immediately lower layers are the same and their lines of instruction(s) to generate the next layers are the same. However, when one instruction is modified, the corresponding layer and any layers above it cannot take advantage of the reuse mechanism.

Recognizing the severity of the issue, a recent work, Slacker, attempts to leave most of remote file retrieval and image building out of the critical path of an instant launching process [1]. It sets up a container file system according to an image on a Tintri VMstore server and makes it accessible over the network. To start the image launching process, one only needs a small set of data in an image. Slacker modifies

the Docker image structure by identifying and marking these data, so that only the data will be retrieved to make the instance quickly launched. Additional image data will be lazily pulled from the VMstore only when they are requested. Though this approach can significantly reduce container launching time, it is not compatible with the Docker Hub framework, and thus is not a general-purpose solution. It requires specialized image formats, and the support of a particular VMstore to leverage its file cloning and snapshotting mechanisms. In this work, we aim to provide a general solution that well fits in the Docker's environment by making most input files required in an image building locally available. This is achieved by caching previously used input files.

C. Locality of Input Files

Once a majority of inputs can be accessed locally, building an container image instance would require only a small amount of Internet access and becomes faster. The success of the caching strategy relies on existence of substantial locality in the access of input files for building different images.

To investigate the locality, we pulled 137 most downloaded official basic images by June 21, 2018 (each downloaded more than 100,000 times at Docker Hub [15]). They are categorized into six groups, as shown in Table I. We also collected all verified images (a total of 2746) that are built iteratively based on these base images from Docker Hub, then divided these images into 137 groups according to the base images. Each image group usually comes with 13-30 Dockerfiles, each representing a version of the customized image derived from a base image. For example, the `php` base image has 20 versions such as 5.0, 5.6, and 6.0. There are two scenarios for the locality to occur during the use of the Dockerfiles. One is that a Dockerfile needs to be iteratively re-configured after its initial image building and instant launching for customization to his preferred setup. For example, an instruction in an Dockerfile for changing the network port is `EXPOSE 80`. A user may want to change it to `EXPOSE 8080` network setup, and re-build the image after the editing. In another scenario, a user may need to change the working directory of the container. Accordingly he changes the instruction `WORKDIR /usr/local` to `WORKDIR /var` and re-builds the image. This kind of Dockerfile changes can be minor, and the set of its input files rarely change. At the granularity of image layer, the access locality may not show up at all. However, access of every individual input file in the re-rebuilding is a reuse of a file that has been accessed in the initial building. A very strong locality is expected in the file accesses.

Another representative scenario for the locality to occur is that users need to upgrade his image version or choose a preferred version among a few by repeated testings. To get an idea on how strong the locality can be, for a series of versions of a base image's Dockerfiles we sequentially build the images in the order of the versions (from the lowest version to the highest). We assume existence of a cache to collect input files requested during building of previous versions, and measure the hit ratio of the cache for building the image with the highest version number. Figure 1 shows the hit ratio in terms of file count and amount of data in the hit files for each of the 137 image groups' Dockerfiles. As shown in the figure, there are around 80-90% of the data in the input files are hit in the cache, suggesting that most of the input file data are locally available during building of the image. In another assumed scenario, the median version is built when input files of its four previous versions have been stored in the cache. The hit ratios are shown in Figure 2. As shown, with input files for building only four previous versions of images are available in the cache, the hit ratio in terms of data amount is around 60-80%, which is also significant. We also measure overlap of input files for building two Dockerfiles' images from different groups. We find that even in these unrelated images, there is often around 30% overlap of input data. Apparently for a

TABLE I
137 MOST POPULAR BASE IMAGES IN THE DOCKER HUB CATEGORIZED IN SIX DIFFERENT USE GROUPS

Categories	Images' Name
Linux Distro	alpine, busybox, amazonlinux, centos, mageia, oraclelinux, cirros, crux, debian, fedora, opensuse, neurodebian, ubuntu, ubuntu-debootstrp, ubuntu-upstart, ros, clearlinux, sourcemage, photon
Database	influxdb, arangodb, orientdb, cassandra,elasticsearch, memcached, crate, neo4j, mariadb, couchdb, couchbase, mongo-express, mysql, percona, postgres, redis, rethinkdb, mongo, aerospike
Language	clojure, gcc, groovy, bash, golang, jruby, python haskell, hylang, java, openjdk, julia, mono, perl, php, pypy, r-base, rakudo-star, ruby, thrift, ibmjava, swift, haxe, julialang, fsharp, erlang, elixir, rust,
Web Component	nginx, kong, glassfish, tomee, websphere-liberty, httpd, iojs, jetty, php-zendserver, tomcat, django, kibana, node, rails, storm, rapidoid, lightstreamer, redmine, haproxy, telegraf, kapacitor, zookeeper, flink, irssi, celery, hipache, notary, traefik, eclipse-mosquitto,
Application Platform	mediawiki, known, nuxeo, gazebo, backdrop, xwiki plone, geonetwork, convertigo, ghost, joomla, rocket.chat, wordpress, drupal, mextcloud, owncloud, jenkins, rabbitmq, odoo, nats, nats-streaming, solr, sonarqube, piwik,
Other	znc, bonita, eggdrop, spiped, composer, docker, gradle, kaazing-gateway, vault, registry, logstash, swarm, buildpack-deps, maven, docker-dev, voice-gateway, chronograf, sentry

server where many images have been built, it is likely that most of input files required for building a new image can be locally available if a cache is maintained to store all the history input files.

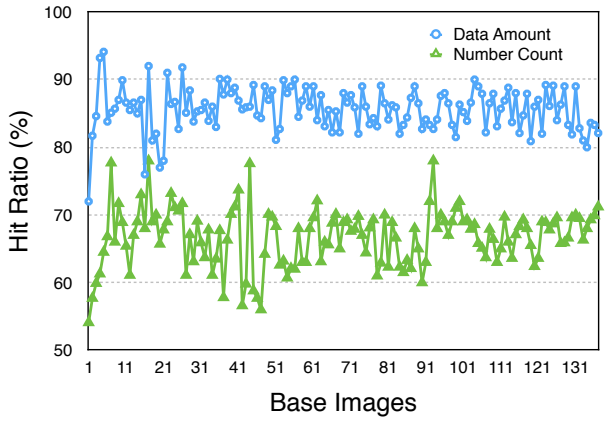


Fig. 1. *Hit ratios in terms of file count and data amount for the highest version among all versions for each of the 137 base images' Dockerfiles*

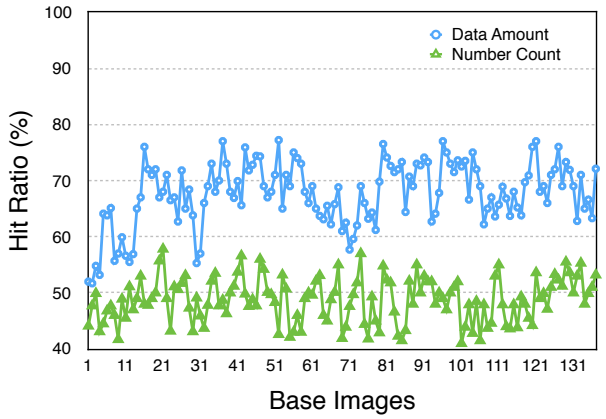


Fig. 2. *Hit ratios in terms of file count and data amount for the median version among all versions for each of the 137 base images' Dockerfiles assuming input files for building its four previous versions of images are in the cache*

To understand implication of substantial reuse of the input files, we measure and compare the time used for building an image and that for retrieving all required remote input files. As shown in Figure 3, consistently more than 60%, 71% by average, of the image building time is spent on the remote input file access. By removing a majority of the remote file access, we expect a significant improvement of the image building time.

III. DESIGN OF FASTBUILD

We use our findings to design a new caching component, named FastBuild, that can seamlessly fit into the Docker framework to enable fast image building without requiring any special hardware supports or modification of stock Docker images. FastBuild includes three efforts on accelerating the process of image building. First, it uses a fine-grained caching mechanism. Instead of using image layer caching, FastBuild caches individual input files to make full use of local image data aiming to significantly remove remote file retrieval time from image building time. Second, FastBuild can overlap operations of instructions' execution to further accelerate the image building. Third, instead of pulling base images from Docker Hub, FastBuild can quickly build base images locally by taking advantage of locally available input files.

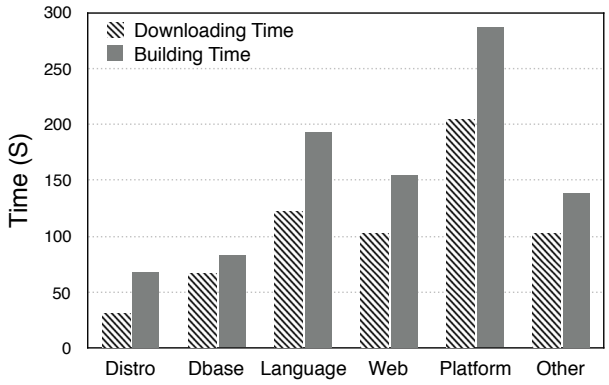


Fig. 3. *Comparison of the times for image building and for retrieving remote input files. Each time is an average over all the times for different Dockerfiles versions of different base images in a base image group*

IV. THE DESIGN CHALLENGE

While the idea of setting up a local cache for collecting input files when they are retrieved from the Internet and then supplying them locally when they are requested again for image building is straightforward, a major challenge is to implement it in the existing Docker's framework in a non-intrusive manner. In particular, the solution should not require any changes of content and format of Docker images, and Dockerfiles, layered union file system, as well as architecture of the image building process.

The key technique is to obtain requests for input files and re-direct them to FastBuild for processing transparently in the existing image building process. The Docker image building process is orchestrated by the Docker Daemon, which retrieves the base image and a Dockerfile. It will build the image layer by layer, starting from the base image. A new layer is built on top of an existing layer, which corresponds to an image file. To build the new layer, the Daemon creates a container instance using the image layer file. It also reads the corresponding line of instruction(s) and runs the instruction(s) in the container. For example, if the line is "`RUN apt-get install gcc`" for installing gcc software packages, Docker Daemon then sends the command "`apt-get install gcc`" to the container and runs it. Requests for input files are generated in the container executing the instruction. When the execution is completed, the Daemon commits the container to an image file about the new layer and the container is destroyed. These steps are repeated for each line of the Dockerfile to produce the corresponding image layer.

We can see that Docker Daemon is only responsible for initializing an instruction execution environment (the container

instance) and writing back an image layer. It does not know what input files are requested in the execution.

To obtain the knowledge about input files, one seems to have to understand and be involved in the instruction execution. However, once a facility is built into the execution environment, it would not be a generic solution, and it may leave footprints in the file system and pollute contents of the resulting image file. To address the issue, we need to be minimally involved in the container environment as long as the requests for input files can be obtained. Therefore, FastBuild only has its presence in the container's network component.

A. Interception of Requests for Input Files

A Docker container isolates its resources into different Linux namespaces, including Mount, UTS, IPC, PID, User, and Network namespaces. Among them, the Network namespace that isolates the container's virtual network device and IP address from the server's physical network device. By having a FastBuild process in a namespace that is independent of other namespaces, such as the Mount namespace for isolation of file system, FastBuild cannot access the image data in the file system.

FastBuild has one daemon process in an entire system and communicates with the Docker Daemon. Whenever Docker Daemon launches a container to build an image, FastBuild's daemon generates the aforementioned process and places it to the container's Network namespace. To this end, FastBuild obtains the container's name and resolves the name to get the id of the main process (`init`) in the container by using the Docker API interface (Step 1 in Figure 4). By reading the process's `proc` file system (Step 2), FastBuild knows the container's Network namespace. It then forks a child process and attaches it to the namespace by calling the `setns()` system call (Step 3).

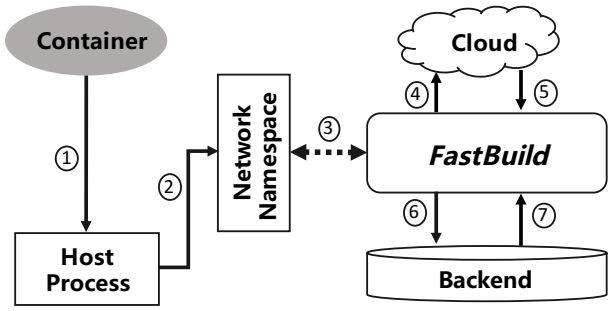


Fig. 4. *Steps for FastBuild to redirect requests for input files and to access the cache*

Requests for input files generated by the instruction execution in a container are sent to the virtual network device file (`veth0`) in its Network namespace. The FastBuild child process then intercepts the network requests by listening to the file using system library `libpcap`, and passes them to the FastBuild daemon process running outside of the container network namespace to serve requests. Sending data between FastBuild's daemon and child processes is implemented through a shared Unix pipe file.

When the daemon process receives the redirected requests, it gets requested files' URLs. For each requested file, it then searches FastBuild's local cache to see whether it is in the cache (Steps 6-7). If it is, FastBuild will first read its last modification time. It also generates a request for the file's last modification time at the server on the Internet specified by its URL (Steps 4-5). Only when the two modification times match, indicating the file in the cache is up to date, will the file be retrieved from the cache and sent to the child process, which will deliver it to the process running the instruction(s) in the container using `libpcap`. If the file is not in the cache or the modification times do not match, the file is remotely retrieved as usual. After receiving the file, FastBuild daemon stores it into the local cache, possibly overwriting the out-of-date file in the cache, and delivers it via the child process.

B. Overlapping Instruction Execution and Image Commitment

With a local cache for supplying input files from the local disks, the image building process becomes potentially much faster. With the reduced time of instruction execution, another operation, which is to commit the data to the disk as a new image file, can become substantial. In addition, the average number of line of instruction(s) in a Dockerfile in Docker Hub, or number of layers in an image, is large (about 23). It takes 11.6% of the total building time on average to repeatedly launch and shut down containers, each of an image layer.

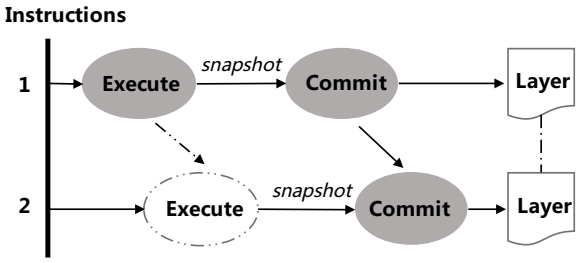


Fig. 5. *Overlapping instruction execution and image commitment operations and building multiple layers of an image in one container instance*

To further reduce image building time, FastBuild overlaps the instruction execution and image commitment times, and uses only one container instance throughout building an image with a Dockerfile. To this end, FastBuild daemon intercepts the `commit` signal, which is sent from Docker Daemon upon completion of a line of instruction(s) in the Dockerfile. After this FastBuild daemon takes over the image building, it accesses the Dockerfile and issues the remaining lines of instructions to the container for execution and commitment.

As execution of the next instruction will be in parallel with the image commitment, the daemon first makes a snapshot of the data in the top layer (read-write layer) of the container's union file system by using `mksnap_ffs` command. To send the next instruction to the container for execution, FastBuild identifies all namespaces, as it does for the Network namespace, and sets the `/proc/self/exe` argument to execute the instruction. In the meantime, the last layer of image is committed, as illustrated in Figure 5. Instruction execution

can proceed without being synchronized with the commitment. Although instructions are executed at the same layer of the union file system, commitment of each layer would be the snapshot.

C. Quickly Obtaining Base Image

Building one Docker image requires a base image. Docker Hub hosts various base images as compressed files for faster downloading. However, it needs to be decompressed after downloading at a user's server. For a typical base image size of 500MB, it may take around 5 seconds for the decompression with the CPU being fully occupied. With the previously described optimization techniques (caching input files and overlapped instruction execution/image commitment), a base image can be made faster by leveraging the techniques. Fortunately, in Docker Hub each base image comes with its Dockerfile. FastBuild can download the Dockerfile, instead of the base image itself, and build the image locally more efficiently.

V. EVALUATION

In this section, we evaluate performance of the FastBuild system, in particular, the performance impact of techniques of caching input files and overlapping instruction execution and image commitment operations. All the performance measurements are made on a host machine with 2.3 GHz Xeon CPUs(E5-2620), 64GB RAM and one Intel Gigabit CT PCIE Network Adapter (EXPI9301CTBLK), and a hard disk (West Digital WD60PURX) with its write and read throughput of 190MB/s and 240MB/s, respectively. All images and Dockerfiles were obtained from the official Docker Hub and during the time period from 12/5/2018 to 12/8/2018 when the experiment measurements reported in the paper were collected. The server was connected to the Internet via China Education and Research Network (CERNET) with a measured download bandwidth of 930.23Mbps and upload bandwidth of 820.08Mbps.

FastBuild is prototyped by instrumenting source code of Docker 17.12. FastBuild is implemented using about 2600 lines of Go code. Among them, approximately 300 LoC are in the Docker server module for redirecting the Dockerfile instructions. About 500 LoC are responsible for optimizing container runtime in Docker core engine module. Rest of the code is for cache lookup and matching. In the implementation, to obtain environment variable info such as `init` process id and namespace id, we use Linux's system syscalls, rather than Docker's APIs for similar functionalities, for higher portability as Docker's API is likely to be updated.

A. Build and Launch Time

Each of the 137 popular base images listed in Table I, has a set of about 13-30 Dockerfiles. Each Dockerfile in a set has a version number. For a set of the Dockerfiles, in the order of their version numbers one at a time, we build an image by executing a Dockerfile and launching the corresponding container. In the experiments about FastBuild the base image is also built using its Dockerfile, instead of directly downloaded, to offer FastBuild with more opportunity to optimize. For the stock Docker, base images are downloaded. At the beginning of the series of experiments for a base image's Dockerfiles, we always clear the cache to give it a clean start. Figure 6 shows the aggregate build and launch times for the sequence of Dockerfiles in a set for a base image with the stock Docker and with FastBuild for each of the 137 image groups. To understand contribution of FastBuild's individual techniques, the figure also shows the times for FastBuild with only input file caching and that with only overlapping of instruction execution and image commitment in addition to the full FastBuild.

As shown, though the time for different image group's Dockerfiles can be very different (from 63 seconds to 208 seconds), FastBuild consistently takes about only 25% of the time, or about 4X faster. For example, for the base image `Nginx` with 27 Dockerfiles, the total build and launch time for the stock Docker and FastBuild are 198 and 51 seconds, respectively. The improvements are significant. In terms of individual techniques' contributions, the figure shows that caching contributes about 2/3 of the improvement and overlapping contributes the remaining.

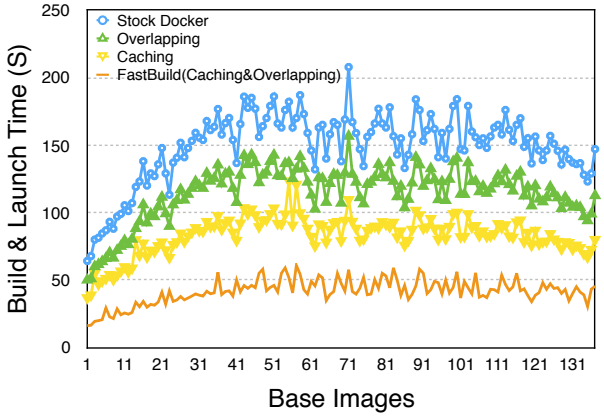


Fig. 6. *The build and launch time for each sequence of Dockerfiles associated with a base image. The times for each of the 137 base images are shown for the stock Docker, FastBuild with only overlaping technique, FastBuild with only caching technique, and FastBuild with both techniques. The base images are shown in the order as they appear in Table I*

To further understand the contribution made by removing remote access of input files, we show the build and launch time for each Dockerfile in the set of Dockerfiles with the `zookeeper` image. The experiment is carried out in order of their version numbers, so are the times presented in the Figure 7. As expected, time reduction (or the time gap between the two curves in the figure) keeps increasing. When more images are built, more input files are added into the cache. When the cache warms up, it has a higher hit ratio and removes more remote file access out of the critical path. Another observation is that major portion of the performance benefit from caching is achieved after execution of only a few Dockerfiles. For example, after execution of six Dockerfiles, FastBuild is 3.2X faster than the stock Docker. But after execution of twenty-

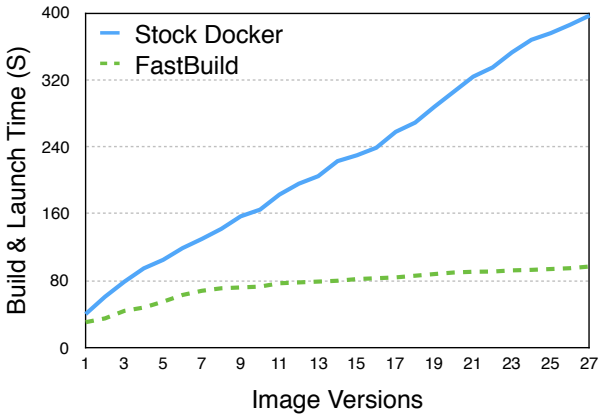


Fig. 7. *The build and launch times of the stock Docker and FastBuild for the set of Dockerfiles associated with the* `zookeeper` *image*

six Dockerfiles it is only 4X faster. This indicates that the caching can become effective on the execution of a Dockerfile after building only a few Dockerfiles belonging to the same base image, which is a common scenario in the use of Docker containers. To confirm this, in Figure 8 we further show the time for the last Dockerfile in each sequence of Dockerfiles associated with each of the 137 base files when only 1, 2, 4, 6, 8, or 10 Dockerfiles preceding to the last one have been executed to warm up the cache. While only one or two previous executions are not sufficient, execution of six or more previous versions of Dockerfile obtains more than 90% of the benefit. This is an encouraging result, demonstrating that FastBuild's performance advantage can be easily realized. Note that if a Dockerfile is executed again after a small change, the second execution is likely to receive almost of its requested input files from the cache.

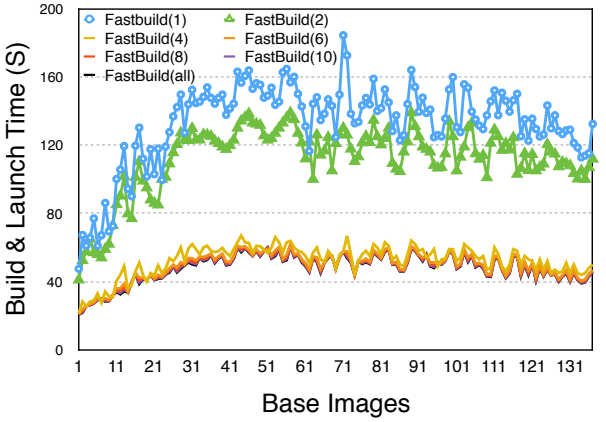


Fig. 8. *The build and launch time for the last Dockerfile in each sequence of Dockerfiles associated with each of the 137 base files when only 1, 2, 4, 6, 8, or 10 Dockerfiles preceding to the last one have been executed.*

B. Remotely Downloaded Data

In the stock Docker, building an image needs to download base images and input files. With FastBuild, base image is not downloaded any more, and only input files missing in the cache are. Figure 9 shows the amount of data downloaded in the stock Docker and FastBuild for executing each sequence of Dockerfiles with each of the base images. As mentioned Docker supports image-layer caching. A image-layer file can be re-used (without rebuilding it) if both the lower-layer image and the instruction for building the current layer match. However, such a reuse may be unsafe as the execution outcome may also rely on the execution environment (e.g., `RUN apt-get update`) which may change. Using an out-of-date image layer may lead to unexpected execution behavior. To this end, some users may opt to turn off the Docker's caching option. Therefore, Figure 9 also includes results for Docker without the caching function.

As shown in Figure 9, FastBuild can significantly reduce amount of remotely downloaded data. This reduction is positively correlated to the improvement of the container's build and launch time shown in Figure 7. The average amount of downloaded data for building one image with the stock Docker is 784.6 MB. If its image-layer caching function is turned off, the amount increases to 1073.2MB. When FastBuild is used, the amount reduces to only 220.5MB, representing 71.9% and 79.5% reduction, respectively.

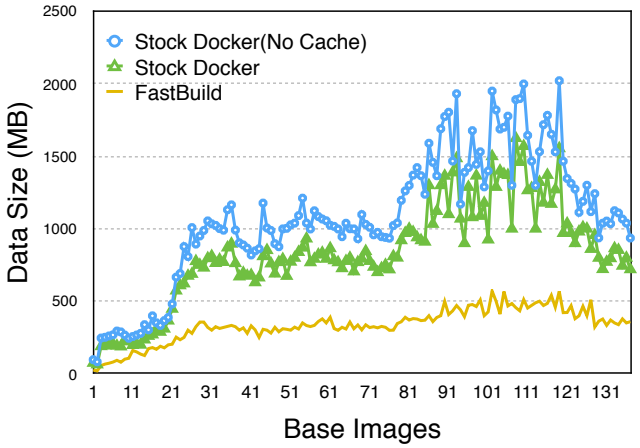


Fig. 9. *The amount of data downloaded in the stock Docker and FastBuild for executing each sequence of Dockerfiles with each of the 137 base images. For the stock Docker, results for turning off its option of image-layer caching is turned off are also presented as "Stock Docker NoCache".*

Figure 10 shows the amount of remotely downloaded data when the last Dockerfile in each base image's Dockerfile sequence is executed assuming different number of preceding Dockerfiles have been executed. These results correspond to those about build and launch time presented in Figure 8. Results in the two figures are consistent. As long as there are six or more previous versions of the Dockerfile have been built to warm up the cache, a majority amount of downloaded data can be obtained locally. The high ratio indicates high redundancy among input files of building different images, which suggests that space demand on the cache is limited. For example, after we build all eight Dockerfiles associated with the popular base files listed in Table I, the cache size is 9.59GB. Considering the abundant capacity in today's disks,

we do not impose a size limit on the cache. If indeed there is such a need, it will be a minor effort by applying a replacement algorithm such as LRU to enforce a space limit.

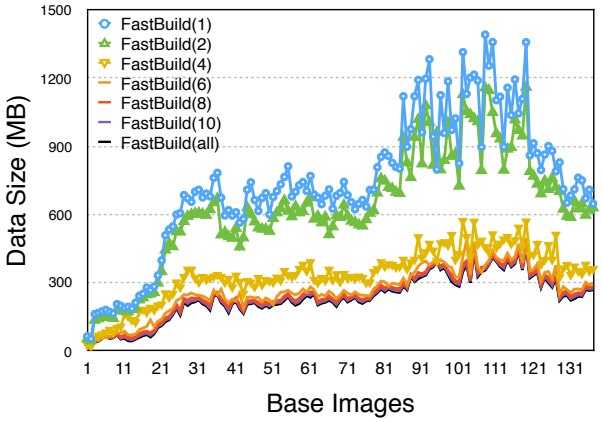
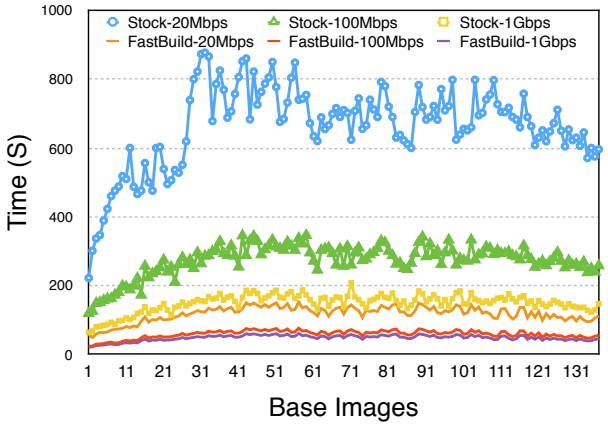


Fig. 10. *The amount of data remotely downloaded during building the last Dockerfile in each sequence of Dockerfiles associated with each of the 137 base files when only 1, 2, 4, 6, 8, or 10 Dockerfiles preceding to the last one have been executed.*

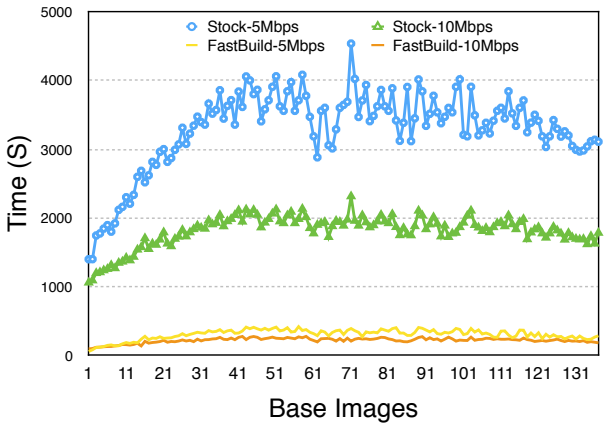
C. Impact of Network Bandwidth

For a software system whose performance heavily depends on the Internet performance, the network can have a major impact. Current process of building and launching of a Docker image is sensitive to the network performance, as often hundreds of megabytes or even a few gigabytes of data for input files and a base image have to be retrieved from the Internet. The environment where a Docker image is built and launched varies. It can be a well-equipped data center with 1 Gbps or multi-Gbps Internet access. It may be a small lab or office environment with under 1Gbps bandwidth. It can even be mobile computing devices at the edge of Internet with the bandwidth usually less than 20Mbps. Actually with increasingly more applications hosted on smartphones and accelerated development of the Internet of Things, the container-based lightweight virtualization technology is gradually being adopted in the devices [42]–[44]. Accordingly mobile devices with limited Internet access bandwidth can be a common use environment for the container techniques. Therefore, we evaluate the bandwidth's impact on FastBuild's relative performance advantage by varying the speed of the network. The actual bandwidth is around 1Gbps. To vary the bandwidth, we use the `tc` command to limit the bandwidth on the server's network device.

Figure 11(a) shows the build and launch time for a sequence of Dockerfiles associated with a base image. As expected, with a higher bandwidth, the impact of the remote file downloading is reduced and the stock Docker's time becomes smaller. FastBuild's relative performance accordingly reduces. For example, in terms of the time averaged over all the base images, FastBuild is 3.1X faster than the stock FastBuild at the 1Gbps network. In contrast, it is 4.9X and 7.0X faster at the 100Mbps and 20Mps bandwidths, respectively. If very low bandwidths (10Mbps or 5Mbps) are assumed (for mobile devices), FastBuild can be 8.2X and 10.6X faster, respectively. For the 5Mbps case, FastBuild reduces the average build and launch time from one Dockerfile from 651 seconds to 62 seconds. This makes use of lightweight containers a viable choice on bandwidth-constrained devices or Internet-limited use environments.



(a) Higher Network Bandwidth



(b) Low Network Bandwidth

Fig. 11. *Build and launch time for a sequence of Dockerfiles associated with each of the 137 base images with different network speeds*

VI. RELATED WORK

As container is designed to be a lightweight virtual machine technology, its performance is more sensitive to the cost of downloading images and other data from the network. Accordingly substantial efforts have been made to reduce the cost to make it truly lightweight.

Among the efforts, Slacker aims to move cost of downloading image data out of the critical path of launching a container as much as possible [1]. By observing that only a small set of data are needed to launch a container instance and much of other data are required only after the instance starts to run, Slack modifies the image format and marks the essential set of data. With a support of dedicated image server where a full union file system associated each image is maintained, Slacker leverages the lazy downloading technique to significantly reduce the time for an instance to become available.

However, future data access may become slower as they will be on-demand retrieved from the Internet. This technique is not compatible with the Docker Hub framework and requires support of specialized image server. In contrast, FastBuild is fully compatible with the existing Docker container build and launch framework, and can be readily adopted by existing users. Furthermore, as most input files can be locally available, especially if they are also buffered in the memory, the on-line access of image data can be very fast and the lazy access technique becomes less necessary.

The Docker Hub can become a performance bottleneck when many users attempt to download images from it simultaneously. The FID scheme introduces a *peer-to-peer* (P2P) technique to make downloaded images be shared across users. However, when using Dockerfiles to build images on site is the preferred manner to launch containers, most of downloading is for input files, rather than the base images and Dockerfiles, and cannot be accelerated. Similarly, to ameliorate the performance at Docker Hub, the Anwar scheme prefetches base images that are likely to be accessed from the disk to the memory, so that they can be quickly supplied [36]. In contrast, with FastBuild only Dockerfiles, which are very small, are downloaded from the Docker Hub, making it less likely to be a performance bottleneck. In addition, it avoids downloading input files via Internet.

As an extension of Docker's caching function for image layers, the CoMICon [37] scheme makes the layers be shared across servers in a cluster. By leveraging the cluster as a larger and cooperative cache more data can be retrieved from the local cluster. The Wharf scheme adopts a similar strategy [46]. It is a middleware in Docker to allow Docker image layers to be shared among different Docker Daemons in a distributed storage system. However, as we have shown, the hit ratio in terms of image layer is usually low. Once one layer is modified, all layers above it will be different. In contrast, FastBuild exploits locality at the granularity of individual input files, and makes rich opportunity of data reuse available to enable fast local data access.

Because building an image from a base image can be slow due to reasons such as downloading many input files, the CNTR scheme separates core functionality data from the image into a slim image, and leaves the remaining data in a fat image [41]. In this way it can build a container from a slim image, which can be much faster. However, if the fat image is modified, the entire image has to be rebuilt from the base image and the separation operation has to be conducted again. In contrast, effectiveness of FastBuild only depends on existence of locality of input files, which has been demonstrated to be common.

VII. CONCLUSIONS

To retain container's major advantage of being lightweight, we propose FastBuild, a technique to transparently accelerate Docker image's building and launching process. An in-depth analysis of the set of popular images in Docker Hub shows that input files used in the image building represent a significant build cost and are also likely to be reused. FastBuild is motivated by the observation. Without changing the Docker's system architecture, it maintains a local cache and non-intrusively intercepts and redirects request for the input files to the cache. With real-world Docker base images and Dockerfiles, we experiment with a prototype FastBuild system and show that a significant percentage of remote access of input files can be removed once a few previous versions of the image have been built to warm up the cache. FastBuild also incorporates the technique for overlapping execution of Dockerfile instructions and writing back image layers to further improve efficiency of image building. Experiment results show that compared with the stock Docker, FastBuild can speed up the build and launch process by up to 10x and remove 72% of remote data downloading for the most popular images at Docker Hub.

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their tremendous feedback. This paper is based upon work supported by National Key Research and Development Program under grant 2016YFB1000501, National Science Foundation of China under grant No.61732010.

REFERENCES

[1] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Slacker: fast distribution with lazy docker containers," in Proceedings of USENIX Conference on File and Storage Technologies (FAST'16), pp. 181–195, 2016.

[2] M. Dirk,"Docker: lightweight linux containers for consistent development and deployment,", Linux Journal, 2014, pp.2.

[3] Z. Jun, Z. Jiang, and X. Zhen, "Twinkle: A fast resource provisioning mechanism for internet services," in Proceedings of IEEE International Conference on Computer Communications (INFOCOM'11), pp. 802–810, 2011.

[4] S. Constantine, C. Ramesh, P. Ben, C. Jim, L. Monica, and R. Mendel, "Optimizing the migration of virtual computers," in ACM SIGOPS Operating Systems Review, vol. 36, pp. 377–390, 2012.

[5] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. C. Arpaci-Dusseau, and R.H. Arpaci-Dusseau, "SOCK: Rapid Task Provisioning with Serverless-Optimized Containers," in Proceedings of USENIX Annual Technical Conference, 2018.

[6] N. Bogdan, B. John, K. Kate, and A. Gabriel, "Going back and forth: Efficient multideployment and multisnapshotting on clouds," in Proceedings of International Symposium on High Performance Distributed Computing (HPDC'11), pp. 147–158, 2011.

[7] Q. Chen, L. Liang, Y. Xia, H. Chen, and K. Hyunsoo, "Mitigating Sync Amplification for Copy-on-write Virtual Disk," in Proceedings of USENIX Conference on File and Storage Technologies (FAST'16), pp. 241–247, 2016.

[8] K. Jayaram, C. Peng, Z. Zhang, M. Kim, H. Chen, and H. Lei, "An empirical analysis of similarity in virtual machine images," in Proceedings of the Middleware Industry Track Workshop, 2011.

[9] A. Sergei, T. Bohdan, G. Franz, K. Thomas, M. Andre, P. Christian, L. Joshua, M. Divya, O'K. Dan, and S. Mark, "SCONE: Secure Linux Containers with Intel SGX," in Proceedings of 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16), vol. 16, pp. 689–703, 2016.

[10] H. Stephen, D. Ayush, L. Aaron, M. Sean, M. Jose, Y. Yang, S. R. Seelam, and T. Michela, "Resource Management for Running HPC Applications in Container Clouds," in Proceedings of International Conference on High Performance Computing (HPC'16), pp. 261–278, 2016.

[11] R. Kaveh and K. Thilo, "Scalable virtual machine deployment using VM image caches," in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'13), 2013.

[12] C. Peng, M. Kim, Z. Zhang, and H. Lei "VDN: Virtual machine image distribution network for cloud data centers," in Proceedings of IEEE International Conference on Computer Communications (INFOCOM'12), pp. 181–189, 2012.
[13] D. Rajdeep, A.R. Raja, and K. Dharmesh, "Virtualization vs containerization to support paas," in Proceedings of 2014 IEEE International Conference on Cloud Engineering (IC2E'14), pp. 610–614, 2014.
[14] B. Tak, C. Isci, S. Duri, N. Bila, S. Nadgowda, and J. Doran, "Understanding Security Implications of Using Containers in the Cloud," in Proceedings of 2017 USENIX Annual Technical Conference (ATC'17), Santa Clara, CA, pp. 313–319, 2017.
[15] "Docker Hub," https://hub.docker.com, 2018.
[16] "Docker Docs," https://docs.docker.com, 2018.
[17] V. Abhishek, P. Luis, K. Madhukar, O. David, T. Eric, and W. John, "Large-scale cluster management at Google with Borg," in Proceedings of European Conference on Computer Systems (EUROSYS'15), 2015.
[18] V. Ben, G. Anoop, and R. Mendel, "Performance isolation: sharing and isolation in shared-memory multiprocessors," in ACM SIGPLAN Notices, pp. 181–192, 1998.
[19] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, L. Timoteo, D. Rose, and A. F. Cesar, "Performance evaluation of container-based virtualization for high performance computing environments," in Proceedings of 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing, pp. 233–240, 2013.
[20] P. Max, F. Lena, and P. Andreas, "A Performance Survey of Lightweight Virtualization Techniques," in Proceedings of European Conference on Service-Oriented and Cloud Computing (SOCA'17), pp. 34–48, 2017.
[21] B. David, "Containers and Cloud: From LXC to Docker to Kubernetes," in Proceedings of IEEE Cloud Computing, pp. 81–84, 2014.
[22] M. Roberto, K. Jimmy, and K. Miika, "Hypervisors vs. lightweight virtualization: a performance comparison," in Proceedings of 2015 IEEE International Conference on Cloud Engineering (IC2E'15), pp. 386–393, 2015.
[23] T. K. Kuppusamy, T. Santiago, D. Vladimir , and C. Justin, "Diplomat: Using Delegations to Protect Community Repositories," in Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI'16), pp. 567–581, 2016.
[24] A. Gulati, I. Ahmad, and A. W. Carl, "PARDA: Proportional Allocation of Resources for Distributed Storage Access," in Proceedings of USENIX Conference on File and Storage Technologies (FAST'09), 2009.
[25] W. Dietz, J, Cranmer, N. Dautenhahn, and S. A. Vikram, "Slipstream: Automatic Interprocess Communication Optimization," in Proceedings of USENIX Annual Technical Conference (ATC'15), 2015.
[26] R. Joshua, L. Oren, B. Eli, S. Alex, M. Vishal, N. Jason, and R. Dan, "VMTorrent: scalable P2P virtual machine streaming," in Proceedings of International Conference on Emerging Networking Experiments and Technologies (CoNEXT'12), pp. 289–300, 2012.
[27] H. Yang, M. Song, and T. Li "Towards Full Containerization in Containerized Network Function Virtualization," in Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17), pp. 467–481, 2013.
[28] A. Samer, S. Dinesh, S. Prasenjit, and R. Matei, "VMFlock: virtual machine co-migration for the cloud," in Proceedings of International Symposium on High Performance Distributed Computing (HPDC'11), pp. 159–170, 2011.
[29] M. Hibler, L. Stoller, J. Lepreau, R. Ricci, and C. Barb, "Fast, Scalable Disk Imaging with Frisbee," in Proceedings of USENIX Annual Technical Conference (ATC'03), 2003.
[30] W. Romain, C. Tony, M. Belmiro, R. Ewan, G. Manuel, G. Sebastien, and S. Ulrich, "Image distribution mechanisms in large scale cloud providers," in Proceedings of 2010 IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom'10), 2010.
[31] O. Steven, S. Dinesh, S. Gong, and N. Jason, "The design and implementation of Zap: A system for migrating computing environments," in ACM SIGOPS Operating Systems Review, vol. 36, pp. 361–376, 2002.
[32] S. Stephen, P. Herbert, M. E. Fiuczynski, B. Andy, and P. Larry, "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors," in ACM SIGOPS Operating Systems Review, vol. 41, pp. 275–287, 2007.
[33] "AUFS," http://aufs.sourceforge.net/aufs.html, 2018.
[34] P. John, "Nimda Worm Shows You Cant Always Patch Fast Enough," in Gartner FirstTake (FT-14-5524), vol. 19, 2001.
[35] S. David, and M. D. Ernst, "An experimental evaluation of continuous testing during development," in ACM SIGSOFT Software Engineering Notes, vol. 29, pp. 76–85, 2004.
[36] A. Anwar, M. Mohamed, V. Tarasov, M. Littley , L. Rupprecht, Y. Cheng, N. Zhao, D. Skourtis, A. Warke , H. Ludwig, D. Hildebrand, and A. R. Butt, "Improving Docker Registry Design Based on Production Workload Analysis," in Proceedings of USENIX Conference on File and Storage Technologies (FAST'18), 2018.
[37] S. Nathan, R. Ghosh, T. Mukherjee, and K. Narayanan, "CoMICon: A Co-Operative Management System for Docker Container Images," in Proceedings of 2017 IEEE International Conference on Cloud Engineering (IC2E'17), pp. 116-126, 2017.
[38] "Kubernetes," https://kubernetes.io, 2018.
[39] "Mesos," http://mesos.apache.org, 2018.
[40] "Swarm," https://docs.docker.com/engine/swarm, 2018.
[41] J. Thalheim, P. Bhatotia, P. Fonseca, and B. Kasikci, "Cntr: Lightweight OS Containers," in Proceedings of USENIX Annual Technical Conference (ATC'17), 2018.
[42] N. Chau and S. Jung, "Dynamic analysis with Android container: Challenges and opportunities," in Digital Investigation, 2018.
[43] R. Morabito, "Virtualization on Internet of Things Edge Devices With Container Technologies: A Performance Evaluation," in IEEE Access, vol. 5, pp. 8835-8850, 2017.
[44] K. Lee, Y. Kim, and C. Yoo, "The Impact of Container Virtualization on Network Performance of IoT Devices," in Mobile Information Systems, 2018.
[45] D. Abts, B. Felderman, "A Guided Tour of Datacenter Networking," in Communications of the ACM, vol. 55, pp. 44–51, 2012.
[46] C. Zheng, L. Rupprecht, V. Tarasov, D. Thain M. Mohamed, D. Skourtis, A. Warke, and D. Hildebrand, "Wharf: Sharing Docker Images in a Distributed File System," in Proceedings of ACM Symposium on Cloud Computing (SoCC'18), 2018.
[47] K. Wang, Y. Yang, Y. Li, H. Luo, and L. Ma, "FID: A Faster Image Distribution System for Docker Platform," in Proceedings of 2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W'17), pp. 191-198, 2017.