Carving Perfect Layers out of Docker Images

Dimitris Skourtis
IBM Research

Lukas Rupprecht
IBM Research

Vasily Tarasov
IBM Research

Nimrod Megiddo
IBM Research

Abstract

Container frameworks such as Docker have changed the way developers package, share, and deploy applications. *Container images* form a cornerstone of this new model. Images contain applications and their required runtime dependencies and can be easily versioned, stored, and shared via centralized *registry* services. The ease of creating and sharing images has led to a rapid adoption of container technology but registries are now faced with the task of efficiently storing and serving millions of images. While in theory identical parts of Docker images can be shared across images and stored only once as *layers*, in practice this provides limited benefits as layers are rarely fully identical.

In this paper, we argue that Docker image layers should be reorganized in order to maximize their overlap and, thereby, reduce storage and network consumption. This argument is based on the observation that many layers only differ in a small number of files but would otherwise be identical. We present a set of design challenges that need to be solved when realizing such a reorganization approach, e.g., how to optimally reorganize layers, how to deal with the scale of current registries, and how to integrate the approach into the container image lifecycle. We then present a mathematical formulation of the problem and evaluate it on a set of real Docker images. Our preliminary results provide storage savings of 2.3×, and indicate that promising network savings are possible.

1 Introduction

Containers have played a pivotal role in realizing the *cloud native* vision [16]. In this vision, developers deploy their applications without worrying about infrastructural concerns such as resource provisioning, availability, or scalability, and thereby, place more focus on application logic. Due to their portability and low overhead compared to virtual machines, containers provide a perfect management unit to assemble cloud native applications. They can be spawned fast, moved easily, and deployed with high density [33].

While containers have been available in Linux for more than a decade [26, 30], the advent of containerization frameworks such as Docker [5] has drastically increased their popularity. Docker provides a management layer, which facilitates the process of packaging, shipping, and deploying applications as containers. Developers use Docker to create *container images*, which encapsulate an application and all its required runtime dependencies. Images consist of a set of *layers*, each containing part of the image contents (a subset of files). Images can be versioned and tagged, and shared in a *container registry* such as Docker Hub [6], and *pulled* from the registry to be deployed on a Docker-enabled host. This process significantly simplifies the sharing and deployment of software across environments.

The ease of packaging applications in Docker images has resulted in an explosion of the amount of available images. For example, Docker Hub contains more than 2 million *public* images alone [7], which combined occupy hundreds of TB of storage space. Additionally, every day more than 1,500 images are added [13]. As containers are becoming a major option of distributing and deploying applications on premises and in the cloud, this trend is only expected to continue.

The large size and growth rate of registries is challenging to deal with for the underlying storage infrastructure [23, 28]. To reduce the amount of stored data, Docker uses a content-addressable storage scheme which allows to identify duplicate layers and only store a single copy of each. This can reduce the storage utilization on the registry and end hosts, and the network in between as the same scheme can be applied to locally stored images. In practice, however, we found that only some layers are equal, which diminishes the potential storage savings. For example, of the 10,000 most popular Docker Hub images, consisting of 104,667 layers, only 36% of the layers were identical, resulting in storage reduction of only 1.48× while still comprising 80% duplicate files.

One approach to combat the growing storage requirements of a registry is to use deduplication-capable storage. Such solution, however, reduces neither the network traffic nor the client storage footprint. We argue that to optimize container image storage, we need to change the way image layers are constructed. If we can rearrange the contents of existing layers, we can maximize their overlap and increase their reusability across images. This argument is based on the observation that often, layers are "almost equal", i.e. they only differ in few files while the bulk of the data is identical. If we can split "almost equal" layers into their shared and unique contents, we can reuse the newly created shared layers in more images and thereby, reduce storage utilization and network traffic.

In this paper, we present an initial approach for optimizing container image layers for reusability. Implementing such an approach poses several questions and challenges, which we

lay out in detail in §4. Our aim in this paper is not to provide concrete answers to all of them but rather demonstrate the feasibility and potential benefits of a layer rearrangement. Hence, we focus on the questions of how layers could be rearranged and how an optimal rearrangement might be computed.

To answer the first question, we formulate an optimization problem (§5). As the numbers of parameters can be in the order of millions, instead of using a solver, we develop a greedy algorithm based on a cost function that considers storage, network, and image depth. We evaluate our approach on a sample dataset of 100 images downloaded from Docker Hub (§6). The dataset comprises a total of 855 layers with an initial storage redundancy of 3.14×. After computing a rearrangement, the resulting layers have a redundancy of only 1.33× while also increasing network savings.

2 Background

Docker containers are managed by the Docker daemon. To create a container, the daemon first has to retrieve the container's image from a centralized image registry. In this section we provide relevant background information about the structure of Docker images and the role of the image registry.

Docker images. A Docker image contains directories and files that form the root file system of a container. An image consists of a stack of read-only layers, each holding some portion of the container's root file system. As different image layers can contain files with the same name and path, layers are ordered and only the file in the highest layer will be displayed as part of the root file system. Special *whiteout* files can be placed in higher layers to mask files from lower layers.

When the Docker daemon starts a container, it creates a writable layer on top of the read-only layers. The writable layer is empty at the start of a container but accumulates changes (e.g., modified or newly created files) as the container updates its root file system. Changes from existing layers are stored in the writable layer via copy-on-write. To unify layers into a single logical file system, the Docker daemon employs overlay technologies such as Overlayfs2 [11] or Aufs [3].

Image registry. Docker images and their layers are stored in public (e.g., Docker Hub [6]) or private, organization-managed registries. Images in a registry are bucketed in per-project repositories and each repository contains tagged images. For example, the `ubuntu` repository in Docker Hub contains images tagged with `18.04` and `devel` (among others). The majority of repositories store an image tagged `latest`, containing the latest version of the corresponding application.

Layers are stored in the registry and transferred to clients in compressed tar form and have an associated layer ID. The ID is based on the layer content and hence, uniquely identifies each layer. This allows both the registry and the Docker daemon to detect duplicate layers and only store a single copy of a layer. A single layer can be referenced by multiple images.

When the Docker daemon downloads an image, it first requests an image *manifest* from the registry. The manifest, among other information, contains the ordered list of layer IDs, which are used by the daemon to download the layers that are not yet present in client's storage. When pushing a new image to the registry, the daemon first uploads the layers missing in the registry and then the manifest referencing them.

3 Redundancy in Image Registries

The increased popularity of containerized software drives the storage requirements of modern image registries to new heights. As of March 2019, the Docker Hub registry lists over 2 million public repositories. In raw format, the 10,000 most popular images already occupy over 5.3 TB of space. Grossly underestimating that each repository contains only one image, the public part of Docker Hub would utilize at least 1 PB of storage! The actual number, accounting for private repositories and multiple images per repository, is likely several times higher and will continue to grow in the foreseeable future.

To cope with such massive amounts of data, Docker currently applies two data reduction techniques. First, images are split into layers, allowing a single layer to be shared by multiple images. Second, every layer is independently compressed using local compression. Based on the 10,000 most popular images, our estimates show that layering decreases the dataset size by 1.48× while compression provides an additional 2.38× reduction, totaling to an overall reduction of 3.54×. While this results in a smaller estimated 1 PB/3.54 = 290 TB dataset, it still requires a hefty storage infrastructure budget. For example, if an organization uses AWS S3 standard storage, its annual cost of storing these images would be $75,000–$130,000 depending on the AWS region [2]. Additionally, there can be significant networking costs.

The problem is especially acute for companies providing registries as a service, such as Docker Hub [6], Jfrog Artifactory [10], Quay [12] and cloud providers [1,4,8,9]. However, other organizations that resort to maintain their own registries (e.g., for security and performance concerns) are also sensitive to the high cost of maintaining storage for container images.

We make two observations that explain why registry datasets contain significant unexploited redundancy. First, unlike earlier approaches to software packaging (e.g., RPM or DEB [19,25]) Docker images *must remain self contained*. As a result, the images of completely unrelated programs may rely on common components and hence, contain duplicate files. For example, by looking at files >1 MB in our 10,000 image dataset, we found that libraries such as `libslang`, `libstdc++`, or `libc` are present in over 1,000 images.

Our second observation is that *existing layering cannot catch all redundant files*. In our sample dataset, 67,047 unique layers in 10,000 images still contain almost 80% duplicate files! This is due to the fact that developers create their images

independently and hence, often end up with layers that share a large number of but not all files with existing layers.

We further notice that while existing file-level deduplication approaches [18, 27, 29] could be used to reduce storage utilization at the registry, they are not sufficient due to two main reasons. 1) Registry-side deduplication does not reduce the amount of network traffic between the registry and clients. Furthermore, the clients still store the images in their original, non-deduplicated form and therefore, require excessive storage space. These limitations could be addressed by coordinating the deduplication process between the client and the registry. However, this would require client-side changes which is overly disruptive. 2) Deduplication can add significant memory, CPU, and I/O overhead. On a layer push, the deduplication-capable registry would need to decompress the layer, compute the hashes of all files, and update the in-memory and on-disk hash index. On a layer pull, the layer needs to be reassembled again from its segments triggering multiple per-file I/Os. This increases registry infrastructure cost and detriments image pull and push times.

4 Design Considerations

We argue that, to effectively reduce storage consumption in container registries, an intelligent image registry that can rearrange layers to increase their reusability is necessary. Such a registry analyzes its contents and computes new optimal layers that can be assembled to form existing images. The primary motivation for the optimization is to reduce the registry's storage footprint while keeping the layered structure of images. However, the problem is multi-dimensional and other aspects need to be taken into account. For example, a straightforward approach would be to keep a layer per file. Since every layer is content addressable, no duplicate files will be stored. The problem with this approach is that images would consist of a very high number of layers (as many as there are files in an image) and unification technologies on the client side experience performance overheads when many layers are merged [22]. Therefore, we would like to limit the number of layers per image while still eliminating as much redundancy across files as possible.

Below we identify a set of important questions that arise when designing such a registry and present our preliminary thoughts on addressing them.

Challenge 1: How to rearrange layers in order to minimize redundancy? Computing a rearrangement is the core challenge and has various dimensions and constraints to ensure correctness and performance. First, no files can be removed from existing images. While new files can be added, those should be minimized to keep image sizes small. Additionally, the number of layers per image should not increase significantly. We provide an initial formulation of this challenge as an optimization problem, which allows us to capture the most important constraints and reason about its feasibility (§5).

Challenge 2: How to compute a rearrangement efficiently at scale? Modern registries contain billions of files and millions of layers and images. Computing an optimal rearrangement at that scale might be infeasible and hence, more efficient methods are required. There are different ways of making the problem tractable, e.g., relaxing less important constraints, using fast heuristics, or computing partial rearrangements on smaller subsets of images. Each approach comes with its trade-offs, which need to be analyzed and evaluated. We demonstrate the cost and quality of a rearrangement based on our problem formulation in §6.

Challenge 3: What is the impact on Docker clients? Rearranging layers and images necessarily affects the clients who consume those images. The best way to deal with this disturbance is an open challenge. It is unclear, whether the changes should be completely transparent to the client, or if clients can be included in the rearrangement process. Inclusion options range from using client data to determine subsets of images to rearrange, computing partial rearrangements on the clients themselves, or even guide clients to construct better layers in the first place. Here, we only focus on the case of computing a registry-wide rearrangement without involving clients.

Challenge 4: When should a rearrangement be computed? A container registry is in constant churn so rearrangements become suboptimal after a certain time and need to be recomputed. Dealing with this churn comes with several questions such as how frequently should layers be rearranged, should recomputation happen in online or offline mode, and can optimization happen incrementally based on the previous rearrangements. So far, we only deal with computing an initial rearrangement to demonstrate the potential benefits but a complete solution needs to be able to address the churn.

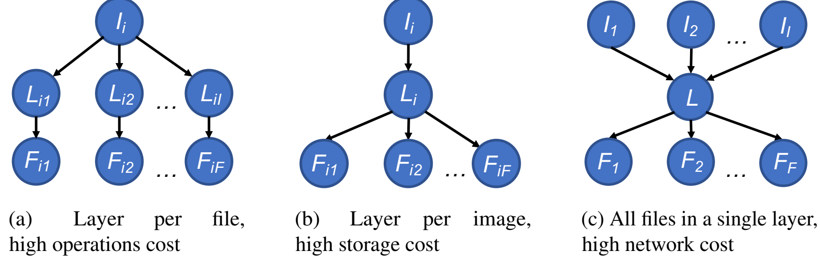
Challenge 5: What is the acceptable impact on images? Layer restructuring might change the order of layers in an image, which can lead to new images that are not perfectly equal to their original ones. This is due to the way Docker deals with duplicate files *inside* a single image, i.e. using layer ordering to deal with identical file names and whiteout files to mask deleted files. Hence, a rearrangement solution needs to ensure, that added files and layer order changes do not cause existing files to change or disappear. Another consequence is the breaking of the existing layer chain in images and thus the image history.

5 Optimizing Layers

In this section we formulate the problem of registry rearrangement and then describe the algorithm we developed as a first step towards the solution.

5.1 Problem Formulation

A registry stores and serves images over the network. It is in the interest of a registry to reduce storage redundancy as well



(a) Layer per file, high operations cost

(b) Layer per image, high storage cost

(c) All files in a single layer, high network cost

Figure 1: Motivating examples of optimal (but undesired) solutions when optimizing only for a *subset* of costs.

as network traffic. Additionally, it is beneficial to keep images "shallow" (small number of layers per image) for performance reasons. To illustrate the above points, consider three possible structuring solutions as shown in Figure 1. Note that these are undesired solutions and only presented here to motivate the problem formulation.

In the first solution (Figure 1a), each file is put in a separate layer. This eliminates storage redundancy and optimizes network cost. However, the total and per-image number of layers becomes significantly high (e.g., the top-100 images in Docker Hub have an average of only 8.55 and a maximum of only 21 layers). The high number of layers incurs performance overhead which we refer to as *operation cost*.

In the second solution (Figure 1b), each image consists of a single unique layer with all required files. This is ideal for the operation cost, however, there can be significant redundancy across the layers, and the network cost is non-ideal for the registry as there is no layer reuse at the client side. In the third solution, there is only a single global layer in the registry, containing all files (Figure 1c). Though unrealistic, this solution demonstrates that optimizing solely the registry storage and operation cost can lead to unacceptable network cost.

From the above we identify three costs: storage, network, and operation. The total cost, to be minimized, may be expressed as a linear combination. Next, we present a mathematical formulation of the problem. The model takes the file sizes and per-image pull frequency into account.

Denote by I the set of images, by J the set of layers, and by K the set of files. Denote by E the set of pairs $(i,k)(i \in I, k \in K)$ for which image i requires file k. Additionally, denote by g_k the size of file k, and by f_i the (usage) frequency of image i. We are looking for the solution consisting of boolean decision variables $x_{j,i}$, $y_{k,j}$, and $z_{i,j,k}$, where $x_{j,i} = 1$ if and only if layer j is in image i, $y_{k,j} = 1$ iff file k is in layer j, and $z_{i,j,k} = 1$ iff layer j is in image i and file k in layer j. We define a cost function to minimize as a weighted combination of storage, network, and operation costs: $cost = \alpha * operation + \beta * storage + \gamma * network$. The *operation* cost counts the number of image-to-layers edges (weighted by image frequency), expressed as $operation = \sum_i f_i \sum_j x_{j,i}$. The *storage* cost counts the number of files, or layer-to-file edges (weighted by file size), expressed as $storage = \sum_k g_k \sum_j y_{k,j}$. The *network* cost counts the number of image-layer-file paths (weighted by image frequency), expressed as $network = \sum_i f_i \sum_j \sum_k g_k z_{i,j,k}$. The constraint of this problem is to satisfy the image-to-file edges, E, i.e., post rearrangement, an image must contain all of its original files.

5.2 Computing New Layers

Given the above formulation, it is possible to compute an optimal solution using a mathematical optimization software for integer programming problems [15,20]. However, for large datasets, the problem can quickly become infeasible due to its large variable space. Hence, as an initial solution, we develop a greedy algorithm using the above *cost* function. Note that other algorithms may be applicable and exploring them is left as future work.

Our algorithm starts with an empty set of layers and constructs layers based on the requirements in E. It does not use the layers in the existing registry structure or take hints. The algorithm considers all files in a random order. For each file it iterates over the images requiring it, and either picks an already created layer (not necessarily referenced by the image) or creates a new one.

There are five cases to guide this decision as illustrated in Figure 2. Each case comes at a different cost depending on the state of the solution up to that point. In case 1, we consider using an existing layer l, created in a previous step, already containing the current file (solid line) but not yet referenced by the image (dotted line). The case requires the addition of a new edge from the image to l (converting the top dotted line to solid). In this case, the operation cost would increase by 1 as we add an image-to-layer edge, the storage cost remains unchanged, while the network cost depends on the files already contained in l. That is, if we connect image i to layer l, image i will need to serve additional files. If those additional files are needed by image i we discount them, otherwise, we would create too many layers unnecessarily. Table 1 describes all five cases. Once all cases are evaluated, we select the layer with the minimum incremental cost. Note that the described algorithm does not account for layer reordering (see §4, Challenge 5). The algorithm, however, may be extended to skip edges that create incorrect orderings.

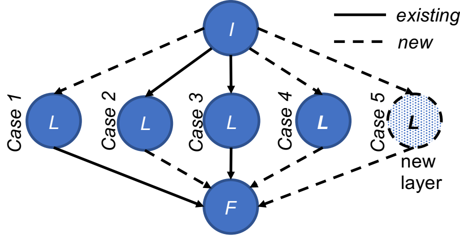


Figure 2: Cost cases. Solid lines show existing relationships between images, layers, and files. Dotted lines indicate the connections that the corresponding case proposes to add.

Case	Addition	Oper.	Storage	Network
1	(i,l)	1	0	$\lvert Files(l)\rvert -$ $\lvert Files(l)$ required by $i\rvert$
2	(l,f)	0	1	$\lvert Images(l)\rvert -$ $\lvert Images(l)$ requiring $f\rvert$
3	$-$	0	0	0
4	$(i,l),(l,f)$	1	1	$Network(Case1)+$ $Network(Case2)+1$
5	$l,(i,l),$ (l,f)	1	1	0

Table 1: Per-case costs based on counting.

6 Evaluation

In our experience the algorithm can scale to hundreds of thousands of files. However, due to the network cost computation, the performance degrades beyond that. Due to this limitation we only use a subset of the dataset described in §3 for the evaluation (we leave further algorithm improvements as future work). We use the 100 most popular images (tagged `:latest` and for `amd64` architecture) with a total of 855 layers from Docker Hub. The total number of files is 1,559,901 out of which 495,697 are unique, giving a maximum possible storage saving of 3.146×. The total number of layers is 855 yielding an average of 8.55 layers per image. The redundant network rate, i.e. files in images shipped but not needed due to layer ordering, is 1.014. Finally, the storage redundancy of this dataset is lower compared to the 80% of the larger 10,000 image dataset from §3, lowering the potential savings.

Setting	(α,β,γ)	Oper.	Storage	Network
1	$(1.0,1.0,1.0)$	2.39	1.78	1.053
2	$(1.0,1.0,0.5)$	2.27	1.75	1.145
3	$(1.0,0.5,1.0)$	2.24	1.79	1.04
4	$(0.5,1.0,1.0)$	9.15	1.33	1.018
-	original	8.55	3.14	1.014

Table 2: Oper., storage, and network overhead comparison.

We focus our evaluation on the case where $f_i = 1$ and $g_k = 1$. In other words, we ignore the file size and image frequency options so that interpreting the results is possible without additional analysis. In our first experiment we set $\alpha = \beta = \gamma = 1$ and run our algorithm over 100 images and 400,00 files. The resulting structure has a file redundancy of 1.78 (compared to 3.14), network redundancy of 1.05 (compared to 1.014), and average image depth of 2.39 (compared to 8.55). Hence, in this case we reduce storage by 1.76× while increasing network cost moderately. Lowering α to 0.5 lowers file redundancy further to 1.33 (vs. 3.14), improves network redundancy to 1.018 (vs. 1.014), and increases average image depth to 9.15 (vs. 8.55). Hence, we reduce files by 2.3× and merely increase network cost or image depth. Similarly, one may adjust the importance of storage and network cost as shown in Table 2. For example, lowering the network cost by setting $\gamma = 0.5$ increases the network overhead. To compute the network cost, we assumed each image is pulled by a *separate* client. This constitutes the worst case as no client savings are possible. Increasing the number of images pulled per client would result in lower network cost as clients would benefit from the improved layer shareability.

7 Related Work

It is well accepted in the industry that storage options and architectures for containers remain challenging [24] and a number of research studies have been published on this topic in recent years [13, 14, 17, 21–23, 28, 31, 34–36]. Some focused on analyzing and improving performance of the client-side ephemeral storage [31, 32, 34, 35] and external Docker volumes [14], while others proposed to distribute images in peer-to-peer fashion [23, 28] or use a distributed storage system to share images across clients [17, 36]. Anwar et al. analyzed Docker registry workloads and proposed effective registry-side caching and prefetching techniques to improve image pull latencies [13]. To reduce container start-up times, Slacker [22] and CernVM-FS [21] transfer image blocks to the clients lazily on access. These works are orthogonal to our proposition. To the best of our knowledge, we are the first to propose exploiting high file redundancy across layers by restructuring them in the registry.

8 Conclusion

We argue that a storage crisis is coming for container registries due to the increasing amount of images generated each day. Docker's current approach of sharing layers across images to reduce storage utilization is ineffective. We believe this is due to the fact that developers create images independently without considering already existing layers, resulting in many "almost equal" layers. Hence, we argue that the registry needs to provide logic to rearrange and combine layers to improve their shareability. This problem, however, is multi-dimensional and requires a holistic approach. We presented an initial approach based on an optimization problem formulation, and showed its potential to improve storage efficiency.

Discussion for HotCloud'19

By presenting our preliminary work at HotCloud'19 we hope to receive actionable feedback on the work, discuss several controversial points, and debate on the open issues.

Feedback we hope to receive. We expect to get feedback centered around the following three questions:

1) Is this the right way of tackling the problem? In this paper, we advocate for a layer restructuring solution to optimize storage in the registry (and, indirectly, at clients). As we demonstrate, such a solution is complex. We chose this design to stay compatible with the current format of Docker images, however, one may wonder if the image format itself is the right choice or if it can be improved to become more storage efficient. For example, instead of having fixed layers stored in the registry, layers could be constructed on-the-fly as part of an image pull request, based on what data the client already has stored locally.

2) Should we hide rearrangement from clients or actively involve them? As we describe in Challenge 3 in §4, the rearrangement approach might affect resulting images and hence clients. We hope to receive feedback from the audience on whether clients should be aware of this fact or rearrangement should happen transparently. Client awareness can improve the rearrangement but it may also complicate operations.

3) What is the experience of registry providers in terms of the amount of generated image data? Finally, we would be interested in collecting feedback on how registry administrators currently deal with the large amount of images. Has this been a problem so far or are there more pressing issues to be solved when operating a large-scale container registry?

Controversial points and discussion. We think the most controversial point we make is to opt for an *application-aware* solution rather than a storage-based solution, such as file deduplication. We think this is necessary to provide client-side benefits, i.e. what we call *network cost* and *operation cost* in our problem formulation. However, one may argue that deduplication is a tried-and-true technique which can solve the problem without additional complexity. We hope we can stir discussion in that area.

The other point of discussion we hope to generate is on the rights and wrongs of the current Docker storage format as described above. Is the current layered format really the best choice or do we need a redesign? What are the benefits of the current design and what are possible alternatives?

Open issues. As described in §4, there are several open issues that need to be addressed to implement a complete solution, including the questions of how to include the client, scale the approach up, incrementally compute rearrangements, deal with layer reordering, etc. These are directions we hope to tackle in the future.

The most pressing issue is whether the approach can be scaled to actual registry sizes of millions of images. As we have shown, the problem formulation is complex and algorithms can already take hours to complete on relatively small datasets. Even though we remain positive about achievable improvements, if an efficient way of computing a rearrangement is not possible, our approach would be infeasible.

References

[1] Amazon Elastic Container Registry. https://aws.amazon.com/ecr/.

[2] Amazon S3 Pricing. https://aws.amazon.com/s3/pricing/.

[3] AUFS - Another Union Filesystem. http://aufs.sourceforge.net.

[4] Azure Container Registry. https://azure.microsoft.com/en-us/services/container-registry/.

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

[6] Docker Hub. https://hub.docker.com/.

[7] Docker Hub Image Index. https://hub.docker.com/search?q=\&type=image.

[8] Goole Cloud Container Registry. https://cloud.google.com/container-registry/.

[9] IBM Cloud Container Registry. https://www.ibm.com/cloud/container-registry.

[10] Jfrog Artifactory. https://jfrog.com/artifactory/.

[11] Overlay Filesystem. https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt.

[12] Quay. https://quay.io/.

[13] Ali Anwar, Mohamed Mohamed, Vasily Tarasov, Michael Littley, Lukas Rupprecht, Yue Cheng, Nannan Zhao, Dimitrios Skourtis, Amit S. Warke, Heiko Ludwig, Dean Hildebrand, and Ali R. Butt. Improving Docker Registry Design Based on Production Workload Analysis. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*, 2018.

[14] J. Bhimani, J. Yang, Z. Yang, N. Mi, Q. Xu, M. Awasthi, R. Pandurangan, and V. Balakrishnan. Understanding Performance of I/O Intensive Containerized Applications for NVMe SSDs. In *Proceedings of the 35th IEEE International Performance Computing and Communications Conference (IPCCC)*, 2016.

[15] Christian Bliek, Pierre Bonami, and Andrea Lodi. Solving Mixed-integer Quadratic Programming Problems with IBM-CPLEX: A Progress Report. In *Proceedings of the 26th RAMP Symposium*, 2014.

[16] Eric A. Brewer. Kubernetes and the Path to Cloud Native. In *Proceedings of the 6th ACM Symposium on Cloud Computing (SoCC)*, 2015.

[17] Lian Du, Tianyu Wo, Renyu Yang, and Chunming Hu. Cider: A Rapid Docker Container Deployment System through Sharing Network Storage. In *Proceedings of the 19th IEEE International Conference on High Performance Computing and Communications (HPCC)*, 2017.

[18] Ahmed El-Shimi, Ran Kalach, Ankit Kumar, Adi Ottean, Jin Li, and Sudipta Sengupta. Primary Data Deduplication–Large Scale Study and System Design. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, 2012.

[19] Eric Foster-Johnson. Red Hat RPM Guide. 2003.

[20] Ambros Gleixner, Michael Bastubbe, Leon Eifler, Tristan Gally, Gerald Gamrath, Robert Lion Gottwald, Gregor Hendel, Christopher Hojny, Thorsten Koch, Marco E. Lübbecke, Stephen J. Maher, Matthias Miltenberger, Benjamin Müller, Marc E. Pfetsch, Christian Puchert, Daniel Rehfeldt, Franziska Schlösser, Christoph Schubert, Felipe Serrano, Yuji Shinano, Jan Merlin Viernickel, Matthias Walter, Fabian Wegscheider, Jonas T. Witt, and Jakob Witzig. The SCIP Optimization Suite 6.0. Technical Report, Optimization Online, 2018.

[21] N Hardi, J Blomer, G Ganis, and R Popescu. Making Containers Lazy with Docker and CernVM-FS. *Journal of Physics: Conference Series*, 1085(3), 2018.

[22] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Slacker: Fast Distribution with Lazy Docker Containers. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, 2016.

[23] Wang Kangjin, Yang Yong, Li Ying, Luo Hanmei, and Ma Lin. FID: A Faster Image Distribution System for Docker Platform. In *Proceedings of the 2nd IEEE International Workshops on Foundations and Applications of Self* Systems (FAS* W)*, 2017.

[24] Mark Lamourine. Storage Options for Software Containers. *;login: The USENIX Magazine*, 40(1), 2015.

[25] Fabio Mancinelli, Jaap Boender, Roberto Di Cosmo, Jerome Vouillon, Berke Durak, Xavier Leroy, and Ralf Treinen. Managing the Complexity of Large Free and Open Source Package-based Software Distributions. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2006.

[26] Paul Menage. Adding Generic Process Containers to the Linux Kernel. In *Linux Symposium*, 2007.

[27] Dutch T Meyer and William J Bolosky. A Study of Practical Deduplication. *ACM Transactions on Storage (TOS)*, 7(4), 2012.

[28] Senthil Nathan, Rahul Ghosh, Tridib Mukherjee, and Krishnaprasad Narayanan. CoMICon: A Co-Operative Management System for Docker Container Images. In *Proceedings of the 5th IEEE International Conference on Cloud Engineering (IC2E)*, 2017.

[29] João Paulo and José Pereira. A Survey and Classification of Storage Deduplication Systems. *ACM Computing Surveys (CSUR)*, 47(1), 2014.

[30] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, 2007.

[31] Vasily Tarasov, Lukas Rupprecht, Dimitris Skourtis, Wenji Li, Raju Rangaswami, and Ming Zhao. Evaluating Docker Storage Performance: From Workloads to Graph Drivers. *Cluster Computing*, Online First, 2019.

[32] Vasily Tarasov, Lukas Rupprecht, Dimitris Skourtis, Amit Warke, Dean Hildebrand, Mohamed Mohamed, Nagapramod Mandagere, Wenji Li, Raju Rangaswami, and Ming Zhao. In Search of the Ideal Storage Configuration for Docker Containers. In *Proceedings of the 1st Workshop on Autonomic Management of Large Scale Container-based System (AMLCS)*, 2017.

[33] Jörg Thalheim, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. Cntr: Lightweight OS Containers. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, 2018.

[34] Xingbo Wu, Wenguang Wang, and Song Jiang. Total-COW: Unleash the Power of Copy-on-Write for Thin-Provisioned Containers. In *Proceedings of the 6th Asia-Pacific Workshop on Systems (APsys)*, 2015.

[35] Qiumin Xu, Manu Awasthi, K Malladi, Janki Bhimani, Jingpei Yang, and Murali Annavaram. Performance Analysis of Containerized Applications on Local and Remote Storage. In *Proceedings of the 33rd International Conference on Massive Storage Systems and Technology (MSST)*, 2017.

[36] Chao Zheng, Lukas Rupprecht, Vasily Tarasov, Douglas Thain, Mohamed Mohamed, Dimitrios Skourtis, Amit S Warke, and Dean Hildebrand. Wharf: Sharing Docker Images in a Distributed File System. In *Proceedings of the 9th ACM Symposium on Cloud Computing (SoCC)*, 2018.