Beyond Layers: Container Registries for Files Distribution and On-Demand Image Partitioning

Giovanni Bartolomeo[♯*] Navidreza Asadi[♯*] Wolfgang Kellerer[♯] Jörg Ott[♯] Nitinder Mohan[♮]
[♯]TU Munich, Germany, {name.surname}@tum.de [♮]TU Delft, The Netherlands, n.mohan@tudelft.nl

Abstract

Thanks to OCI standardization, containers now describe a complete end-to-end environment for packaging, distributing, and deploying software and its dependencies. Unfortunately, the complexity of modern applications creates use cases that are not yet supported by this technology. For example, packaging machine learning model weights in containers requires extensive build times, reduces flexibility, and forces developers to seek alternative, custom solutions. Another example is platform-specific drivers and binaries, which are often included in container fat images but involve cache invalidations for every minor change. In our work, we extend the container layered structure with a new two-dimensional filesystem layer type, specifically designed for efficient handling of large data. The proposed layer type allows highly parallelized image builds and fine-grained layer caching, while also providing a mechanism for on-demand partitions. Such partitions enable developers to request a subset of the image, e.g., containing only a portion of a machine learning model or a specific set of drivers for a given architecture. The proposed implementation is fully OCI-compliant, allowing distribution of such customized images to any container runtime with no additional effort. Our implementation, called 2DFS, achieves 56x faster build times and 25x better caching efficiency compared to Docker, while providing on-demand image partitioning with no overhead.

1 Background and Motivation

Containers have emerged as a de facto standard for packaging applications and dependencies and have seen widespread adoption in the industry. Furthermore, thanks to Open Container Initiative (OCI) standardization [6], the creation, execution, and sharing of container images has become more streamlined. Even in the context of machine learning (ML) applications, containers have become the preferred method for packaging and deploying ML models [2]. With the advent of split computing, models can be partitioned on demand and distirbuted across different devices, calling for a flexible way to package and distributed the model weigths [4]. Unfortunately, frequent model re-training can led to frequent re-builds of the container images, while in split computing scenarios, multiple images must be built, one for each split. Additionally, it is common practice to store application libraries' binaries and drivers in container images [1, 7], making them self-contained and portable across different environments, but resulting in severely bloated images [8]. Unfortunately, dependencies download at runtime leads to increased startup times and bandwidth usage. While solutions using dynamic volumes mounts require additional engineering efforts, especially at the edge where object stores are not widely available. State of the art build systems, like BuildKit [5], are optimized for efficient computation of container changesets but are not meant for handling large independent files. On the other hand, by addressing content by digest, container registries offer an excellent mechanism for object retrieval. In our work, we extend the OCI image format to recycle the well-enstablished container image registries to distribute not only code and dependencies but also binaries, model parameters, and any large file without affecting the complexity of the CI/CD pipelines. By building a specialized two-dimensional container layer, the ML model splits, drivers and dependencies can be independently cached. Registries can then provide each client with image partitions, including only the necessary files.

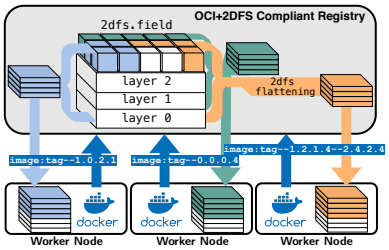


Figure 1. 2DFS image distribution workflow.

2 2DFS High Level Design

This work presents the initial implementation efforts to realize 2DFS, a two-dimensional filesystem build and distribution framework for containers. Essentially, 2DFS is an extension of the conventional layered filesystem used in container images. We introduce a new layer type called `2dfs.field`. A field is a sparse hash-pointer matrix of *allotments* representing a self-contained, non-overlapping, and independent filesystem space. Each one of these allotments links one or more files. As shown in fig. 1, the field datastructure is appended to the layers list. As this is a sparse matrix, each allotment can either point to a file or be empty. The two-dimensional data structure has been selected to simplify the definition of partitions. In fact, we propose a partitioning operation that can select subsets of the field, allowing on-demand retrieval

* Both authors contributed equally to this research.

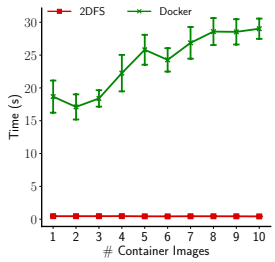


Figure 2. MNv2LBuild time.

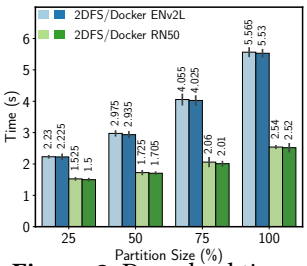
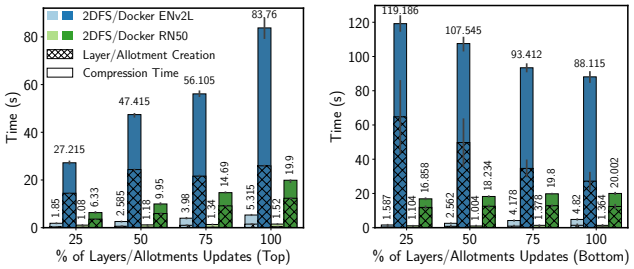


Figure 3. Download time.



(a) Top-down update strategy. **(b)** Bottom-Up Update Strategy.

Figure 4. Build time after model updates with image caching.

of the data. A field, expressed by rows and columns, can be partitioned drawing a sub-field. E.g., in fig. 1, the first worker node is requesting only the allotments from `<row 1, column 0>` to `<row 2, column 1>` of the image. Partitions can be requested by appending the `--x1.y1.x2.y2` operator to the image tag as shown in the figure. All the OCI+2DFS compliant registries must support this operation.

Image partitioning is performed by flattening the allotments into regular OCI layers, ensuring compatibility of the downloaded images with any OCI-compliant runtime, such as Docker. The 2DFS builder genearates the allotments as non-overlapping independent regions, making the flattening operation performed by the registries just an image index manipulation operation, extremely lightweight and efficient.

A `2dfs.field` structure is generated by the 2DFS builder. A builder uses a base OCI image, and a field descriptor file, both provided by the developer, to create an extended OCI+2DFS image which can be uploaded to any compliant registry. A field descriptor file, called `2dfs.json`, contains the list of allotments, their position in the field (row and column), and the source files to be included in each allotment. Essentially, instead of including the files with the traditional Dockerfile ADD command, the developer specifies the files to be included in the allotments in the `2dfs.json` file. The builder exploits the allotment independence to parallelize the build process, significantly reducing the build time.

3 Early Results

We evaluate 2DFS's performance across build time and caching efficiency, comparing it to Docker, the most popular container-building tool [3]. Because 2DFS uses a novel layering approach, we often needed multiple Docker images to replicate the functionality of a single 2DFS image partition. For instance, a three-partition 2DFS image required three separate Docker images for comparison. Our evaluation uses MobileNetV2 (MNv2L), ResNet50 (RN50) and EfficientNet-V2L (ENv2L) models. We apply split-computing techniques to these models, generating multiple splits for each model wich we map to different allotments. We compare the performance of 2DFS while varying the amount of splits of a model.

Figure 2 shows the build time of MNv2L with 10 splits. While for 2DFS we only need to build a single image with all the splits as allotments, for Docker we build 10 images, one for each model split. The average build time difference shows that 2DFS build is 55.9× faster. 2DFS high parallel build and reduced filesystem copy operations, result in faster build times. We omit the total build time because it is not comparable across the two approaches.

Figure 3 shows the retrieval time for ENv2L and RN50 split models from a registry. For each model partition, we compare the download time of a custom pre-built Docker image against a 2DFS image that is partitioned on-demand. We observe that the download time for the partitioned images is comparable to the pre-built images, with an average overhead of 20 *ms*. The partitioning and flattening operation at the registry introduces minimal overhead.

Figure 4 shows the re-build time caused by changes in a container layer. On the left we increasingly update the layers from the top to the bottom, while on the right we update the layers from the bottom (worst case scenario for Docker). We update from 1 to 100% of the layers in the image. On average 2DFS re-build time is 25× faster, with a peak of 75× faster build time for the bottom-up scenario. Files updates in 2DFS do not trigger a full cache invalidation but rather the reconstruction of the local allotment. The field reconstruction will recycle the cached allotments resulting in better performance and improve cache utilization.

Acknowledgments

Work supported by the Federal Ministry of Education and Research of Germany (BMBF) project 6G-Life (16KISK002).

References

[1] D. Aivaliotis. Tupperware: Container deployment at scale. 2015.
[2] D. J. Beutel et al. Flower: A friendly federated learning research framework. *arXiv:2007.14390*, 2020.
[3] Docker. Most-used developer tool. https://www.docker.com/blog/docker-stack-overflow-survey-thank-you-2023/, 2023.
[4] Y. Matsubara, M. Levorato, and F. Restuccia. Split computing and early exiting for deep learning applications: Survey and research challenges. *ACM Computing Surveys*, 55(5):1–30, 2022.
[5] Moby. Buildkit. https://github.com/moby/buildkit, 2024.
[6] OCI. Oci specification: Image specification. https://github.com/opencontainers/image-spec/blob/main/spec.md, 2024.
[7] H. Saokar et al. {ServiceRouter}: Hyperscale and minimal cost service mesh at meta. In *OSDI 23*, 2023.
[8] H. Zhang et al. Machine learning systems are bloated and vulnerable. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2024.