FILEgrain: Transport-Agnostic, Fine-Grained Content-Addressable Container Image Layout

github.com/AkihiroSuda/filegrain
Who am I

- **Software Engineer at NTT**
  - Largest telco in Japan

- **Docker Moby Core maintainer**

> In April, Docker [as a project] transited into Moby. Now Docker [as a product] has been developed as one of downstream of Moby.

- **BuildKit initial maintainer** ([github.com/moby/buildkit](https://github.com/moby/buildkit))
  - Next-generation `docker build`
  - Executes DAG vertexes of Dockerfile-equivalent concurrently
• New image format that allows doing `docker run` before `docker pull` 100% finishes

• More benefits: deduplication, etc.

• OCI compatible

Summary

You just need to pull 20% for running the official OpenJDK 8 image! (The rest 80% can be pulled on demand)
The current status of Docker / OCI format
OCI Image Spec

• In July 2017, Open Containers Initiative (OCI) announced the first release of its industry-standard container image spec
  • CoreOS's appc/ACI spec maintainers has also joined OCI

• The data structure is almost identical to Docker Image Manifest V2.2, but made separate from Docker Registry HTTP API

• Agnostic to distribution protocol; Any protocol with directory-like semantic will work. e.g. HTTP, NFS, IPFS
  • Some extra coordinator is needed though (e.g. lock for the image index)
  • I'm +1 for IPFS (P2P filesystem)

• Note: the spec is unrelated to Dockerfile syntax / instructions
Structure of Docker / OCI image format

- Composed of TAR archives of AUFS layers
  - AUFS-style tar format is used as the lingua franca across different snapshot drivers (e.g. OverlayFS, Devicemapper, BtrFS, ZFS)

```
FROM ubuntu:17.04
RUN apt-get install foobar
COPY foobar.conf /etc

/bin/bash
/bin/ls ...

mount -t overlay -o lowerdir=0,upperdir=1 ..
mount -t overlay -o lowerdir=1,upperdir=2 ..
```

- added & modified files
- file deletion info ("whiteout")
Structure of Docker / OCI image format

- Blobs such as TARs are stored with a Merkle tree structure
- Generally, distributed via Docker Registry, with (e.g.) S3/Swift backend

```
/index.json
{
  "schemaVersion": 2,
  "manifests": [
    {
      "mediaType": "application/vnd.oci.image.manifest.v1+json",
      "size": 7143,
      "digest": "sha256:e692418e4cbaf90ca69d05a66403747baa33ee08806650b51fab815ad7fc331f"
    }
  ]
}
```

/blobs/sha256/e692418e... (Next slide)
Environment variables and so on

Base layer

Diff layer

Diff layer
Structure of Docker / OCI image format

- Merkle tree ensures reproducibility of an environment (`docker pull foobar@sha256:e692418e..`)

```
```

```
```

```
```

```
```

```
```
Problems of Docker / OCI image format

• TAR = Tape ARchiver, appeared in circa. 1979 (UNIX 7th Edition)
• TAR was originally designed for magnetic tapes
  → Not suitable for Docker/OCI workloads in 2017

PDP-11, the target architecture of UNIX in 1970s, with TU56 DECTape drives
Problem 1: TAR requires scanning the whole "tape"

Without scanning the whole "tape"...

• Files cannot be listed up → Can't be mounted as a filesystem

• File offsets cannot be detected → Files can't be accessed
  • We could create a separate index, but it is useless when a TAR is gzipped, as it can't be seeked
Problem 1: TAR requires scanning the whole "tape"

• If we could solve this problem, we can mount(2) an image and start a container without pulling the whole image
  • Only metadata are needed for mount(2)
  • Files are lazily pulled on demand

• → Shorter start-up time & Less network traffic

New containers can start immediately on newly added hosts

Faster testing and deployment cycle
Problem 1: TAR requires scanning the whole "tape"

Detailed usecases:

• Web apps with huge number of HTML files and graphic files

• Jupyter Notebook with various big data samples
  • Academic papers will be immediately reproducible by just running `docker run some-single-huge-image@sha256:deadbeef.`

• Full GNOME/KDE/Windows(potentially) Desktop

• Java / dotNET runtimes

• Integration testing environment...
Problem 2: No deduplication

- A registry might contain very similar images
  - Different versions
  - Different architectures
  - Different configurations

- TARs of these images are likely to contain identical files, but waste storage without any data deduplication

```bash
FROM ubuntu:17.04
RUN apt-get install foo

FROM ubuntu:17.04
RUN apt-get install foo bar

FROM debian:9
RUN apt-get install foo

FROM ubuntu:17.04
RUN echo ... > /etc/apt/source.list
RUN apt-get install foo
```
Problem 3: No concurrency

• It might be good to pull a large TAR layer with multiple connections
  • Especially when multiple servers are available

   ![Diagram](Diagram.png)

   - Range 0-1023
   - Range 1024-2047

• But not all protocol allows that
  • RFC7233 says HTTP/1.1 Range Requests are OPTIONAL
Problems of Docker / OCI image format

1. TAR requires scanning the whole "tape"

2. No deduplication

3. No concurrency

Image: https://en.wikipedia.org/wiki/Magnetic_tape
New image format: FILEgrain

https://github.com/AkihiroSuda/filegrain
FILEgrain overview

• Single large TAR blob → Many small blobs
• Metadata blob with content-addressability
• Only metadata blob is needed for mounting the image
  • File blobs can be lazily pulled on demand

```
| Metadata 0   | File 0 |
| Metadata 1   | File 1 |
| ...          |       |
| Metadata {n-1} | File {n-1} |
| Terminal zero bytes |

TAR
```

```
| Metadata 0   |
| Metadata 1   |
| ...          |
| Metadata {n-1} |

File 0

| File 1          |

File 1

| File {n-1} |

File {n-1}

content-addressable

File name, permission bits, SHA256 digest...

continuity
```
Metadata format: continuity

• continuity: filesystem metadata manifest system used in containerd / Moby community ([github.com/containerd/continuity](http://github.com/containerd/continuity))
  • Serializes file names, permission bits, XAttrs, and digest values as a ProtocolBuffers message structure

```protobuf
define Resource {
    repeated string path;
    int64 uid;
    int64 gid;
    uint32 mode;
    uint64 size;
    repeated string digest;
    repeated XAttr xattr;
    ...
}
```

• Similar format: mtree(8)
Highly compatible with traditional OCI spec

- A single OCI image can contain both FILEgrain manifests and traditional OCI manifests

```
`docker pull foo:v1-filegrain`
```

```
`docker pull foo:v1`
```

```
"v1-filegrain": "sha256:a8e3..", "v1":"sha256:e692.."
```

```
docker pull foo:v1
```

FILEgrain Manifest

Index

Traditional OCI Manifest

```
docker pull foo:v1-filegrain
```

```
docker pull foo:v1
```

```
"v1-filegrain": "sha256:a8e3..", "v1":"sha256:e692.."
```

```
docker pull foo:v1
```

```
"v1-filegrain": "sha256:a8e3..", "v1":"sha256:e692.."
```

``` index.json ```

```
"v1-filegrain": "sha256:a8e3..", "v1":"sha256:e692.."
```

```
docker pull foo:v1
```

```
"v1-filegrain": "sha256:a8e3..", "v1":"sha256:e692.."
```

```
docker pull foo:v1
```

```
"v1-filegrain": "sha256:a8e3..", "v1":"sha256:e692.."
```

```
docker pull foo:v1
```

```
"v1-filegrain": "sha256:a8e3..", "v1":"sha256:e692.."
```

```
docker pull foo:v1
```

```
"v1-filegrain": "sha256:a8e3..", "v1":"sha256:e692.."
```

```
docker pull foo:v1
```

```
"v1-filegrain": "sha256:a8e3..", "v1":"sha256:e692.."
```

```
docker pull foo:v1
```

```
"v1-filegrain": "sha256:a8e3..", "v1":"sha256:e692.."
```

```
docker pull foo:v1
```

```
"v1-filegrain": "sha256:a8e3..", "v1":"sha256:e692.."
```

```
docker pull foo:v1
```

```
"v1-filegrain": "sha256:a8e3..", "v1":"sha256:e692.."
```

```
docker pull foo:v1
```

```
"v1-filegrain": "sha256:a8e3..", "v1":"sha256:e692.."
```
• FILEgrain layers can be put on top of traditional OCI layers
  • Traditional OCI layers might be still suitable for frequently used files or large number of small files

```json
{
  ...
  "layers": [
    {
      "mediaType": "application/vnd.oci.image.layer.v1.tar+gzip",
      "size": 33554432,
      "digest": "sha256:e692..."
    },
    {
      "mediaType": "application/vnd.continuity.layer.v0+pb",
      "size": 16724,
      "digest": "sha256:a18b..."
    }
  ]
}
```
The problems are solved!

1. TAR requires scanning the whole "tape"
   → Only metadata (continuity) blob is needed for mounting. Other stuff can be lazily pulled on demand.

2. No deduplication
   → Deduplication in the granularity of files

3. No concurrency
   → Concurrency in the granularity of files
Cons

1. Larger number of blobs
   • /blobs/sha256 directory will contain huge number of files when laid out on filesystem, and may make readdir(3) slow
     • Generally, this issue could be solved by "sharding" /blobs/sha256/deadbeef.. like /blobs/sha256/de/deadbeef.., but not in this case, for compatibility with OCI.

2. More RPC overhead
   • Single traditional TAR is still suitable for small files so as to reduce number of RPCs

3. Less compression rate
   • Single traditional TAR is still suitable when the layer contains similar files

But anyway, these cons can be easily mitigated by composing FILEgrain layers on top of traditional OCI layers
Alternative ideas?

 Alternative 1: Use some external blob store?
→ No, because it depends on certain protocols. Also, it is unlikely to work with OCI Merkle tree.

Related work:


• Loopback-mount an ext4 image located on NFS

• Support lazy-pulling and deduplication in granularity of blocks
Alternative ideas?

Alternative 1: Use some external blob store?

→ No, because it depends on certain protocols. Also, it is unlikely to work with OCI Merkle tree.

Related work:


• CernVM FS
  • CernVM FS has its own Merkle tree

• Support lazy-pulling and deduplication in granularity of files (as in FILEgrain)
Alternative ideas?

Alternative 1: Use some external blob store?
→ No, because it depends on certain protocols. Also, it is unlikely to work with OCI Merkle tree.

- IPFS is also attractive protocol
  - P2P, content-addressable

- FILEgrain is made agnostic to transportation protocol as in OCI image spec; it works with HTTP / NFS / CernVM FS / IPFS / whatever
Alternative ideas?

Alternative 2: Seek TAR?
→ No
  • Compressed TAR is not seekable
  • Even uncompressed TAR is sometimes not seekable, depending on transportation protocol
  • Larger number of request for fetching the whole metadata
Alternative ideas?

Alternative 3: Use (e.g.) ZIP instead of TAR?  → No

- still unseekable depending on transporation protocol
- poor metadata support

<table>
<thead>
<tr>
<th>Extra metadata 0</th>
<th>Compressed file 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extra metadata 1</td>
<td>Compressed file 1</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Extra metadata {n-1}</td>
<td>Compressed file {n-1}</td>
</tr>
<tr>
<td>Metadata 0</td>
<td>Metadata 1</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Metadata {n-1}</td>
<td>Footer</td>
</tr>
</tbody>
</table>

Metadata can be pulled at once firstly? (No)
Evaluation
Implementation

• Implemented as a read-only FUSE filesystem

• No support for write operations
  • "Cattle" containers should be immutable; it is anti-pattern to do any write operation against rootfs
    • /tmp and /run should be mounted as tmpfs
    • persistent data should be written to bind-mounted Ext4/XFS

  • Actually, even Docker built-in storage drivers (overlayfs, AUFS) don't fully support write operations (github.com/AkihiroSuda/issues-docker)
    • e.g. Yum is known not to work on overlayfs, without installing yum-plugin-ovl
      (neither a bug of overlayfs nor yum!)
Implementation

- Docker doesn't support running a container with a rootfs that is not managed by the Docker daemon

- So current FILEgrain POC is evaluated using runc

- TODO: reimplement as a containerd plugin
# Images for evaluation

<table>
<thead>
<tr>
<th>Image</th>
<th>Description</th>
<th>rootfs size (after tar+gzip expansion)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>openjdk:8</code></td>
<td>Debian 9.1, OpenJDK 8u141</td>
<td>671.7MB</td>
</tr>
<tr>
<td><code>sha256:5da842d59f76009fa27ffde06888ebd560c7ad17607d7cd1e52fc8757c9a45fb</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>kdeneon/all</code></td>
<td>Ubuntu 16.04, KDE Plasma 5.10, Firefox 54..</td>
<td>4.8GB</td>
</tr>
<tr>
<td><code>sha256:e3e7f216a5f8f1fdc4e4eab8807d7afcd291c050899ab3e8a8355b7b28a19247</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>kaggle/python</code></td>
<td>Debian 8.5, Various machine learning frameworks, NLTK (natural language toolkit) dataset..</td>
<td>8.3GB</td>
</tr>
<tr>
<td><code>sha256:335103c998aea22a5608c2eeca7dcf109e0828ed233b75f5098182c5b058fe98</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

See [https://github.com/AkihiroSuda/filegrain/issues/17](https://github.com/AkihiroSuda/filegrain/issues/17) for detailed information
Evaluation: blobs needed to start a container (uncompressed)

• openjdk:8 (blobs in total = 671.7MB + meta 5.4MB)
  • mount: 5.4MB ( 2 blobs)
    • only metadata are needed (manifest and continuity)
  • `sh`: 7.3MB ( 8 blobs) in total
  • `java -version`: 88.2MB (30 blobs) in total
  • `javac HelloWorld.java`: 137.3MB (50 blobs) in total

• kdeneon/all (4.8GB + 34.5MB)
  • mount: 34.5MB ( 2 blobs)
  • `sh`: 36.7MB ( 8 blobs)
  • `startkde`: 742.7MB (4,267 blobs)
  • `firefox`: 866.6MB (4,506 blobs)

note: commands are executed sequentially
Evaluation: blobs needed to start a container (uncompressed)

• kaggle/python (8.3GB + 38.2MB)
  • mount: 38.2MB (2 blobs)
  • `sh`: 40.1MB (8 blobs)
  • `ipython --c 'echo("hello")'`: 75.4MB (1033 blobs)
  • `ipython --c 'import nltk'`: 352.0MB (2799 blobs)

less than 1/24
## Evaluation: compression

<table>
<thead>
<tr>
<th>Image</th>
<th>rootfs</th>
<th>TAR at once + gzip -9</th>
<th>FILEgrain + gzip -9 against each of blobs</th>
</tr>
</thead>
<tbody>
<tr>
<td>openjdk:8</td>
<td>671.7MB</td>
<td>261.3MB</td>
<td>260.7MB (-645,604B)</td>
</tr>
<tr>
<td>kdeneon/all</td>
<td>4.8GB</td>
<td>2.1GB</td>
<td>2.1GB (-489,361B)</td>
</tr>
<tr>
<td>kaggle/python</td>
<td>8.3GB</td>
<td>3.6GB</td>
<td>3.6GB (+4,345,701B)</td>
</tr>
</tbody>
</table>

No more than a rounding error in these cases
(If an image contained similar files, its compression rate would get worse)
Evaluation: deduplication

kdeneon/all
(4.8GB)

kaggle/python
(8.3GB)

75.4MB deduplication
(KDE and Kaggle are mutually unrelated, but have some common Debian files)
Evaluation: FUSE overhead

Time required for archiving /usr of openjdk
(X: n-th experiment, Y: seconds)

Docker (overlay2) | FILEgrain FUSE

Docker 17.06 / Fedora 26 / 2 vCPUs, 2GB RAM, 2GB swap (VMware Fusion, MacBook Pro 2016)
Evaluation: FILEgrain

For FILEgrain, the first data contains the time for pulling some blobs needed to start a container (But no network overhead, as localhost is the remote in this evaluation)

But the Docker data doesn't contain the time for pulling, as a container can be started only after pulling the whole image.
Evaluation: FUSE overhead

Time required for archiving
(X: n-th experiment, Y: seconds)

For some implementation reason, in-kernel cache doesn't seem working..

But this cache should be easily enabled, as the filesystem is always read-only by design

Getting faster due to in-kernel caching
Evaluation: others

- **Overhead of larger number of RPC for pulling blobs**
  - Depends on the transportation protocol
  - TODO: implement Docker Registry API client and evaluate the overhead
    - Current POC just uses a local directory as a mock registry
Future work
Future work

• Even finer granularity (*CHUNK*grain?)
  • For large files, it might be good compute SHA256 digests against each of chunks, and serialize the digests in some format
    • Probably, this serialization would be separate from continuity manifest itself.
    continuity#85

• Use traditional OCI TAR for files that are very likely to be needed immediately to start a container
  • We can easily detect such files by starting a container and capture FUSE calls before pushing the image
Integration to the ecosystem

- containerd is the next standard runtime in the container industry
Integration to the ecosystem

- containerd plugin system allows new feature to be added without modifying the containerd upstream
Integration to the ecosystem

- FILEgrain will be reimplemented as set of containerd plugins
- Can be easily integrated to the higher-level engines without modification (Moby/Docker, Kubernetes, ..)

containerd v1.0

- runtime service:
  - runc plugin
- snapshot service:
  - overlayfs plugin
  - btrfs plugin
- differ service:
  - generic plugin
- content service:
  - generic plugin

FILEgrain plugins
Recap
Recap

• New image format that allows doing `docker run` before `docker pull` 100% finishes

• More benefits: deduplication, etc.

• OCI compatible

You just need to pull 20% for running the official OpenJDK 8 image! (The rest 80% can be pulled on demand)
Demo