FAST'25

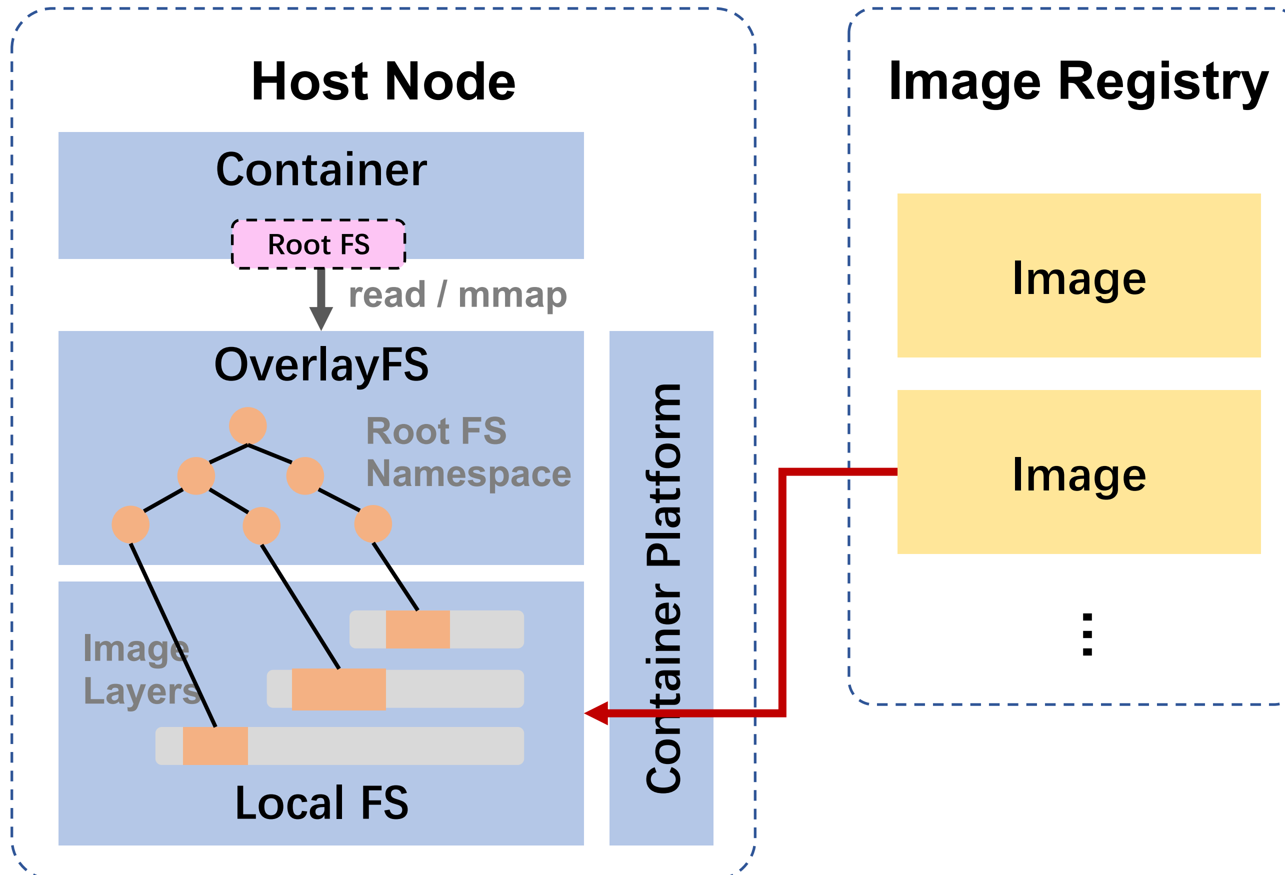
FlacIO: Flat and Collective I/O for Container Image Service

Yubo Liu, Hongbo Li, Mingrui Liu, Rui Jing, Jian Guo, Bo Zhang, Hanjun Guo, Yuxin Ren, and Ning Jia

Huawei Technologies Co., Ltd.

Container Image Service

A Naïve Solution: Full Image Loading



Full Image Loading is inefficient!

- Large container (e.g., Pytorch) startup takes minutes
- Dozens of times I/O amplification

OPT1: Cold Startup Acceleration

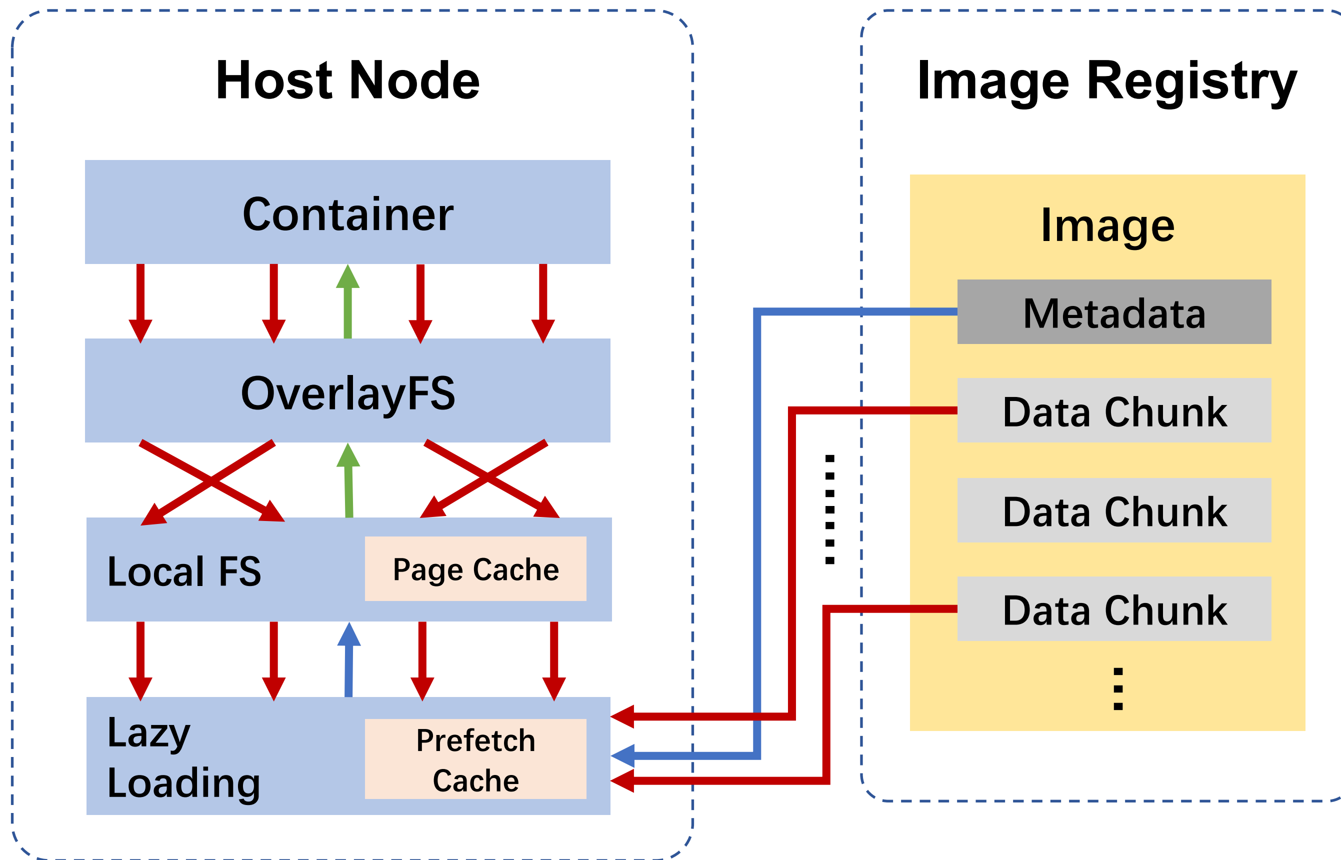
- Lazy loading – On demand I/O
- Prefetching (e.g., expanded, prioritize files, trace)

OPT2: Cold Startup Mitigation

- Host-side caching/sharing
- Forking
- P2P loading

Container Image Service

Typical Process of Lazy Loading



Stage 1: Deploy (→)

Obtain the metadata required for building the container runtime

Stage 2: Running (→)

Create the container runtime in the host (e.g., *cgroup*)

Stage 3: Ready (→)

Launch the service and executes the entrypoint in the container

Cold Startup Overhead Analysis

Solution	Latency Breakdown				I/O Behavior	
	Deploy	*Running*	*Ready*	*Total*	*I/O Amp.*	*Net. Pkg.*
Full Image						
CRFS						
Nydus						
DADI						
DADI-Trace						

CRFS / Nydus:
- File-level lazy loading
- Expanded prefetch -- loads extra data beyond the missed range

DADI / DADI-Trace:
- Block-level lazy loading
- Trace prefetch – replays I/O according the historical trace

Experiment Setup

- Host and registry connected by 10Gbs Net.
- Cold startup *Pytorch* container

Cold Startup Overhead Analysis

Solution	Latency Breakdown				I/O Behavior	
	Deploy	*Running*	*Ready*	*Total*	*I/O Amp.*	*Net. Pkg.*
Full Image	124.6s	1.6s	1.7s	127.9s	47.5X	573K
CRFS	1.8s	1.2s	24.1s	27.1s	1.8X	99K
Nydus	0.8s	2.9s	21.4s	25.1s	1.6X	90K
DADI	0.6s	2.6s	17.0s	20.2s	3.1X	171K
DADI-Trace	0.7s	2.2s	17.1s	20.0s	3.0X	166K

Experiment Setup

- Host and registry connected by 10Gbs Net.
- Cold startup *Pytorch* container

Observation 1

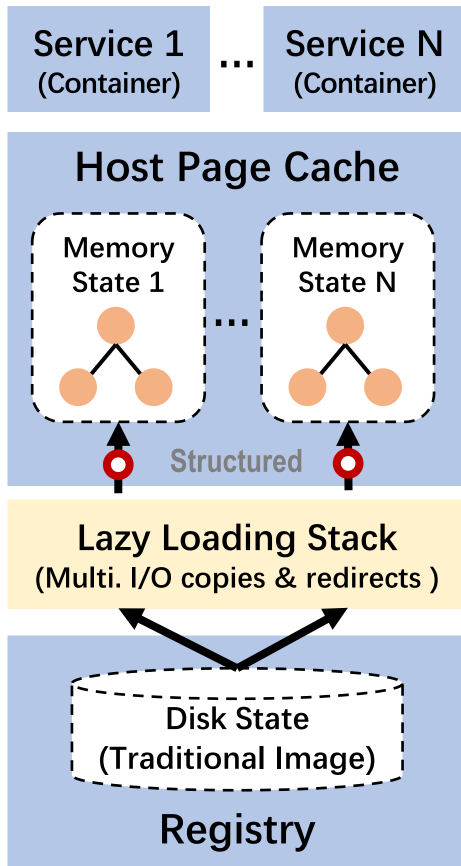
- Lazy loading solutions speedup the "*Deploy*" stage, but they surfer from high overhead in the "*Ready*" stage.

Observation 2

- Severe I/O amplification and inefficient network accesses are the main bottlenecks

Motivation: A New Image Abstraction

Traditional Image Abstraction



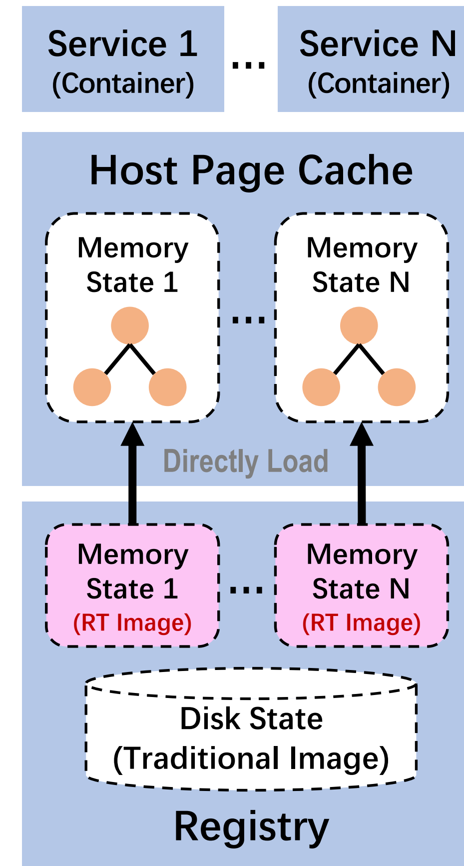
Global-Oriented: One image for multiple services

Storage-Oriented: Record the disk state

Cons:

- I/Os are difficult to aggregate
- I/O amplification is hard to eliminate
- I/O locality is difficult to optimize
- Need complex I/O forwarding

Runtime (RT) Image Abstraction



Service-Oriented: One image for one service

Memory-Oriented: Record the memory state

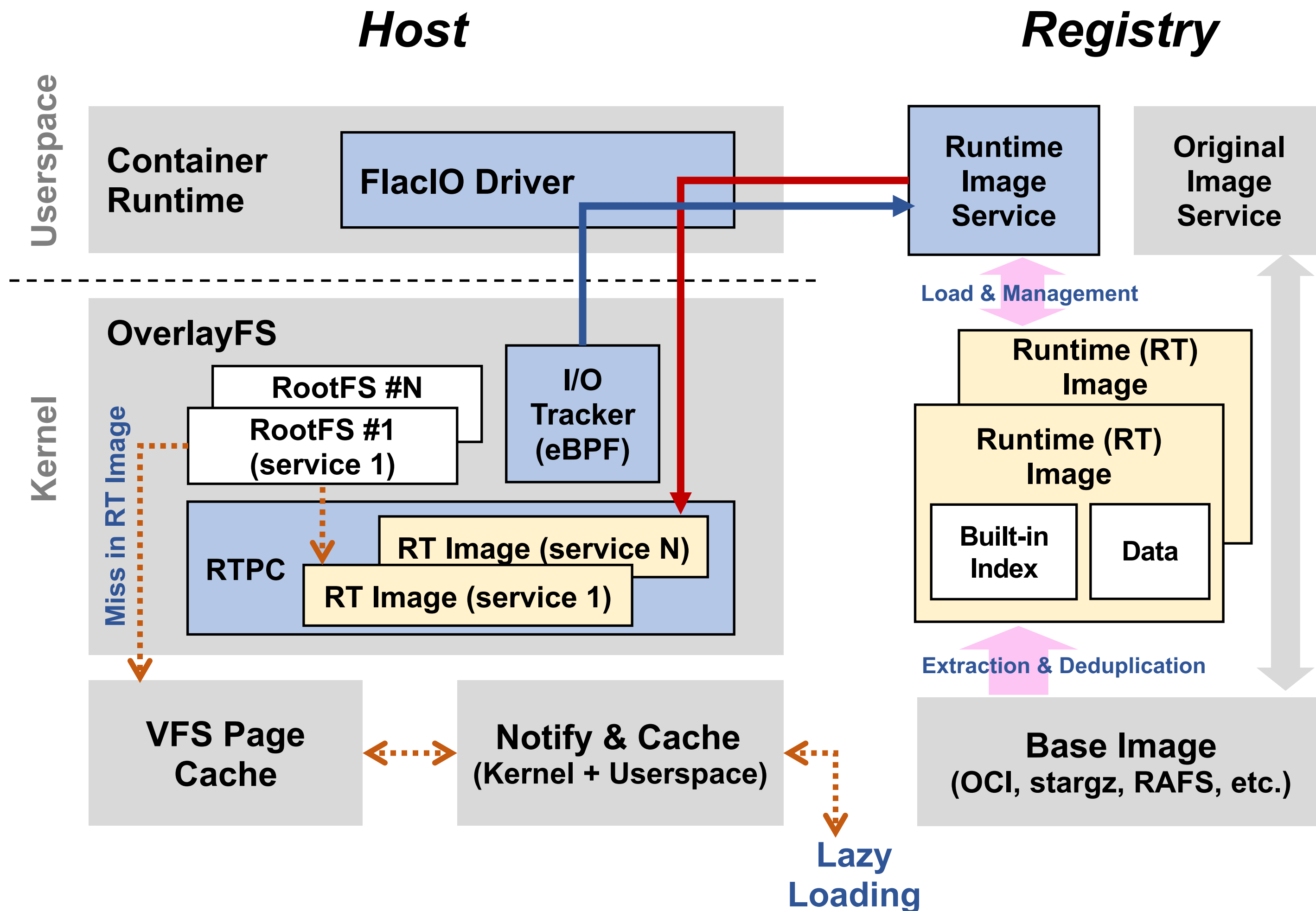
Pros:

- Efficient for network transfer
- Support fast root file system construction

FlacIO Design: Challenges

- **How to organize the runtime image in an efficient manner based on the new image abstraction?**
  - Accurate I/O tracing
  - Compact size
  - Ecosystem compatibility
- **How to build a lightweight I/O stack on the host for the runtime image?**
  - New OS primitives and cache mechanism
  - Compatible with the legacy I/O stack

FlacIO Design: Overview



Key Components

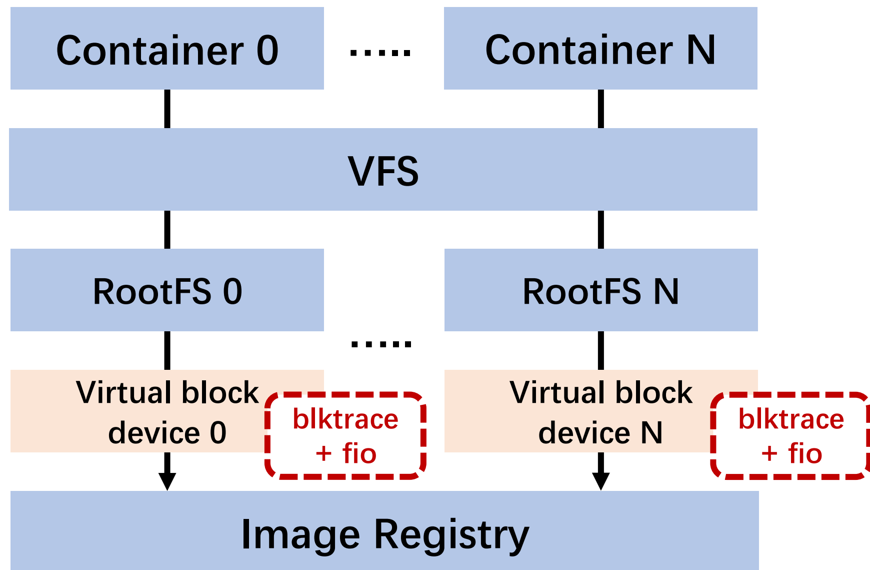
- FlacIO Driver
- I/O Tracker
- Runtime Page Cache (RTPC)
- Runtime Image service

Key Workflows

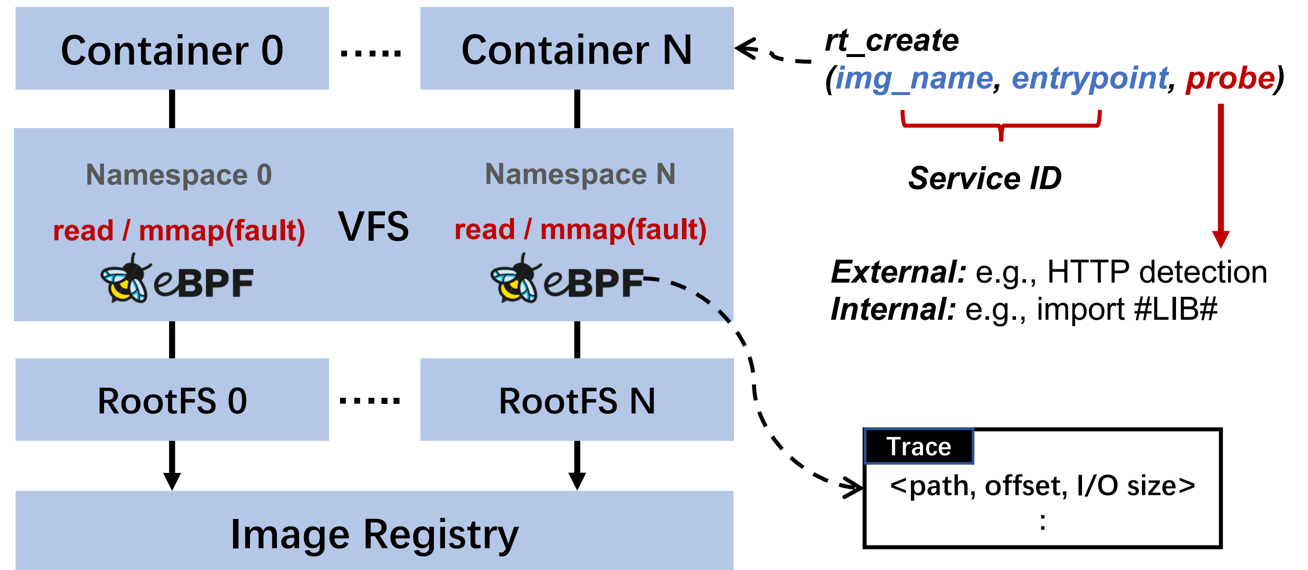
- RT Image Generation (blue arrow)
  - Track I/O in the file-level
  - Deduplication
- RT Image Loading (red arrow)
  - Incremental load by large I/O
  - Load-as-use
- File *read* / *mmap* (orange dotted arrow)
  - Compatible with legacy I/O

FlacIO Design: Runtime Image

Block-Level Tracing



Probe-based File-Level I/O Tracing



Disadvantages

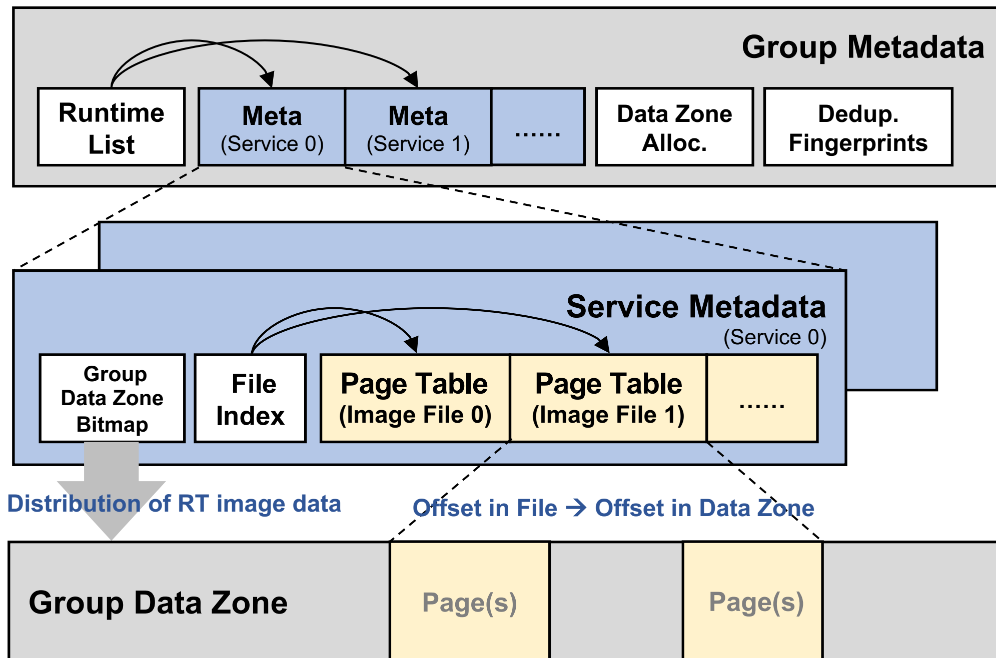
- Difficult to set tracing window accurately
- Rely on virtual block device

Advantages

- Stop tracing when the probe captures the corresponding event
- Suitable for any lazy loading system

FlacIO Design: Runtime Image

RT Image Organization



Definition of "Group"

- Consists of services that belong to the same base image (e.g., different entrypoints)
- Data belonging to the same group is *deduplicated*

Group Metadata

- Index the service metadata
- Record the information of the group data zone

Service Metadata

- Index the pages in the group data zone
- Record the page distribution on the data zone

Group Data Zone

- Continuous address space (as a mapping file)

FlacIO Design: Runtime Image

RT Image Generation Process (offline)

I/O Trace

Group Metadata

Runtime List | Meta (Service 0) | Meta (Service 1) | ...... | Data Zone Alloc. | Dedup. Fingerprints

Service Metadata (Service 0)

Group Data Zone Bitmap | File Index | Page Table (Image File 0) | Page Table (Image File 1) | ......

Distribution of RT image data

Offset in File → Offset in Data Zone

Group Data Zone | Page(s) | Page(s)

(New Page)

Initializing Meta

Parse Base Image

Dedup. & Create RT Image

RT Image Generation

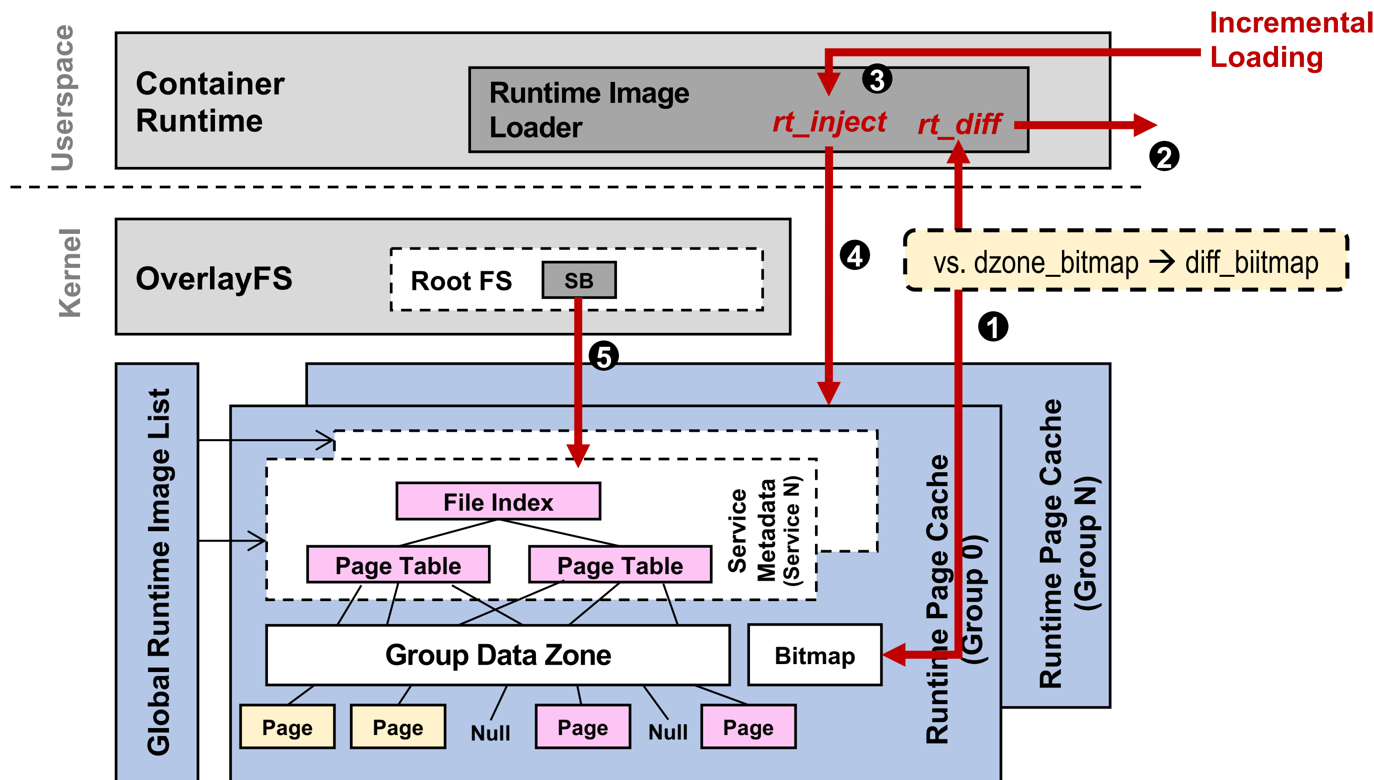
Base Image

OCI, stargz, RAFS ···

1. Initializes the relative group metadata and service metadata
2. Parses out the pages are in the record from the base image
3. Calculate the fingerprint of the page and check if it exists
4. (If new page) Update the dedup. fingerprint and add page to the data zone
5. Update the page table and the bitmap in the service metadata

FlacIO Design: Runtime Page Cache (RTPC)

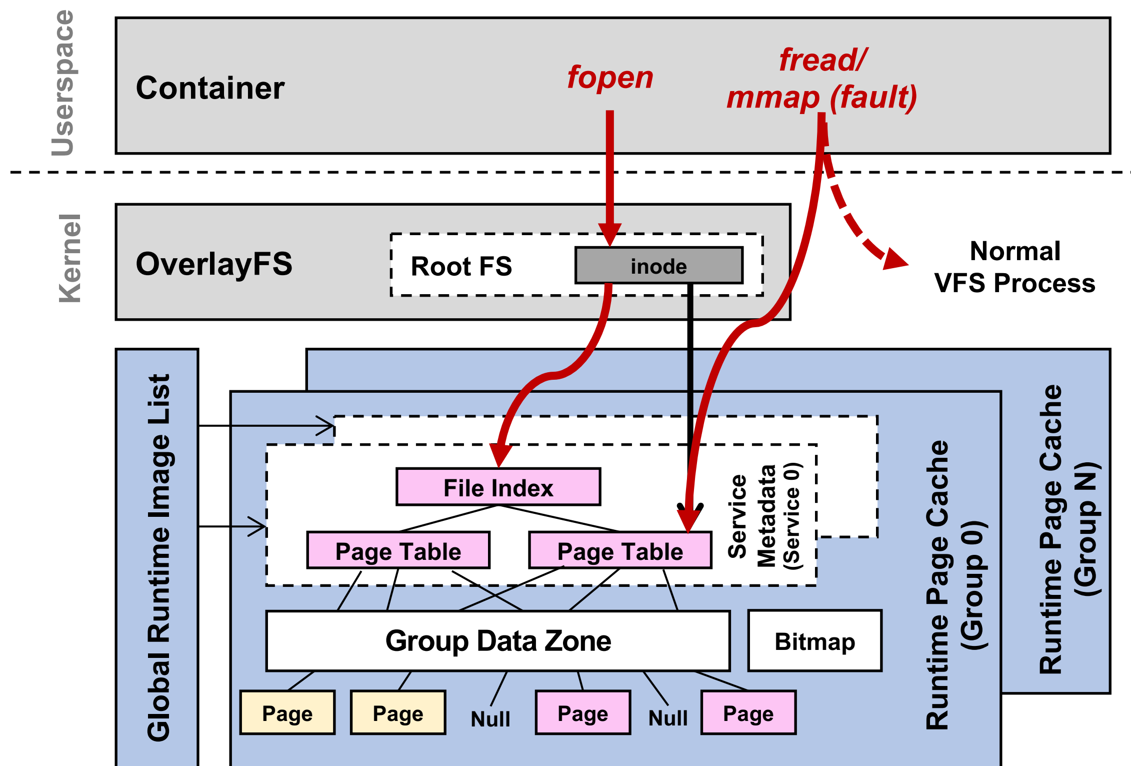
Incremental Loading & Root FS Construction



1. Get service metadata and call ***rt_diff*** to obtain the positions of the pages that are not loaded, return *diff_bitmap*
2. Send *diff_bitmap* to the image registry to request data
3. Pull the missed runtime image data from by a single large network I/O
4. Call ***rt_inject*** to load the service metadata and missed pages to RTPC
5. Call ***mount -rt service_id*** to build the root FS based on the runtime image

FlacIO Design: Runtime Page Cache (RTPC)

File Operations on RTPC



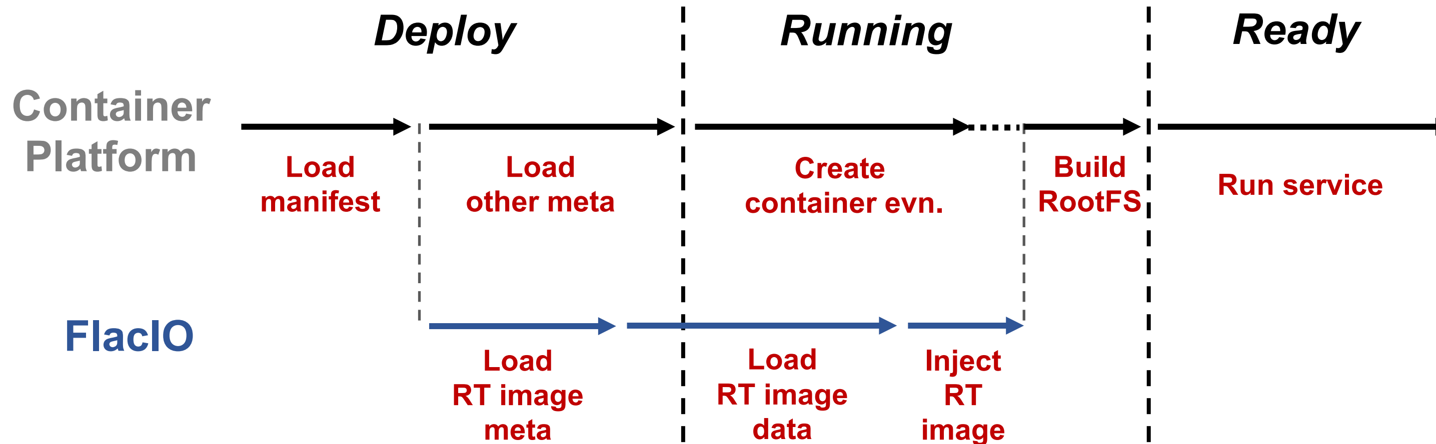
File open

- Search the file (by the path) in the file index in the service metadata
- (If exist) Points the inode to the corresponding page table

File read / mmap page fault

- Searche the page table in the service metadata based on the access offset
- (If no RTPC) Roll back to the traditional VFS process

Putting Everything Together



Implementation

- openEuler 22.03 LTS with Linux 6.5 kernel
- 188LOC / 182LOC for porting FlacIO to CRFS / Nydus (in Containerd)
- Many FlacIO's processes can be overlapped by the container platform's processes

***vs.* Expanded Prefetch**

- Simply expand the loading unit
- E.g., Nydus, CRFS

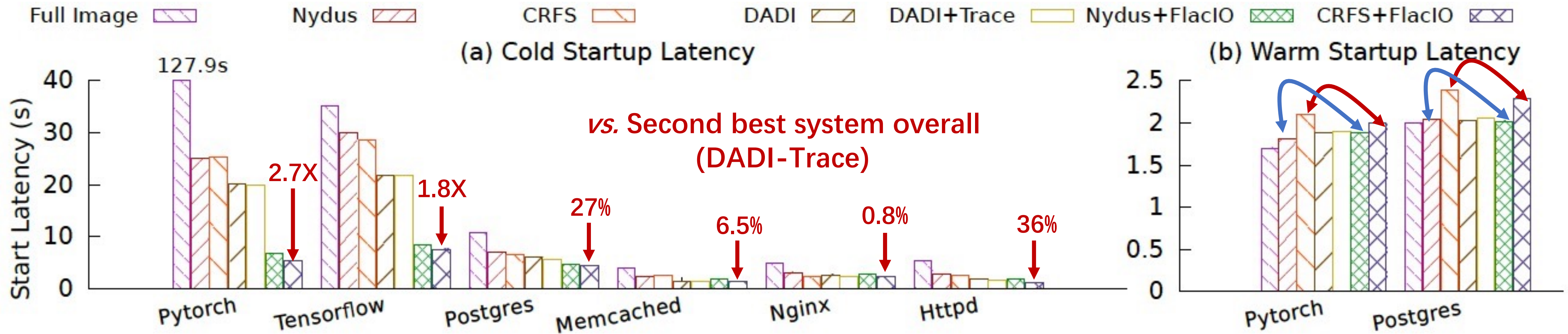
***vs.* Prioritize Files Prefetch**

- Strong reliance on user experience
- Large loading unit (file granularity)
- E.g., CRFS

***vs.* Trace Replay**

- Problem of accurate tracing and efficient aggregation
- Rely on an independent ecosystem
- E.g., DADI

Container Startup Performance



Evaluation Setup

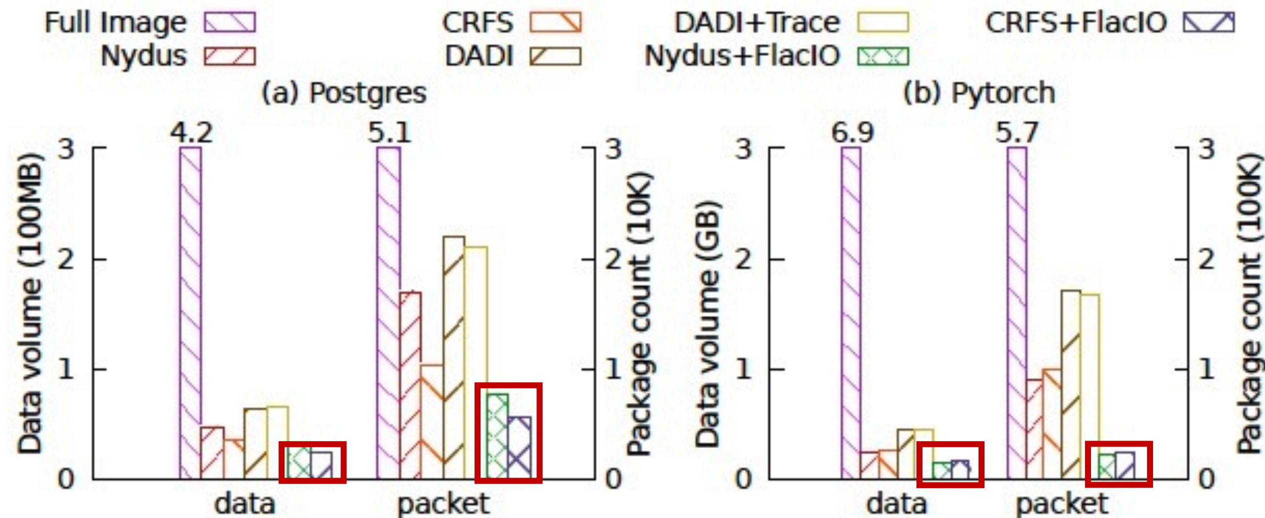
- **Hardware:** 24-core CPU @ 2.30GHz, 256GB DRAM, connected to the image registry through 10Gbs network
- **Container Probe Setting (FlacIO):** Pytorch/Tensorflow -- successful loading of the core libraries; Other -- successful access of the HTTP ports

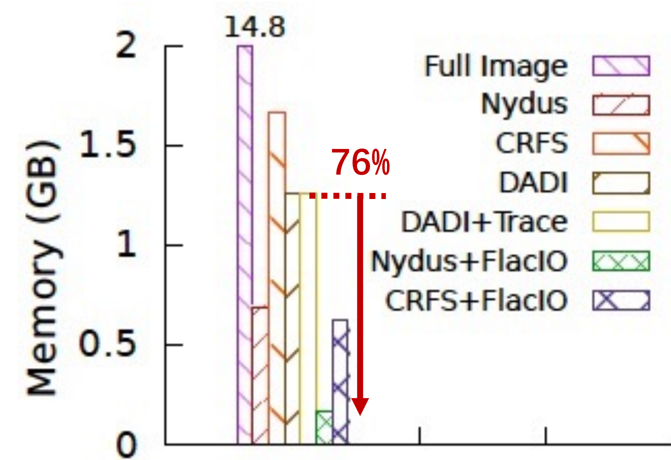
Summary

- Improves performance by up to 2.7X in cold startup (vs. the second best)
- Almost no additional cost for warm startup, compared to w/o FlacIO

Design Analysis

Network & Memory Overhead





Network traffic and Packet Count

- Use *tcpdump* to capture network requests during cold startup
- > 1.6X in data volume and packages than FlacIO (in *Pytorch*)

Memory Footprint on Host

- Use *vmstat* to profile the memory footprint
- 1.1% to 24% of other (Nydus+FlacIO)

Summary: Accurate aggregated I/O and streamlined I/O stack for low network overhead and low memory footprint

Design Analysis

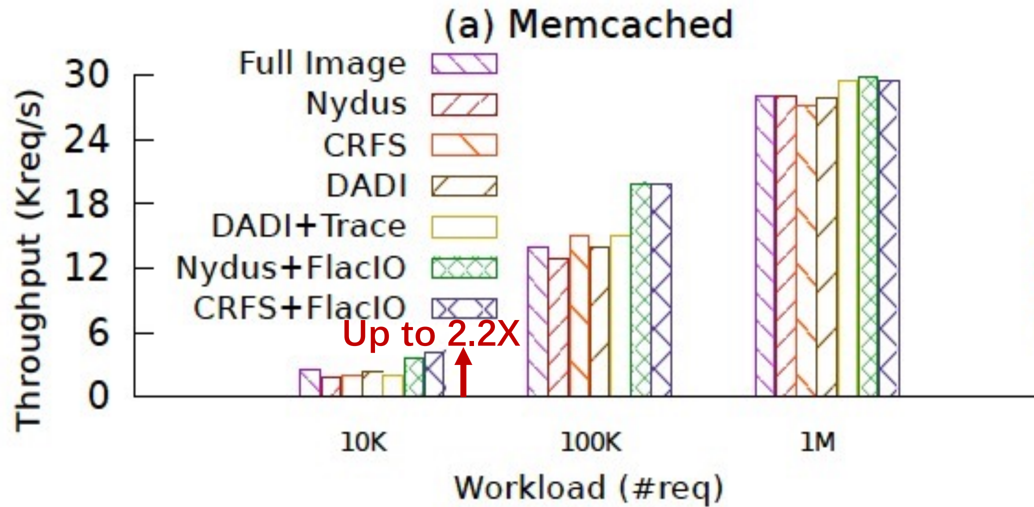
Storage Space Overhead

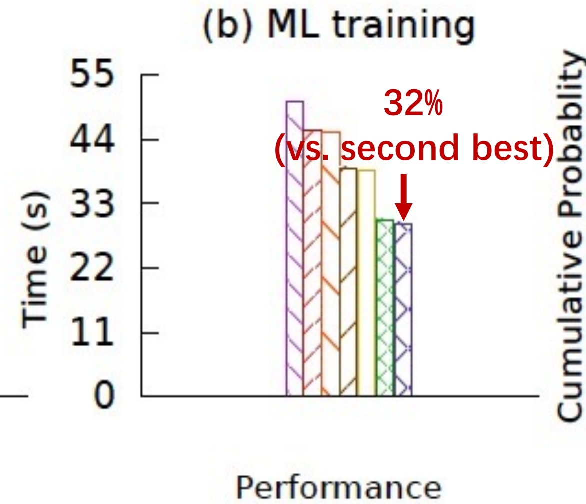
Unit: MB	CRFS Img. (compressed)	Nydus Img. (compressed)	Runtime Img.	% (CRFS / Nydus)
Nginx	69.7	94.5	4.3	6.2% / 4.6%
Httpd	64.7	88.5	2.9	4.5% / 3.3%
Redis	51.9	71.8	5.1	9.8% / 7.1%
Memcached	33.6	45.3	2.8	8.3% / 6.2%
Tomcat	213.4	262.5	34.1	16.0% / 13.0%
Wordpress	237.4	321.5	19.1	8.0% / 5.9%
Postgres	152.1	210.9	16.2	10.7% / 7.7%
MySQL	178.1	247.6	31.8	17.9% / 12.8%
Pytorch	3348.5	4229.1	146.5	4.4% / 3.5%
Total	4349.4	5571.7	262.8	6.0% / 4.7%

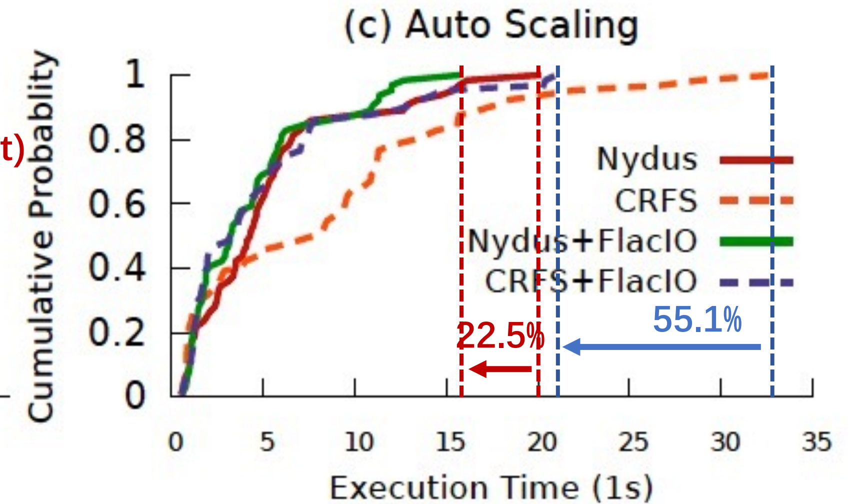
Summary

- Runtime images account for 6%/4.7% (CRFS/Nydus) of the total size
- Pay around 5% of storage overhead for up to 2.4 times speedup
- Storage overhead can be further reduced by compression

Real-World Application







Object Storage Scenario

- Start Memcached from the cold state
- Insert 10K, 100K, 1M 2KB objects

Summary
Make object storage systems on the cloud warm up faster

ML Training Scenario

- Train MNIST dataset with Keras
- Dataset is stored in local FS

Summary
Reduce on-demand loading cost during training

Auto-Scaling Scenario

- 8 node, 8 *Postgre* per node
- Batch cold start

Summary
Reduce tail latency of dynamic expansion

Conclusion

- **Analysis of the bottlenecks of existing lazy loading solutions**
  - Heavy network traffic, high I/O amplification, and complex I/O stack
- **Runtime image, an efficient image abstraction for network and I/O stack**
  - From Global- and Storage-Oriented to Service- and Memory-Oriented
- **FlacIO, an end-to-end solution to accelerate container cold startup**
  - Adapted to mainstream lazy loading systems, with significant performance gains

Thanks :)