CoFS: A Filesystem for Fast Container Startup

Li Wang, Jinxu Du, Yang Yang, Qingbo Wu, Tao Liu, and Haoze Wu

KylinSoft Co., Ltd.

Contact: laurence.liwang@gmail.com

The Problem: Container Cold Start

76%

of startup time spent pulling images

6.4%

of pulled data is actually read

(Harter et al., FAST '16)

Cold Start Pipeline:

Bottleneck

High startup latency violates SLAs in serverless computing and during burst scaling.

On-Demand Pulling

Core idea: containers run before the entire image is downloaded, fetch data on-demand.

System	Level	Implementation	Key Limitation
Nydus-fuse	Filesystem	FUSE-based	High lookup latency Context switch overhead
eStargz	Filesystem	FUSE-based	Same FUSE overhead
Nydus-erofs	Filesystem	In-kernel erofs + fscache	Long call chain Fscache eviction issues
Overlaybd	Block	Block-level layering	No page cache sharing across containers

FUSE Has Two Costly Bottlenecks

Bottleneck 1: Metadata Lookup

Path traversal is iterative:
/usr/local/bin/app requires 4 separate lookups

Each lookup: VFS → FUSE driver → userspace daemon → response

Cost: multiple context switches + request copies per path component

---

Bottleneck 2: Data Access

Even for cached/downloaded data, read requests are forwarded to userspace

Cost: unnecessary context switches + data copy for every read

The Key Insight

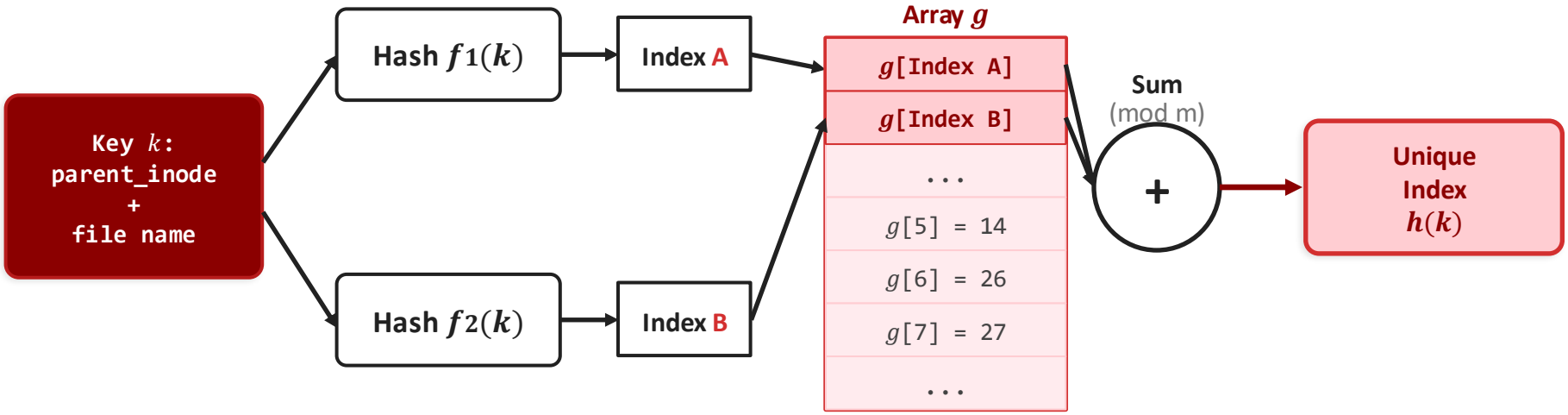
“Container images are built once with a fixed, read-only filesystem tree.”

---

1. The full set of filenames is known at build time
2. A perfect hash function can be precomputed over all entries
3. Lookups become O(1) — no directory traversal needed at runtime

Minimal Perfect Hash Function

A hash function that maps m keys to exactly $[0, m-1]$ with no collisions.



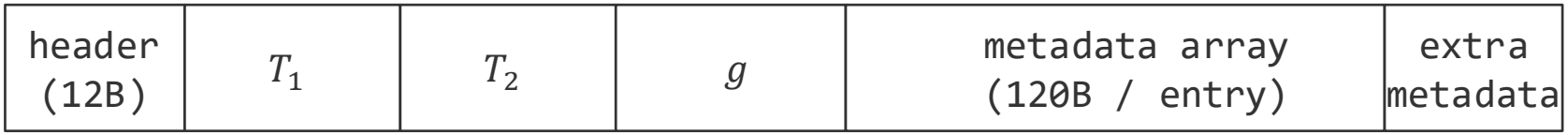
$$h(k) = \left(g\left(f_1(k)\right) + g\left(f_2(k)\right)\right) \bmod m$$

Key properties:

Collision-free · Space-optimal O(m) · O(1) lookup · Linear-time construction

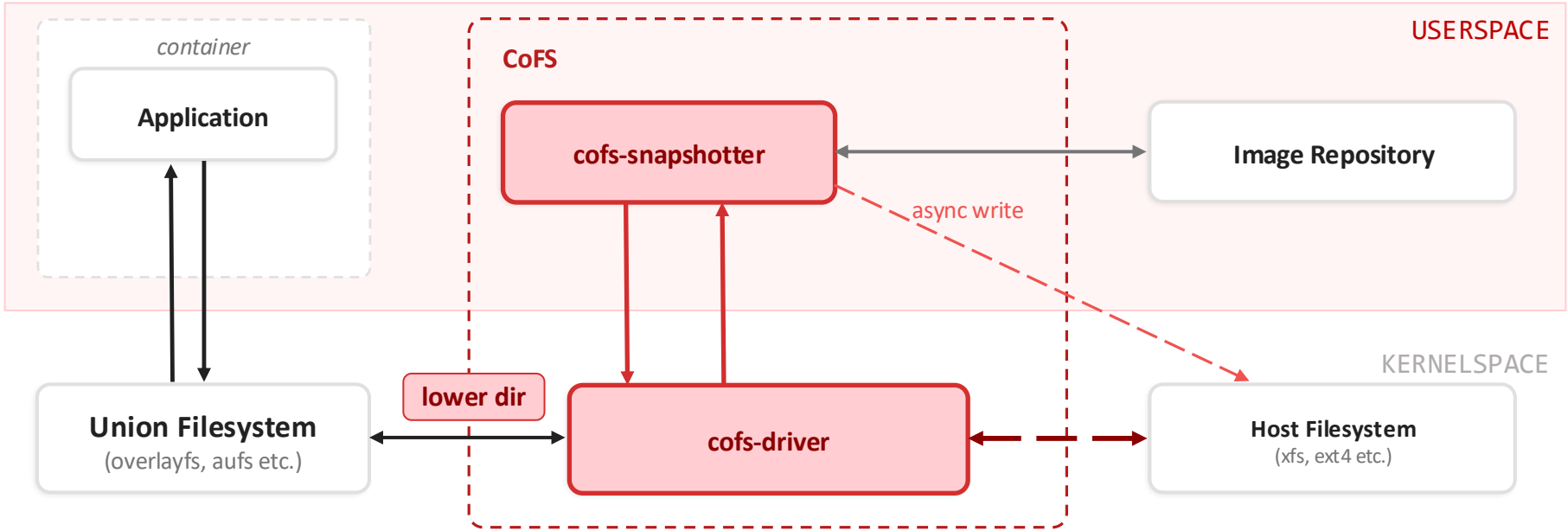
Metadata File Layout: cofs.inode.array

Metadata with a binary format indexed by MPHF hash value.



- Each metadata entry contains all inode metadata
- Filenames ≤ 16 bytes stored inline; longer filenames in extra metadata tail
- Space overhead for MPHF: $O(m)$ — ~9.5 MB for 1M files

CoFS Architecture



Fast Lookup Path

1. Input: parent inode number + filename
2. Compute MPHF hash
3. Use hash value as index for metadata array
4. Read the 120-byte entry
5. Verify: compare parent inode + filename
6. Match → construct in-memory inode;
   no match → file does not exist

vs. ext4 lookup

I/O count depends on:

- Directory size
  (linear scan or htree traversal)
- Directory depth
  (iterative per component)

CoFS: ≤ 1 I/O regardless

Parallel Path Resolution

A second MPHF keyed on full file paths enables parallel inode resolution.

Mechanism:

1. **Detect a image container file is opened**
2. **Dispatch to kernel work queue thread**
3. **CoFS bottom-up inode resolution**
4. **Stop when inode already in memory**

Concurrent Execution:

VFS (top-down):

`/ → usr → local → bin → app`

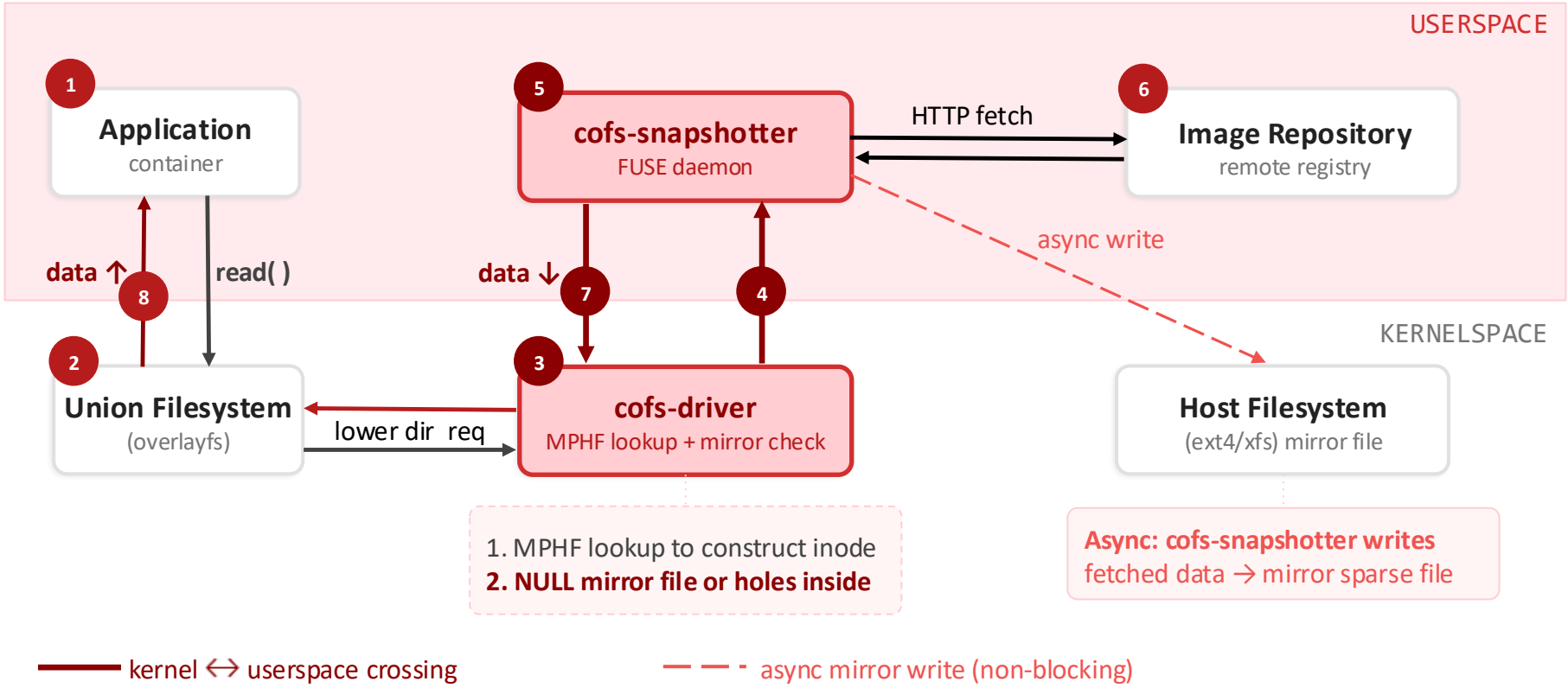
CoFS worker (bottom-up):

`app → bin → local → usr → /`

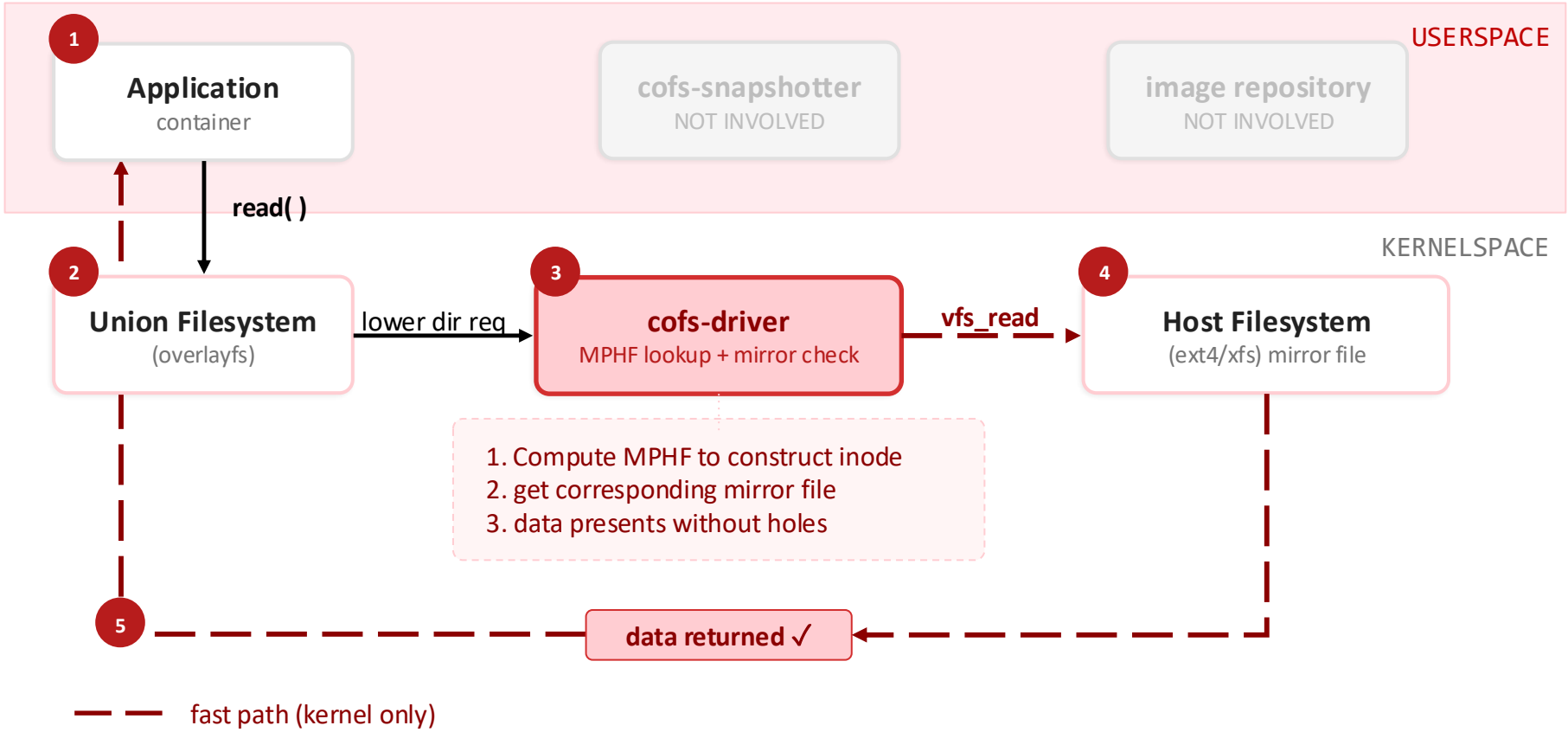
↑ They meet in the middle ↑

Runs concurrently prepopulating inode cache

Data Access: Uncached Data Read

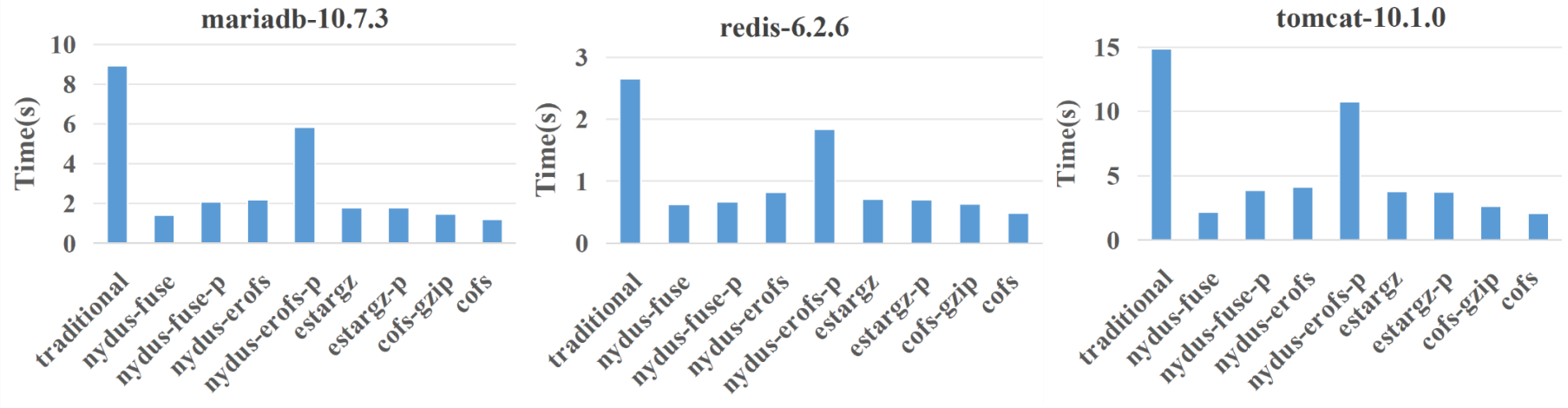


Data Access: Cached Data Read



Evaluation: Cold Startup Time

Dual Xeon E5-2640, 128GB RAM, 4TB HDD, 1GbE — simulates constrained bandwidth.



Key findings:

- CoFS outperforms all systems on every container tested
- Background prefetch degrades performance — wastes bandwidth on unneeded data
- Nydus-erofs worse than Nydus-fuse due to long fscache call chain

Evaluation: Lookup & Read Performance

Lookup Performance (vs. fuse-loopback)

73–86%
improvement in average lookup time

28% additional gain from parallel lookup

Cached Data Read Performance

Kernel-space systems (CoFS, traditional, Nydus-erofs):
Nearly identical bandwidth & latency

FUSE-based (Nydus-fuse, eStargz):
Lower due to userspace overhead

MPHF Construction Overhead

Files (m)	1K	10K	100K	1M
Avg build time	0.016s	0.141s	1.899s	34.042s

Summary

1. **MPHF-Based Lookup**

   Precomputed; O(1) metadata lookup from kernel space;
   No userspace round-trip.

2. **Kernel-Space Fast Path**

   Sparse mirror files cache downloaded data on host FS.
   Cached reads served entirely in kernel.

3. **Parallel Path Resolution**

   Second MPHF keyed on full paths;
   Kernel worker resolves inodes bottom-up, concurrent with VFS.

Result:
CoFS outperforms all state-of-the-art systems; **73–86% lookup improvement** over FUSE.