CHINA LINUX KERNEL
中国Linux内核开发者大会

vivo

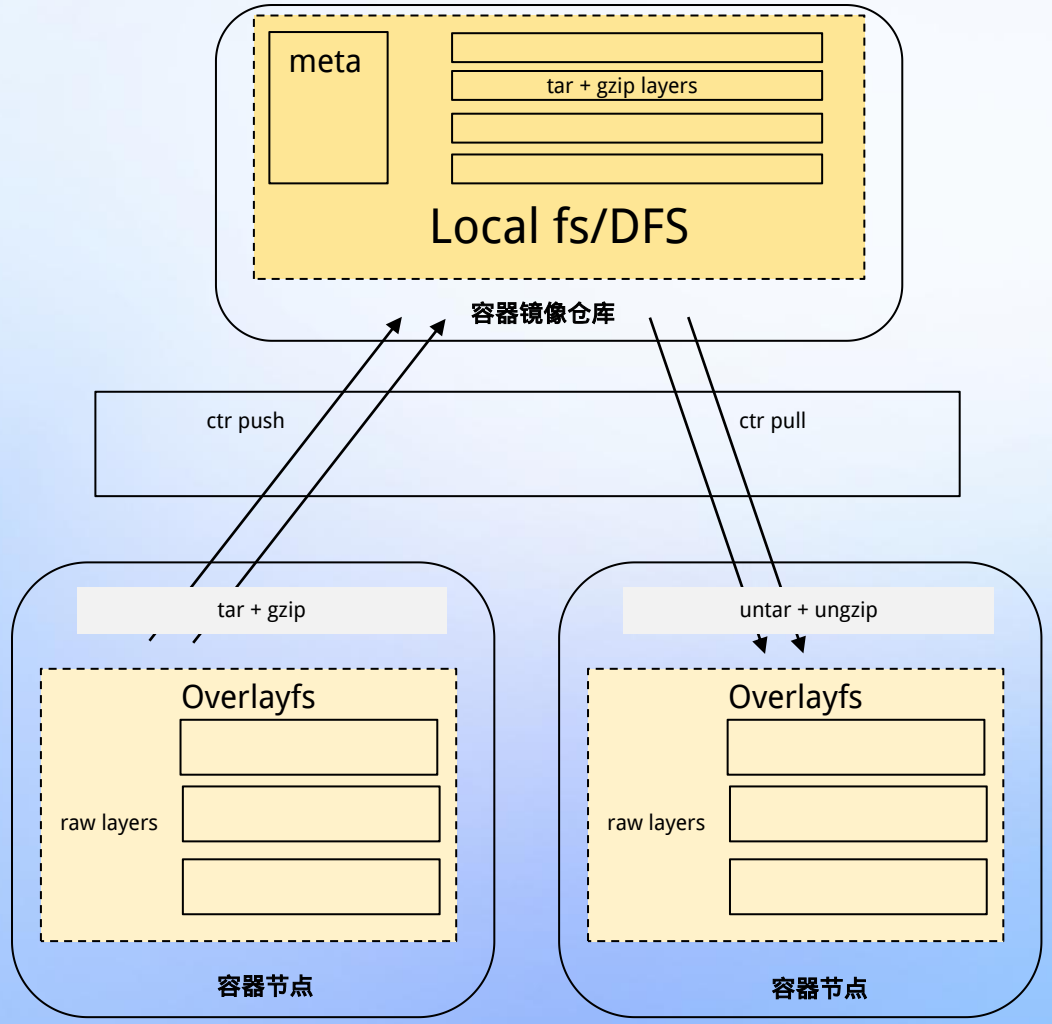
TrIO:利用IO轨迹加速容器冷启动

分享人：黎红波 （华为OS内核实验室）

目 录

CONTENTS

背景介绍：容器生态



容器镜像：是一个静态的、只读的文件，它是应用程序、运行时环境、依赖项和配置等的打包，是容器的基础模板。

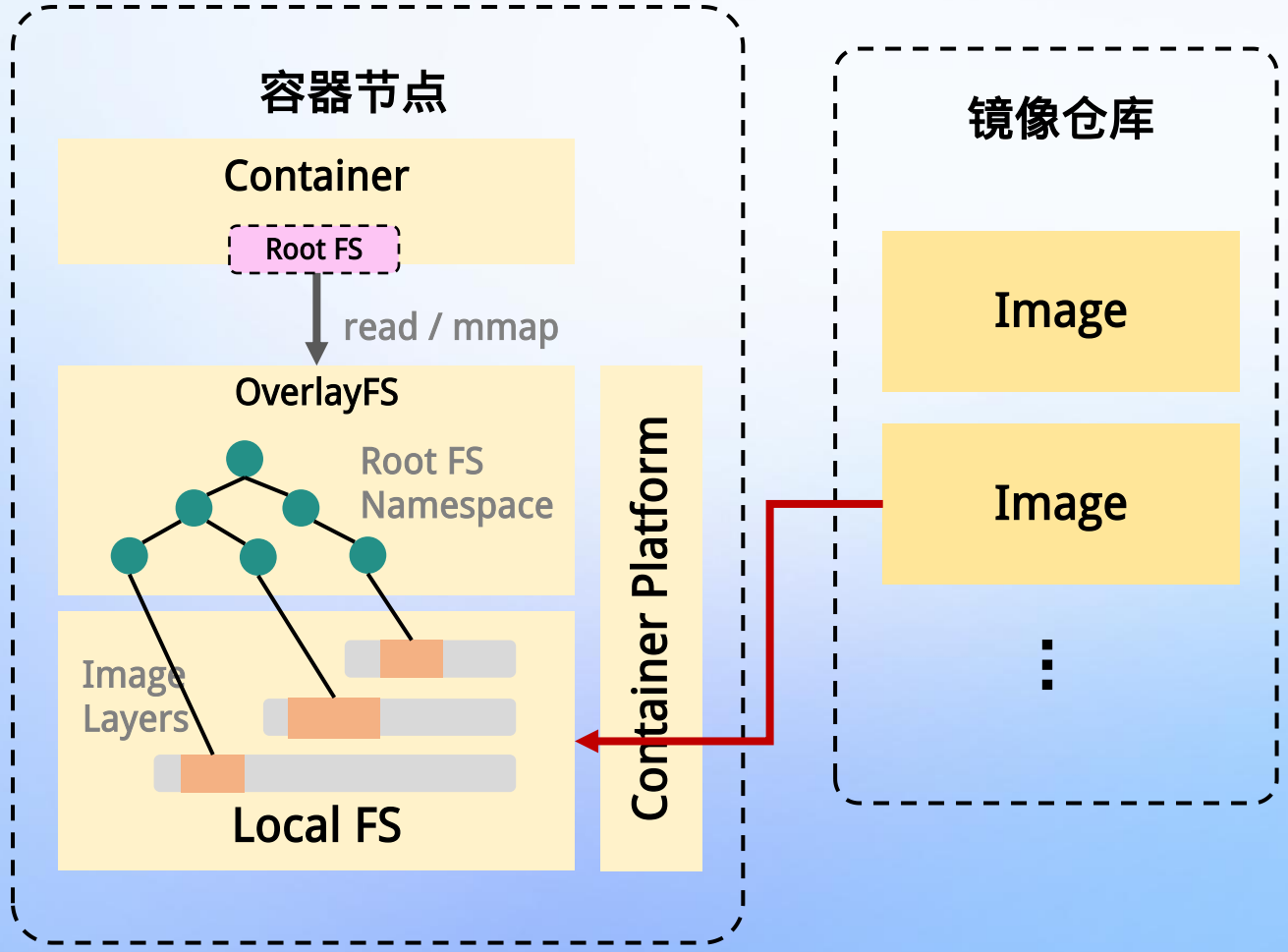
角色：容器节点、镜像仓库，两者通过网络进行数据交互。

生态：容器管理工具负责容器的生命周期管理，包括push、pull、create、run等。

docker/ctr pull：镜像仓库　本地节点

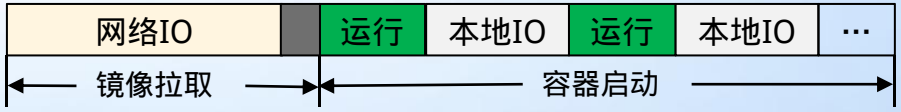
1. 获取manifest文件
2. 获取config文件、Layer索引等
3. 并发下载每一层镜像到本地
4. 解压到本地
5. …

传统方式：容器镜像全量加载



容器启动效率低

- 启动过程需要等待镜像全量拉取到本地，拉取过程网络开销大
- 整个启动过程存在大量的IO放大



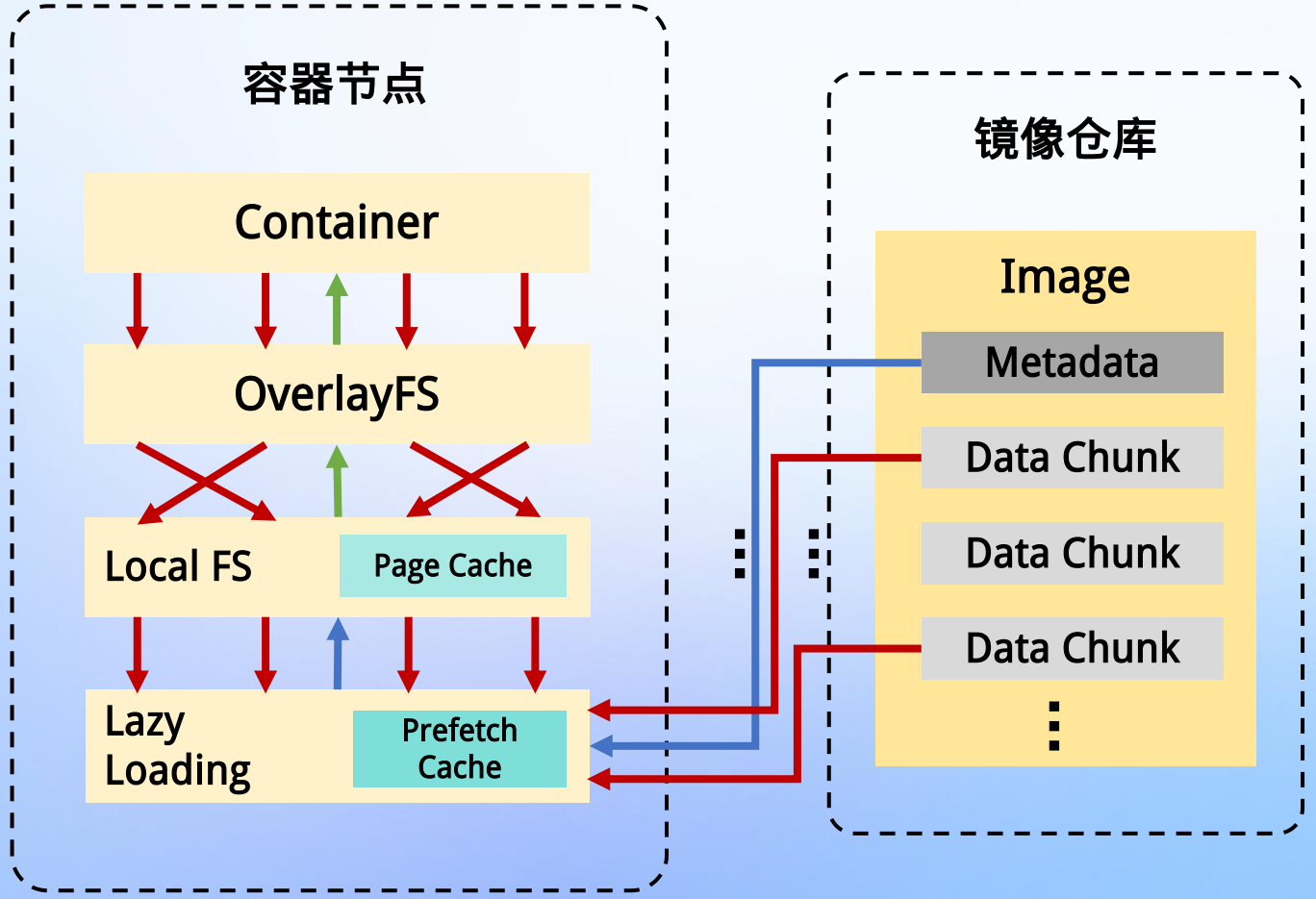
方向1: 直接优化IO

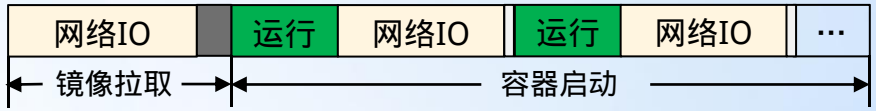
- 主机侧镜像缓存
- 容器镜像按需加载

方向2: 状态迁移

- 通过进程fork方式
- 通过P2P加载镜像

现有优化：容器镜像按需加载





阶段1：Deploy（ → ）

获取构建容器运行RootFS相关的元数据

阶段2：Running（ → ）

创建容器运行时资源（如网络、cgroup等）

阶段3：Ready（ → ）

根据容器的Entrypoint运行容器进程

现有方案对比分析

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

实验环境

- 容器节点与镜像仓库之间网络：10Gbs
- 冷启动一个Pytorch容器

观察 1

- 容器镜像懒加载能明显加速Deploy阶段，但是对Ready阶段带来了额外的开销

观察 2

- 容器镜像懒加载仍然存在IO放大，并且网络IO效率低

现状&动机

问题1：IO存在放大

- 传统方案需要读取所有镜像数据
- 按需加载方案需要读取IO所在的整个文件或数据块
- 传统方案中有效数据约6.4%，按需加载中有效数据约43%

动机1： 容器启动有确定的IO行为，精确拉取所需数据，减少容器启动过程中的IO量

IO轨迹追踪

问题2：IO行为不友好

- 多次小IO：启动过程会读多层镜像的多个文件/数据块，频繁与镜像仓库直接进行交互

动机2： 将所需数据连续存储，采取IO大聚合方式，提高IO效率&带宽利用率

IO聚合

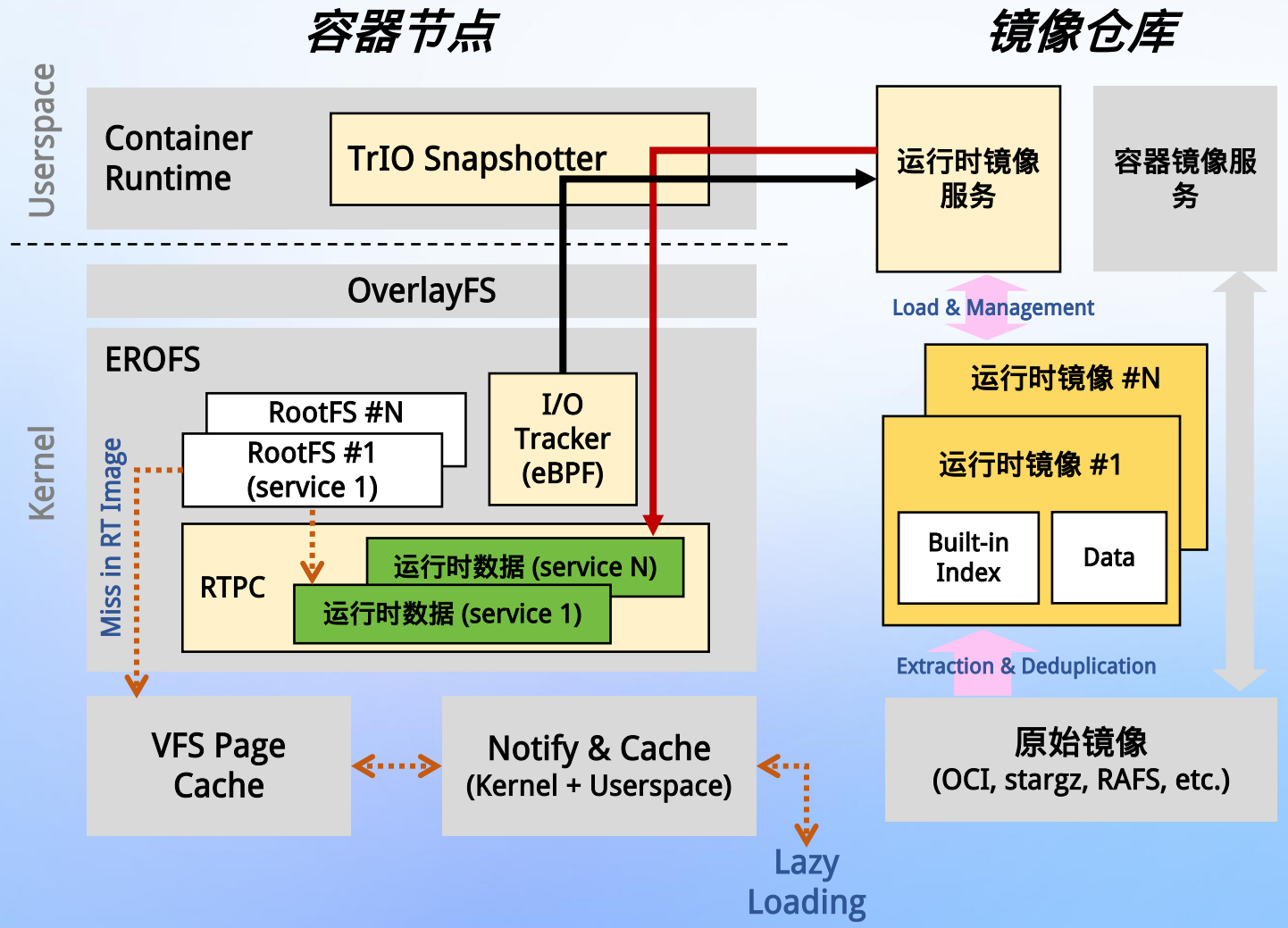
问题3：IO栈复杂

- 当前按需加载方式从镜像仓库拉取的数据无法直接填充进page cache，导致IO路径长

动机3： 将启动所需镜像数据直接加载到内核page cache中，减少IO访问路径

运行时镜像

TrIO：利用IO轨迹加速容器启动



主要组件

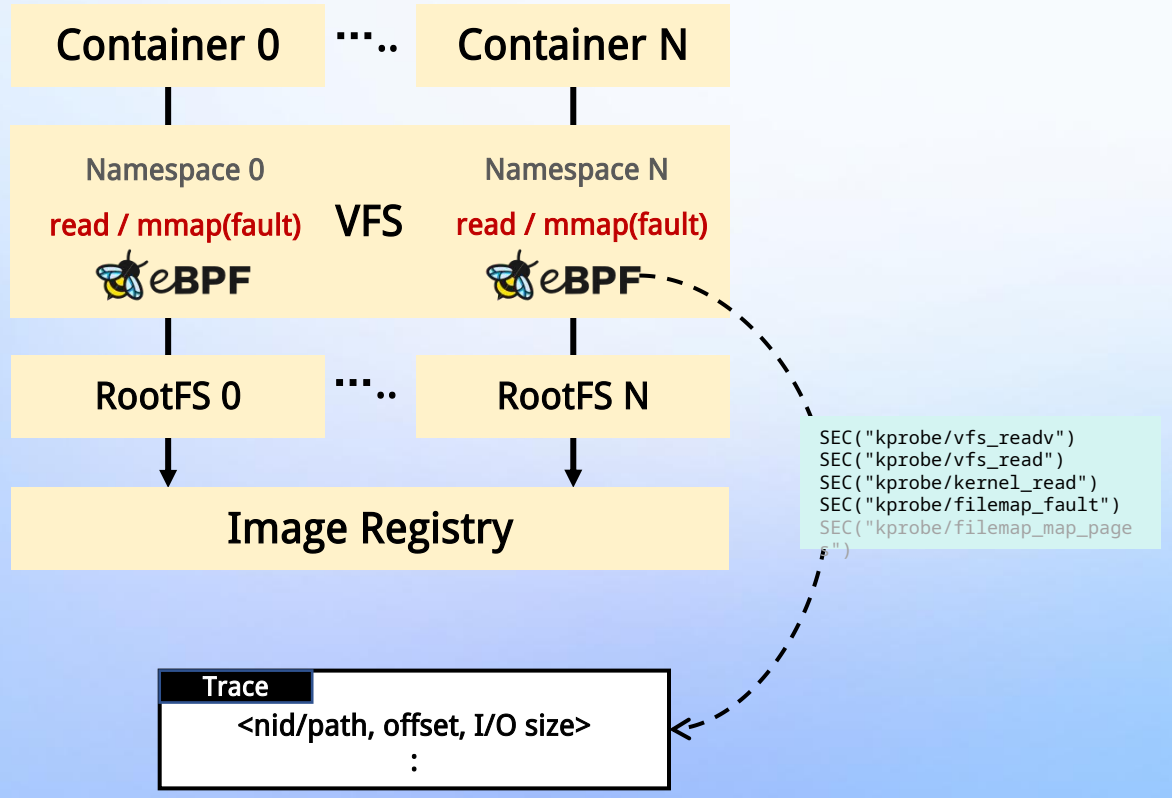
- TrIO Snapshotter
- I/O Tracker
- 运行时Page Cache(RTPC)
- 运行时镜像服务

主要流程

- 运行时镜像生成 (→)
  - 基于文件IO的追踪
  - IO去重
- 运行时镜像加载 (→)
  - 通过大IO进行聚合
  - 服务粒度进行按需加载
- 读取流程：read/mmap (⇢)
  - 与默认流程兼容

TrIO设计：运行时镜像构建

File-Level I/O Tracing



主要步骤：

- 追踪容器启动（直到事件满足）时的IO
- 预处理（去重、合并、筛选等）形成运行时镜像元数据
- 对运行时数据按页对齐方式紧凑编排

初始化iotracker模块
$ insmod rio_tracker_mod/rio_tracker.ko tracker_output="trace.txt"
$ iotracker/.output/iotracker

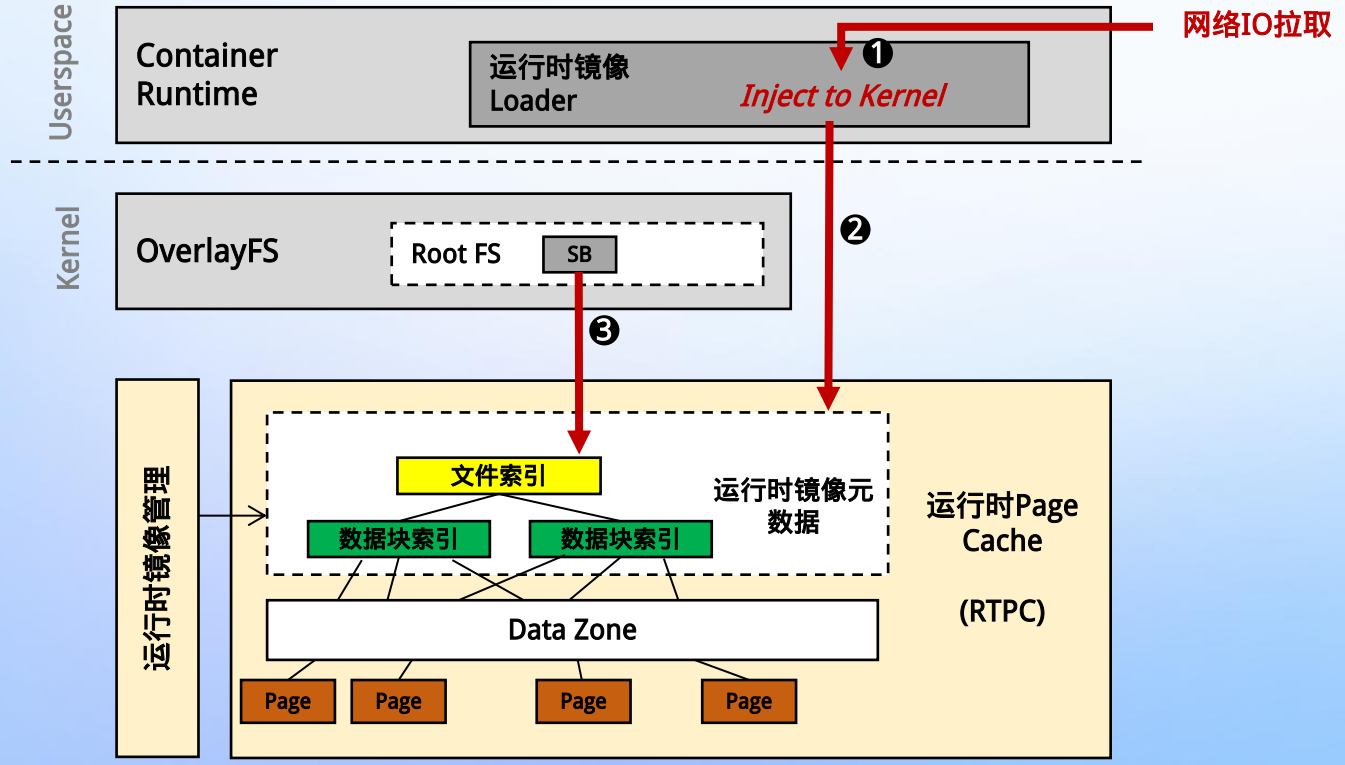
开启Tracing功能
$ echo -n TRACE_HOST_NAME > /sys/kernel/rio_tracker/host_ns
$ echo 1 > /sys/kernel/rio_tracker/enable

运行指定容器
$ 在指定namespace下运行目标容器

停止Tracing
$ echo 0 > /sys/kernel/rio_tracker/enable
$ echo 65536 > /sys/kernel/debug/fault_around_bytes # recovery
$ echo 1 > /sys/kernel/rio_tracker/dump

TrIO设计：运行时镜像加载

RT Image Loading & Root FS Construction



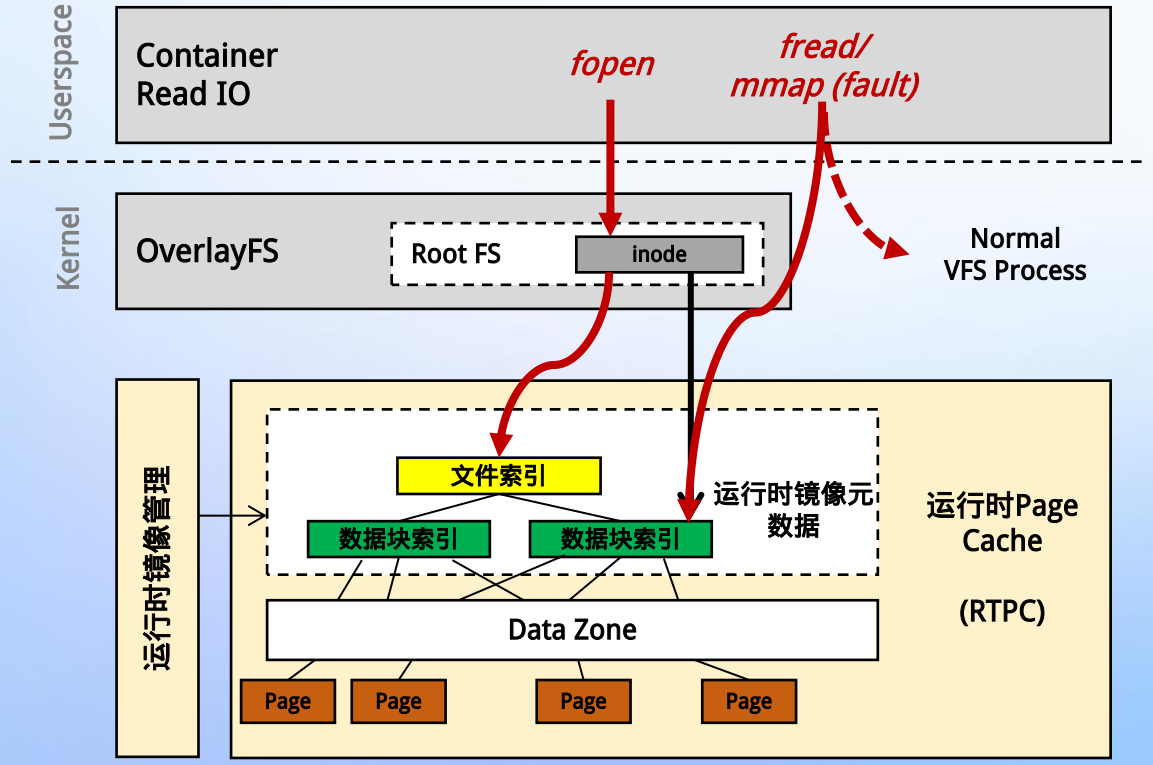
❶ 容器获取manifest文件后以大IO方式从镜像仓库拉取运行时镜像

❷ 将运行时镜像从用户态映射进入内核（ *mount --trio_data #DATAFILE --trio_meta #METAFILE* ）

❸ 读取运行时数据区并形成RTPC，解析元数据构建RootFS初始数据

TrIO设计：运行时镜像IO读取

File Operations on RTPC



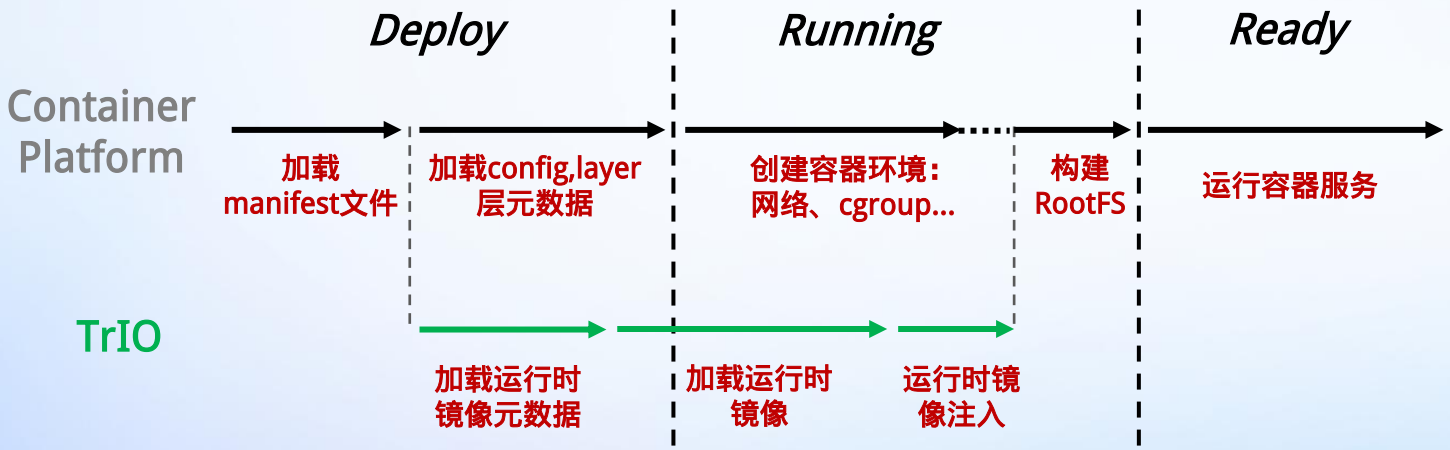
文件open操作

- RootFS的superblock部分记录了image id
- 根据<image id, filepath>从索引树中找到对应文件数据块索引句柄
- 如果存在，则将句柄关联到i_private字段

文件读取操作：read / mmap page fault

- 根据<offset, length>进行区间匹配，获取对应的数据
- 如果不在RTPC中，则退化为原生读取流程

TrIO设计：用户态适配



用户态适配

- 基于CRFS/Nydus snapshotter: ~180LOC (in Containerd)
- 运行时镜像数据的拉取和加载实现堆叠
- 运行时镜像管理服务（可以扩展OCI yaml字段按照原始镜像格式进行存储）

vs. Prefetch

- 盲目加载
- E.g., Nydus, CRFS

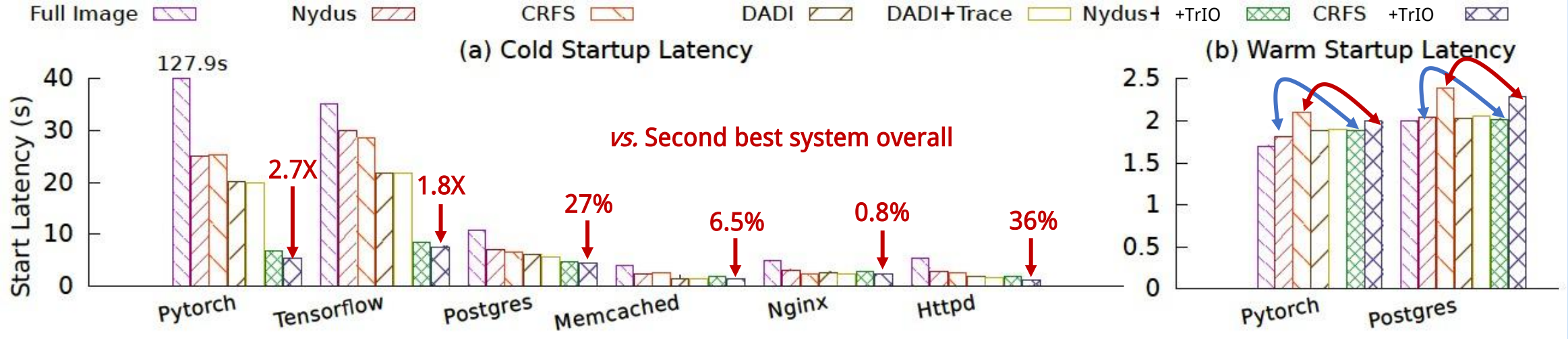
vs. Prioritize Files Prefetch

- 依赖经验
- 文件粒度加载
- E.g., CRFS

vs. Trace Replay

- 难以精确追踪和聚合
- 生态不兼容
- E.g., DADI

TrIO效果验证：典型容器场景



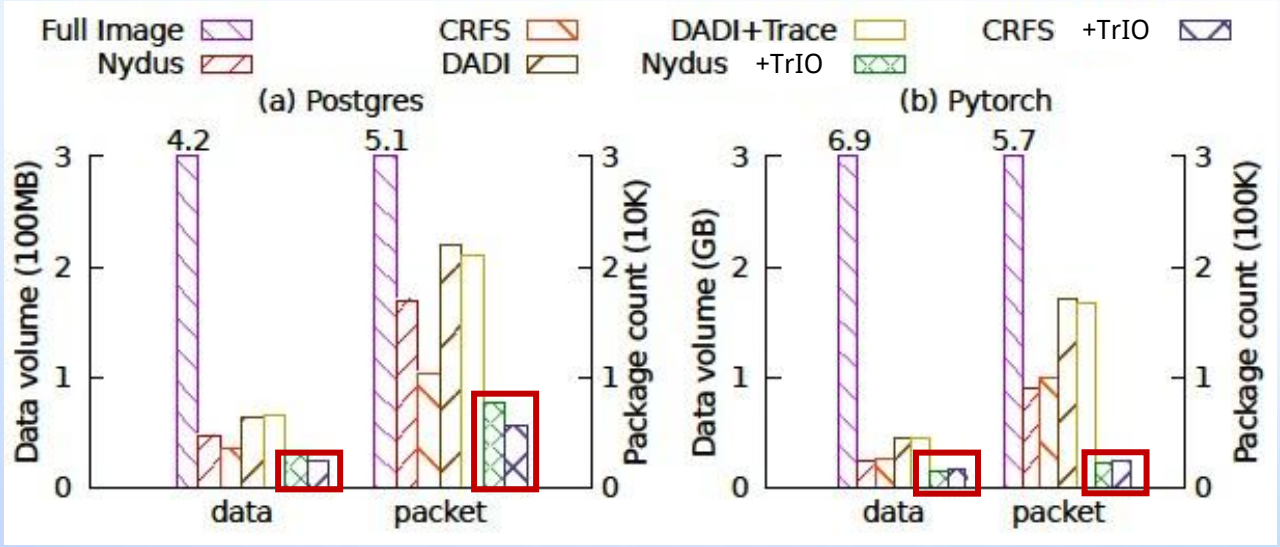
Evaluation Setup

- **硬件平台：**24-core CPU @ 2.30GHz, 256GB DRAM, 和 10Gbs network
- **容器启动标准：**应用类——程序运行成功；服务类——相应端口能够访问成功

总结

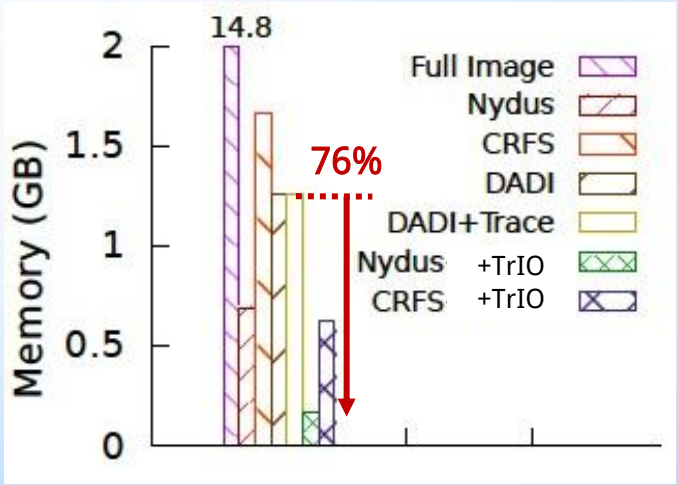
- TrIO 在几种经典的容器（服务类&应用类）使用场景对容器冷启动过程均有提升
- 热启动场景基本无额外开销

TrIO效果验证：低内存开销



网络开销

- 使用*tcpdump*跟踪容器启动过程的网络包开销
- 更低的网络开销，数据量降低> 1.6X (*Pytorch*)



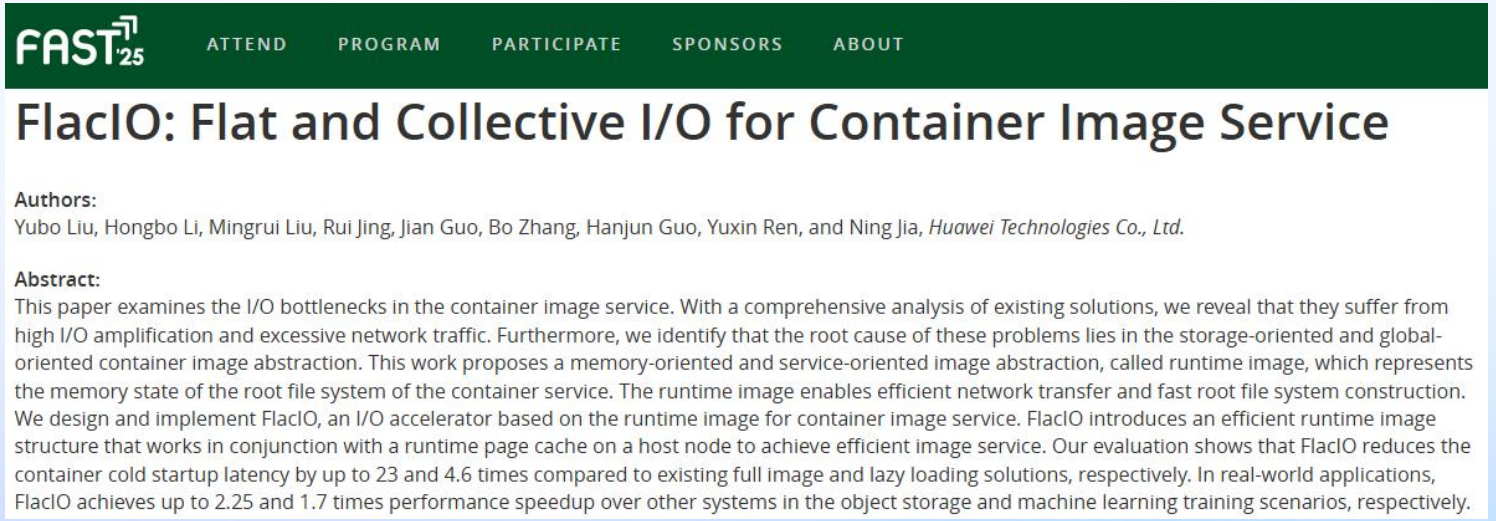
主机侧内存开销

- 使用*vmstat*观察内存使用
- 更低的内存占用开销

总结: 精确的IO聚合和轻量软件栈使得网络开销和内存开销降低

TrIO开源情况

- 相关成果发表在存储顶会 FAST’25
  https://www.usenix.org/conference/fast25/presentation/liu-yubo



FAST'25 ATTEND PROGRAM PARTICIPATE SPONSORS ABOUT

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

Authors:
Yubo Liu, Hongbo Li, Mingrui Liu, Rui Jing, Jian Guo, Bo Zhang, Hanjun Guo, Yuxin Ren, and Ning Jia, *Huawei Technologies Co., Ltd.*

Abstract:
This paper examines the I/O bottlenecks in the container image service. With a comprehensive analysis of existing solutions, we reveal that they suffer from high I/O amplification and excessive network traffic. Furthermore, we identify that the root cause of these problems lies in the storage-oriented and global-oriented container image abstraction. This work proposes a memory-oriented and service-oriented image abstraction, called runtime image, which represents the memory state of the root file system of the container service. The runtime image enables efficient network transfer and fast root file system construction. We design and implement FlacIO, an I/O accelerator based on the runtime image for container image service. FlacIO introduces an efficient runtime image structure that works in conjunction with a runtime page cache on a host node to achieve efficient image service. Our evaluation shows that FlacIO reduces the container cold startup latency by up to 23 and 4.6 times compared to existing full image and lazy loading solutions, respectively. In real-world applications, FlacIO achieves up to 2.25 and 1.7 times performance speedup over other systems in the object storage and machine learning training scenarios, respectively.

- 相关代码已合入openEuler 25.03
  https://gitee.com/openeuler/kernel/pulls/15323

Thanks!

(Q&A)