USENIX ATC '24

StreamCache: Revisiting Page Cache for File Scanning on Fast Storage Devices

Zhiyue Li and Guangyan Zhang

Tsinghua University

Agenda

- Background & Motivation
- Design & Techniques
- Evaluation
- Conclusion

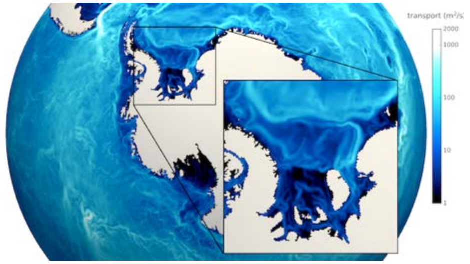
File scanning in data-intensive applications

File scanning

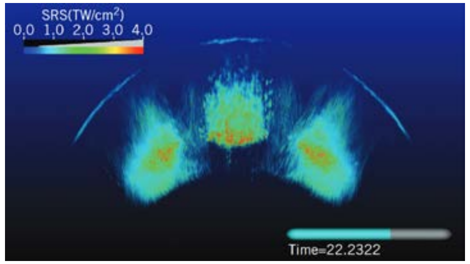
- Most file pages are only accessed once during one I/O stage
- Low ratio of reused data

Common in data-intensive applications

- Scientific computing and AI training
- Initial data loading, checkpoint and restart, and result visulization



Climate simulation[1]



Laser-plasma interaction[2]

ChatGPT ChatGLM
Gemini Meta Llama

Large language model

Examples of common data-intensive applications

[1] E3SM. https://e3sm.org/research/cryosphere-ocean/v1-cryosphere-ocean/.
[2] S. H. Langer, A. Bhatele and C. H. Still. PF3D Simulations of Laser-Plasma Interactions in National Ignition Facility Experiments. 2014.

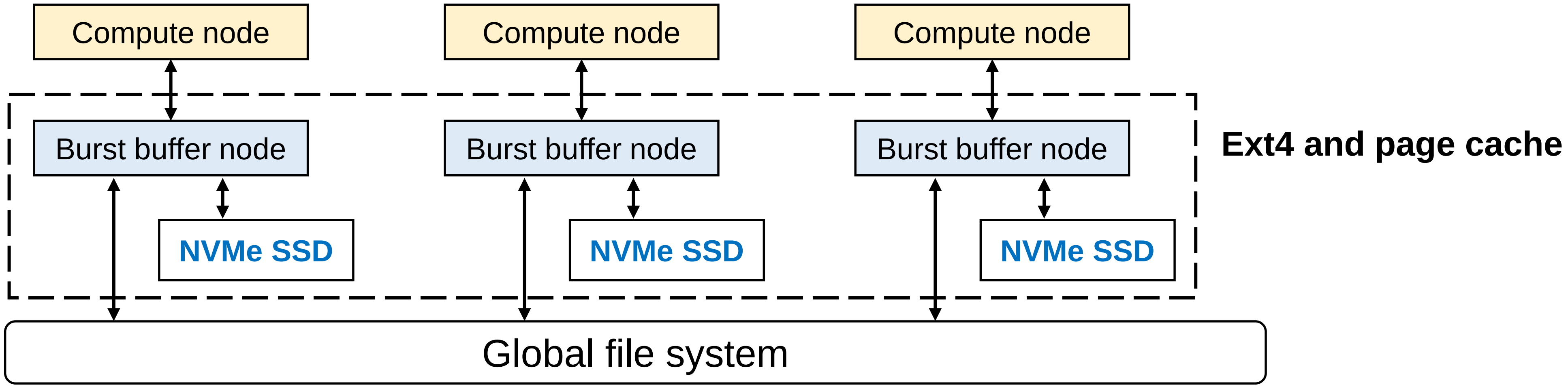
File scanning with the kernel buffered I/O

Buffered I/O is commonly used for file scanning

- Cutting-edge HPC clusters deploy **NVMe SSD-based** burst buffer (BB)
- The BB file system **HadaFS**[1] uses **buffered I/O** on the burst buffer nodes

Advantage of buffered I/O

- Transparent buffering, data aggregation, I/O alignment and prefetching with the **kernel page cache**

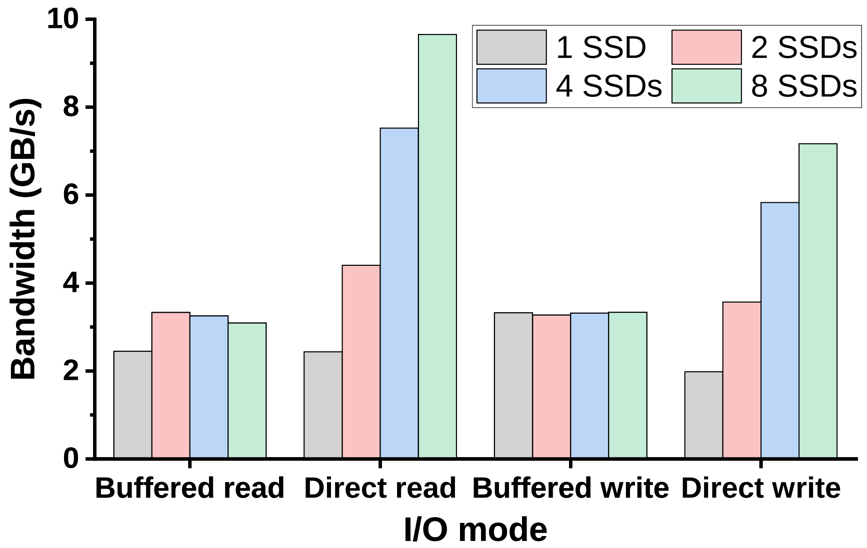


[1] https://www.usenix.org/conference/fast23/presentation/he.

Performance issues on next-generation storage

Issue 1: Poor scalability with the device bandwidth

- Aggregating 8 PCIe 3.0 SSDs to simulate a next-generation storage
- Sequential read/write workloads with **FIO** (10GB file size, 4MB I/O size)
- Direct I/O scales better than buffered I/O under a large I/O size



Buffered read: 35% improvement at most

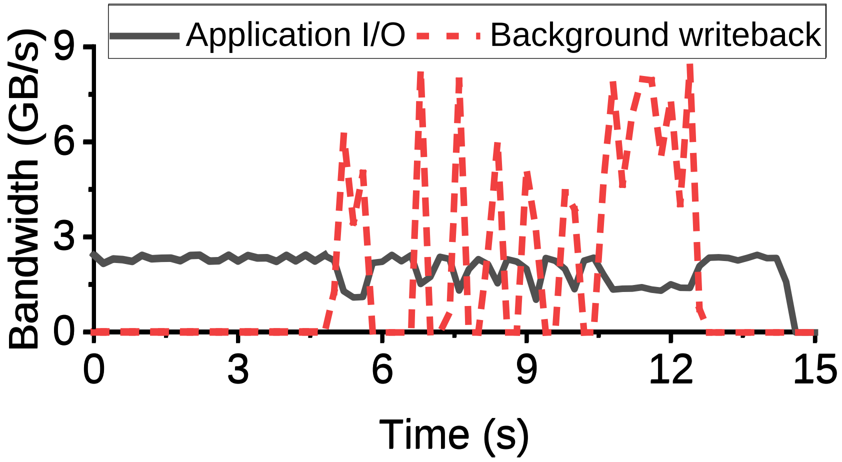
Buffered write: no obvious improvement

Direct read/write: better scalability

The kernel page cache doesn't fit for fast storage devices under file scanning

Performance issues on next-generation storage

- **Issue 2**: High interference from background writeback
  - Sequential write workload with **FIO** (30GB file size)
  - Performance is stable at the beginning
  - The proportion of software overhead increases when writing back to fast storage, severely degrading the buffered write performance



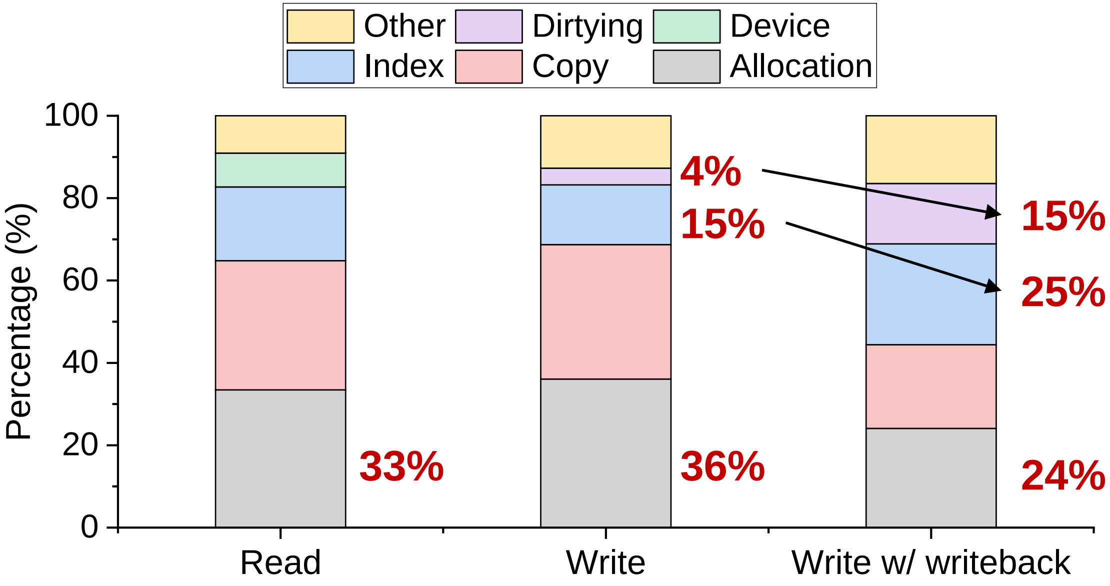
Without writeback: relative stable performance

During writeback: about 32% degradation

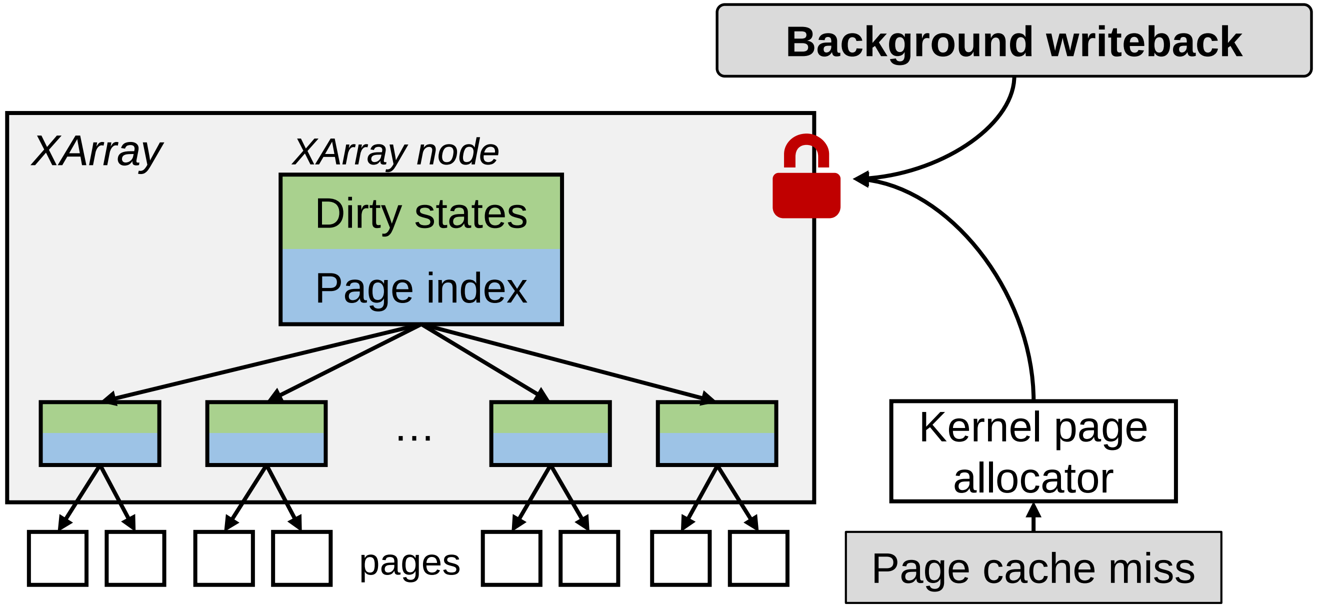
Background writeback on fast storage severely degrades buffered write performance

CPU time breakdown with profiling

- Profiling sequential read, sequential write and sequential write with active writeback using **perf** tool
  - Page allocation occupies major CPU cycles in all workloads
  - Coupled page index and dirty states causes lock contention during writeback
  - Data copy takes non-negligible parts of CPU time



CPU time breakdown of three workloads



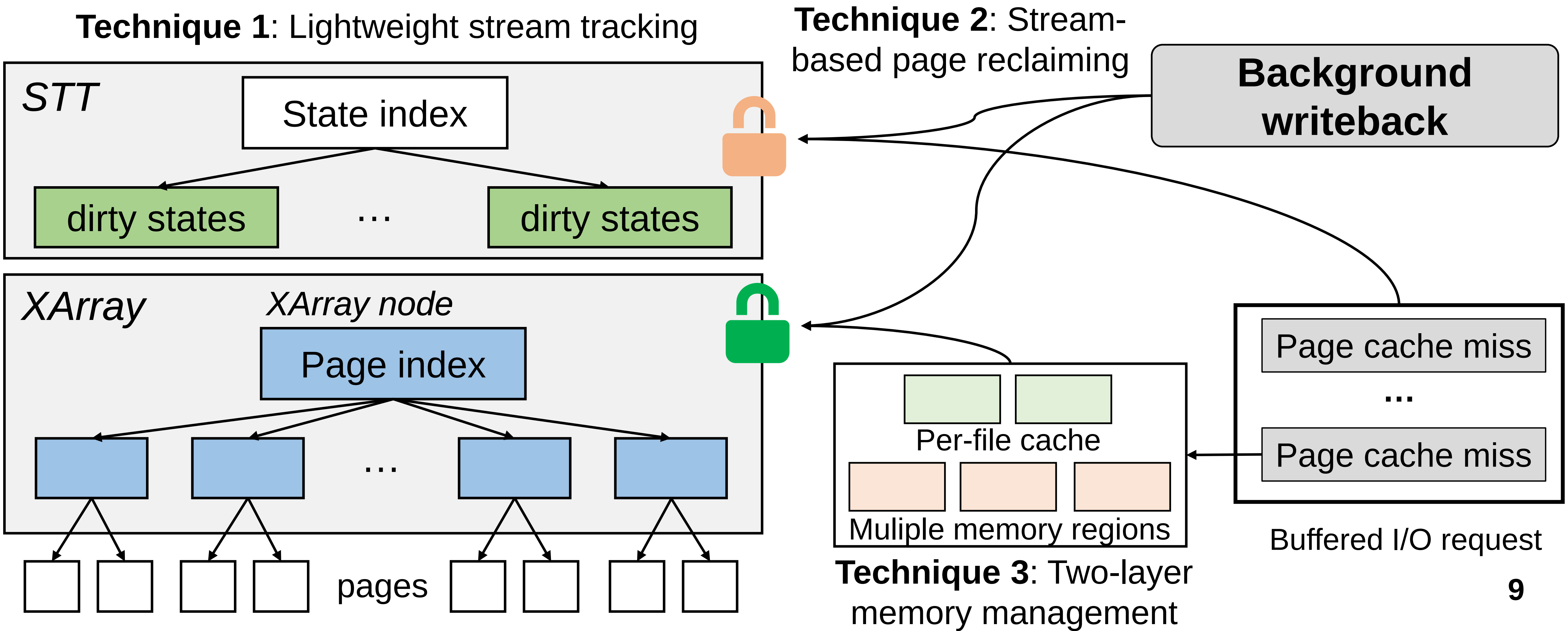
The procedure of file scanning buffered write

Agenda

- Background & Motivation
- Design & Techniques
- Evaluation
- Conclusion

StreamCache overview

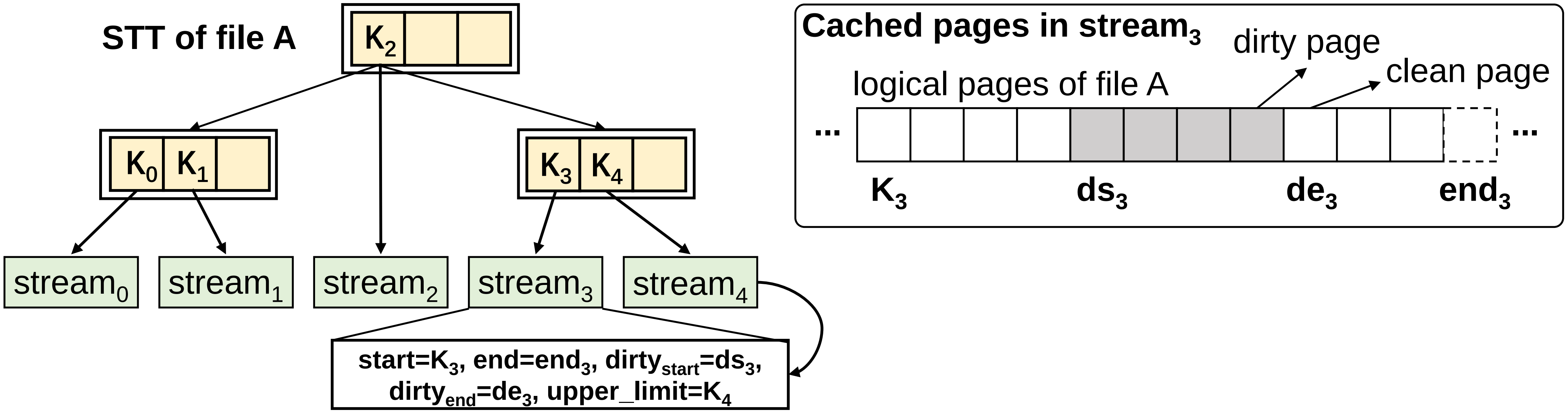
Key idea: Batch updates of dirty states (decicated stream-level index) and fast page allocation (sharded and file-local free-page lists)



Technique 1: Lightweight stream tracking

☐ Stream tracking and stream tracking tree (STT)

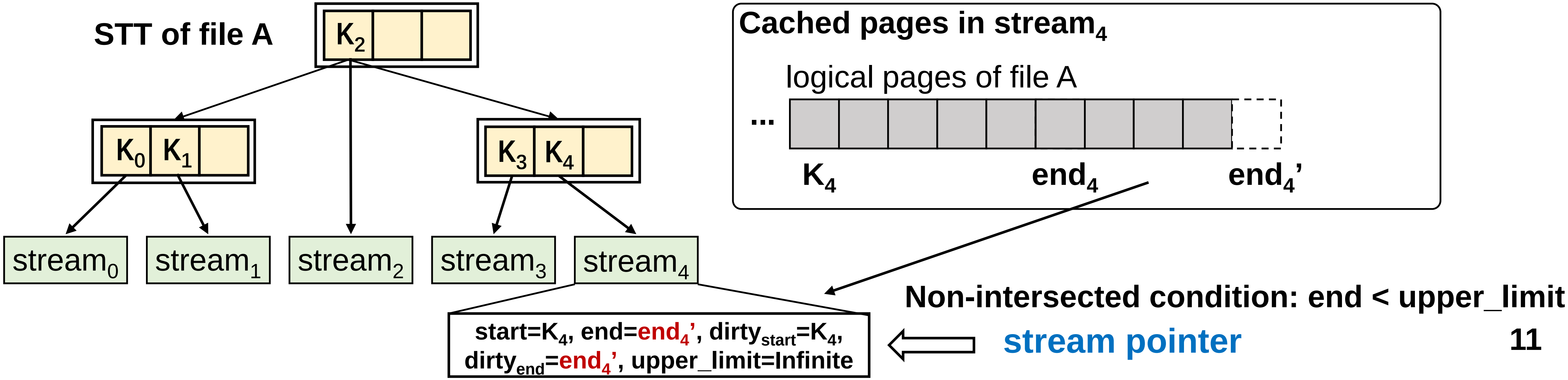
- Stream refers to a range of **logically continuous cached pages**
- STT is the **per-file** tree that indexes streams with their **start page indexes**
  - Better capturing the **I/O patterns** than the system-level tracking
  - Keeping the STT **intact** when a stream is extended
- New streams from buffered I/O requests are merged with existing ones to keep them **non-intersected**



Technique 1: Lightweight stream tracking

☐ Stream tracking optimization with stream pointer

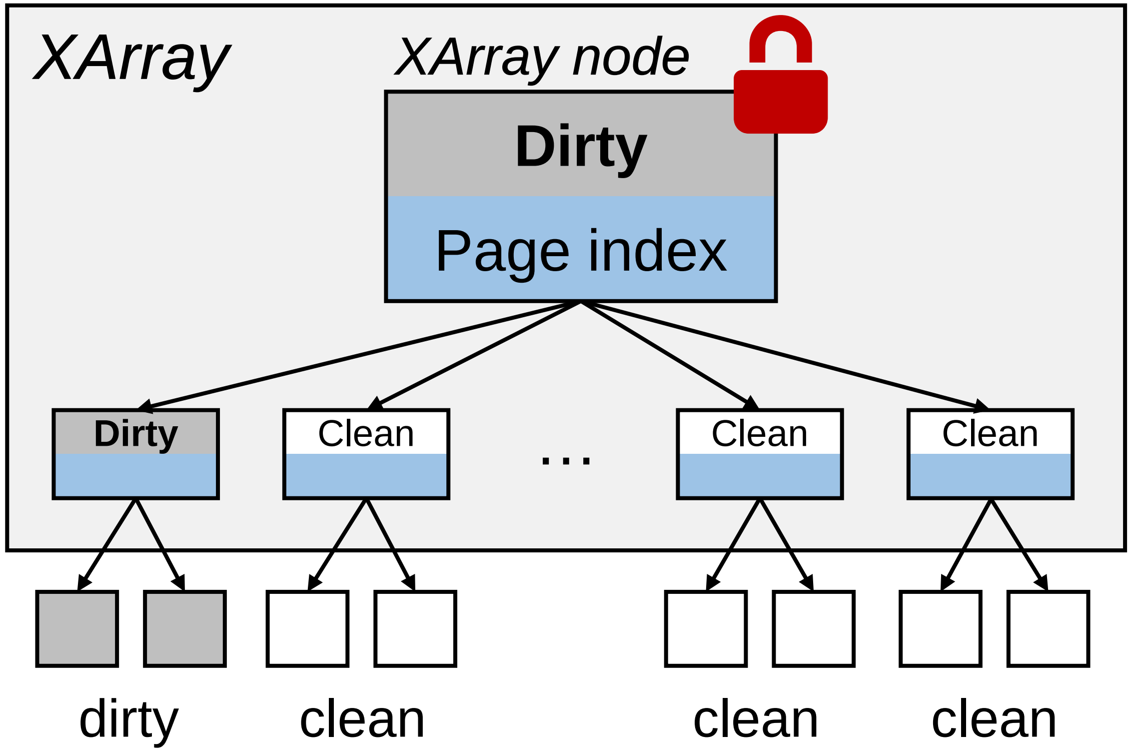
- A **per-file pointer** that points to the **stream of the last I/O**
- Tracking each buffered I/O request firstly **inspects the cached stream**
- Inspecting the "upper_limit" field for any **potential intersection**
- **Accelerating** stream tracking when workload is **sequential**



Technique 1: Lightweight stream tracking

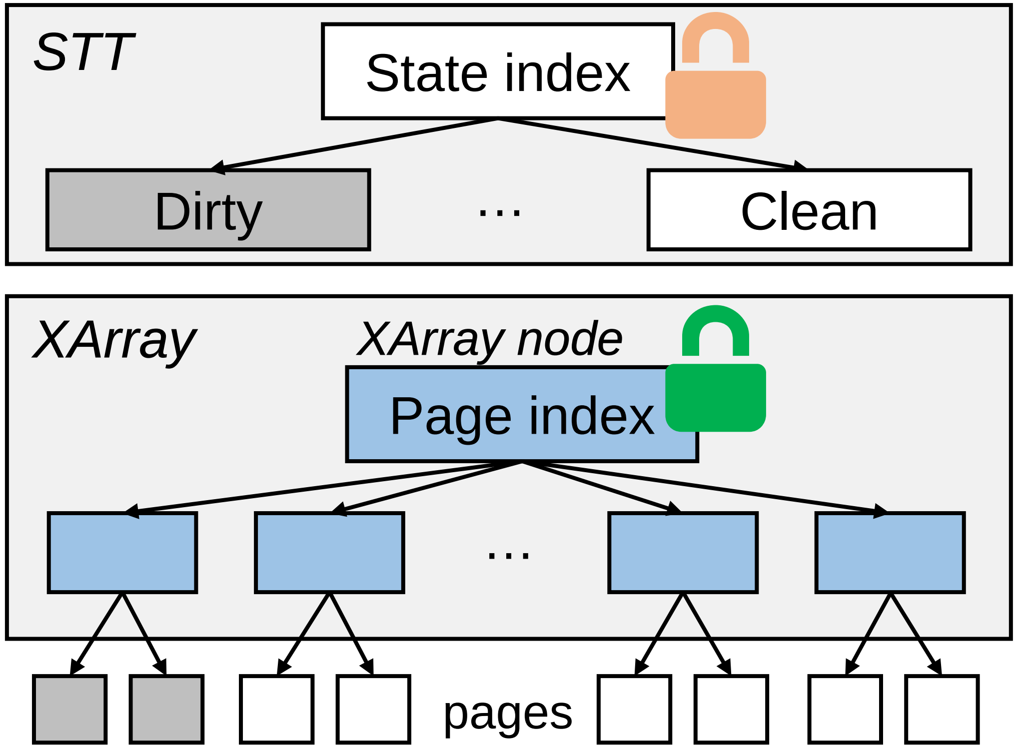
Takeaway: Decoupled dirty states at the stream granularity

Dirty state tracking in existing methods



- Maintaining **at the page granularity**
- Requiring an **exlusive lock** for **each page dirtying**

Dirty state tracking in StreamCache

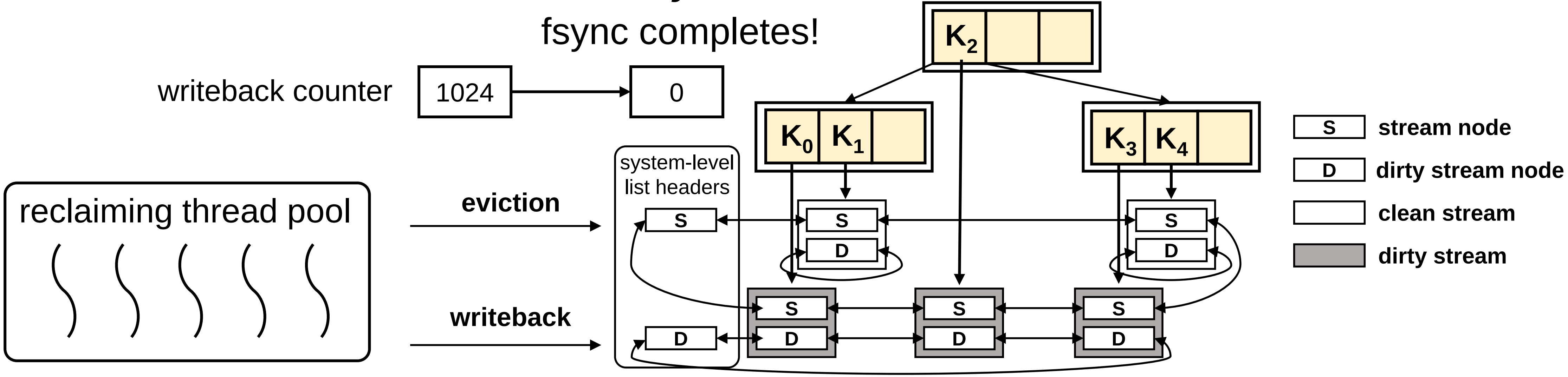


- Maintaining **at the stream granularity**
- Low tracking overhead under **sequential I/Os**

Technique 2: Stream-based page reclaiming

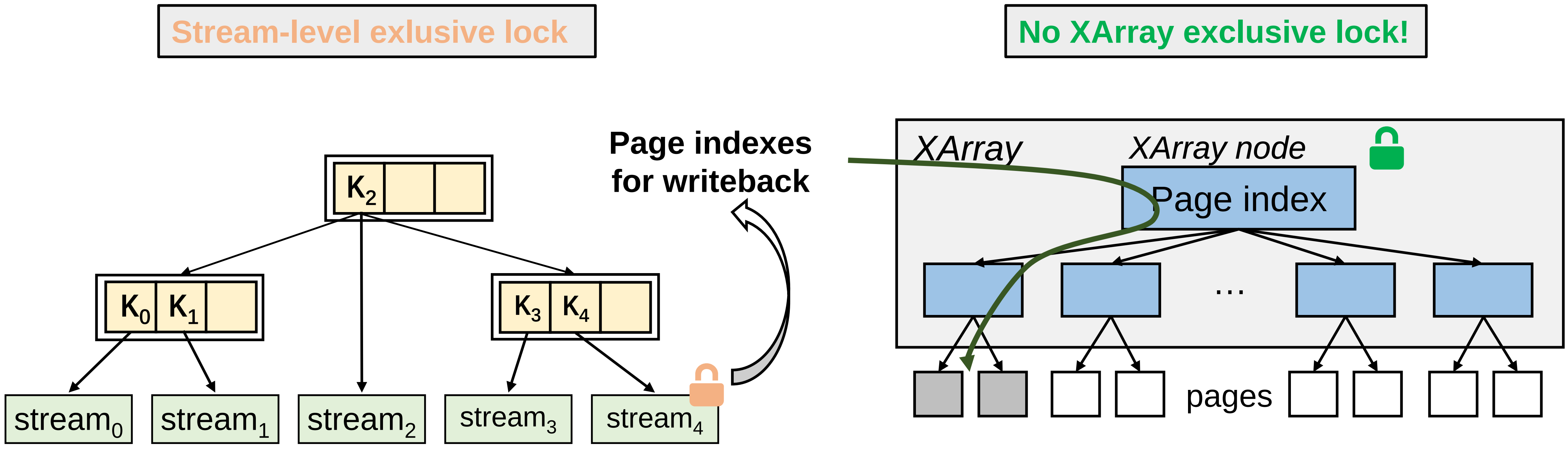
◻ Stream-based page reclaiming based on STT

- Connecting **streams** with double-linked lists for writeback and eviction
- Keeping a pool of reclaiming threads for page writeback and eviction **at the stream** granularity
- The **per-file writeback counters** to denote the completion of writeback in face of the commands like “*fsync*”



Technique 2: Stream-based page reclaiming

- Locating dirty pages in stream-based writeback
  - Extracting the **indexes** of **a range of dirty pages** from the STT
  - Referring to the dirty pages in the XArray **without an exclusive lock**

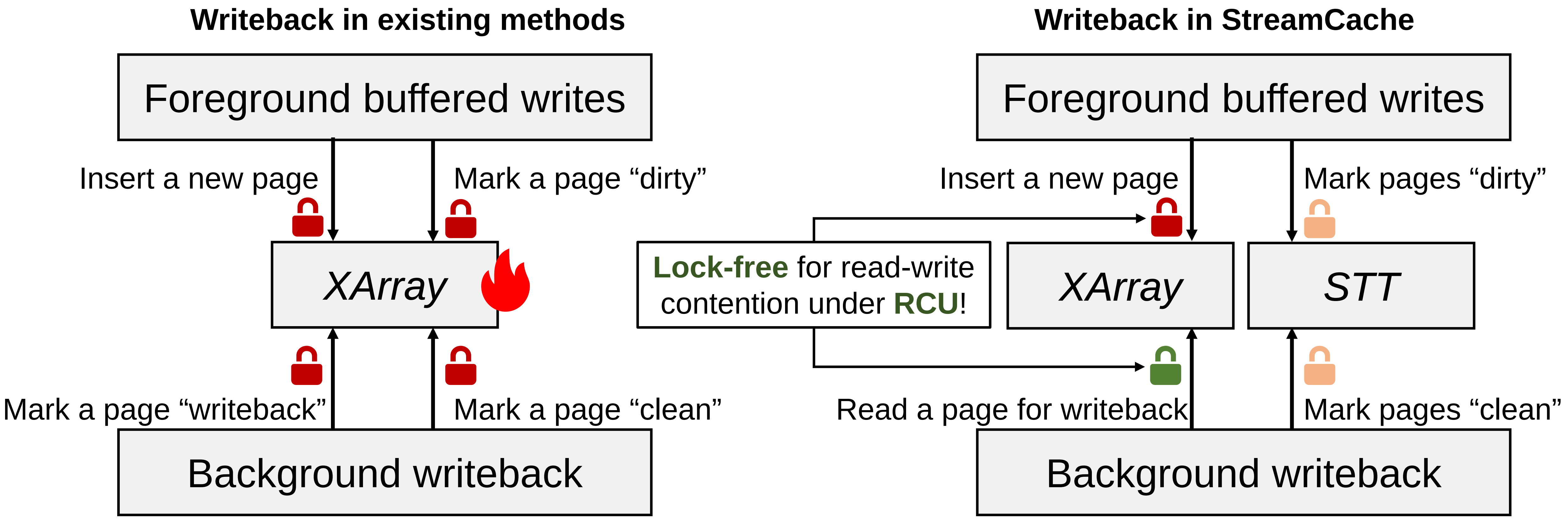


STT of file A

XArray of file A

Technique 2: Stream-based page reclaiming

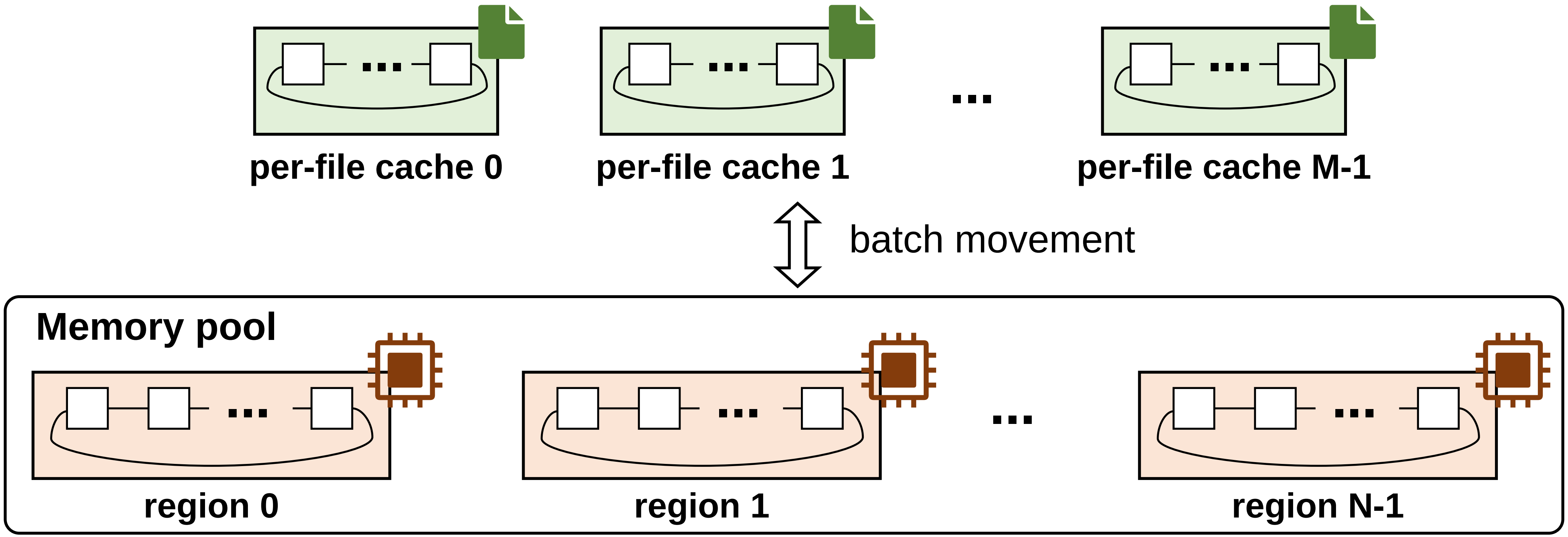
Takeaway: Changing the dirty states at the stream granularity



- Both buffered writes and background writeback needs an **exlusive lock for each page manipulation**
- Page-level **read-write contention** and **stream-level write-write contention**

Technique 3: Two-layer memory management

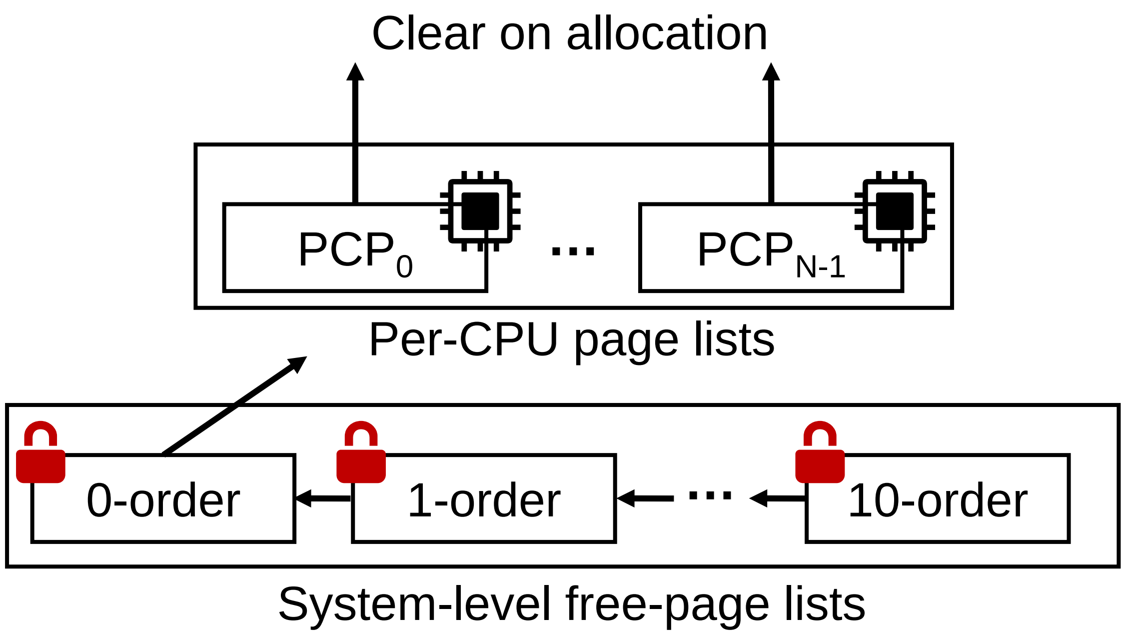
- Two-layer memory management
  - Pre-allocating **zero-order pages** into system-level **per-core** free-page lists
  - Per-file cache for **CPU-cache-friendly** page allocation



Technique 3: Two-layer memory management

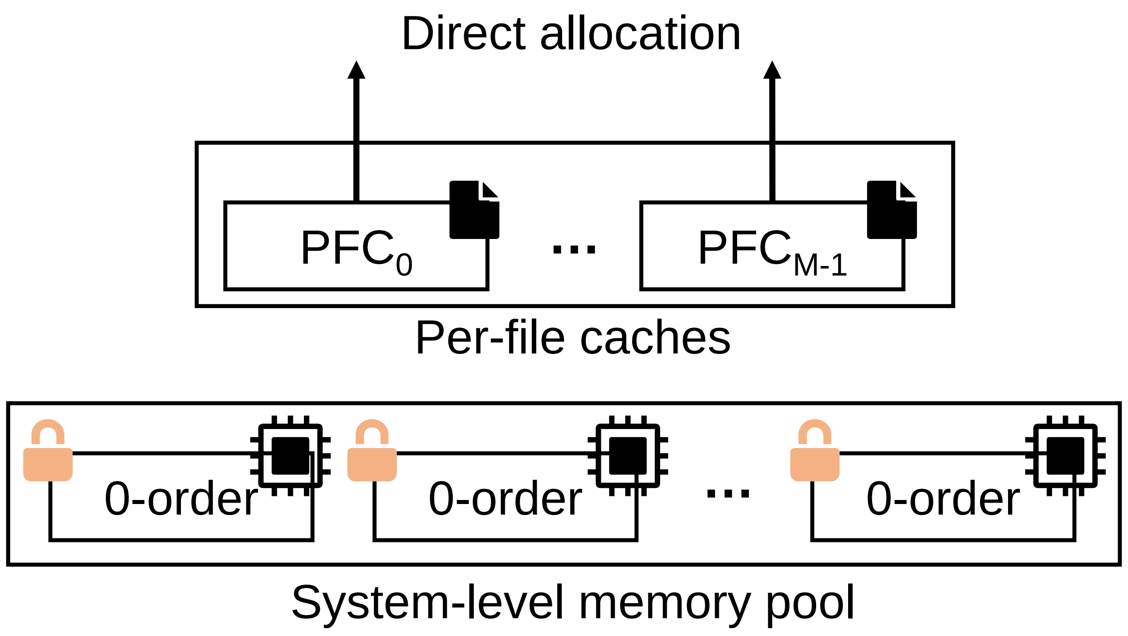
Takeaway: Designing sharded and file-local free-page lists

Page allocation in kernel page cache



- **Page splitting** overhead
- Lock contention on a **single free-page list**
- **Page clearing** overhead on allocation

Page allocation in StreamCache



- **No page splitting** overhead
- Minor lock contention with **multiple free-page lists**
- **Removing page clearing** from the critical path
- File-local lists for **better CPU cache locality**

Agenda

- Background & Motivation
- Design & Techniques
- Evaluation
- Conclusion

Experiment settings

- TestBed
  - Ubuntu 18.04 (kernel version 5.4)
  - 32-core AMD Rome EPYC 7542 CPU, 128GB DRAM
  - RAID-0 of 8 Intel Optane 905p SSDs
- Baseline (all integrated in XFS)
  - Linux kernel page cache
  - FastMap-cache
- Workloads
  - Synthetic workloads (FIO)
  - Real-world workloads (PF3DIO)

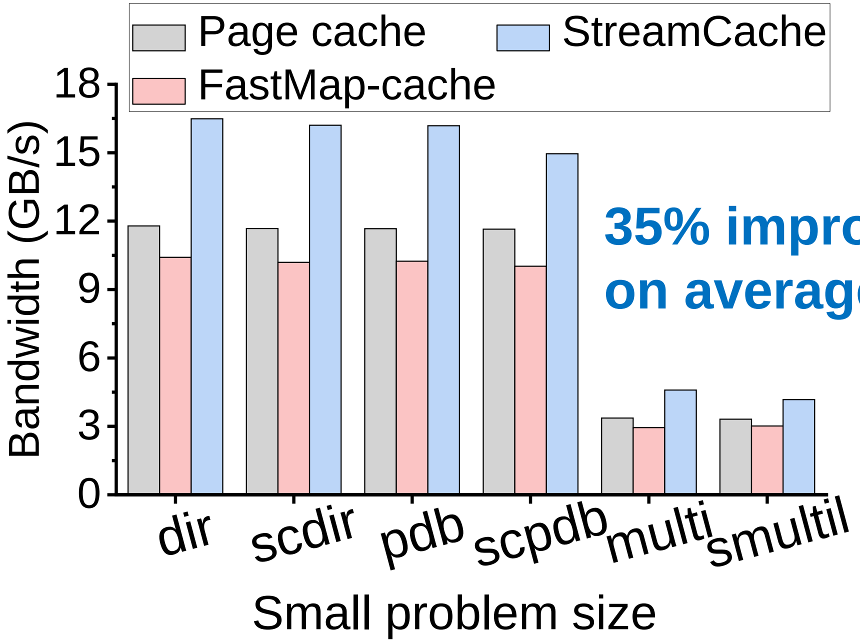
Experiment outline

- StreamCache's performance under **real-world workloads**?
- StreamCache's performance under **different workload parameters**?
- **Effects of individual techniques** in StreamCache?
- More in our paper ...

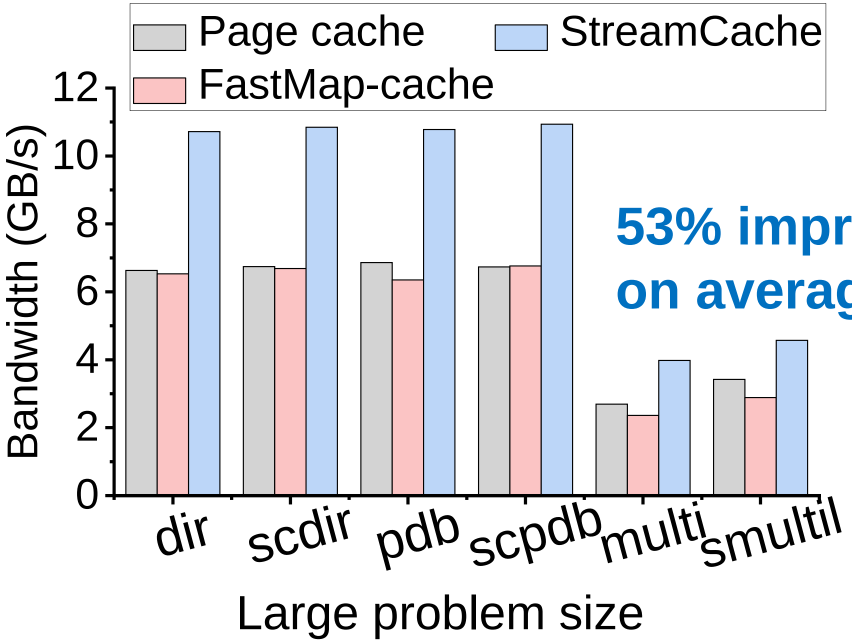
Performance of real-world workload

◻ Scientfic computing I/O benchmark (PF3DIO kernel)

- Writing checkpoint files in six different patterns
- StreamCache outperforms existing methods by **26%-62%**
- Larger problem size triggers background writeback, and the benefit of StreamCache is more obvious



35% improvements on average

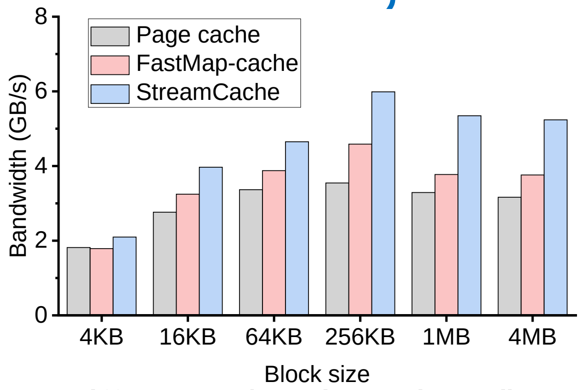


53% improvements on average

Performance of workloads with different parameters

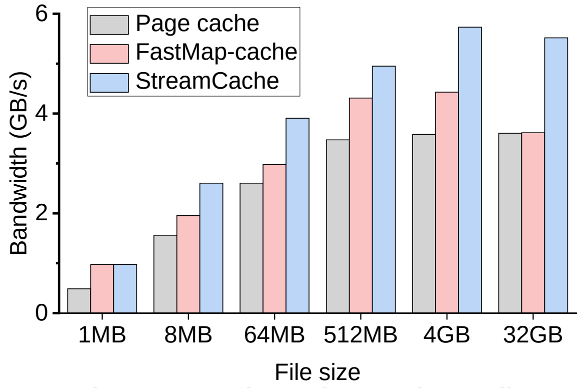
☐ Synthetic workloads generated by FIO benchmark

- File scanning workloads can benefit from StreamCache **despite the I/O size, file size and parallelism**



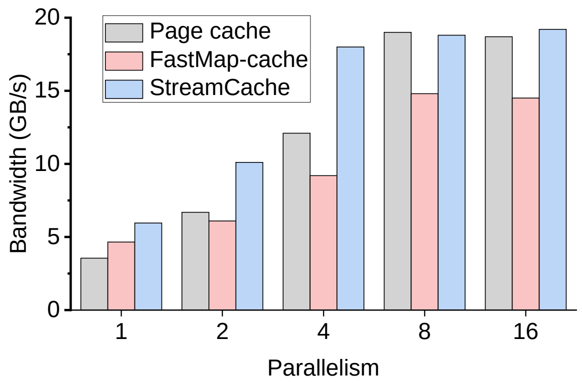
28% on average

Different I/O sizes (read)



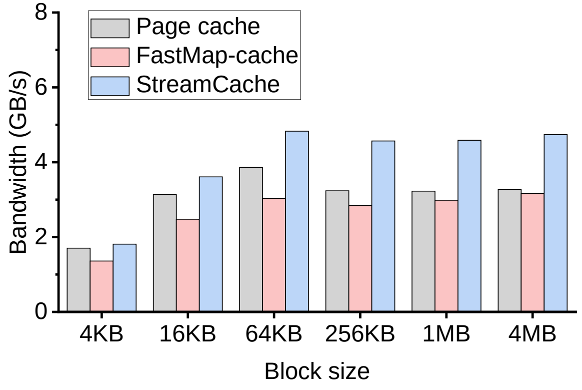
26% on average

Different file sizes (read)



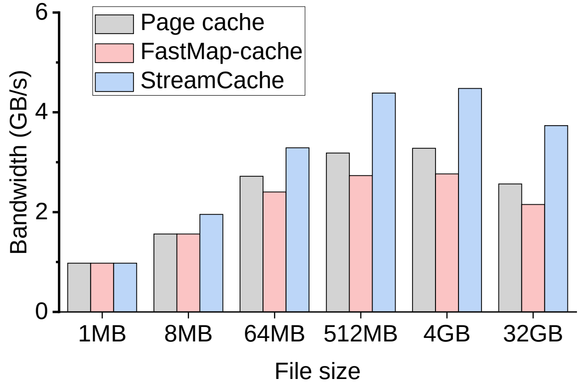
27% on average

Different parallelisms (read)



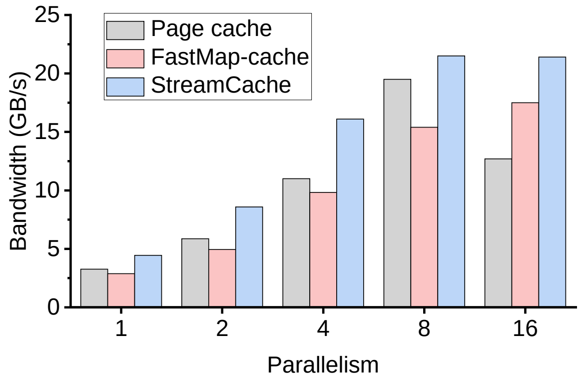
29% on average

Different I/O sizes (write)



32% on average

Different file sizes (write)

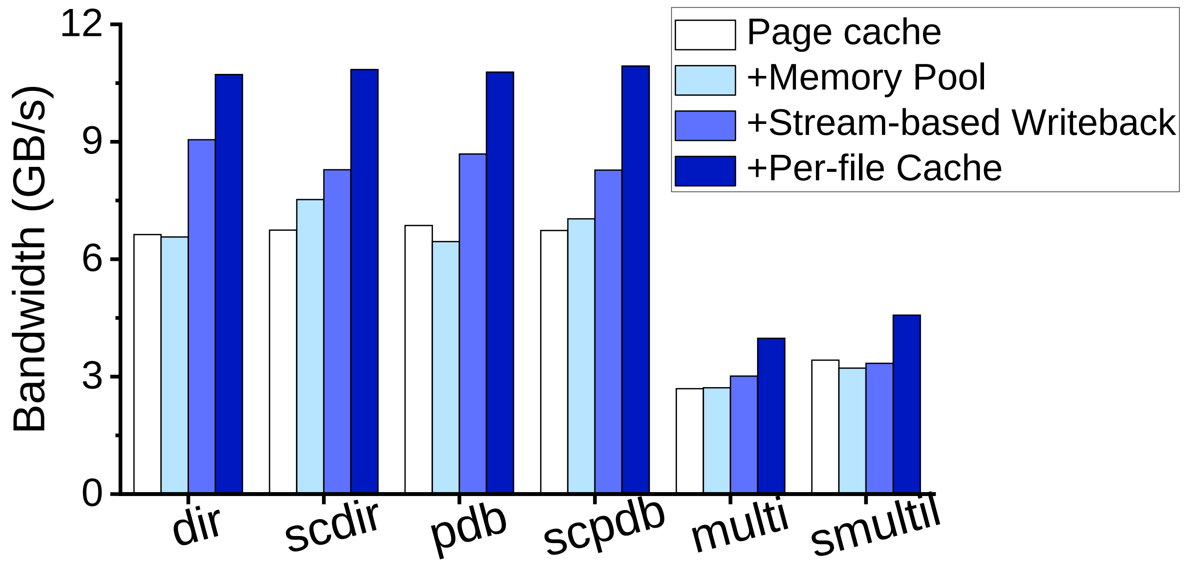


28% on average

Different parallelisms (write)

Effects of individual techniques

- Adding main techiniques incrementally under PF3DIO kernel of large problem size
  - Memory pool brings a **1.3%** improvement
  - Stream tracking and stream-based writeback brings a **21.3%** improvement
  - Per-file cache brings a **27.5%** improvement



Effects of individual techniques with PF3DIO large problem size

Agenda

- Background & Motivation
- Design & Techniques
- Evaluation
- Conclusion

Conclusion

- Problem
  - XArray lock contention and slow page allocation hinder the performance of file scanning with buffered I/O on fast storage devices
- Key idea
  - Separating dirty states from the page cache index and keeping them in the dedicated stream-level index
  - Designing sharded and file-local free-page lists for fast page allocation
- Techniques
  - Lightweight stream tracking
  - Stream-based page reclaiming
  - Two-layer memory management

Thank you!