BFO: Batch-File Operations on Massive Files for Consistent Performance Improvement

Yang Yang, Qiang Cao, Hong Jiang, Li Yang, Jie Yao,

Yuanyuan Dong, Puyuan Yang

Huazhong University of Science and Technology,

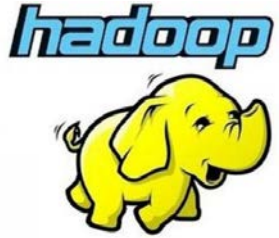
University of Texas at Arlington, Alibaba group

Outline

- Background
- BFO Design
- Evaluation
- Conclusion

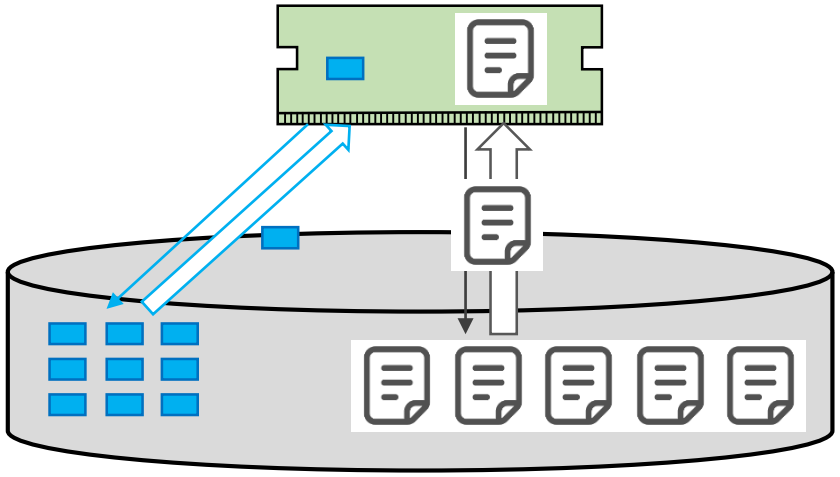
Background

- Batch-file Operations
  - Accessing a batch of files
- Many applications need batch-file operations
  - Backup applications
  - File-level data replication and archiving
  - Big data analytics systems
  - Social media and online shopping websites
- Traditional access approaches access files one by one
  - Called single-file access pattern
  - Inefficient for small files



Background

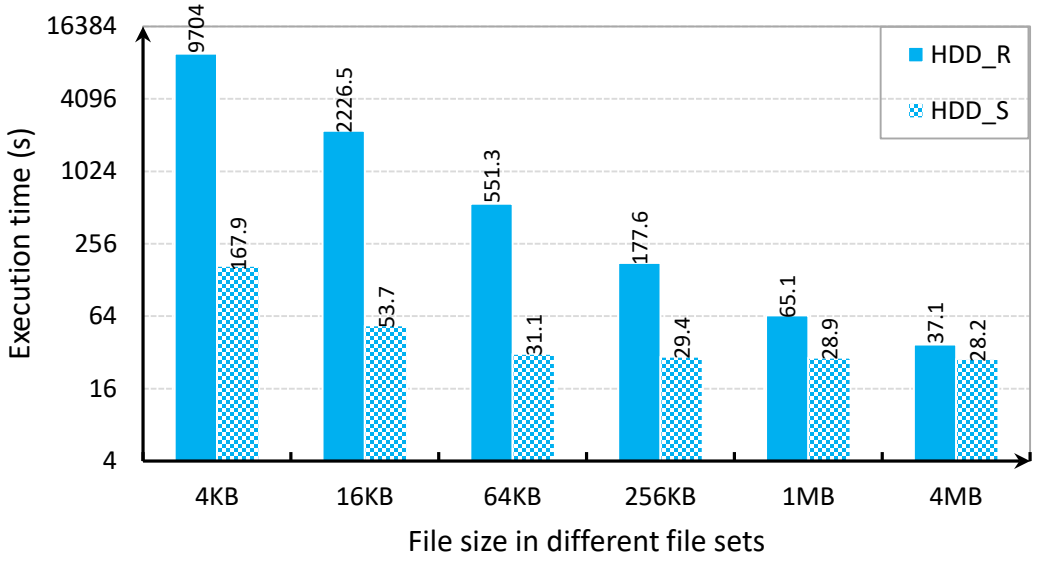
- Small files in file systems
  - Desktop file system: more than 80% of accesses are to files smaller than 32B.
  - Cloud and HPC cluster: 25%~40% files < 4KB.
- Single-file access pattern for small files
  - Accessing metadata
  - Fetching file data, and so on

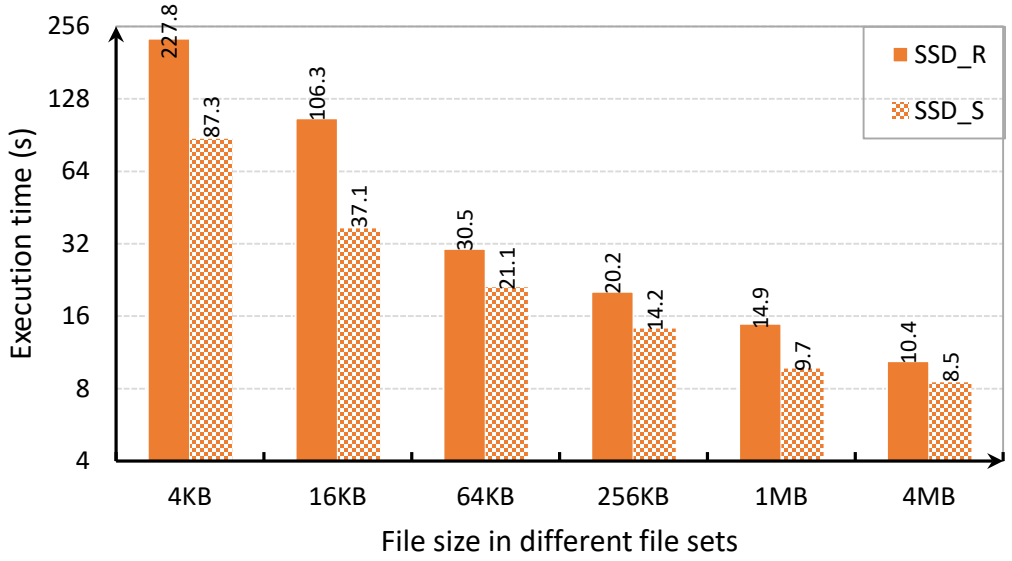


- IO operations dominate batch-file access
  - Metadata access contributes 40% time for accessing a small file on disk.
  - Random data IOs

Overall access performance

- Read performance



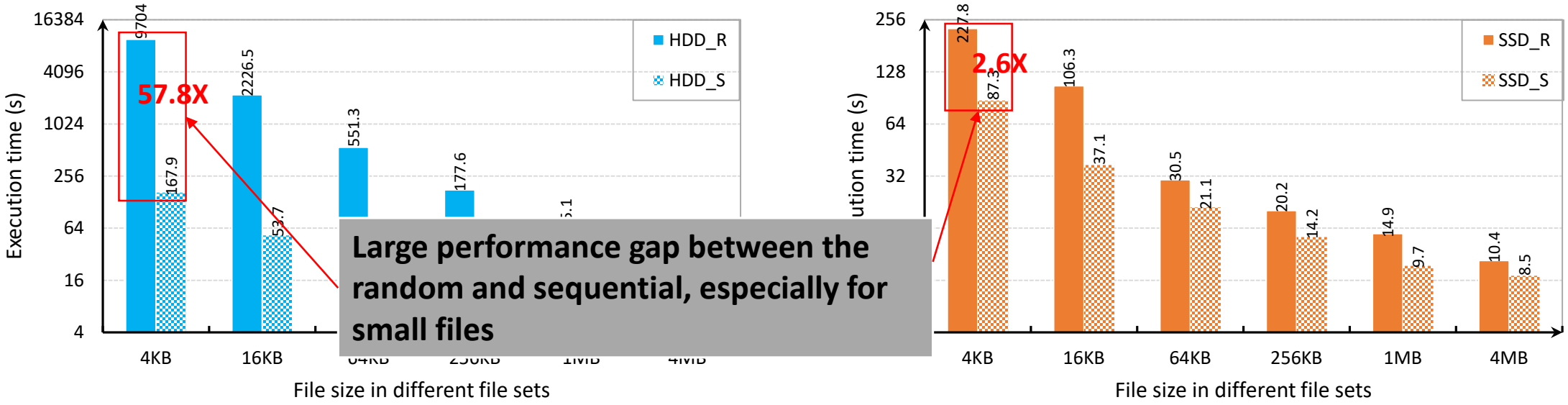


Setup:

- File sets: 4GB data with different file sizes (i.e., from 4KB to 4MB)
- Devices: HDD & SSD
- Orders: Random & Sequential

Overall access performance

- Read performance

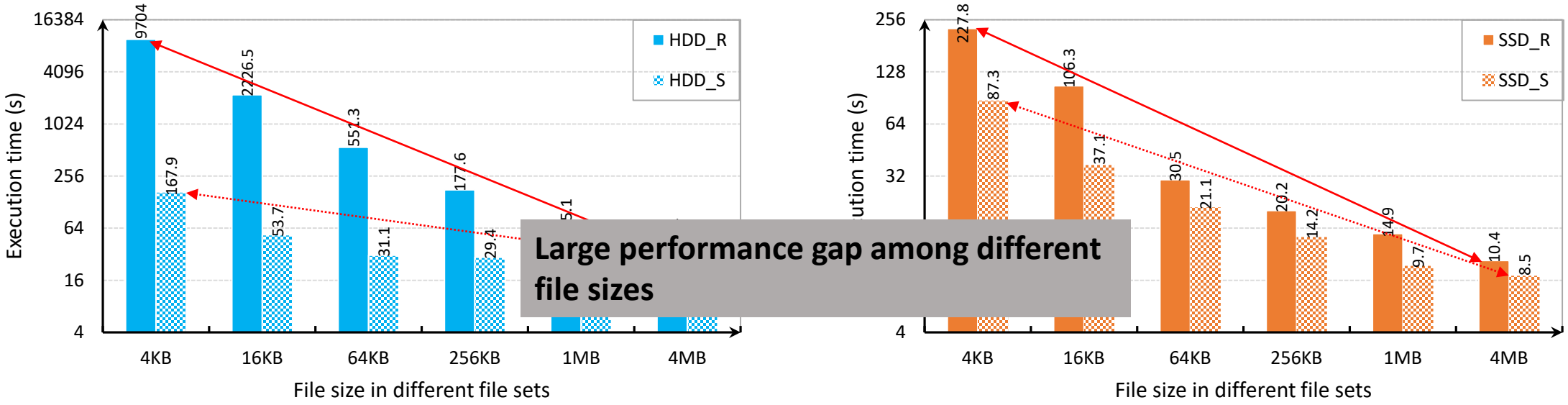


Setup:

- File sets: 4GB data with different file sizes (i.e., from 4KB to 4MB)
- Devices: HDD & SSD
- Orders: Random & Sequential

Overall access performance

- Read performance

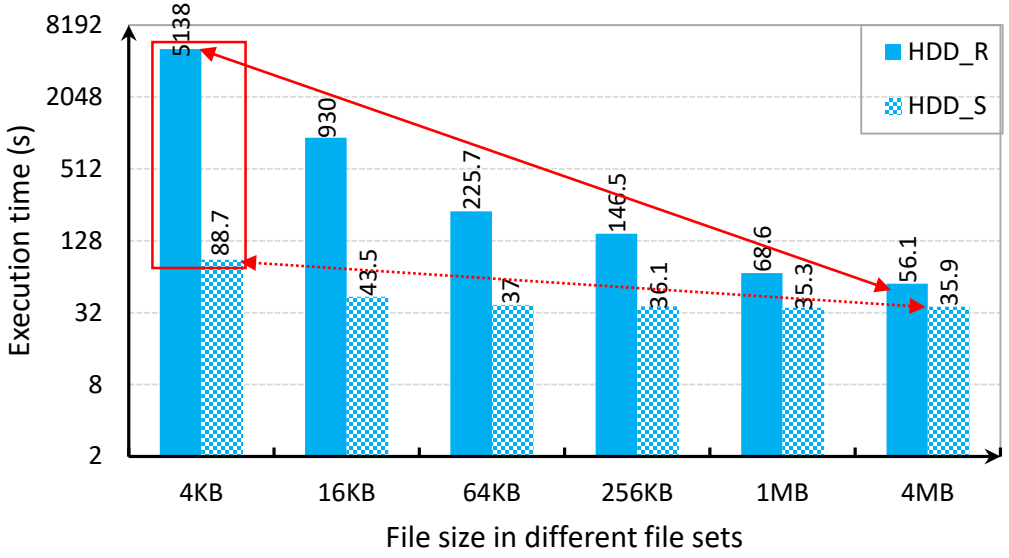


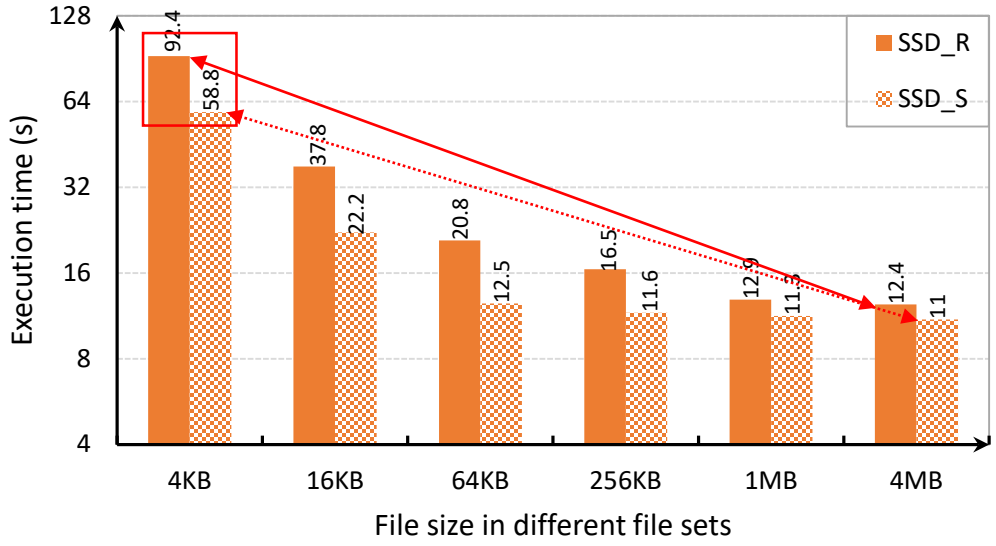
Setup:

- File sets: 4GB data with different file sizes (i.e., from 4KB to 4MB)
- Devices: HDD & SSD
- Orders: Random vs Sequential

Problem

- Write performance





Observation: the single-file access approach is very inefficient

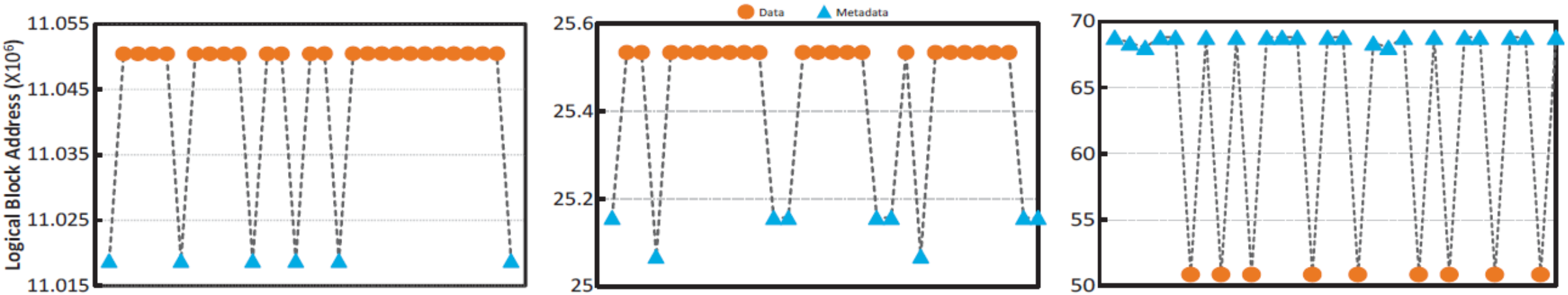
- **for small files (below 1MB);**
- **in a random manner .**

Related Works

- Application-level optimization (Fastcopy)
  - Multi-threading, large buffer
- Prefetching mechanism (Diskseen, ATC'07)
  - Depending on the future access behaviors
- Block-level I/O scheduler (split-level I/O scheduling, SOSP'15)
  - Serializing the file accesses
- Packing metadata and data together (CFFS, FAST'16)
  - Redesigning new file systems

Problem Analysis

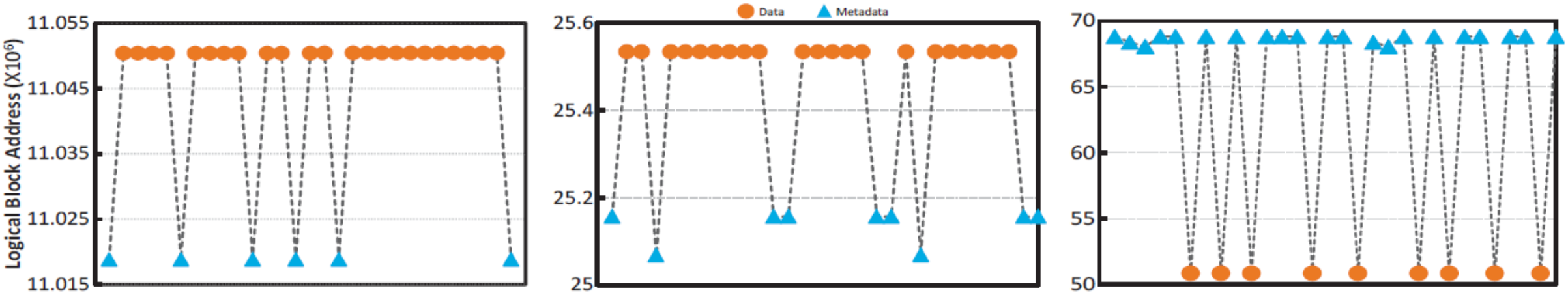
- File Access behaviors
  - Reading a file set with three representative file systems



(a) Ext4 (b) Btrfs (c) F2FS

Problem Analysis

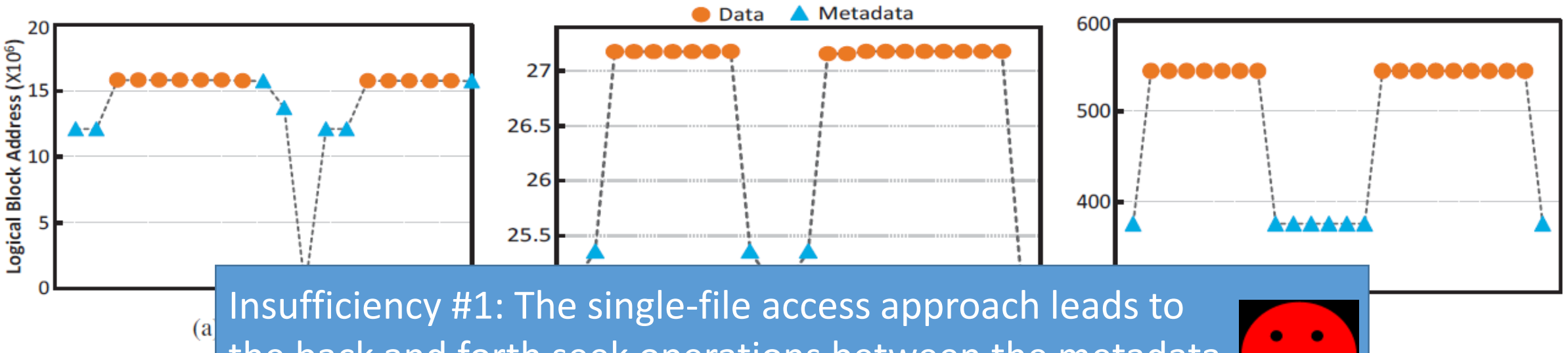
- File Access behaviors
  - Reading a file set with three representative file systems



(a) Ext4 (b) Btrfs (c) F2FS

Problem Analysis

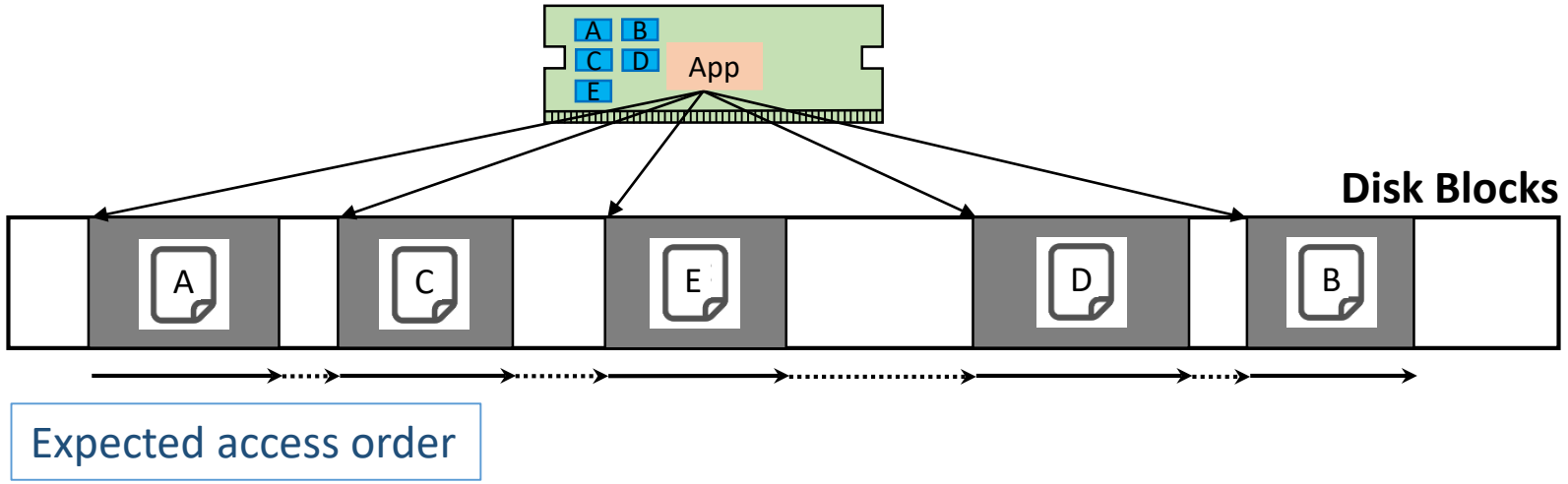
- File Access behaviors
  - Reading a file set with three representative file systems
  - Writing a file set with three representative file systems



Insufficiency #1: The single-file access approach leads to the back and forth seek operations between the metadata area and data area, resulting in many non-sequential I/Os.

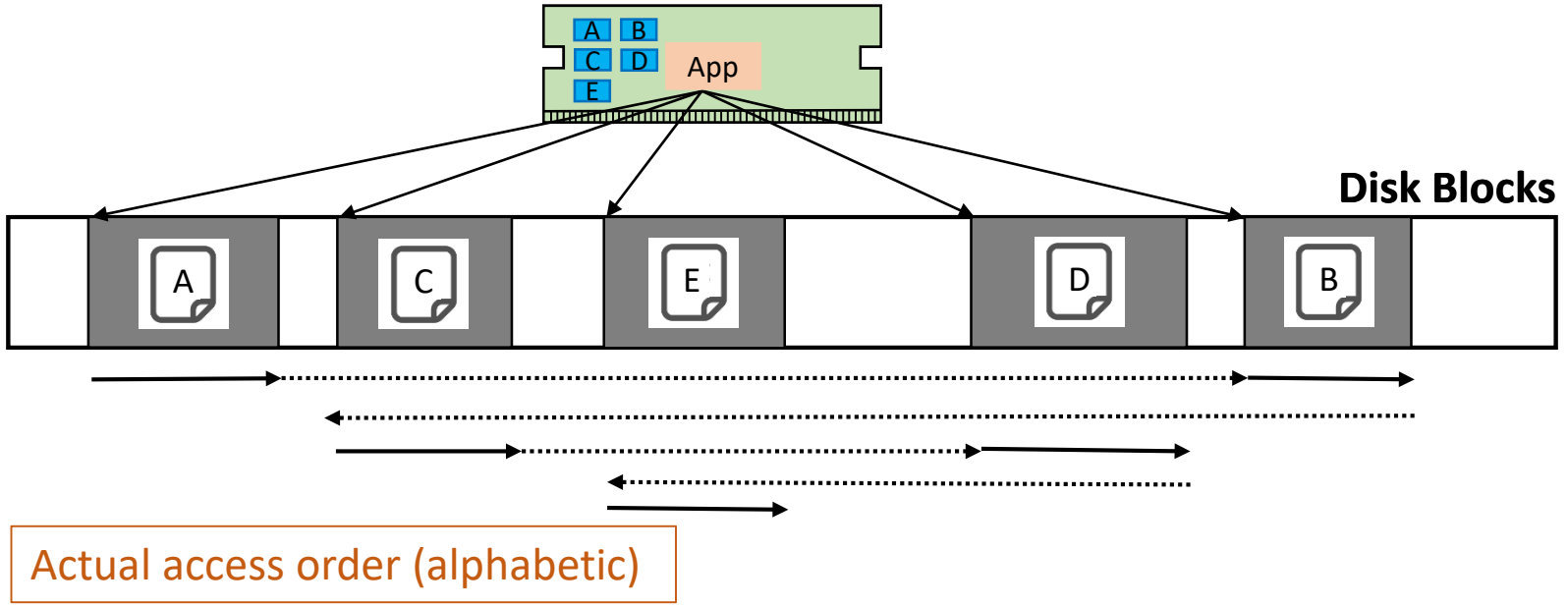
Problem Analysis

- File Access behaviors
- Data Access behaviors (excluding the metadata)



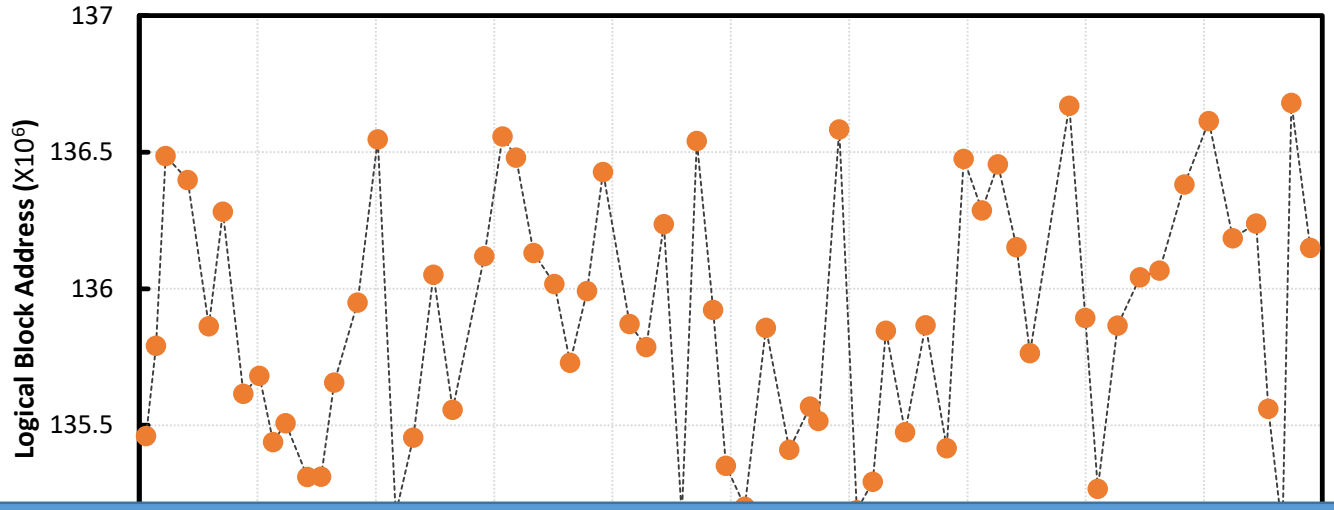
Problem Analysis

- File Access behaviors
- Data Access behaviors (excluding the metadata)



Problem Analysis

- File Access behaviors
- Data Access behaviors (excluding the metadata)



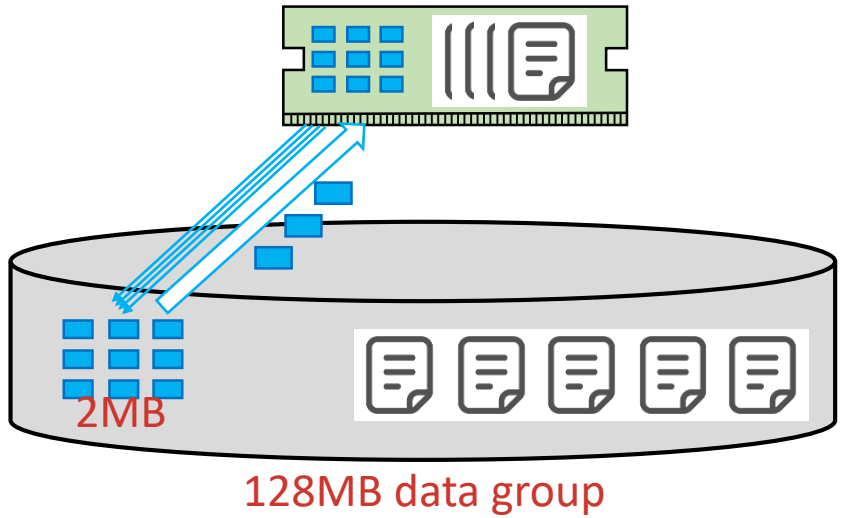
Insufficiency #2: The single-file access approach is unaware of the underlying data layout, and may read these files in any order, also leading to random I/Os.

Outline

- Background
- BFO Design
  - BFOr
  - BFOw
- Evaluation
- Conclusion

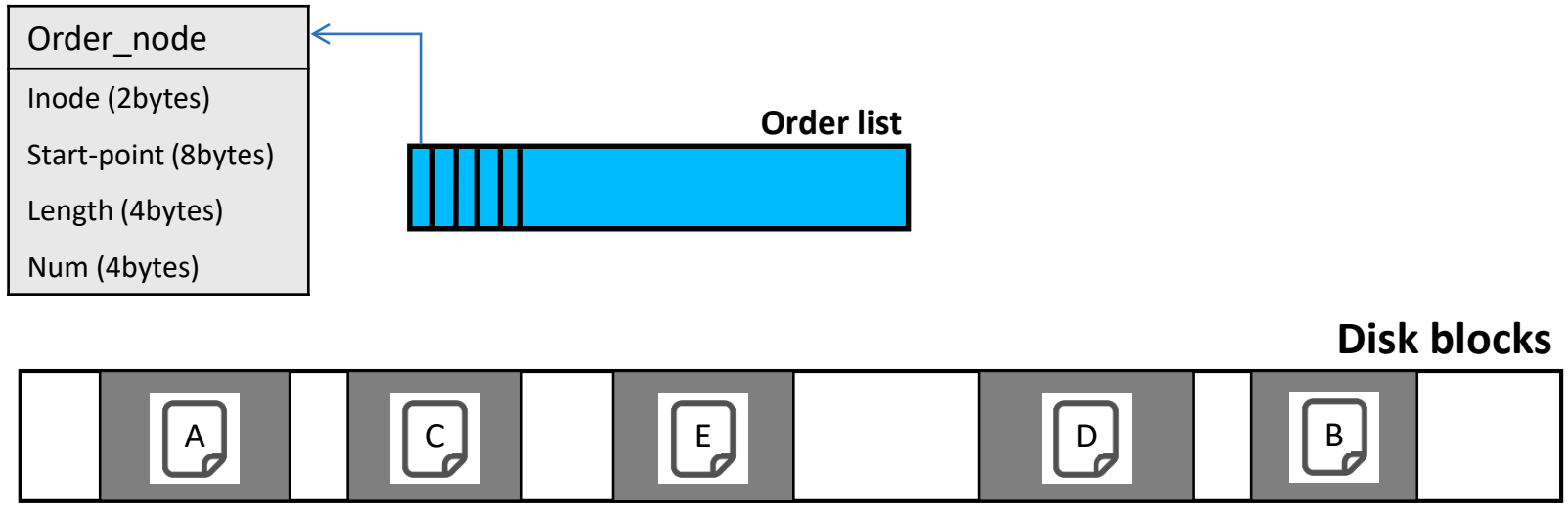
BFOr

- Two-phase read
  - Objective: Separately read the metadata and file data of all accessed files in batches
  - Phase 1: scanning the inodes
  - Phase 2: fetching all files' data
- Layout-aware scheduler



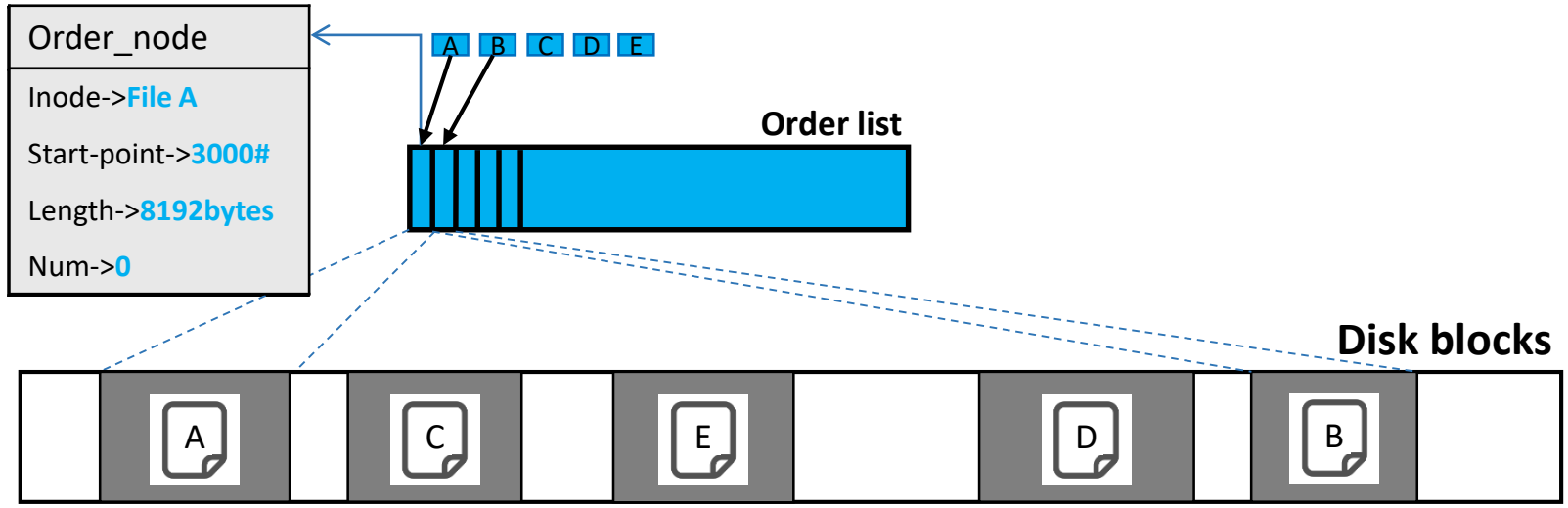
BFOr

- Two-phase read
- Layout-aware scheduler
  - Extracting the addresses from the inodes
  - Sorting the addresses of all files
  - Issuing read I/O in the order of the list



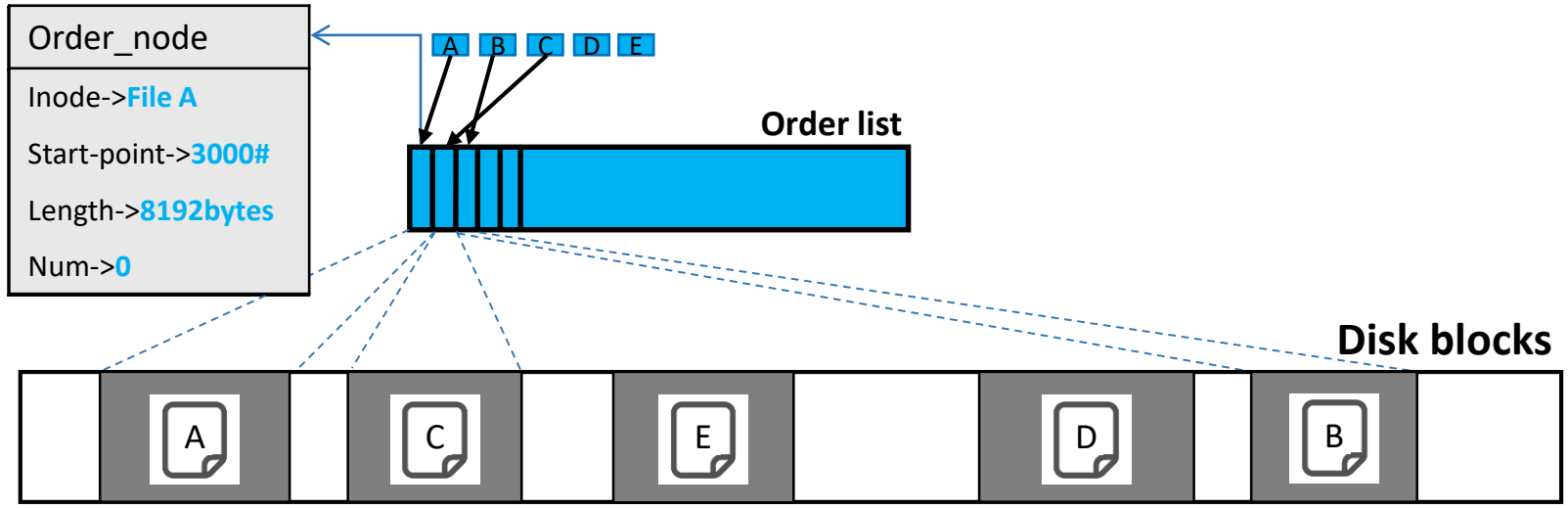
BFOr

- Two-phase read
- Layout-aware scheduler
  - Extracting the addresses from the inodes
  - Sorting the addresses of all files
  - Issuing read I/O in the order of the list



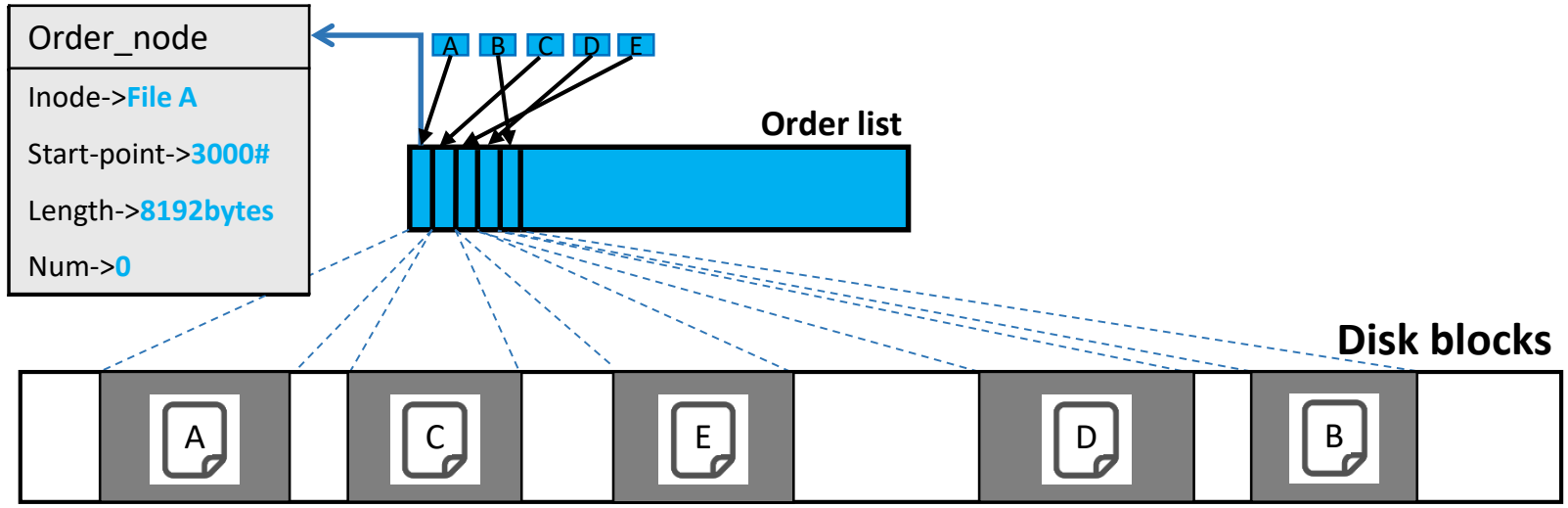
BFOr

- Two-phase read
- Layout-aware scheduler
  - Extracting the addresses from the inodes
  - Sorting the addresses of all files
  - Issuing read I/O in the order of the list



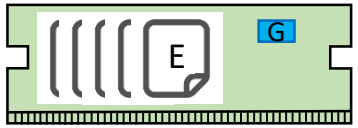
BFOr

- Two-phase read
- Layout-aware scheduler
  - Extracting the addresses from the inodes
  - Sorting the addresses of all files
  - Issuing read I/O in the order of the list



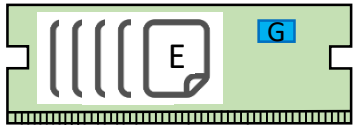
BFOw

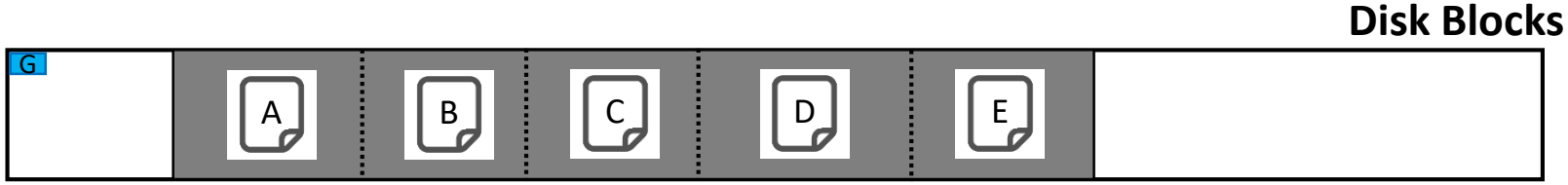
- Two-phase write
  - Phase 1: creating a global file to store all data once
    - Creating G inode for the file
    - Creating Order_list to record the order of the written files
  - Phase 2: creating all inodes for all files
    - Extracting the address from the G inode
    - Creating all inodes with the address information and the Order_list
      - Current_FileAddr = Previous_FileAddr + FileLength
- Light-weight consistency strategy



BFOw

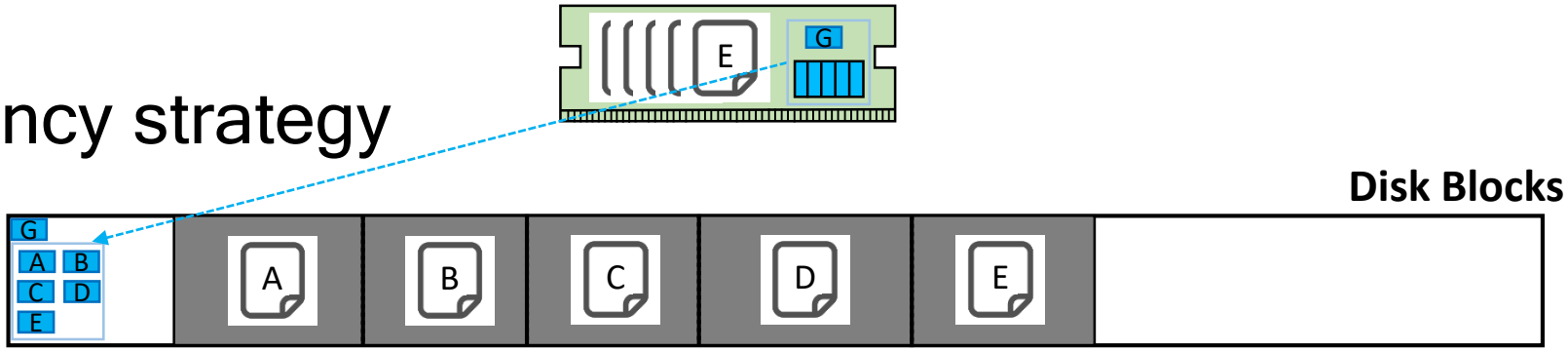
- Two-phase write
  - Phase 1: creating a global file to store all data once
    - Creating G inode for the file
    - Creating Order_list to record the order of the written files
  - Phase 2: creating all inodes for all files
    - Extracting the address from the G inode
    - Creating all inodes with the address information and the Order_list
      - Current_FileAddr = Previous_FileAddr + FileLength
- Light-weight consistency strategy





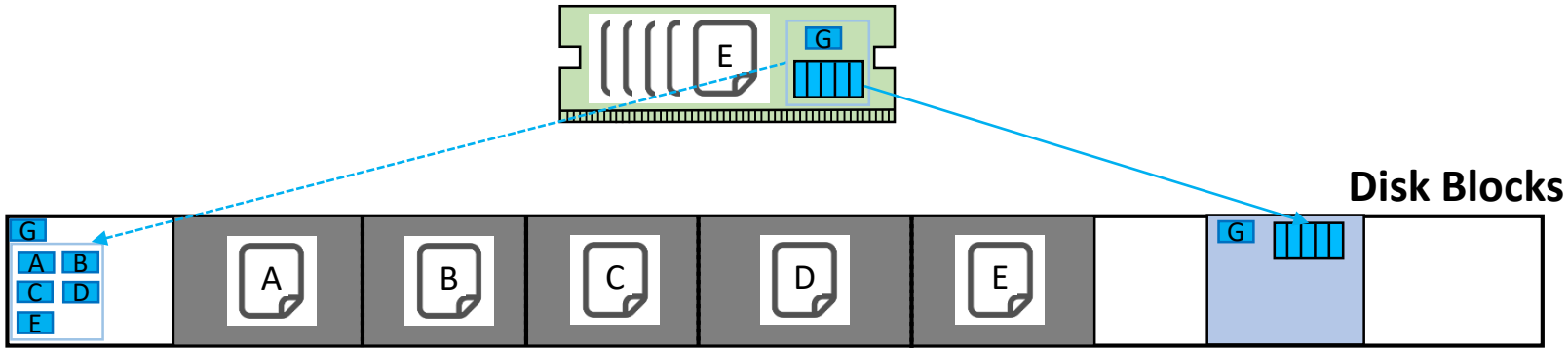
BFOw

- Two-phase write
  - Phase 1: creating a global file to store all data once
    - Creating G inode for the file
    - Creating Order_list to record the order of the written files
  - Phase 2: creating all inodes for all files
    - Extracting the address from the G inode
    - Creating all inodes with the address information and the Order_list
      - Current_FileAddr = Previous_FileAddr + FileLength
- Light-weight consistency strategy



BFOw

- Two-phase write
- Light-weight consistency strategy
  - writing the Order_list into journal files as an atomic operation
  - recreating all inodes with the Order_list and G inode



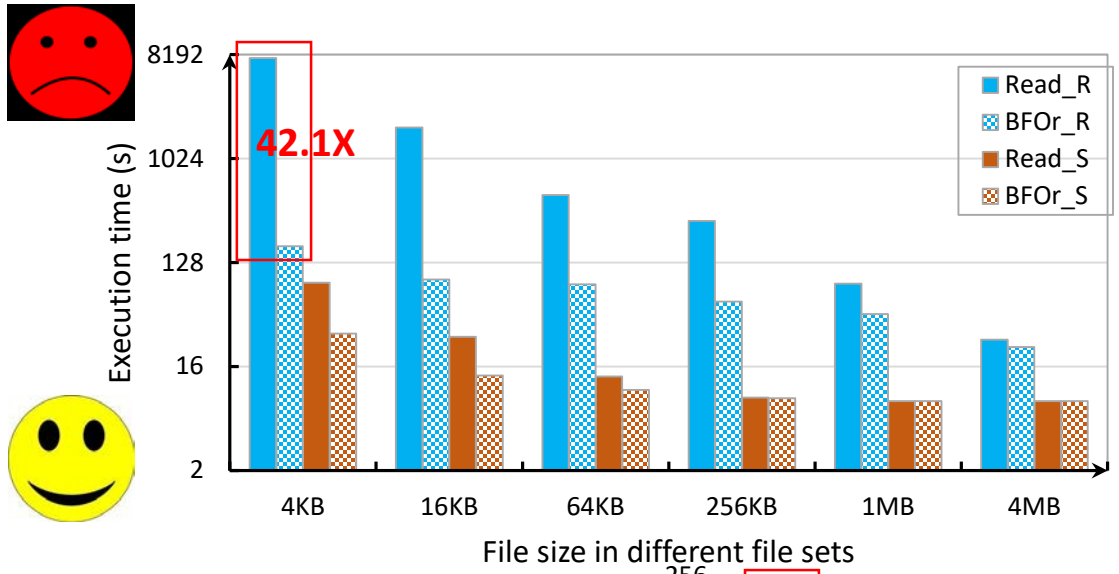
Outline

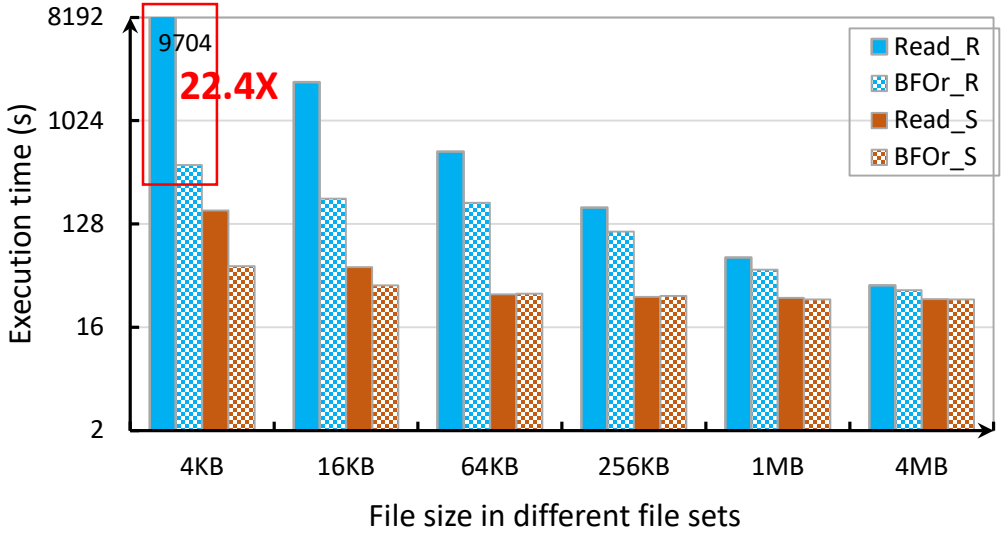
- Background
- BFO Design
- Evaluation
- Conclusion

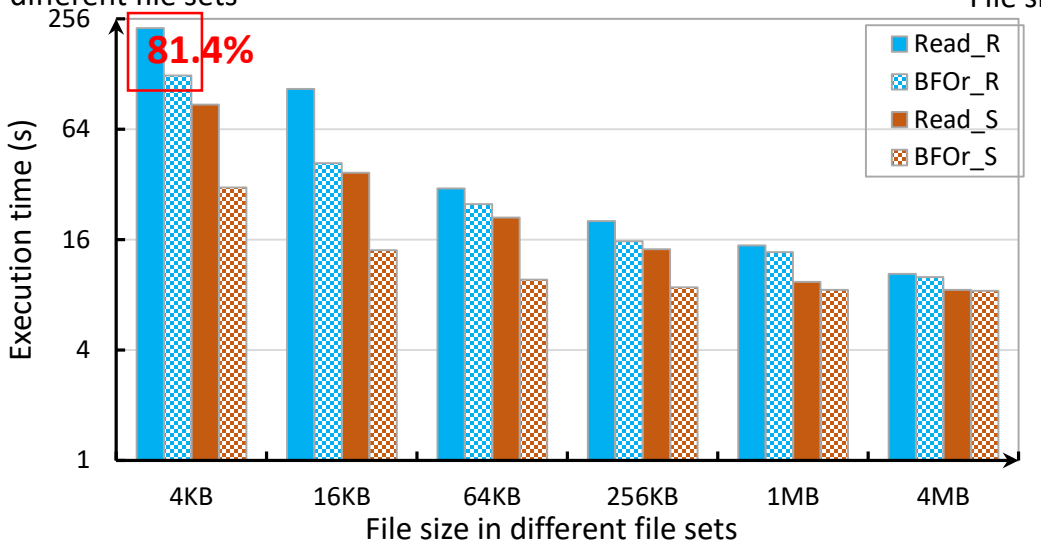
Experimental setup

- Prototyped BFO on ext4
- Intel Xeon E5 2620 @ 2.40GHz and 16GB RAM
- Storage devices
  - RAID0 with 5 Western Digital 7200RPM 4TB SAS HDD
  - A Western Digital 4TB SAS HDD
  - 480GB SAMSUNG 750 EVO SSD
- File sets
  - File sets created by Filebench
    - 4GB data with different file sizes (i.e., from 4KB to 4MB)
  - Linux-kernel source code

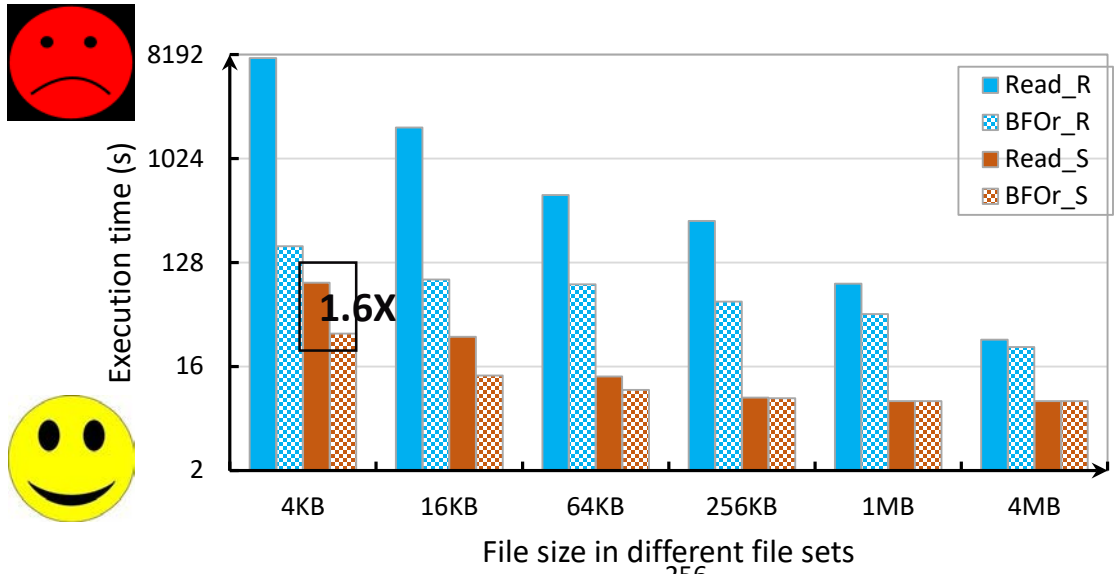
Read Performance

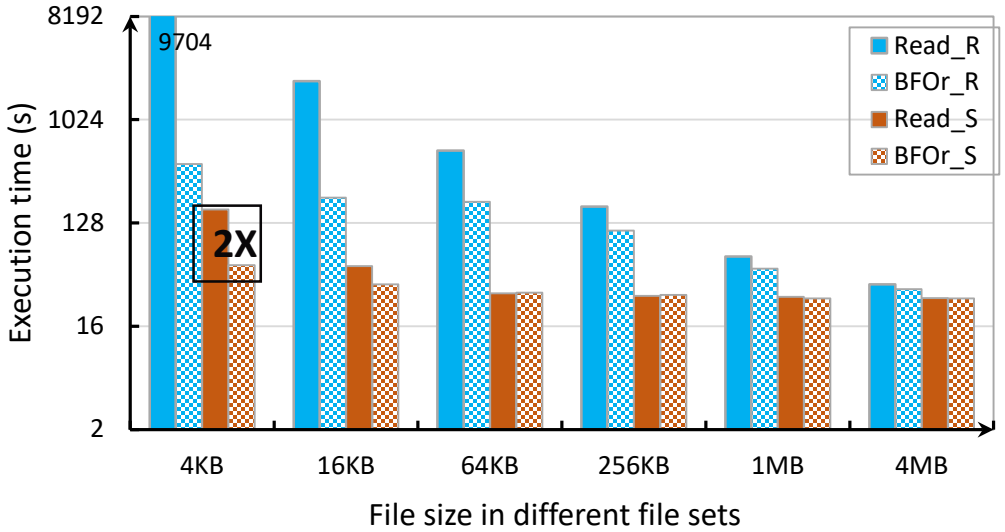


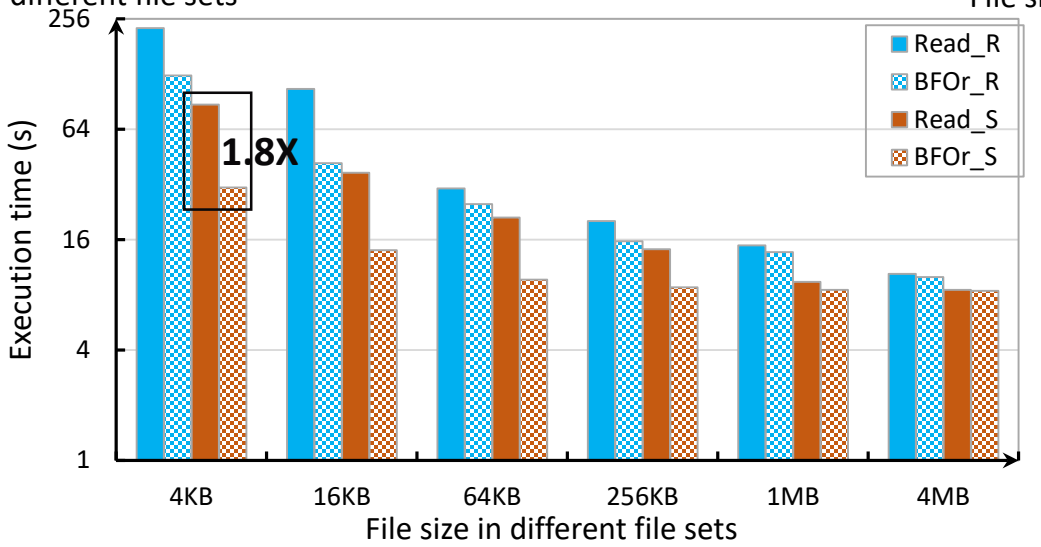




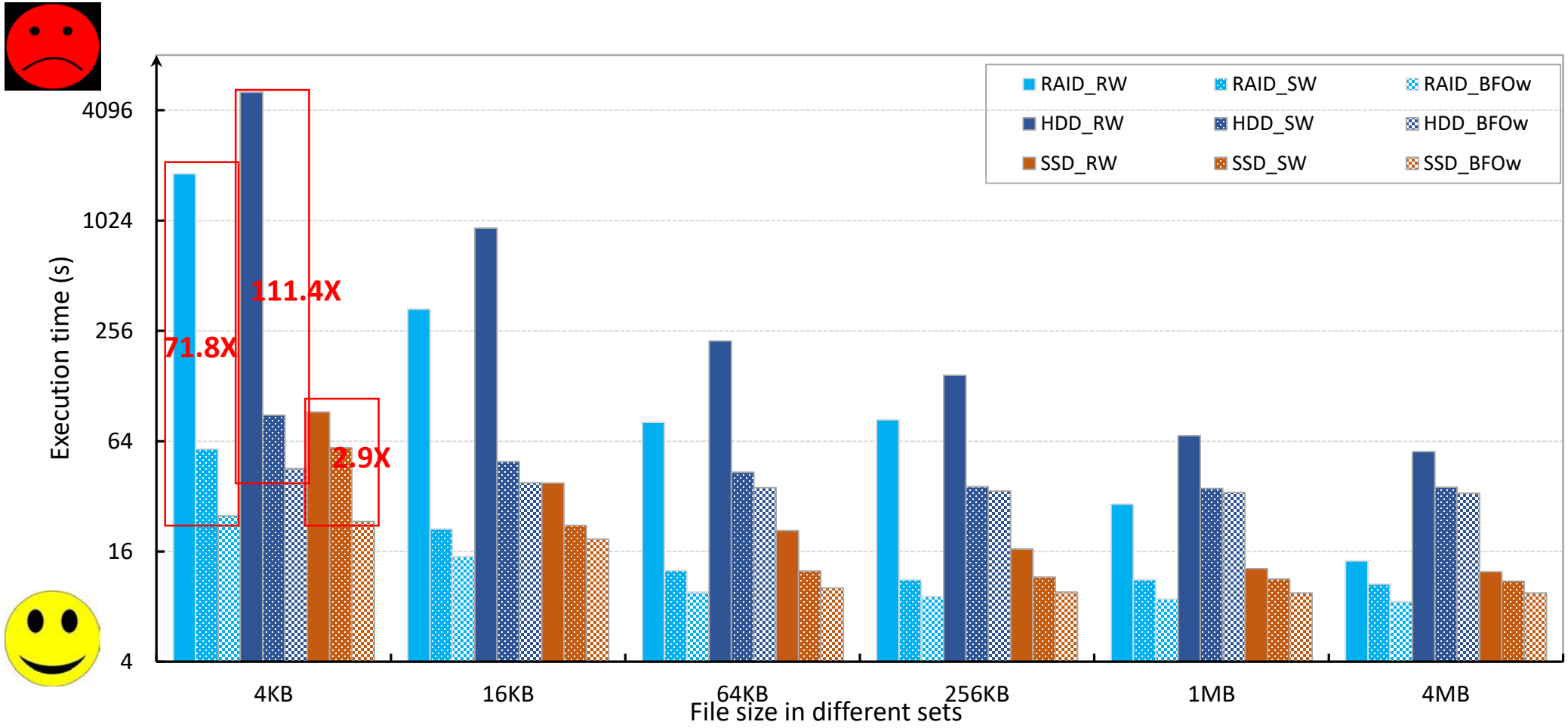
Read Performance



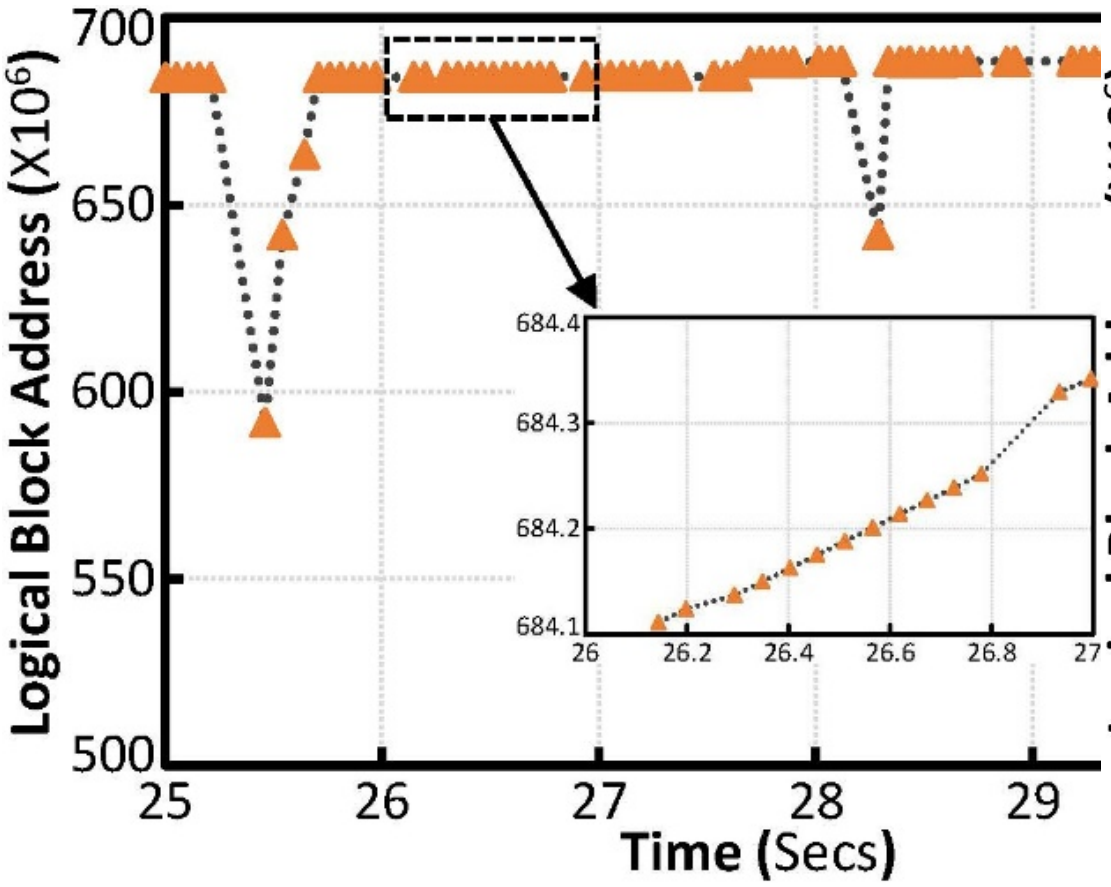


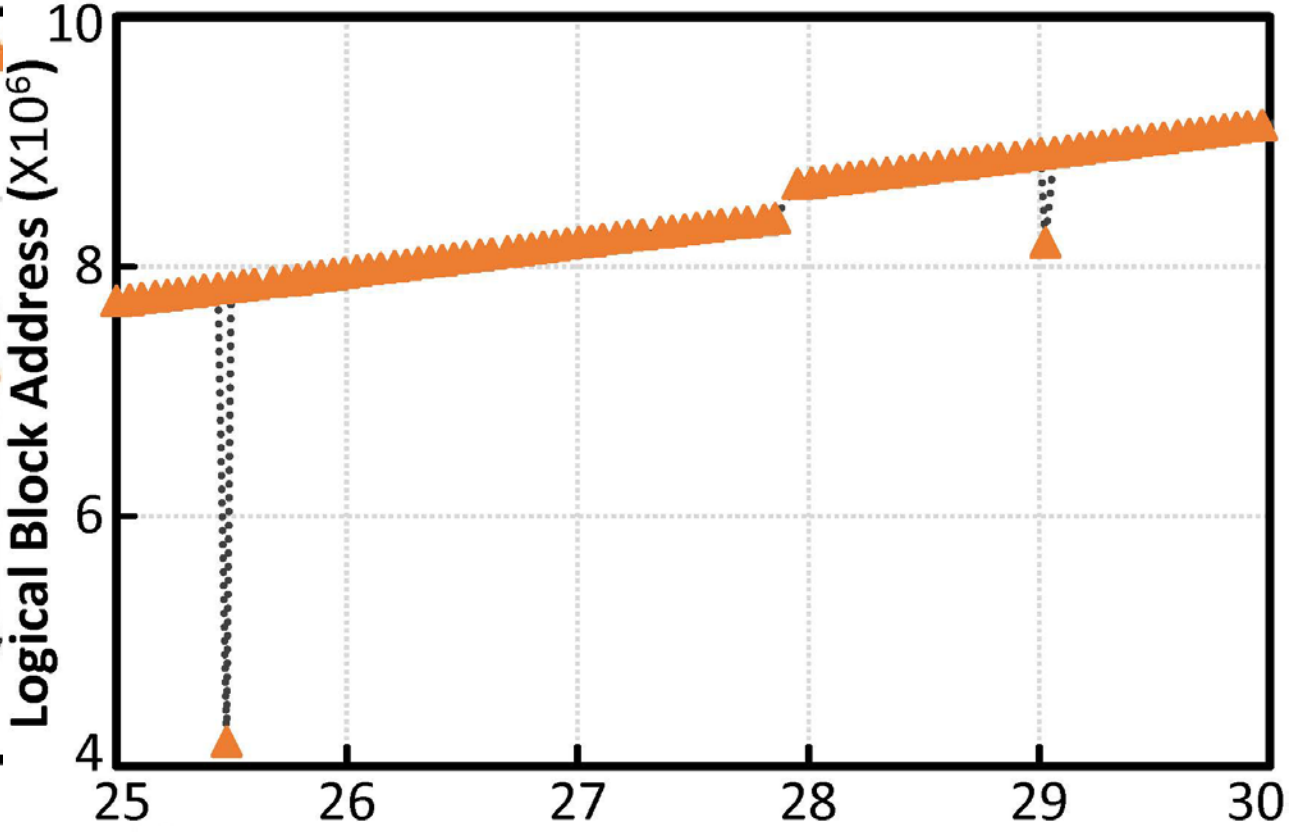


Write Performance

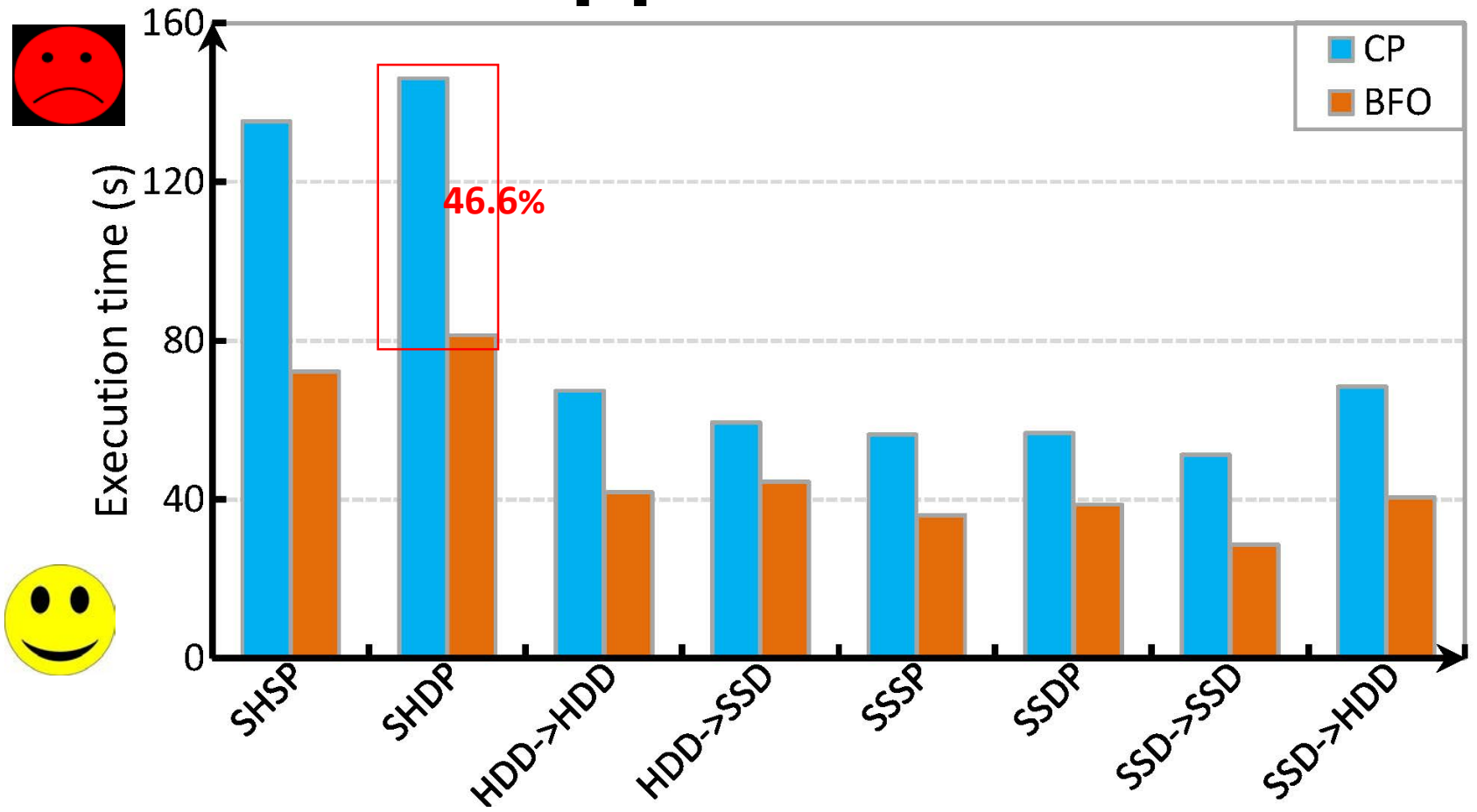


Access Behaviors





Real-world Applications



The execution time of copying a file set with different storage devices. SHSP (SSSP): within the same partition of the same HDD (SSD), SHDP (SSDP): between the different partitions of the same HDD (SSD).

Outline

- Background
- BFO Design
- Evaluation
- Conclusion

Conclusion

- We experimentally investigate the root cause of the inefficiency of the traditional single-file access pattern for batched files.
  - Seeking forth and back between metadata area and data area.
  - Accessing all files in random order.
- We present BFO, for batch-file access, with optimized batch-file read (BFOr) and write (BFOw).
  - Two-phase access.
  - Layout-aware scheduler.
  - Light-weight consistency strategy
- BFO improves the access performance consistently, and removes a significant amount of random and non-sequential I/Os.

Thank You

Q&A