FAST '25

Rethinking the Request-to-IO Transformation Process of File Systems for Full Utilization of High-Bandwidth SSDs

Yekang Zhan[1], Haichuan Hu [1], Xiangrui Yang [1], Qiang Cao [1],
Hong Jiang [2], Shaohua Wang [1] and Jie Yao [1]

[1] Huazhong University of Science and Technology
[2] University of Texas at Arlington

Background: High-Bandwidth SSDs

- SSD bandwidth continues to grow rapidly and is highly cost-effective.
- High-bandwidth SSDs have been widely deployed in IoT, mobile device, servers, and large-scale datacenters.

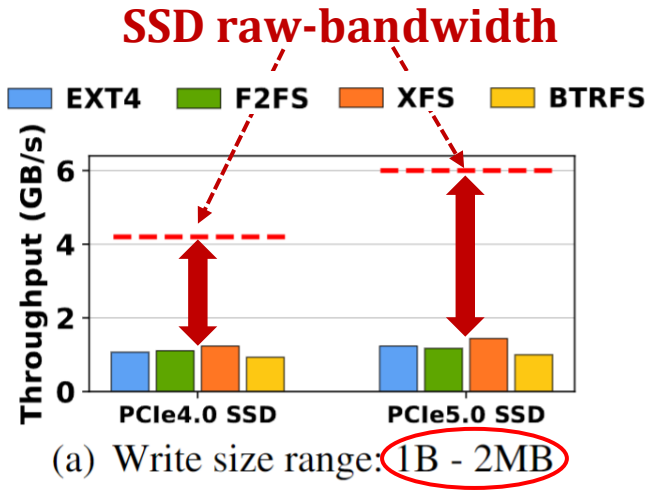
SATA SSD: 500MB/s

PCIe5.0 SSD: 10GB/s

How can emerging applications benefit from high-bandwidth SSDs?

Background: Write Performance of File Systems

- Emerging data-intensive applications often send **writes with diverse offsets and sizes**
  - e.g., graph processing, scientific computing, cloud computing, etc..
- Let's test it out!
  - Observe the performance behavior using an intuitive single-threaded test.
  - FIO test (ioengine=psync, queue-depth=1) with sufficient memory.
  - The throughput of the file systems are about 1/3 and 1/4 of the raw-bandwidth of the PCIe4.0 SSD and PCIe5.0 SSD, respectively.



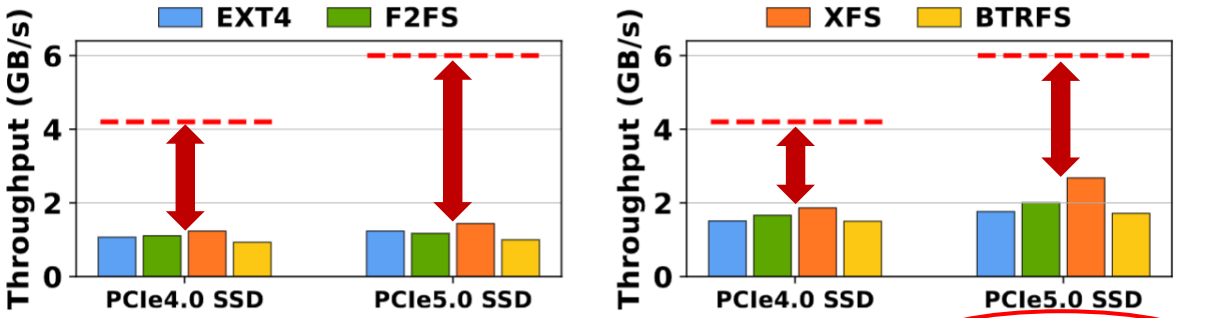
(a) Write size range: 1B - 2MB

Maybe the key performance bottleneck is small writes?

Background: Write Performance of File Systems

- Emerging data-intensive applications often send **writes with diverse offsets and sizes**
  - e.g., graph processing, scientific computing, cloud computing, etc..
- Let's test it out!
  - Observe the performance behavior using an intuitive single-threaded test.
  - FIO test (ioengine=psync, queue-depth=1) with sufficient memory.
  - Maybe the key performance bottleneck is small writes?

  Figure(b): It's far more than that !

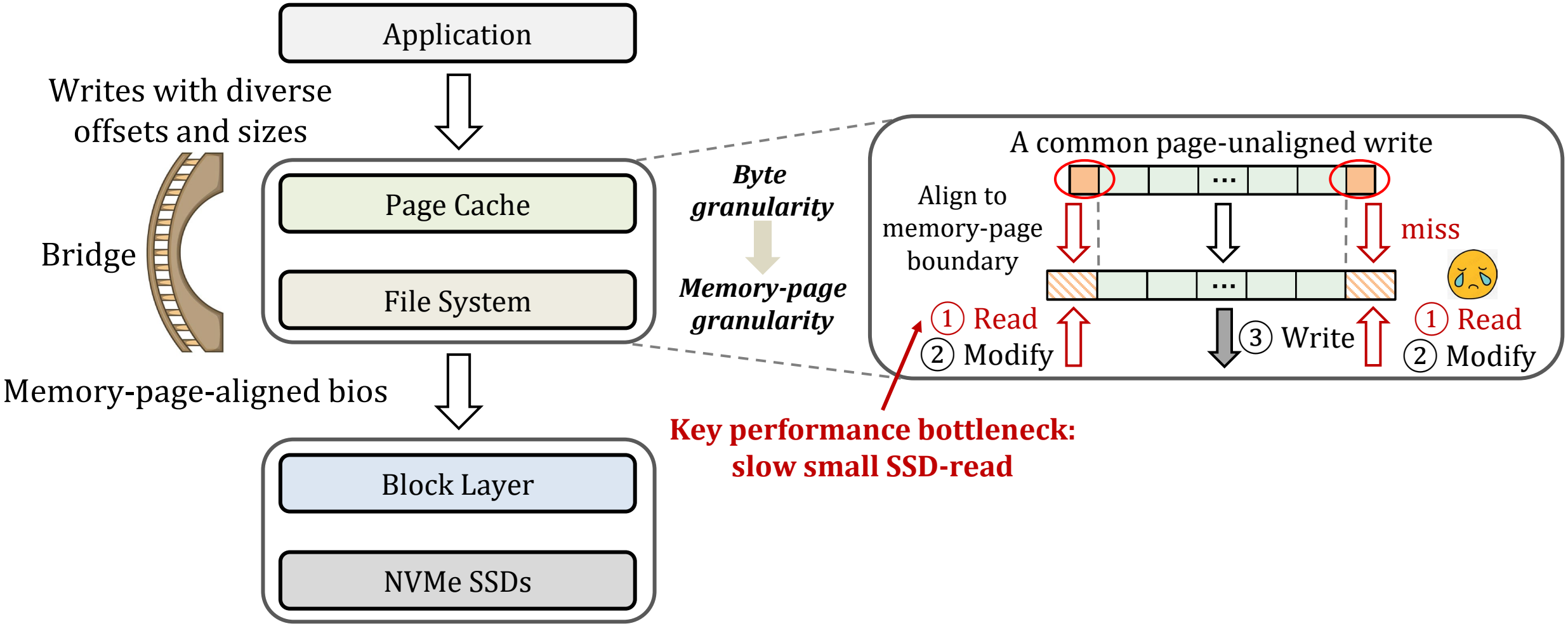


(a) Write size range: 1B - 2MB (b) Write size range: 0.9MB - 1.1MB

What happened with the write operations? 🤔

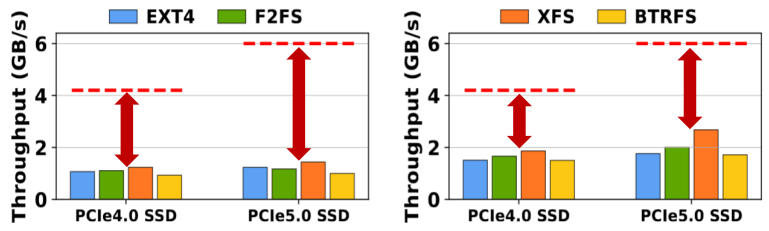
Background: The Write Operation of File Systems

- What happened with the write operations?



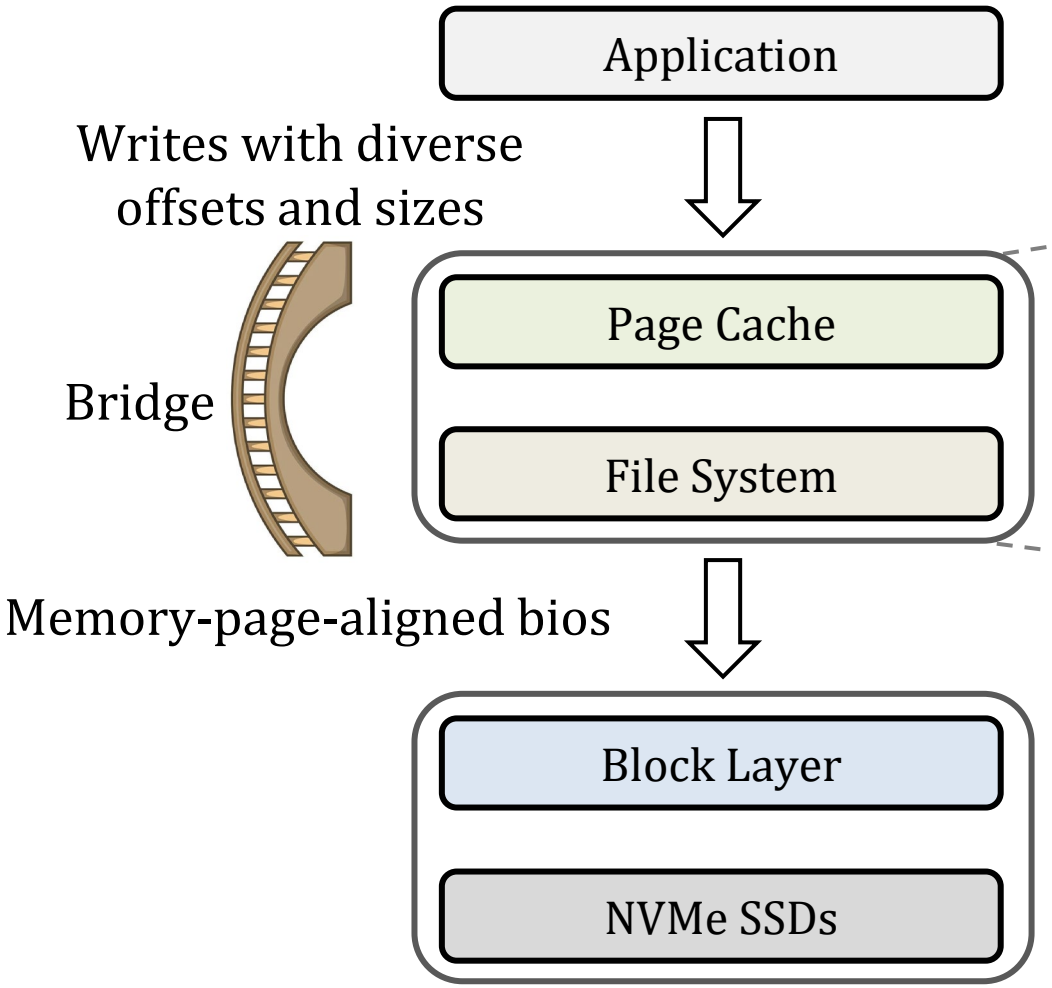
Background: The Write Operation of File Systems

- What happened with the write operations?



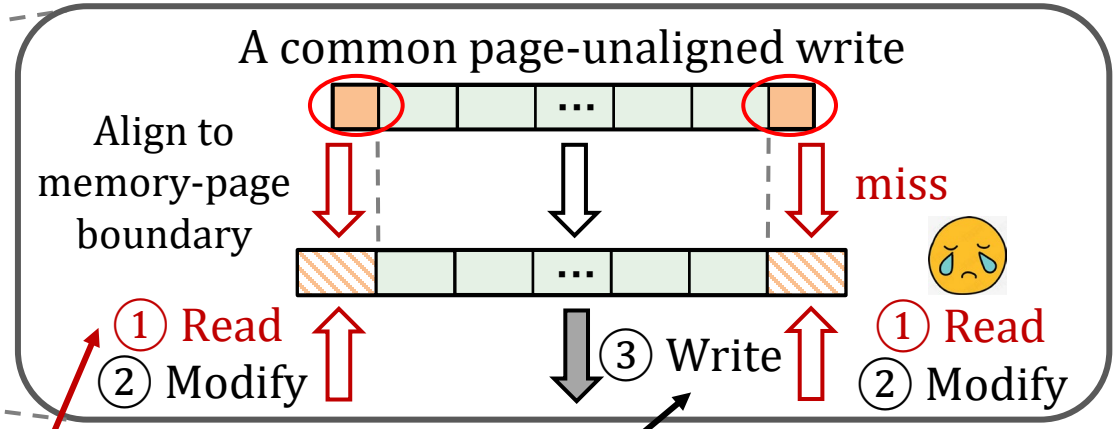
(a) Write size range: 1B - 2MB (b) Write size range: 0.9MB - 1.1MB

Previous FIO test



Byte granularity

Memory-page granularity



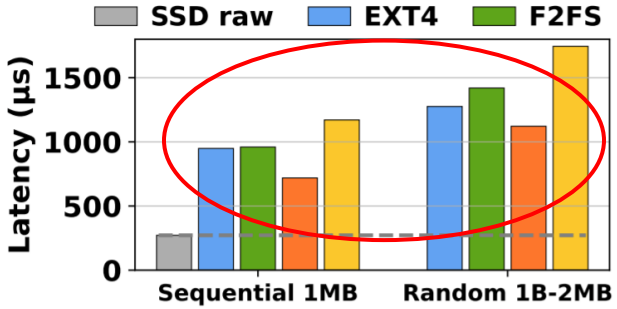
Key performance bottleneck: slow small SSD-read

Due to sufficient memory and without sync-like calls, the ③ write is not executed immediately in the previous FIO test.

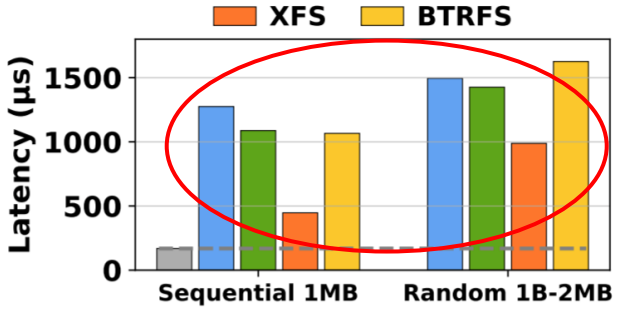
The FIO write performance is entirely derived from page cache.

Background: Write-Inefficiency

- Let's examine the actual write performance unhidden by page cache.
  - FIO test (ioengine=psync, queue-depth=1) with fsync()=1.
  - The write latency of the file systems is 2.64X - 9.62X of the raw SSD latency.
- File systems fail to deliver the performance of high-bandwidth SSDs to applications.
  - We refer to this phenomenon as ***write-inefficiency***.
  - Not only alignment, even aligned large writes suffer from write-inefficiency.



(a) on a PCIe4.0 SSD



(b) on a PCIe5.0 SSD

High additional overhead

Background: Root causes of Write-Inefficiency

- I want to know the root causes of write-inefficiency. 
  - **1) IO Alignment**

    Aligned Direct IO writes vs. Unaligned Direct IO writes.

    Using Read-Modify-Write (RMW) to align unaligned writes on host.

    Direct IO: Exclude the impact of page cache.

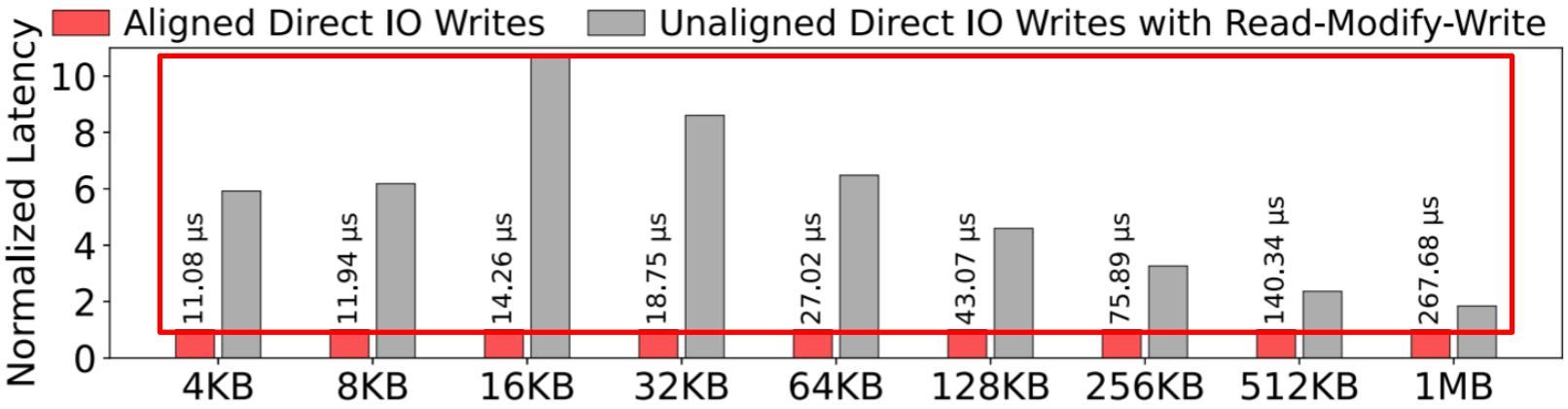
Background: Root causes of Write-Inefficiency

- I want to know the root causes of write-inefficiency. 
  - **1) IO Alignment**

    Aligned Direct IO writes vs. Unaligned Direct IO writes.

    Using Read-Modify-Write (RMW) to align unaligned writes on host.

    Direct IO: Exclude the impact of page cache.



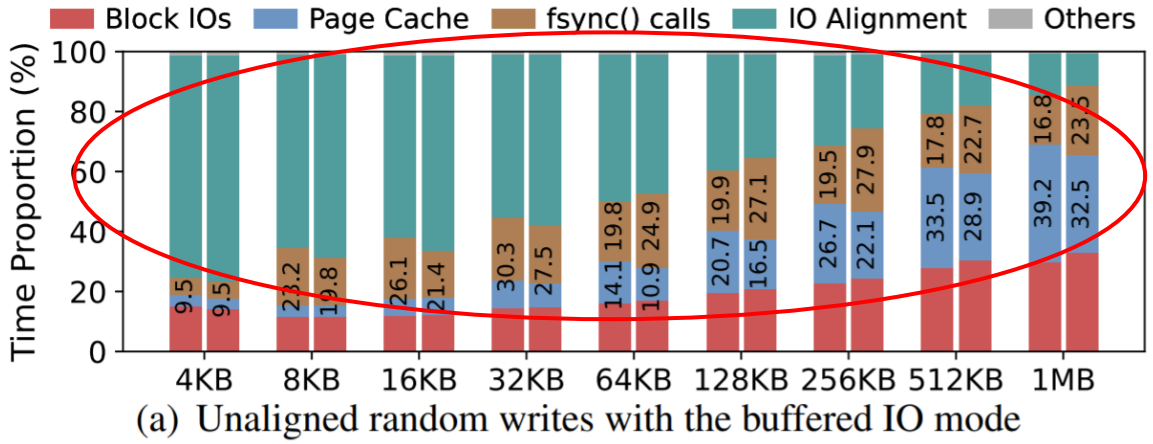
The latency of unaligned direct IO writes is 1.85X-10.71X that of aligned direct IO writes

Background: Root causes of Write-Inefficiency

- I want to know the root causes of write-inefficiency. 
  - 1) IO Alignment
  - **2) Page cache**

    The larger the write size, the more significant the page cache overhead. (4.0%- 39.2%)

    The overhead of fsync() calls (excluding IO times) is also significant. (5.5%-27.9%)



(a) Unaligned random writes with the buffered IO mode

Write time breakdown of EXT4 (left bars) and F2FS (right bars).

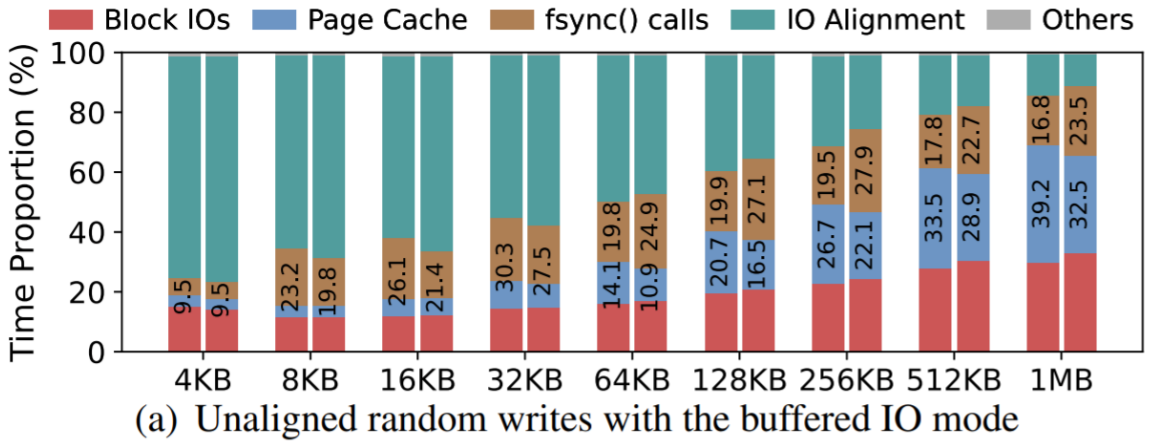
Background: Root causes of Write-Inefficiency

- I want to know the root causes of write-inefficiency. 
  - 1) IO Alignment
  - **2) Page cache**

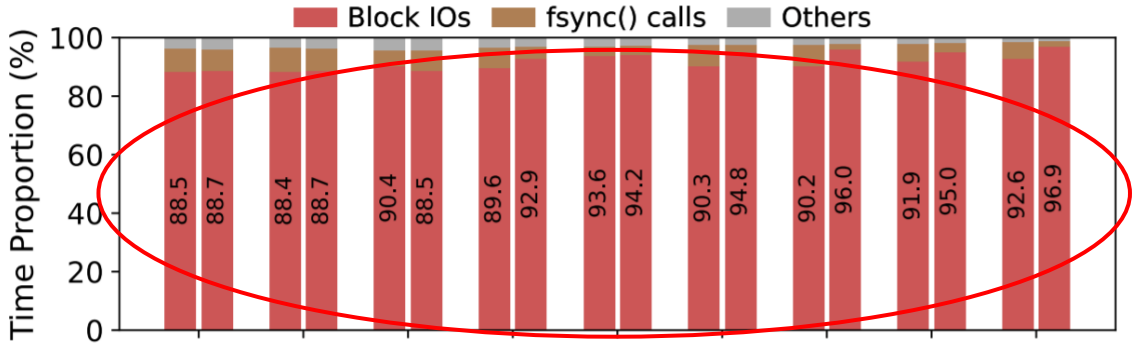
    The larger the write size, the more significant the page cache overhead. (4.0%- 39.2%)

    The overhead of fsync() calls (excluding IO times) is also significant. (5.5%-27.9%)

    In comparison, the direct IO writes bypassing page cache exhibit low fsync() overhead, but requiring strict alignment among the IO size, offset, and memory-buffer address



(a) Unaligned random writes with the buffered IO mode



(b) Aligned random writes with the direct IO mode

Write time breakdown of EXT4 (left bars) and F2FS (right bars).

Background: Root causes of Write-Inefficiency

- I want to know the root causes of write-inefficiency. 
  - 1) IO Alignment
  - 2) Page cache
  - **3) IO Concurrency**

    A single IO thread fails to maximize SSD performance.

    Actively splitting IO into multiple smaller SSD-page aligned IOs and executing in a multi-threaded manner can maximize SSD bandwidth.

Background: Root causes of Write-Inefficiency

- I want to know the root causes of write-inefficiency. 🧐
  - 1) IO Alignment
  - 2) Page cache
  - **3) IO Concurrency**

    A single IO thread fails to maximize SSD performance.

    Actively splitting IO into multiple smaller SSD-page aligned IOs and executing in a multi-threaded manner can maximize SSD bandwidth.

The write and read throughputs in the multi-threaded IO pattern are 1.02X-1.56X and 1.06X-1.65X that of the single-threaded IO pattern

Motivation

- I want to deliver the full performance of high-bandwidth SSDs to applications.
  - Issues:

    1) High IO alignment overhead

    2) High page caching overhead

    3) Insufficient IO Concurrency

  - Intuitive idea:

    For 1), merging writes in memory and then asynchronously writing them back to SSD, but sacrificing persistence and wasting the high bandwidth provided by SSDs.

    For 2), using Direct IO mode to bypass page cache, but how to meet the strict alignment?

    For 3), using proactive IO splitting and multi-threaded IO processing, but how to implement it?

Motivation

- How to deliver the performance of high-bandwidth SSDs to applications?
  - Issues:

    1) High IO alignment overhead
    2) High page caching overhead
    3) Insufficient IO Concurrency

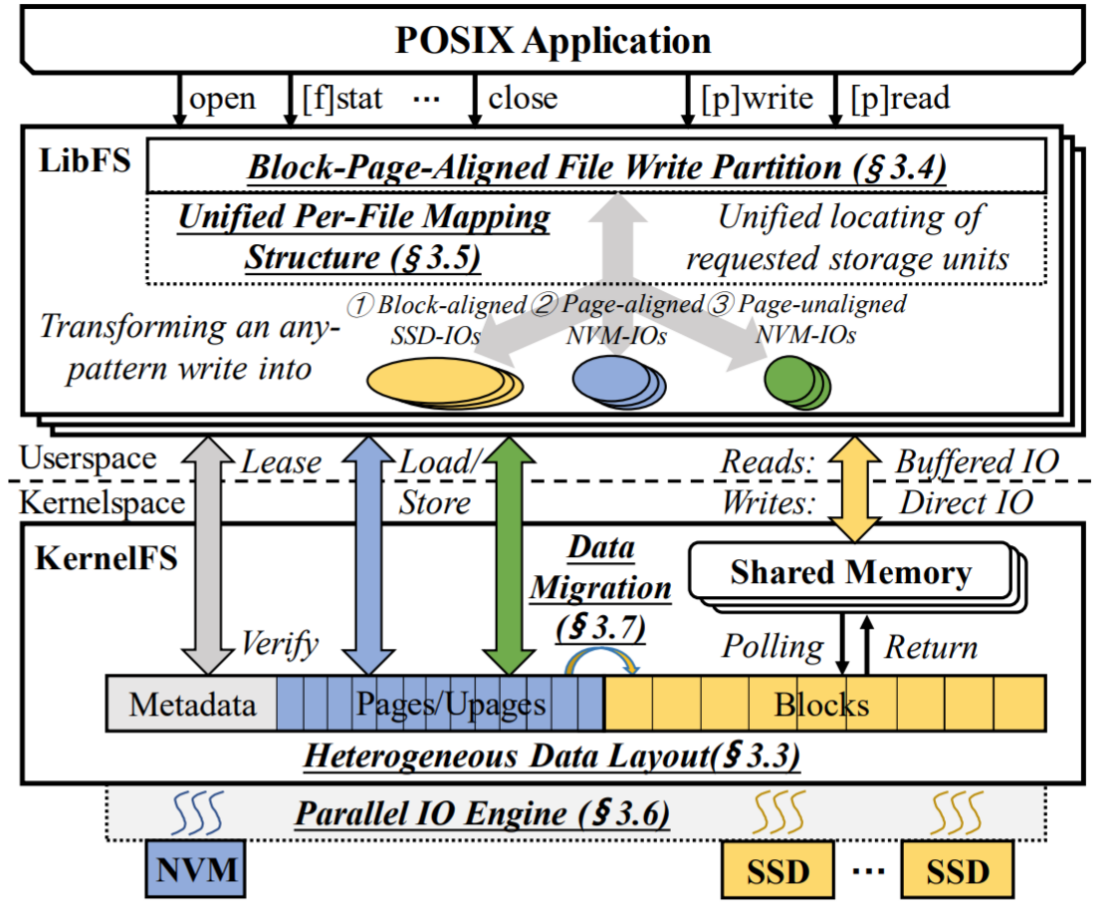
- Our solution:
  - Introducing low-latency NVMs as auxiliary storage to maximize SSD performance.

    Transforming any-pattern file-writes into **SSD-page aligned SSD-IOs** and/or remaining **SSD-page unaligned NVM-IOs**.

    Using direct IO mode and **multi-threaded IO processing** to handle SSD-IOs that are always aligned, thereby maximizing SSD performance.
  - Key idea: **Partitioning** writes and handling them by persistent storage immediately **instead of merging** writes in memory and persisting them asynchronously.

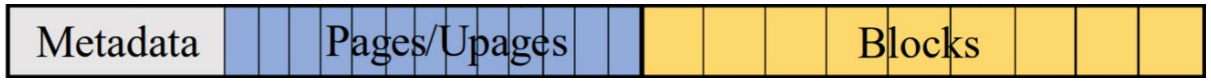
OrchFS Design

- OrchFS: An SSD-NVM heterogeneous-IO orchestrated file system
- Key Technologies:
  - Heterogeneous Data Layout
  - Block-Page-Aligned File Write Partition
  - Unified Per-File Mapping Structure
  - Parallel IO Engine

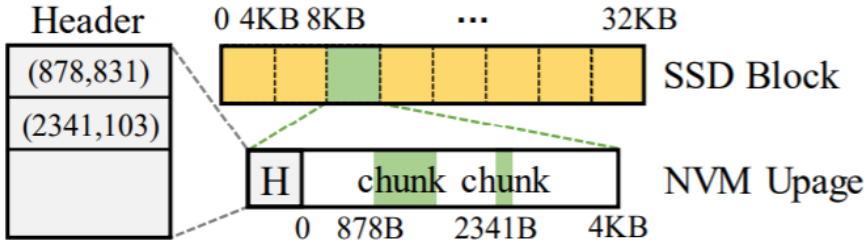


OrchFS Design

- Heterogeneous Data Layout
  - Metadata: exclusively stored on NVM
  - Data: stored across NVM and SSD
  - Three types of data storage unit: NVM pages, NVM Upages, and SSD blocks
  - SSD blocks, e.g., 32KB, used for storing 32KB-aligned data segment
  - NVM pages, e.g., 4KB, used for storing 4KB-aligned data segment
  - NVM Upages (Unaligned pages), e.g., 64B (header) + 4KB (data), used for storing and locating unaligned data segment.



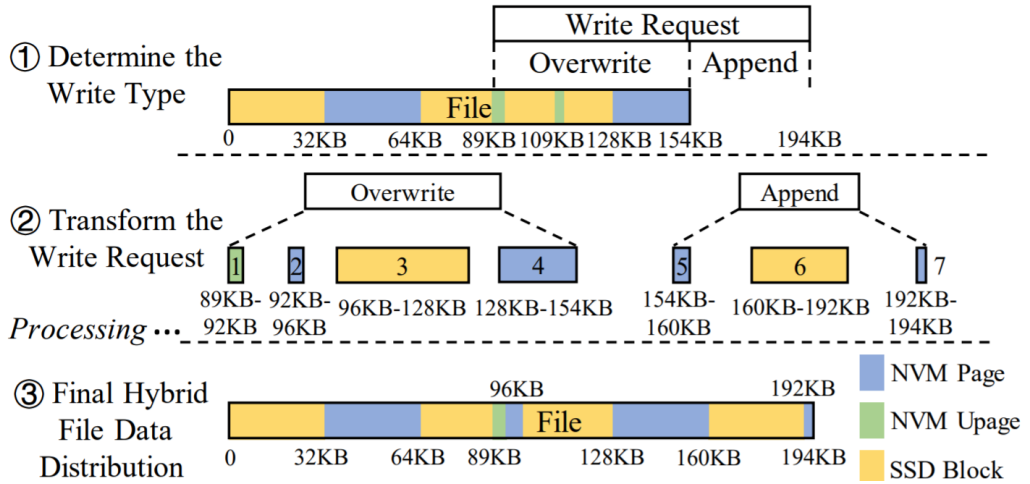
Heterogeneous Data Layout



An NVM Upage example

OrchFS Design

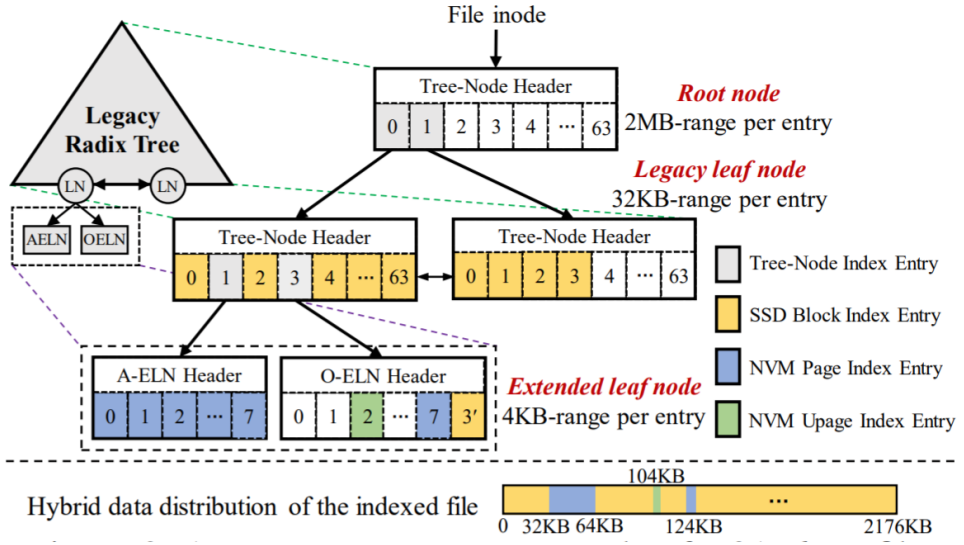
- Block-Page-Aligned File Write Partition
  - Partitioning an any-pattern write into block-aligned SSD-IOs, page-aligned NVM-IOs, and page-unaligned chunk-IOs sequentially.
  - Sending SSD-IOs to SSD blocks, NVM-IOs to NVM pages, and chunk-IOs to NVM Upages.
  - Alignment-prioritizing policy and fragmentation-minimizing policy.
  - See our paper for more details.



A write-partition example

OrchFS Design

- Unified Per-File Mapping Structure: HRtree (Heterogeneous Radix tree)
  - Used for mapping from file's logical offset to blocks, pages, and Upages simultaneously.
  - HRtree = A legacy radix tree + a heterogeneous layer on the bottom
  - The radix tree part: mapping a file's logical offset to a logical block (e.g., 32KB).
  - The heterogeneous layer: recording the data composition of the corresponding logical block.
  - A single traversal can locate all storage units of a file request.



An HRtree structure example

OrchFS Design

- Parallel IO Engine
  - Used for handling SSD-IOs and NVM-IOs in parallel.
  - Using multi-threaded IO processing to handle SSD-IOs.
  - Each dedicated IO thread uniquely executes IOs within its corresponding exclusive address range, thus ensuring data consistency.
  - Triple data paths:

    Using Direct IO mode to handle SSD-write IOs.

    Using Buffered IO mode to handle SSD-read IOs benefiting from mature page cache reading.

    Using popular memory-semantic data path for NVM IOs.

OrchFS Design

- Other designs and optimizations
  - Data defragmentation

    Using merge-on-write approach to handle NVM-Upage writes.

    Data migration mechanism for merging used NVM pages and Upages to corresponding SSD blocks.
  - HRtree optimizations

    Connecting leaf nodes of HRtree similarly to B+tree to speedup range search.

    A readers-writer range lock on HRtree to support fine-grained concurrent accesses.
  - File system architecture

    An extended LibFS-KernelFS architecture to further reduce IO penalties.
  - See the paper for more details

Evaluation: Experimental Setup

Hardware Platform

- Ubuntu 20.04 and Linux kernel 5.18.18
- Dual Intel Xeon Gold 6348 processors (2.60 GHz, 28 CPUs) and 256GB of DDR4 DRAM.
- SSD: a PCIe4.0 3.84TB Samsung PM9A3 SSD (6.8GB/s read and 4.2GB/s write bandwidth)
- NVM: an 128GB Intel Optane PM (6.6GB/s read and 2.3GB/s write bandwidth)

Baseline File Systems

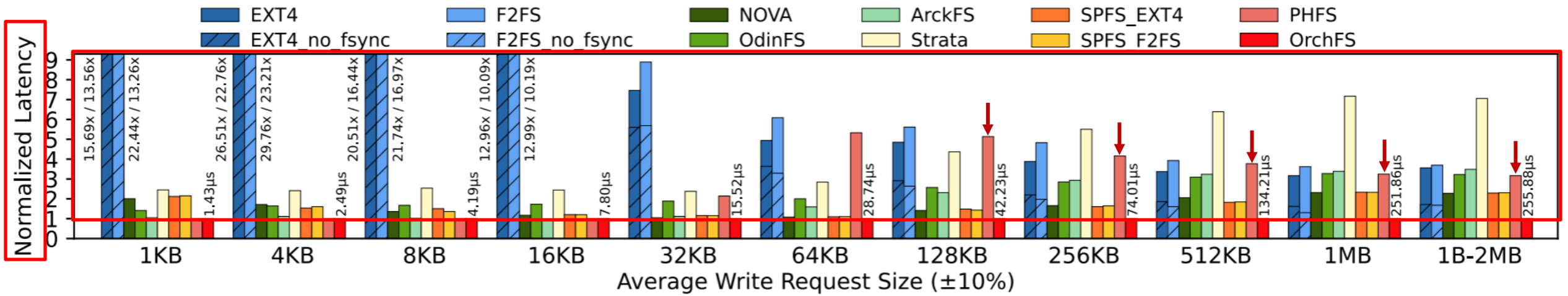
- SSD file systems: EXT4 and F2FS
- NVM file systems: NOVA (FAST'16), OdinFS (OSDI'22) and ArckFS (SOSP'23)
- Hybrid NVM-SSD file systems: Strata (SOSP'17), SPFS (FAST'23) and PHFS*

  SPFS is stacked on EXT4 and F2FS respectively for evaluation.

  PHFS is implemented based on our OrchFS. It stores all metadata in NVM, and sends small and large write requests to NVM and SSD respectively using 32KB as the small-large boundary.

  PHFS is used to intuitively demonstrate the benefits of the OrchFS designs.

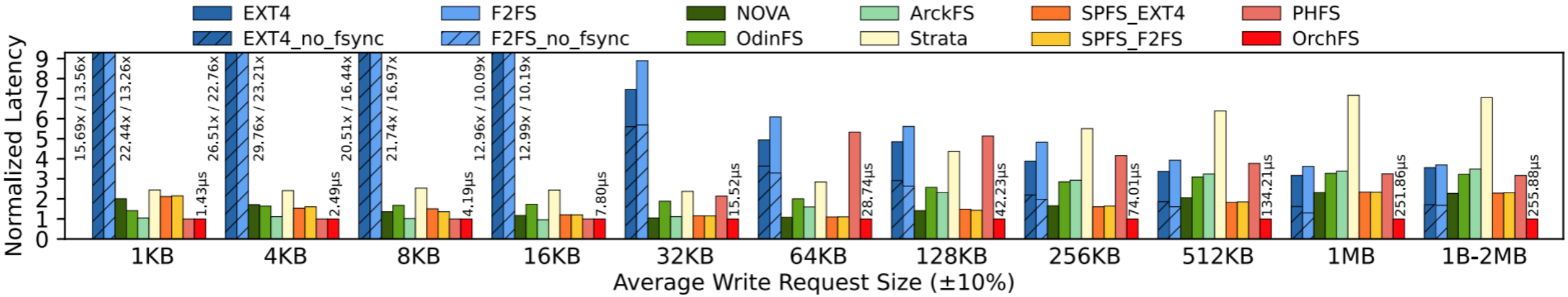
Evaluation: Single-Threaded Writes



Single-threaded random writes

Compared to the SSD file systems, OrchFS improves write latency by up to 96.6%;
Compared to the NVM file systems, OrchFS improves write latency by up to 71.3%;
Compared to the hybrid NVM-SSD file systems, OrchFS improves write latency by up to 86.0%.

Evaluation: Storage Space Usage



Single-threaded random writes

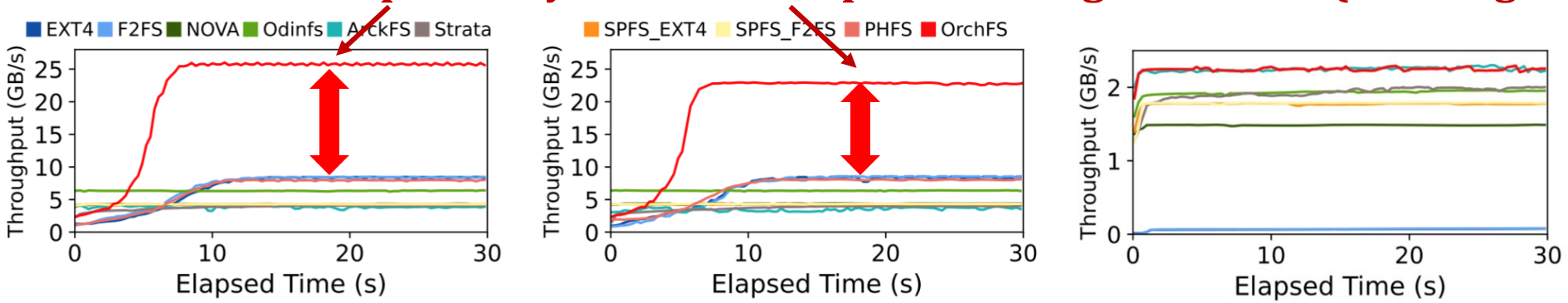
Table 2: OrchFS storage space usage breakdown.

IO Size (±10%)	1,4,8,16KB	32KB	64KB	128KB	256KB	512KB	1MB	1B-2MB
	Space occupation after the 10GB pre-fill (i.e., append writes) (GB)							
NVM Page	10	9.76	5.19	2.50	1.25	0.63	0.31	0.32
NVM Upage	0	0	0	0	0	0	0	0
SSD Block	0	0.24	4.81	7.50	8.75	9.37	9.69	9.68
	Space occupation after the 10GB random writes on the above filled file (GB)							
NVM Page	10	9.89	7.06	4.19	2.34	1.25	0.63	0.64
NVM Upage	0	0.03	0.47	0.42	0.27	0.15	0.08	0.08
SSD Block	0	0.24	4.81	7.50	8.75	9.37	9.69	9.68

In large write cases, most of OrchFS's performance stems from the SSD.

Evaluation: Single-Threaded Reads

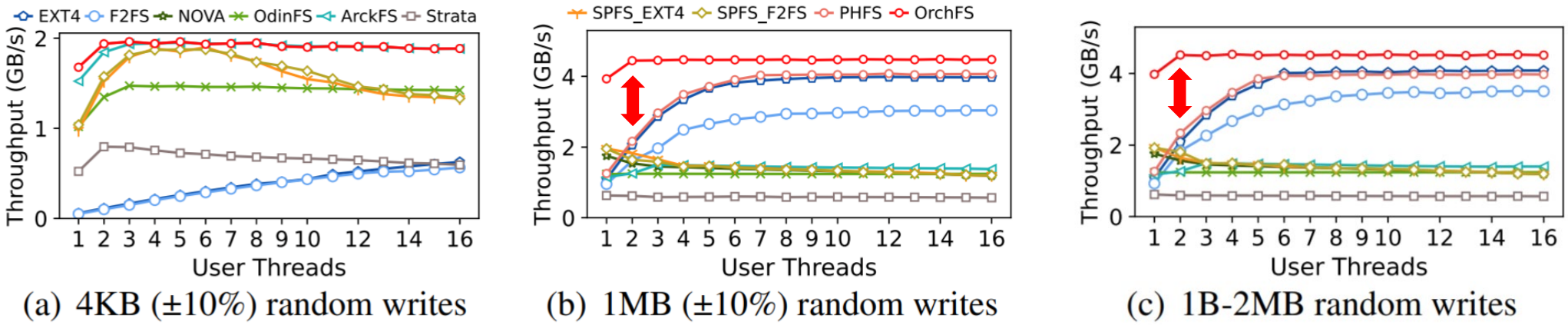
OrchFS's SSD-side reads passively benefit from previous aligned writes (less fragmentation).



Single-threaded random reads

Compared to the SSD file systems, OrchFS improves peak read throughput by up to 3.08X;
Compared to the NVM file systems, OrchFS improves peak read throughput by up to 6.79X;
Compared to the hybrid NVM-SSD file systems, OrchFS improves peak read throughput by up to 6.34X.

Evaluation: Multi-threaded Writes



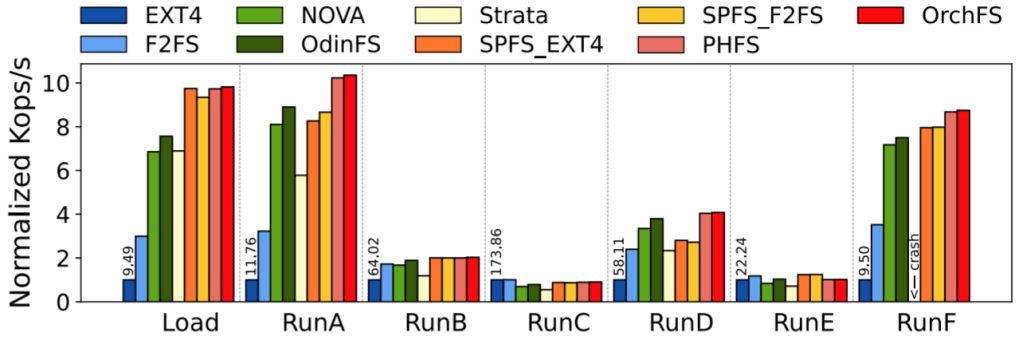
Multi-threaded random writes

OrchFS maximizes SSD performance with a small number of user threads (e.g. 2).

Evaluation: Real-World Applications

Table 4: YCSB workload characteristics.

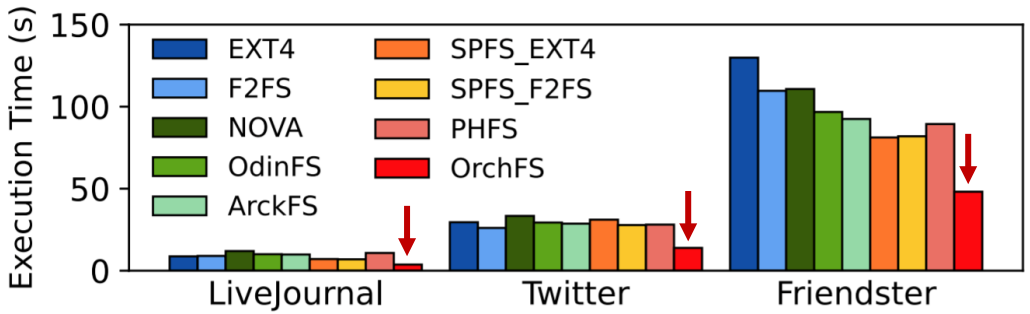
Load	Write only: 100% insert
RunA	Update heavy: 50%/50% read/write mix
RunB	Read mostly: 95%/5% read/write mix
RunC	Read only: 100% read
RunD	Read latest: new records are inserted, and the most recently inserted records are read
RunE	Short ranges: short ranges of records are queried, instead of individual records
RunF	Read-modify-write: read a record, modify it, and write back the changes



YCSB+LevelDB throughput.

Table 5: Workload characteristics of graph processing.

Dataset	|V|	|E|	Dataset Size	Writes	Reads	Average Write Size	Average Read Size
LiveJournal	4.85M	69.0M	539MB	1.8GB	12.8GB	123.1KB	16.5MB
Twitter	61.6M	1.5B	11.5GB	35.4GB	25.5GB	1.02MB	4.9MB
Friendster	68.3M	2.6B	28.2GB	86.8GB	93.4GB	129.4KB	11.0MB



Execution time of graph processing.

In small-access-dominated LevelDB, OrchFS improves throughput by up to 10.31X.
In graph processing with mixed small and large accesses, OrchFS reduces processing time by up to 68.8%.

Conclusion

- We explore and identify the root causes of write-inefficiency of file systems on HBSSDs.
  - High IO alignment overhead
  - High page caching overhead
  - Insufficient IO Concurrency
- We present OrchFS to overcome the write-inefficiency to maximize SSD performance.
  - Heterogeneous Data Layout
  - Block-Page-Aligned File Write Partition
  - Unified Per-File Mapping Structure
  - Parallel IO Engine
  - Other designs and optimizations
- OrchFS outperforms popular SSD file systems, SOTA NVM file systems and NVM-SSD file systems by up to 29.76× and 6.79× in write and read performances, respectively.

Thank You! & QA

Yekang Zhan, Haichuan Hu, Xiangrui Yang,
Qiang Cao, Hong Jiang, Shaohua Wang and Jie Yao

Email: ***zhanyekang@foxmail.com***

Source code: ***https://github.com/YekangZhan/OrchFS***

Appendix: IO Alignment

- I want to know the root causes of write-inefficiency. 🧐
  - **1) IO Alignment**

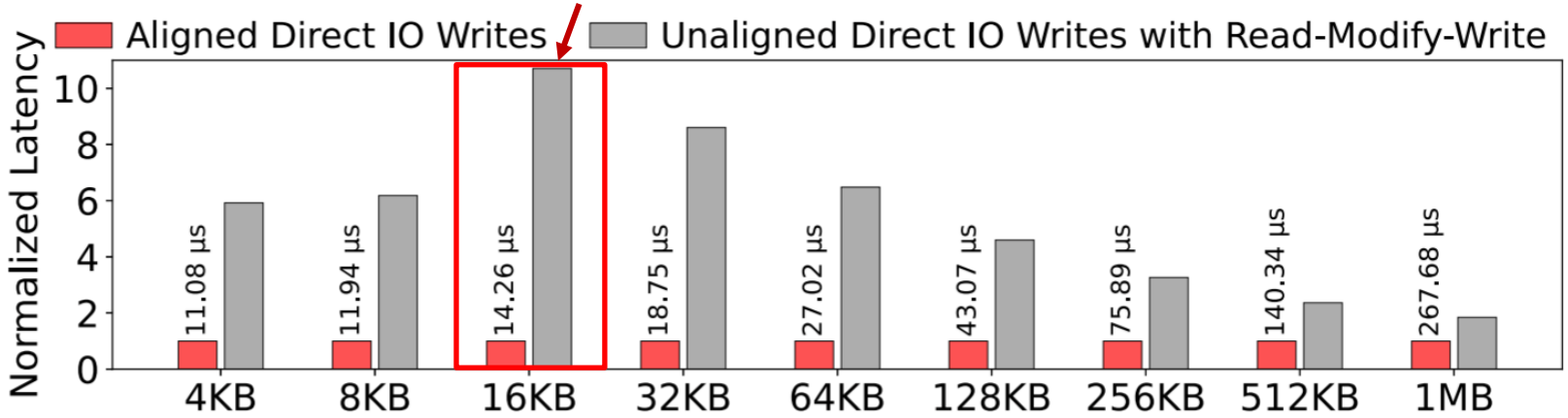
    Aligned Direct IO writes vs. Unaligned Direct IO writes.

    Using Read-Modify-Write (RMW) to align unaligned writes on host.

    Direct IO: Exclude the impact of page cache.

    Not only aligning to memory-page boundary on host, but also aligning to SSD-page boundary (e.g., 16KB) within SSD.

16KB user write -> 20KB write on host -> 32KB write within SSD



The latency of unaligned direct IO writes is 1.85X-10.71X that of aligned direct IO writes

Appendix: IO Alignment