HotStorage 2026
The 18th ACM Workshop on Hot Topics in Storage and File Systems
Sept. 28-29
Prague, Czechia
Co-located with SOSP 2026

Manycore Scalable Logging for Log-Structured Filesystem



Hwanseok Yoo Juwon Kim Youjip Won

KAIST

Background



Log Structured Filesystem

Log Structured Filesystem

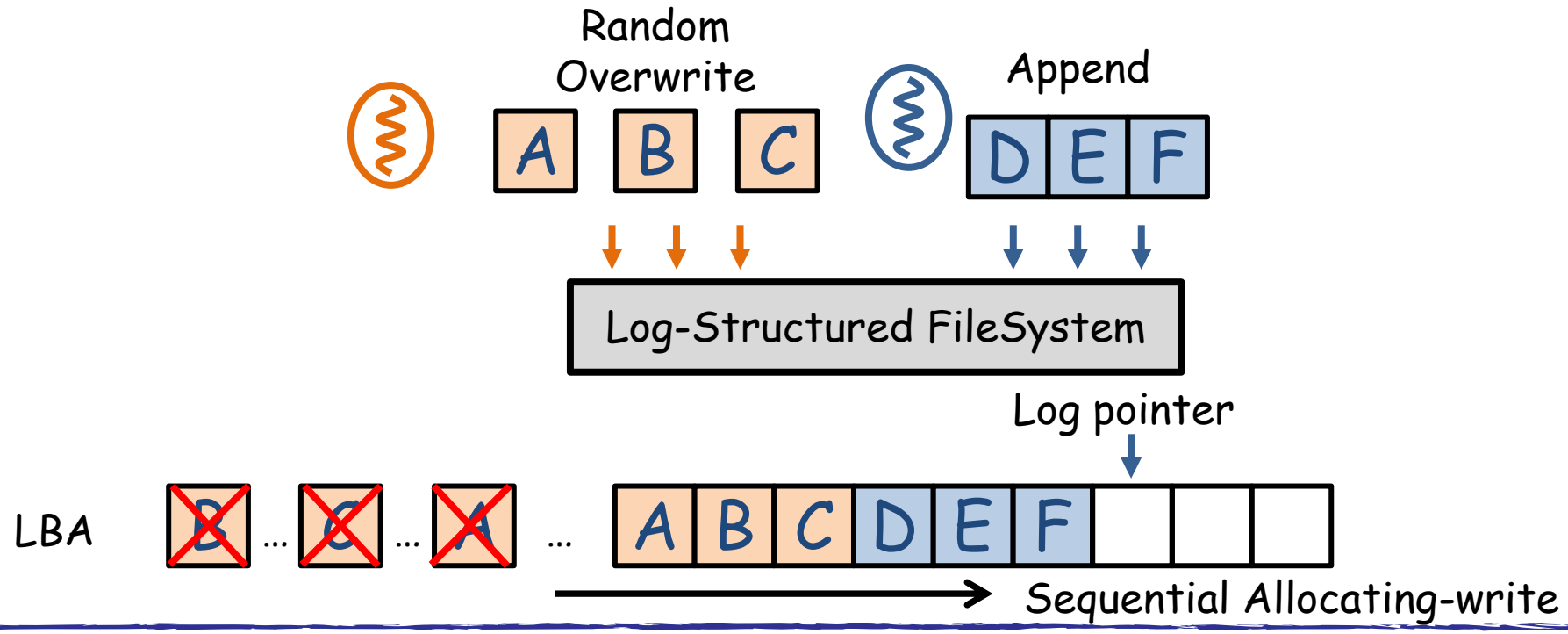
- Out-place update filesystem
- Append-only log
- Writer does not recycle the old block
- Converts random writes into sequential writes



Log Structured Filesystem

Allocating Write?

- write request claims a new block -> allocating write
- write request recycles an existing block -> non-allocating write
- LFS turns every write (random overwrite and append) into a sequential allocating write



Log Structured Filesystem

Why LFS?

- Sequential write is storage-friendly!
  - HDD: less seek time
  - SSD: flash-friendly, reduces the write amplification

For conversion every writer should do additional operation

- Trade off!
  - Random Storage Access (IPU filesystem)
  - Additional operation + Sequential Storage Access

The storage used to be the bottleneck. Paying the additional operation was worth the sequential access.

Our question: Does the trade-off still hold?

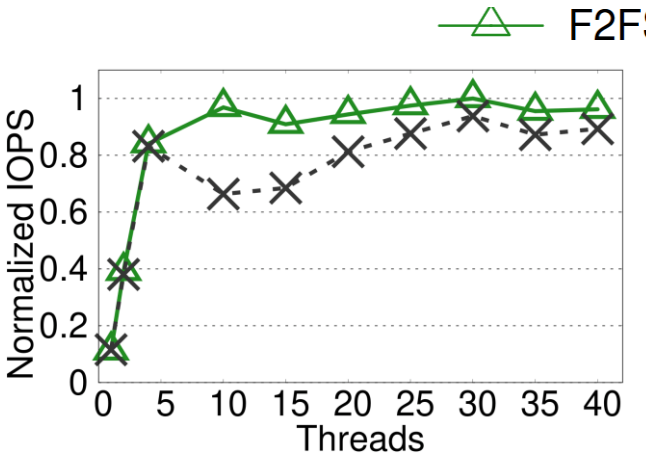
- The storage now delivers millions of IOPS (NVMe, PCIe 4.0, PCIe 5.0)



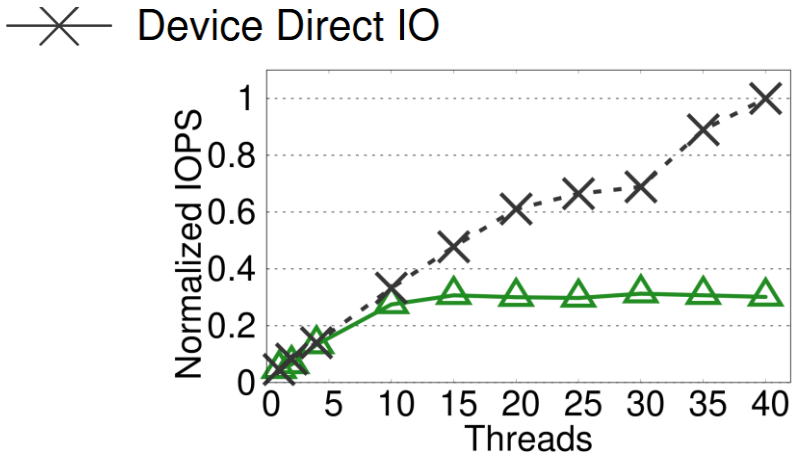
Log Structured Filesystem

LFS Direct write Scalability on fast SSD.

- We observe the bottleneck of LFS on fast SSDs (PCIe 5.0) under Direct Write.
- The CPU cost of (random write -> sequential write) is no longer hidden.



<Slow SATA SSD (100 KIOPS)>

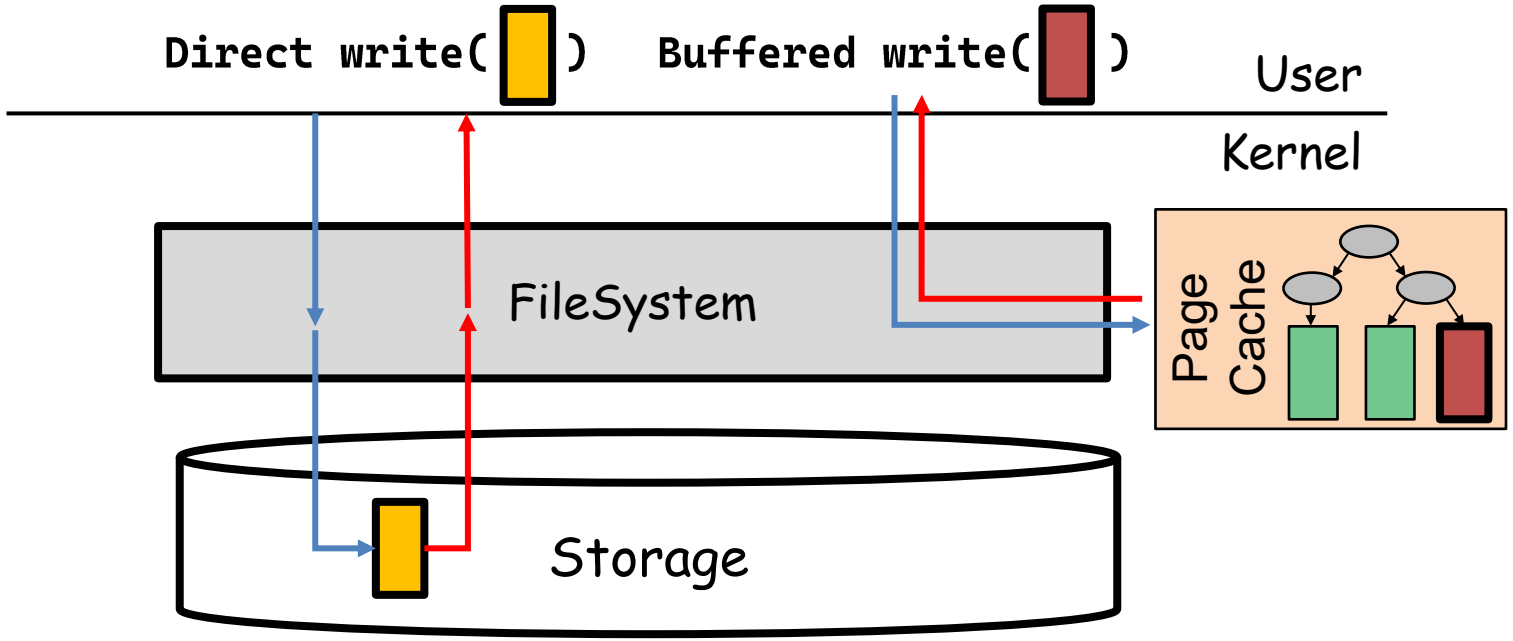


<FAST NVMe SSD (2.6 MIOPS)>

Direct write of LFS

Direct Write?

- A thread issues the IO to the storage, not to the page cache
- Each thread enters the filesystem and does the work for the IO by itself
  - e.g. dnode traversal, LBA mapping, space allocation, waiting for the IO completion (synchronous DIO)
- The storage performance used to dominate the Direct Write performance



Block allocation of LFS (Direct write)

Append only Allocation

- LFS allocates a block by appending the data at the end of the log
- Multiple threads access the same log pointer
- A critical section protects the space allocation from the race condition

Block-at-a-time Allocation

When Thread request write for multiple blocks… (ex. 12 KB for 3 blocks)

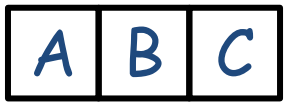
- LFS claims one block per allocation
  - After claiming a block, the thread claims the next one
  - If the new block is not contiguous with the previous one, the thread stops and submits the bio
- To allocate N blocks, a thread repeats the allocation N times



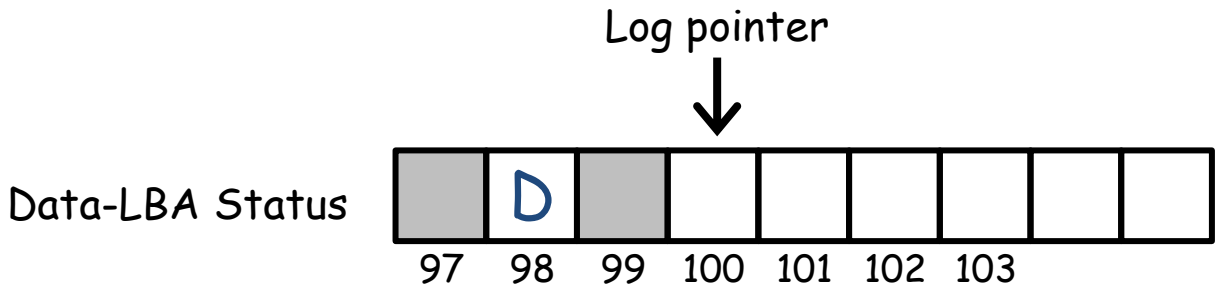
Block allocation of LFS_ example

Write sequence for Thread 1

`write(A, 3)`

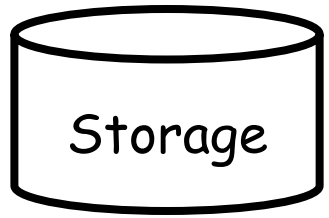


Block allocation (Critical section)



Block Bitmap

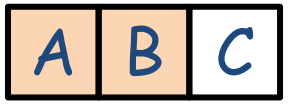
96: 0	100: 0
97: 0	101: 0
98: 1	102: 0
99: 0	103: 0



Block allocation of LFS_ example

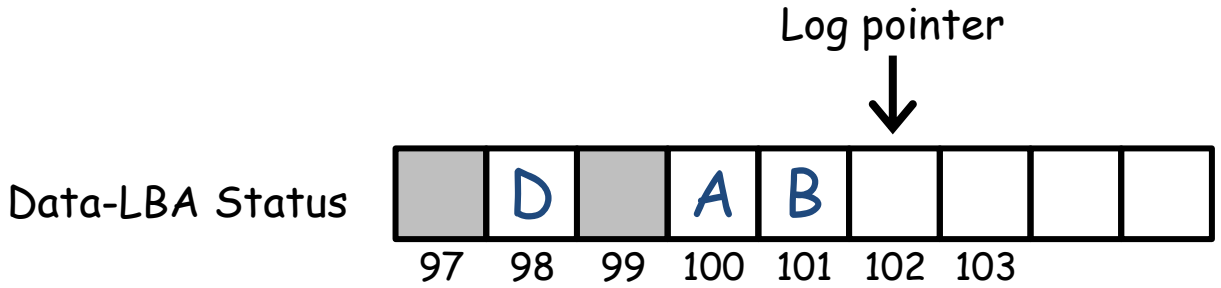
Write sequence for Thread 1

```
write(A, 3)
```



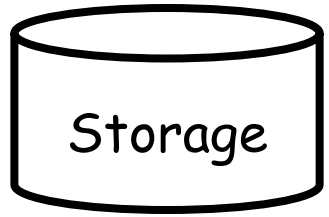
Thread 1

Block allocation (Critical section)



Block Bitmap

96: 0	100: 0 → 1
97: 0	101: 0 → 1
98: 1	102: 0
99: 0	103: 0

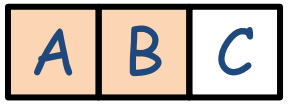




Block allocation of LFS_ example

Write sequence for Thread 1

`write(A, 3)`



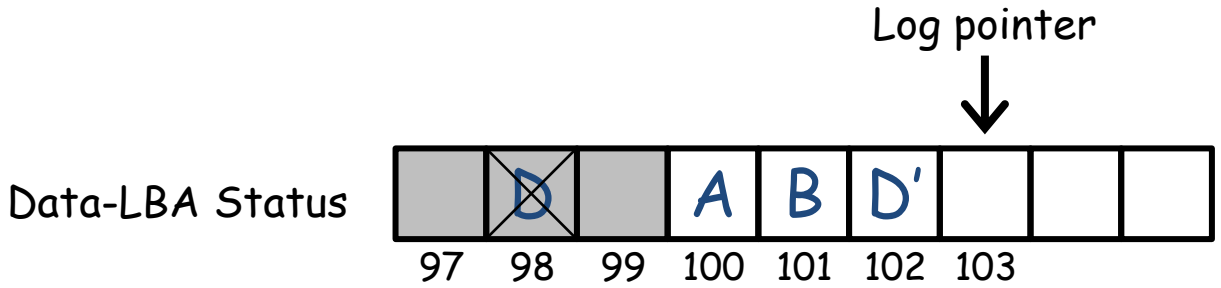
Write sequence for Thread 2

`write(D',1) // D → D'`



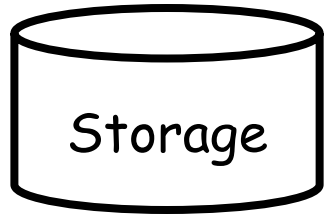
Thread 2

Block allocation (Critical section)



Block Bitmap

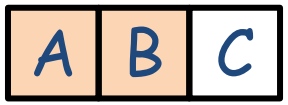
96: 0	100: 1
97: 0	101: 1
98: 1 → 0	102: 0 → 1
99: 0	103: 0



Block allocation of LFS_ example

Write sequence for Thread 1

`write(A, 3)`



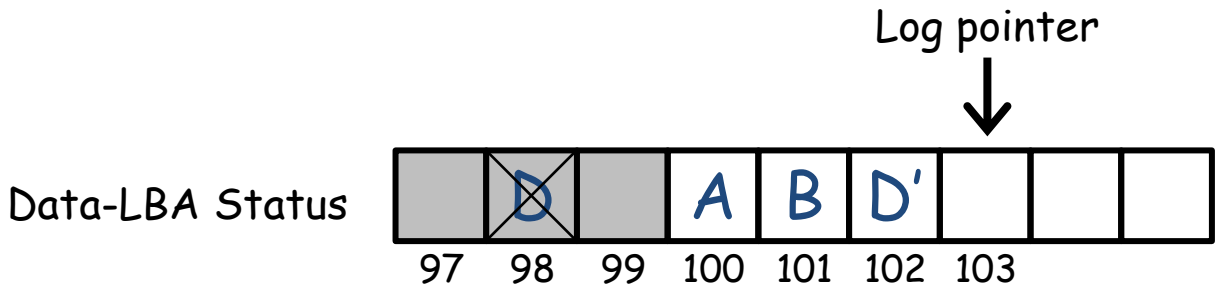
Write sequence for Thread 2

`write(D',1) // D → D'`



Thread 2

Block allocation (Critical section)



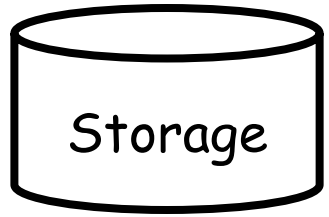
Block Bitmap

96: 0	100: 1
97: 0	101: 1
98: 0	102: 1
99: 0	103: 0

Thread 1 does not wait!

`write(A, 100, 2)`

`write(D', 102, 1)`

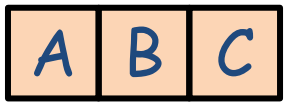




Block allocation of LFS_ example

Write sequence for Thread 1

`write(A, 3)`



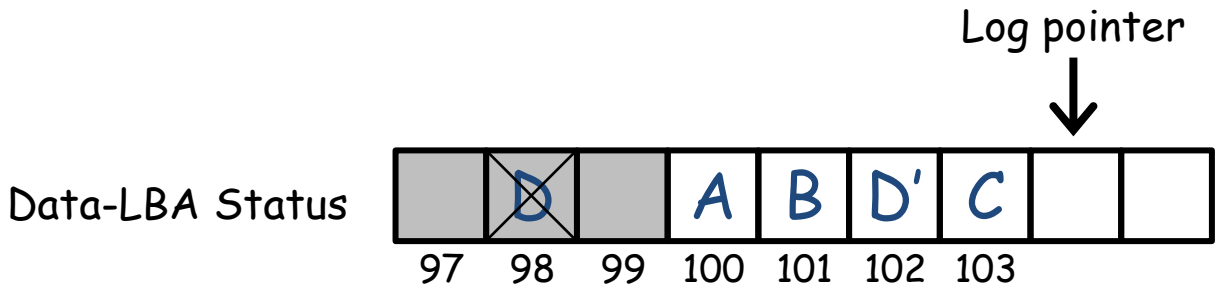
Write sequence for Thread 2

`write(D',1) // D → D'`



Thread 1

Block allocation (Critical section)



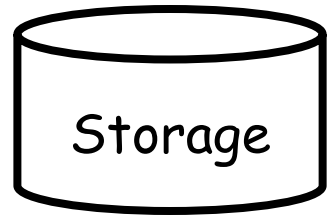
Block Bitmap

96: 0	100: 1
97: 0	101: 1
98: 0	102: 1
99: 0	103: 0 → 1

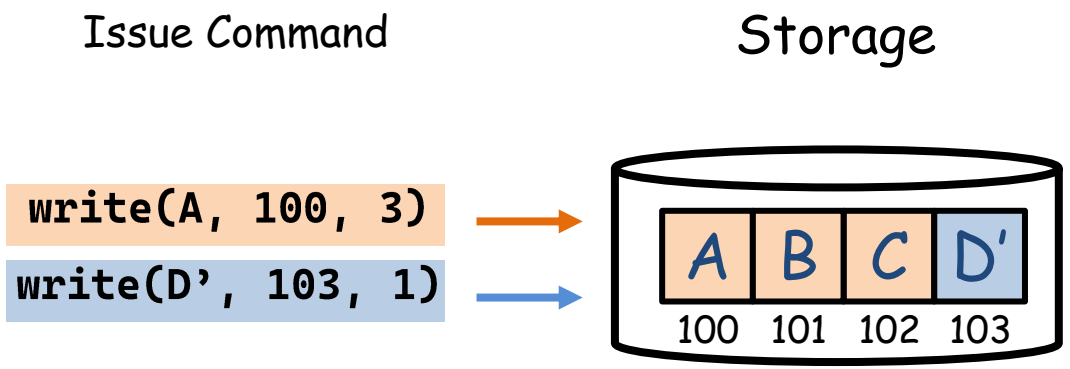
`write(A, 100, 2)`

`write(D', 102, 1)`

`write(C, 103, 1)`

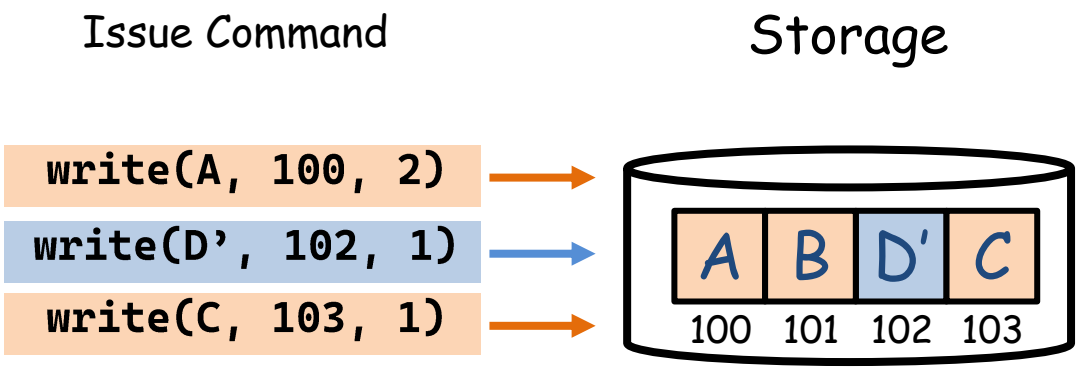


Block allocation of LFS_ example



Critical section entry : 4

<Without Contention>



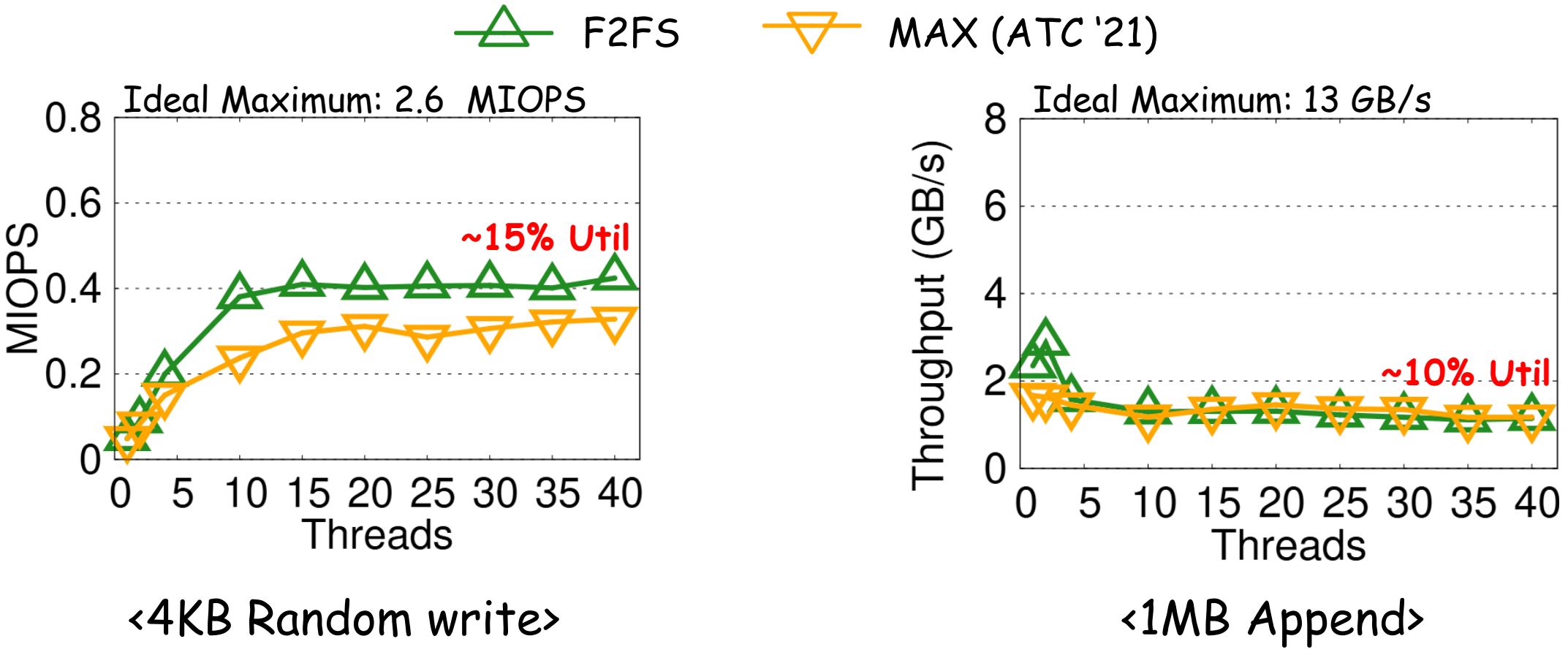
Critical section entry : 4

<With Contention>

Analysis & Motivation



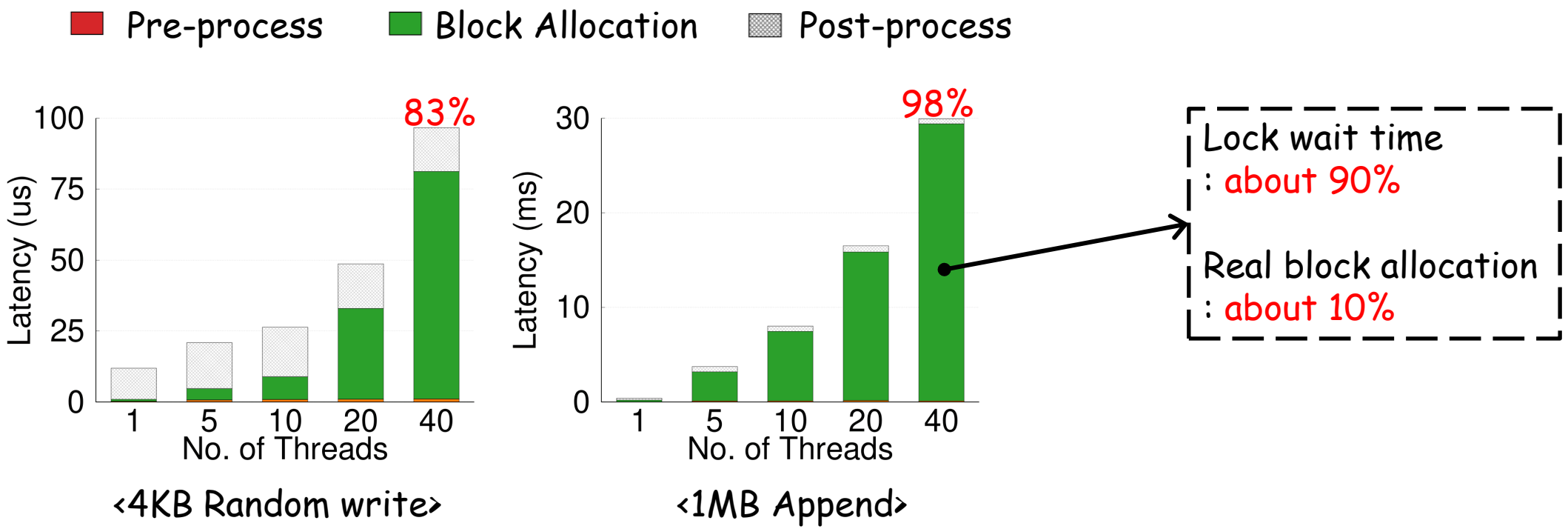
Scalability of throughput



<4KB Random write>

<1MB Append>

Latency breakdown



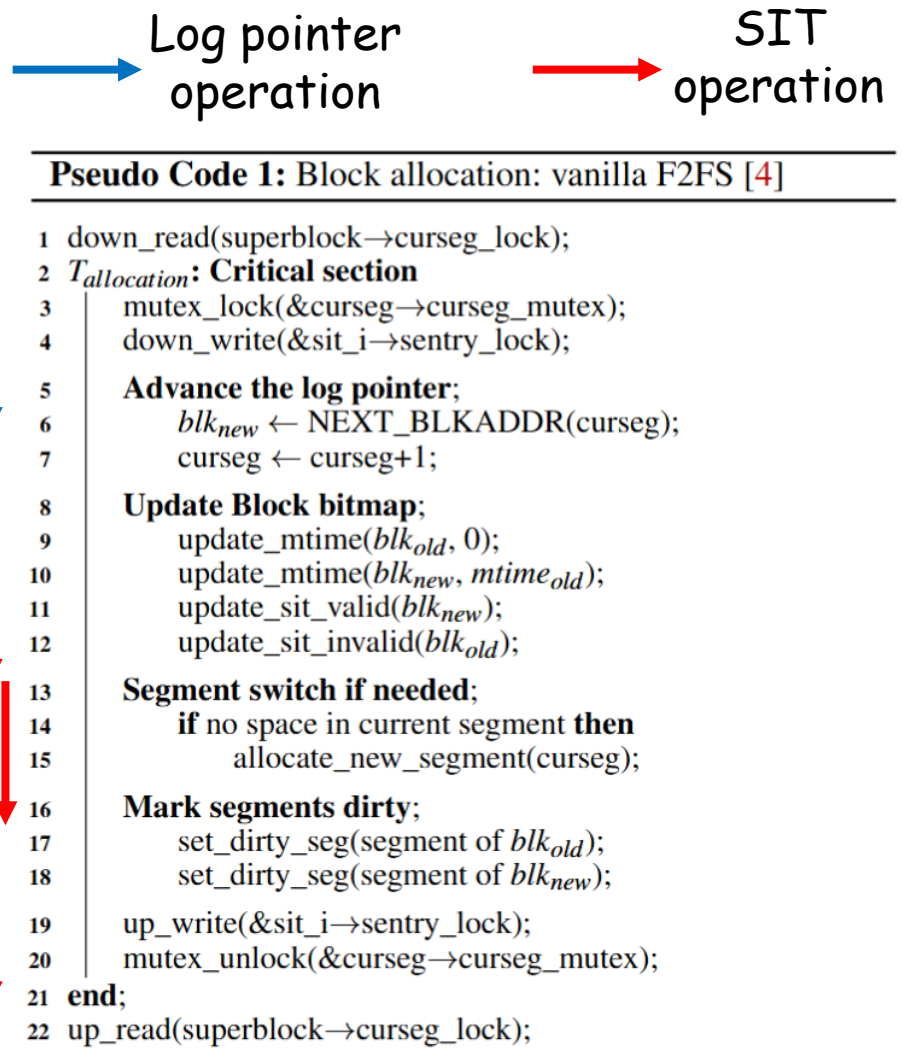
Reason 1. Coarse grained locking

Block allocation transaction protects two data structure

1. Log pointer

2. Segment information table

-> Two transactions are coupled!

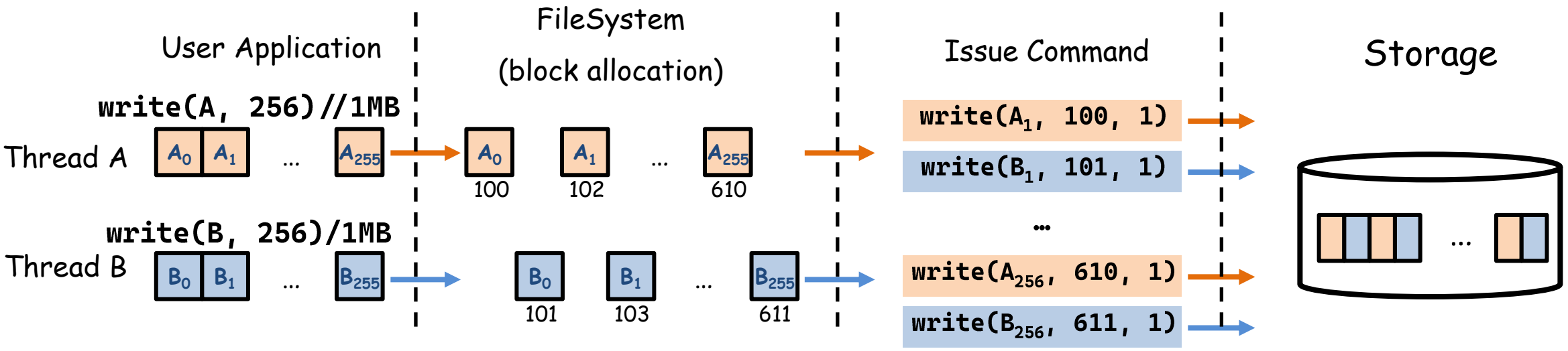


Pseudo Code 1: Block allocation: vanilla F2FS [4]

```
1  down_read(superblock→curseg_lock);
2  T_allocation: Critical section
3      mutex_lock(&curseg→curseg_mutex);
4      down_write(&sit_i→sentry_lock);
5      Advance the log pointer;
6          blk_new ← NEXT_BLKADDR(curseg);
7          curseg ← curseg+1;
8      Update Block bitmap;
9          update_mtime(blk_old, 0);
10         update_mtime(blk_new, mtime_old);
11         update_sit_valid(blk_new);
12         update_sit_invalid(blk_old);
13     Segment switch if needed;
14         if no space in current segment then
15             allocate_new_segment(curseg);
16     Mark segments dirty;
17         set_dirty_seg(segment of blk_old);
18         set_dirty_seg(segment of blk_new);
19     up_write(&sit_i→sentry_lock);
20     mutex_unlock(&curseg→curseg_mutex);
21 end;
22 up_read(superblock→curseg_lock);
```

<Block allocation path in Vanilla F2FS>

Reason 2. Fragmented logging



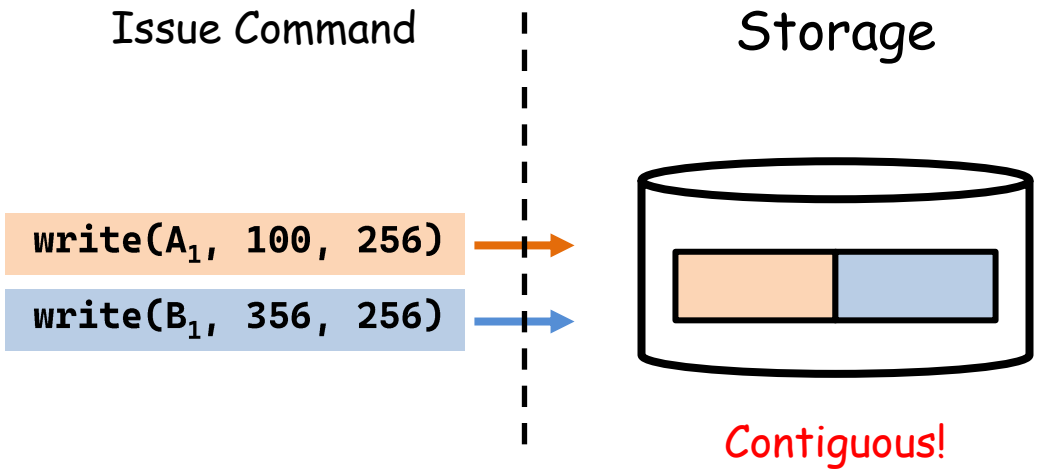
Contiguous user data → Fragmented!

Reason 2. Fragmented logging

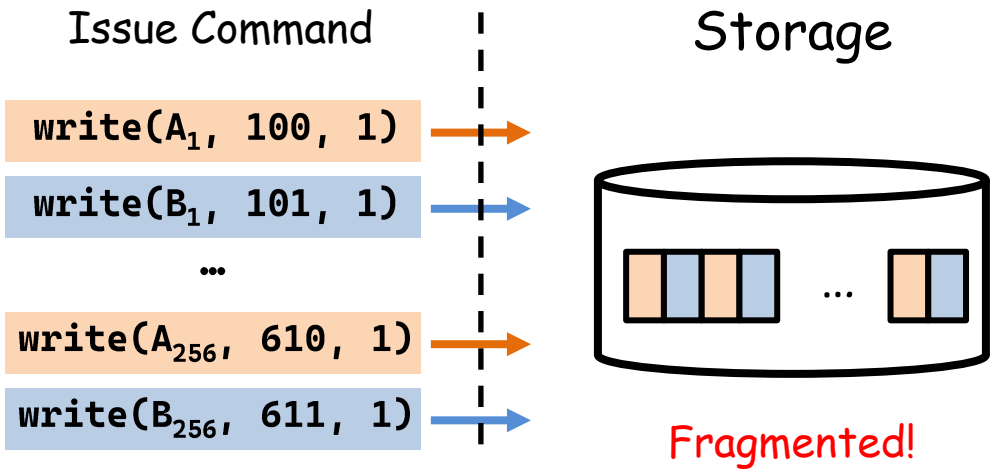
Single Large request -> small many request

Increase entry of critical section

More extents



<W/o Contention>



<W/ Contention>

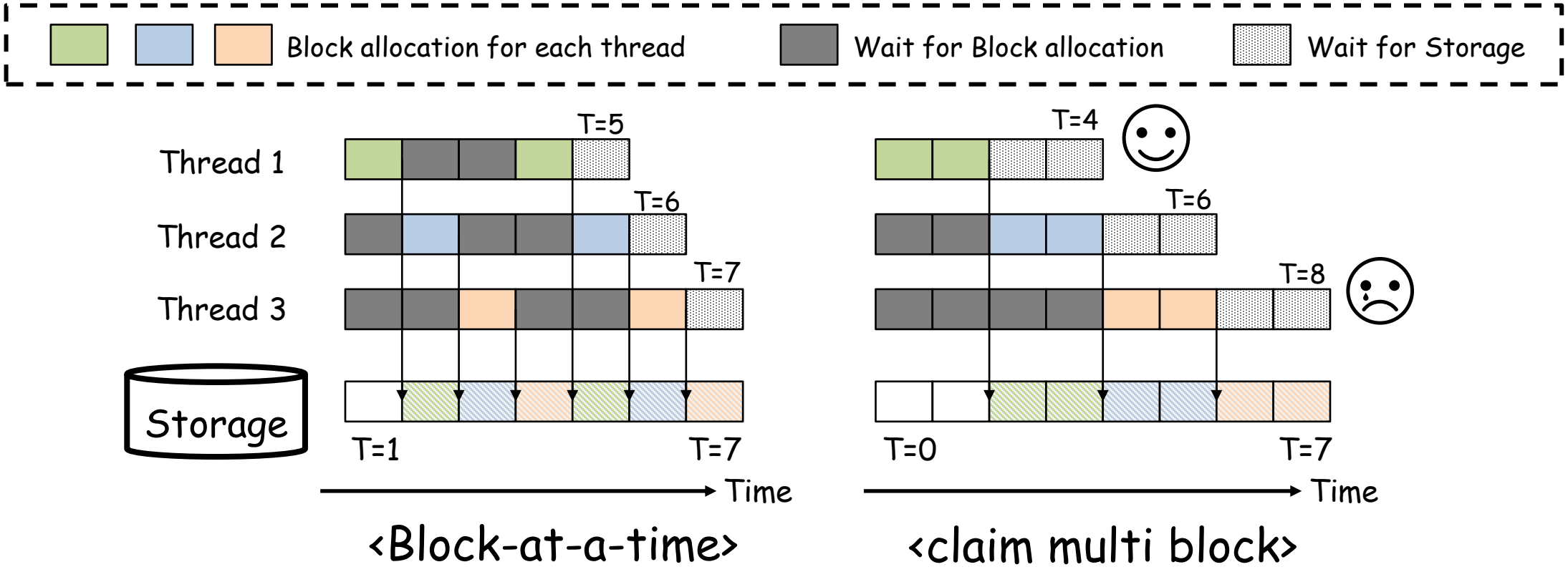
2 thread 1MB write -> 512 bio issue!

Is Block-at-a-time allocation always bad?

Block-at-a-time is a **fine-grained allocation!**

Pros: fairness, no starvation, lower tail latency

Cons: fragmented logging (more extents), more critical section entries



Design

Goal

Goal 1. Decrease critical section

Goal 2. Avoid block-at-a-time allocation

- Goal 2.1 : Allocate N blocks in O(1) manner, not O(N)
- Goal 2.2 : Allocate N blocks in single critical section



Overview

ScaLog : logging method to better scalability of log-structured filesystem

<3 Design>

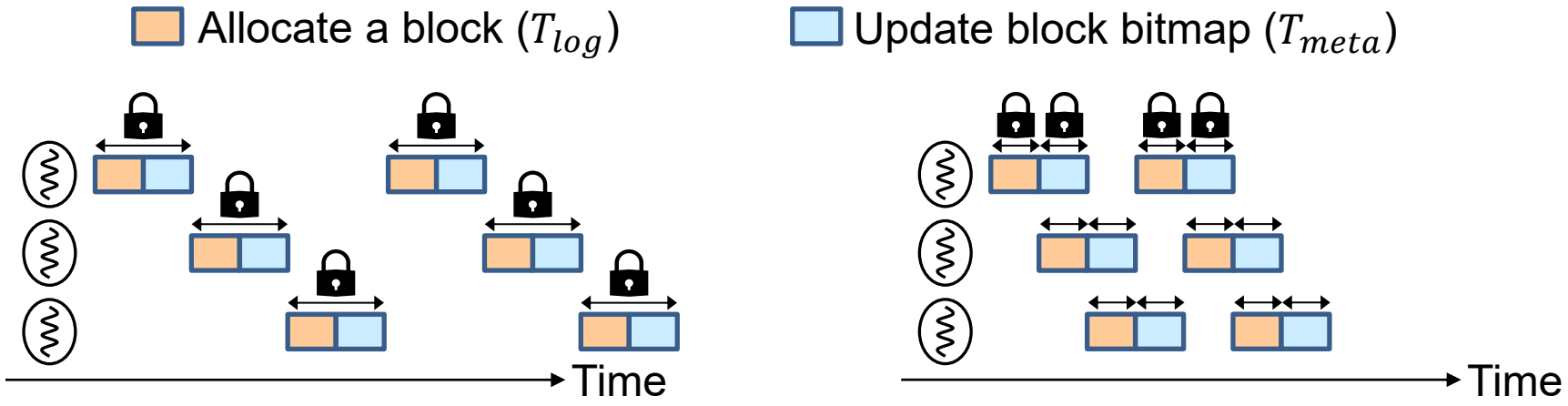
1. Split Transaction → Goal 1. Decrease critical section
2. Per segment range list → Goal 2.1 : Allocate N blocks in O(1) manner, not O(N)
3. Multi-block allocation → Goal 2.2 : Allocate N blocks in single critical section

Design 1. Split transaction

The log pointer advance and the bitmap update are tightly coupled

- Split them into two transactions: T_{log}, T_{meta}
- One thread updates the bitmap while another advances the log pointer

The two locks are held together only when the log needs a new segment



Design 2. Per segment Range logging

Even after Split Transaction, the block bitmap still has inefficiencies.

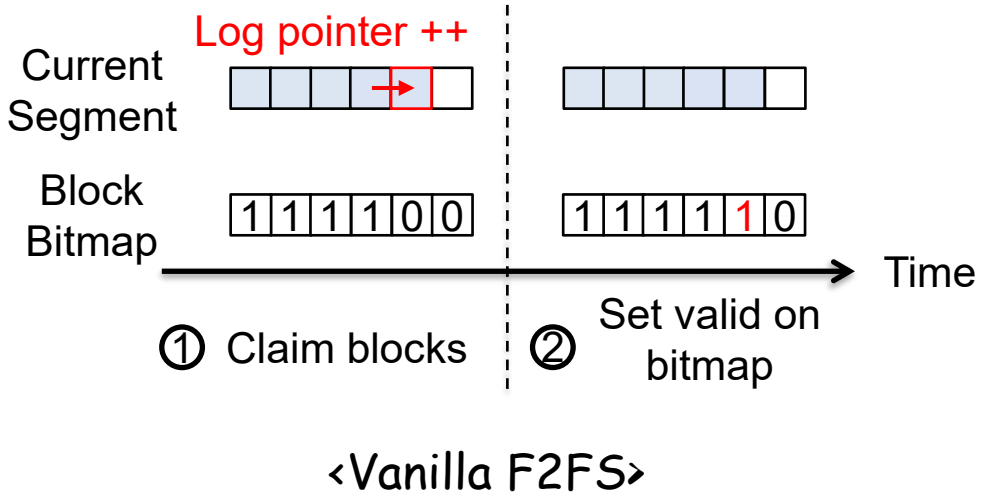
1. Validation and invalidation on different segments have no physical contention, yet the same transaction protects them.
2. Allocating N blocks in a single critical section still costs O(N) bitmap operations.



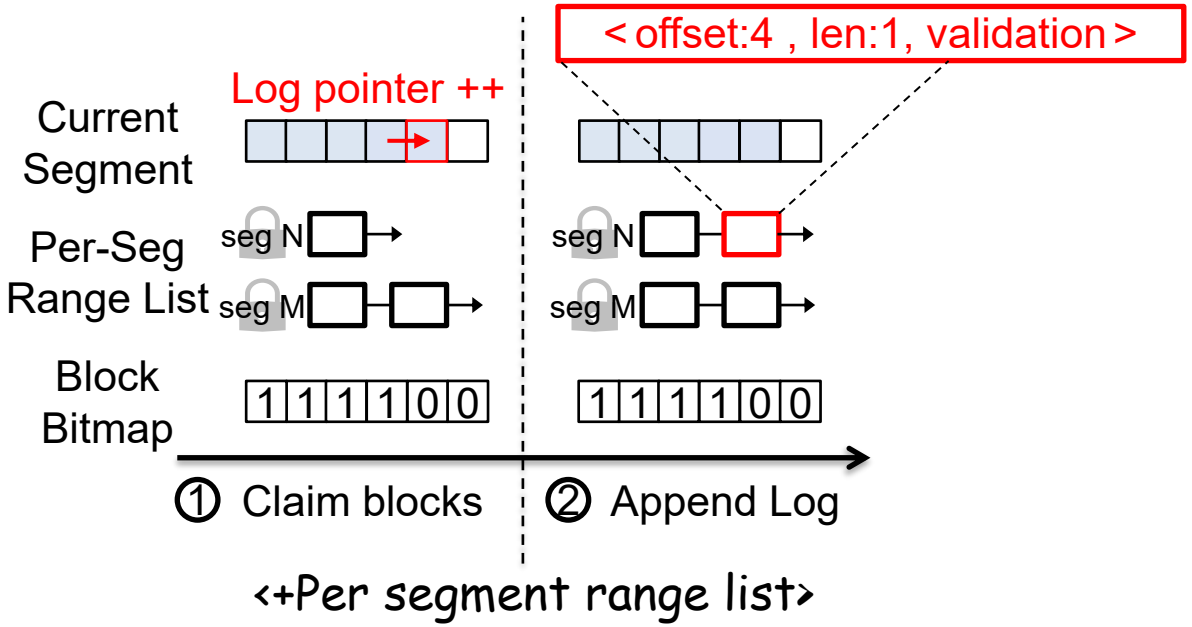
Design 2. Per segment Range logging

Instead of updating the bitmap directly, ScaLog logs the change as a range.

- ScaLog keeps a range list per segment
- The updates to different segments proceed in parallel



<Vanilla F2FS>

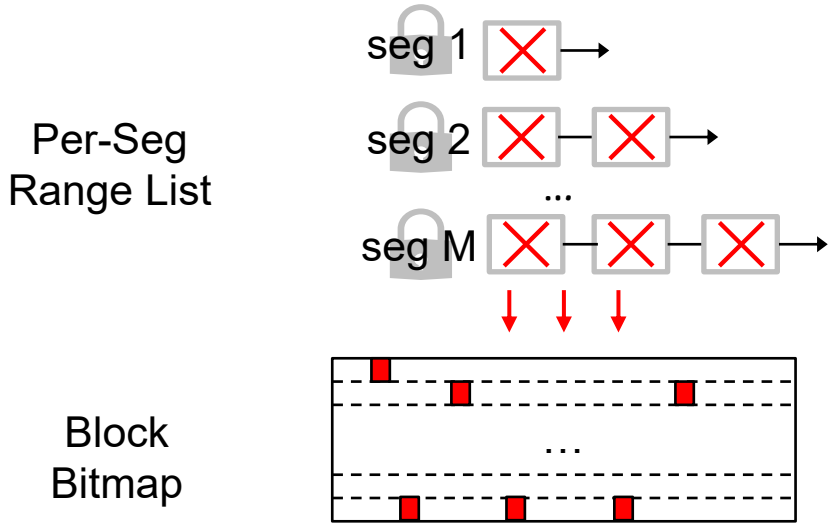


<+Per segment range list>

Design 2. Commit

We need to commit range list to global bitmap!

- Validation sets the bits to 1, and invalidation sets them to 0
- At checkpoint, the global bitmap must be up to date.
- In other words, the checkpoint thread traverses every range list and applies all the changes to the block bitmap.

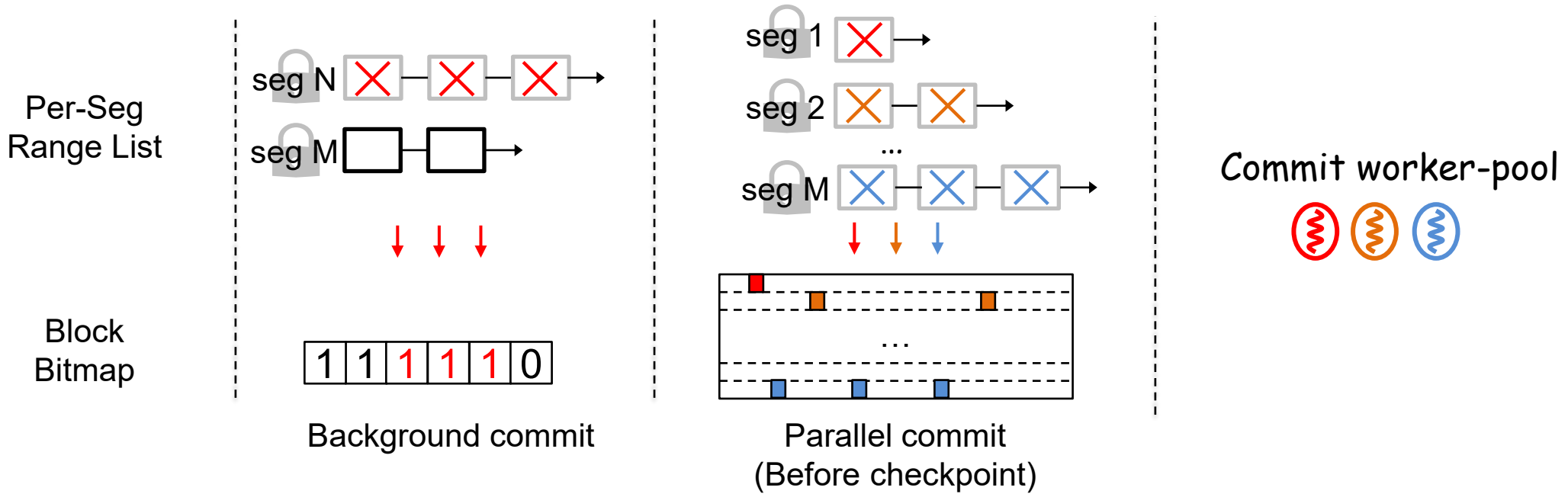


Parallel commit
(Before checkpoint)

Design 2. Commit

If the checkpoint traverses many lists, the checkpoint overhead can increase!

- Background commit
  - When a list grows beyond a threshold, commit the segment in the background
- Parallel commit
  - Before checkpoint, commit all the range lists with multiple threads



Design 3. Multi block allocation

Claim multiple blocks in a single critical section

- Fewer critical section entries
- Less fragmented logging
- Fewer extents per file

How many blocks in a single critical section?

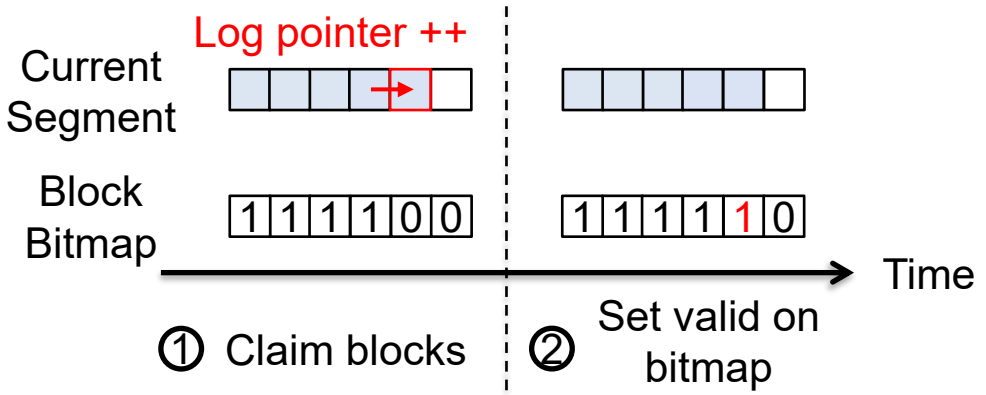
$$
\begin{aligned}
&N_{req} \leftarrow \#\ of\ block\ for\ the\ write \\
&while\ (N_{req} > 0)\{ \\
&\quad start\ critical\ section \\
&\quad\quad N \leftarrow \min(N_{req}, \delta_{node}, \delta_{seg}) \\
&\quad\quad allocate\ N\ block \\
&\quad\quad N_{req} \leftarrow N_{req} - N \\
&\quad end\ critical\ section \\
&\}
\end{aligned}
$$

- δ_{node}: the maximum number of blocks that the current data node can still map
- δ_{seg} : the number of free blocks left in the current segment

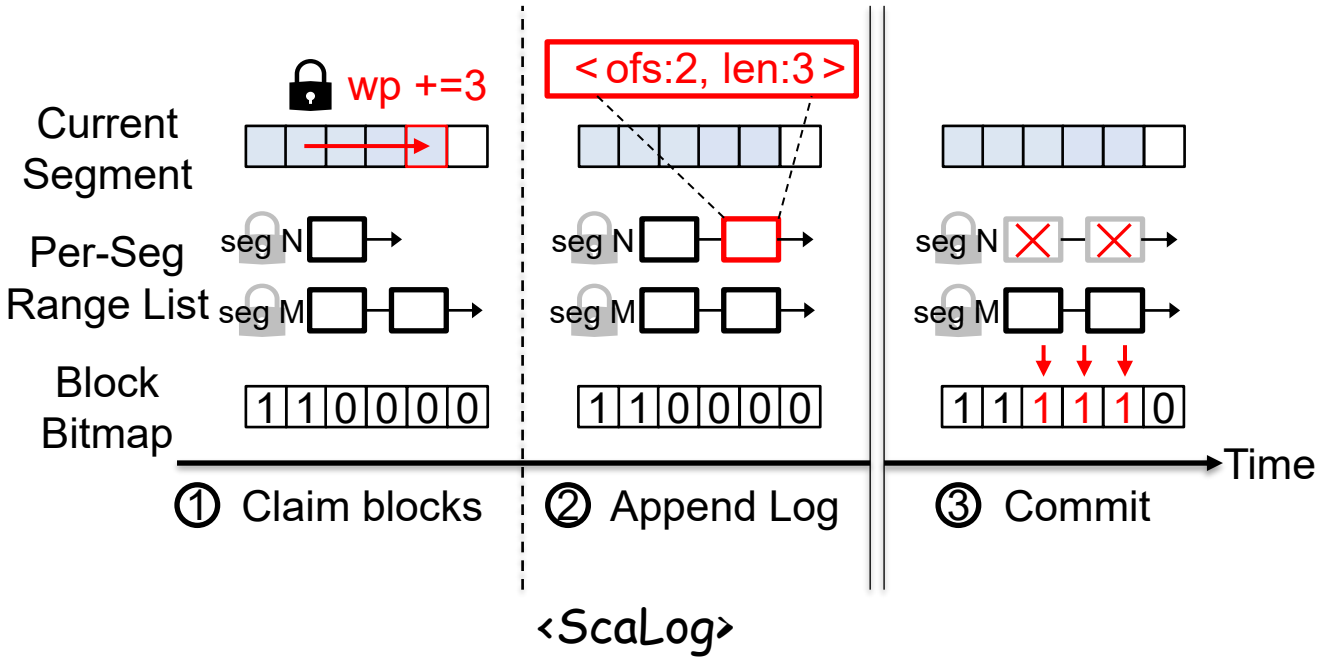


Design 3. Multi block allocation

Scenario: allocating 3 blocks, no overwrite



<Vanilla F2FS>



<ScaLog>

Overall design

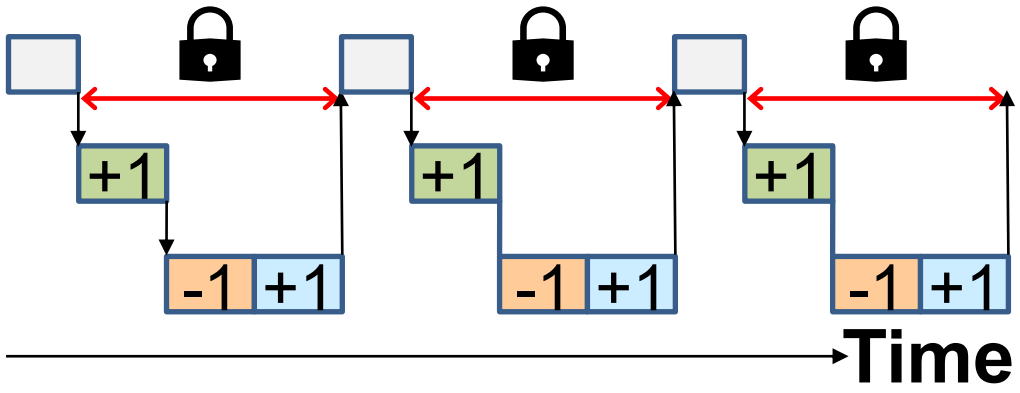
pre-process

-1 Invalidate an old block

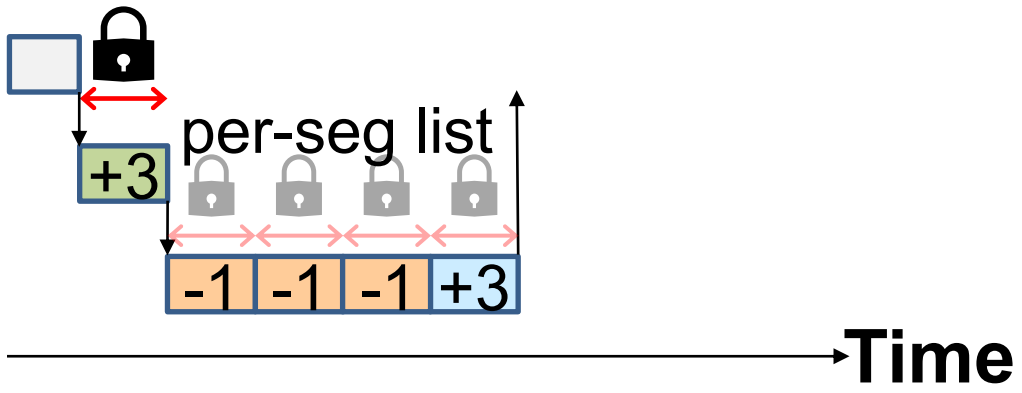
+n Allocate n blocks (log-pointer)

+n Validate n new block

⟷ Critical section



<Vanilla F2FS>



<ScaLog>

Evaluation



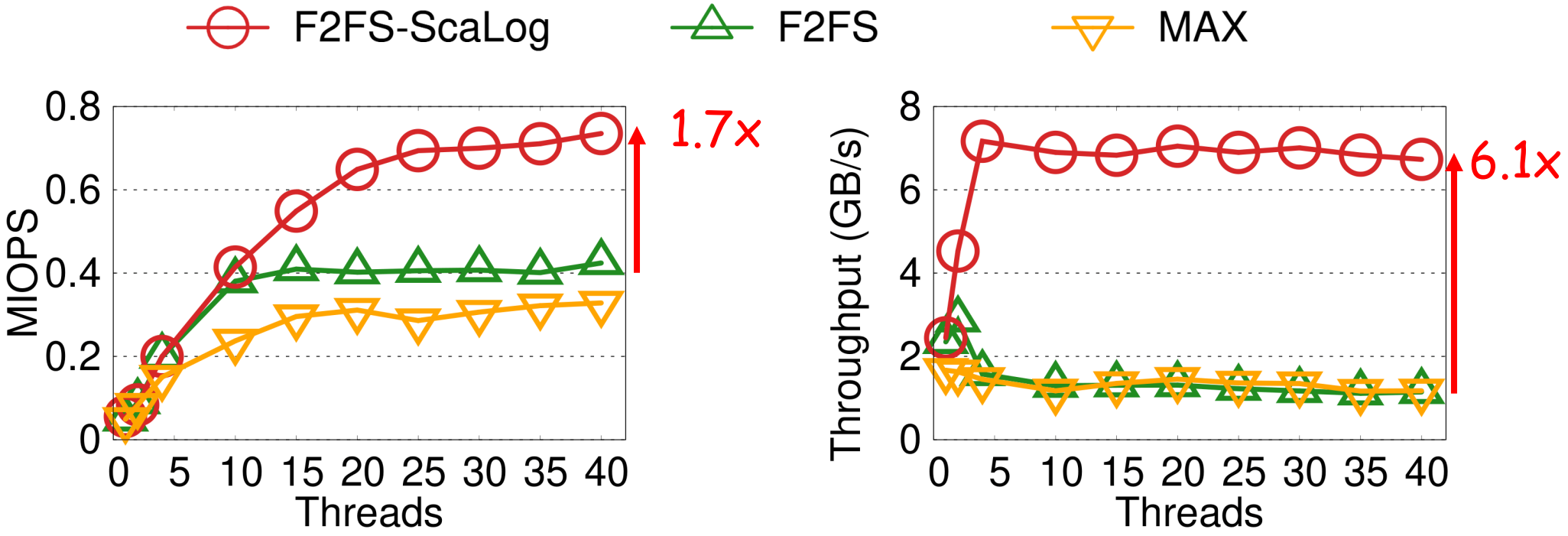
Evaluation

Hardware setup

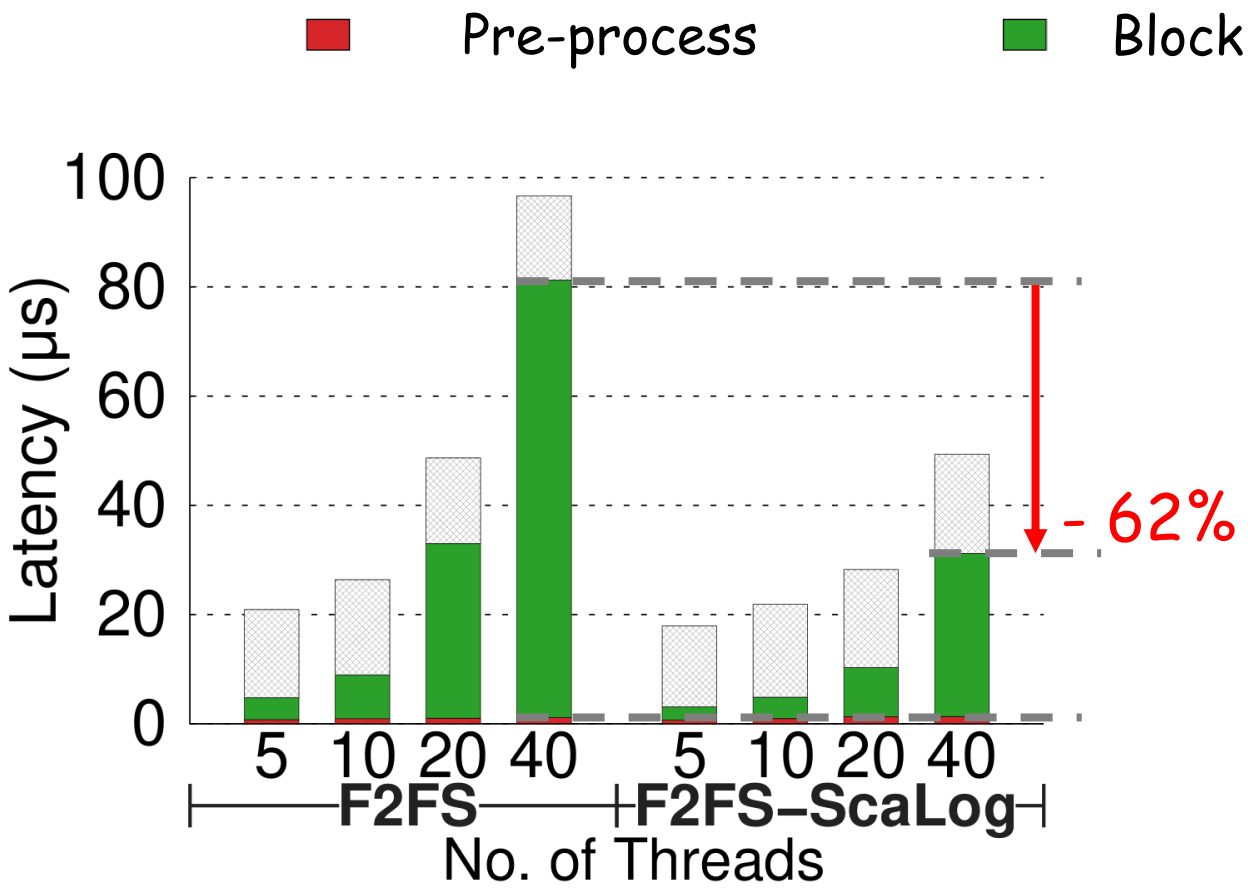
- CPU: 40-core Intel
- Memory: 128 GB DDR5
- Storage: Samsung 9100 PRO NVMe SSD (PCIe 5.0, 2.6 MIOPS)
- Kernel: Linux 6.1.0
- Workload: FIO, psync engine, 4KB random write, 1MB append, O_DIRECT
- FileSystem
  - F2FS-ScaLog
  - F2FS
  - MAX (ATC '21)
- Each thread writes to its own file - Partition filled to at most 60% (no foreground GC)

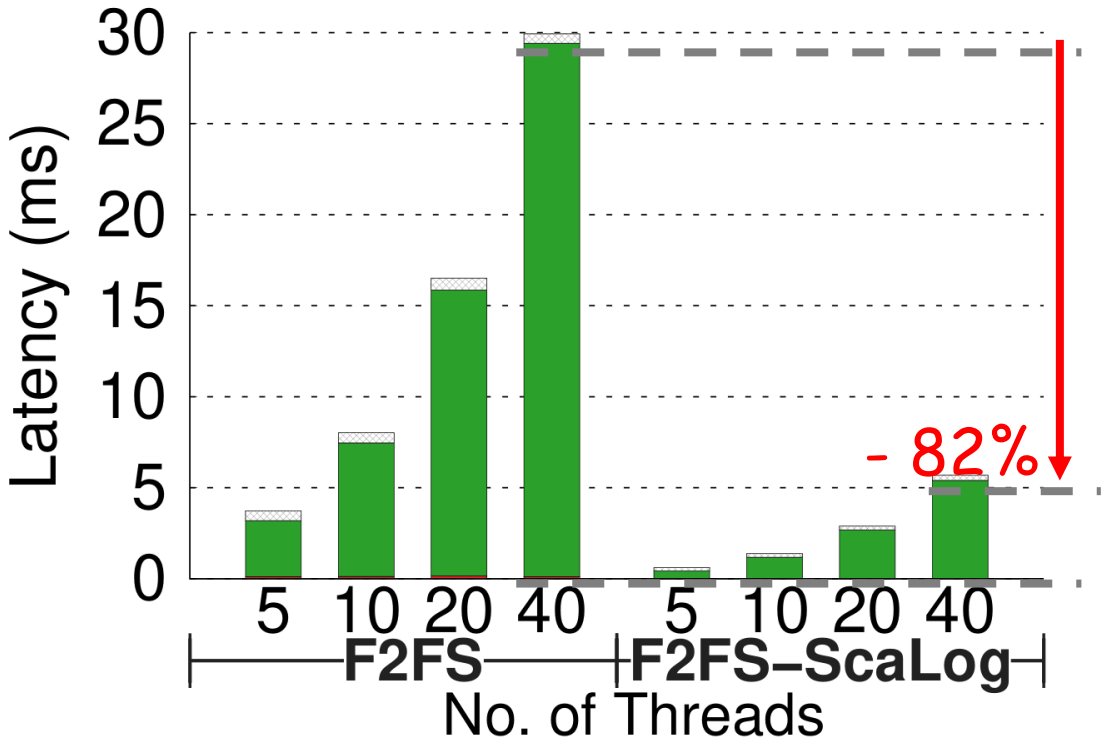


FIO throughput

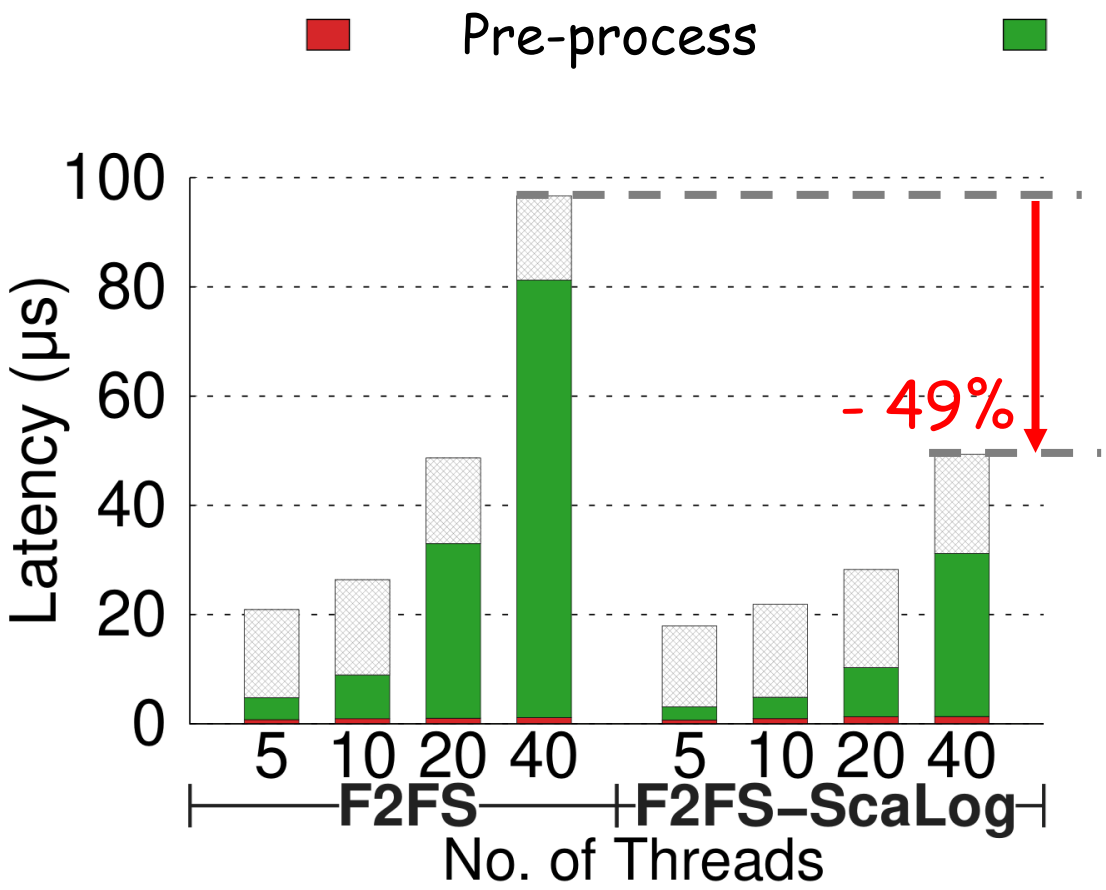


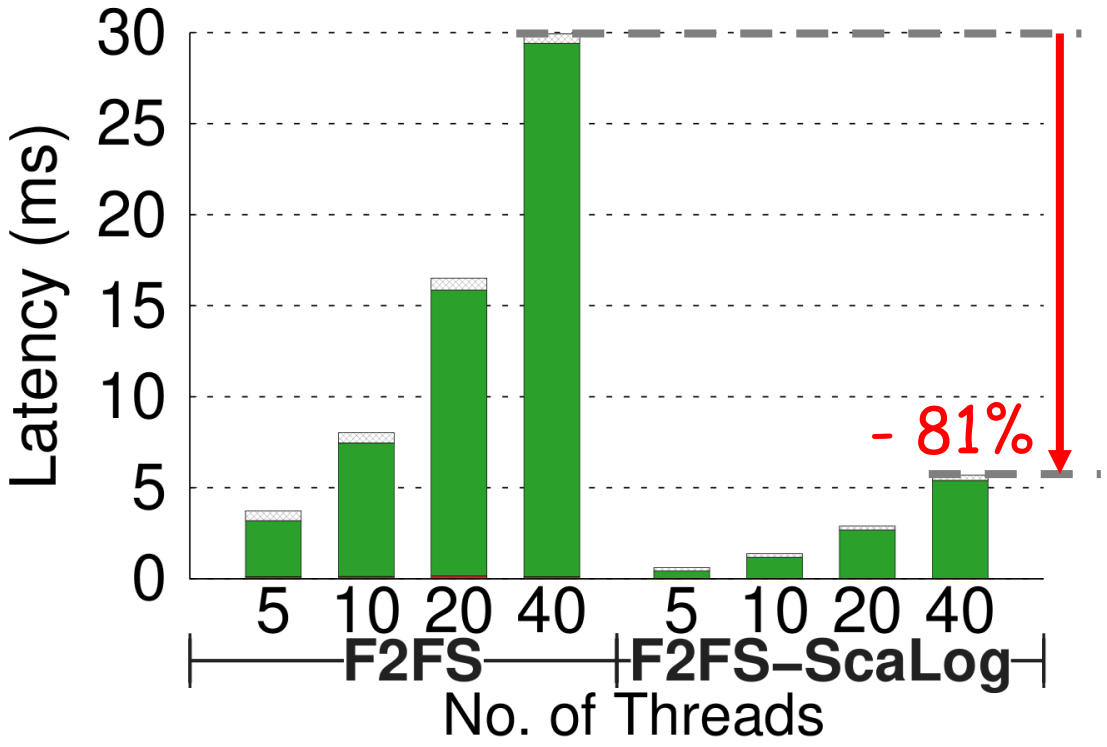
Latency Breakdown





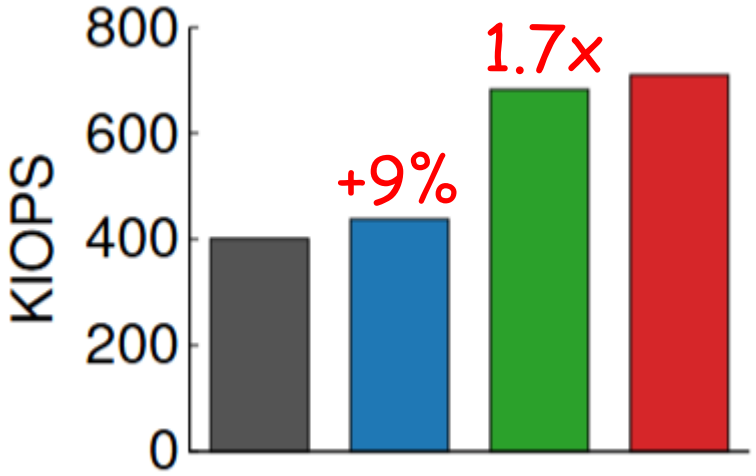
Latency Breakdown

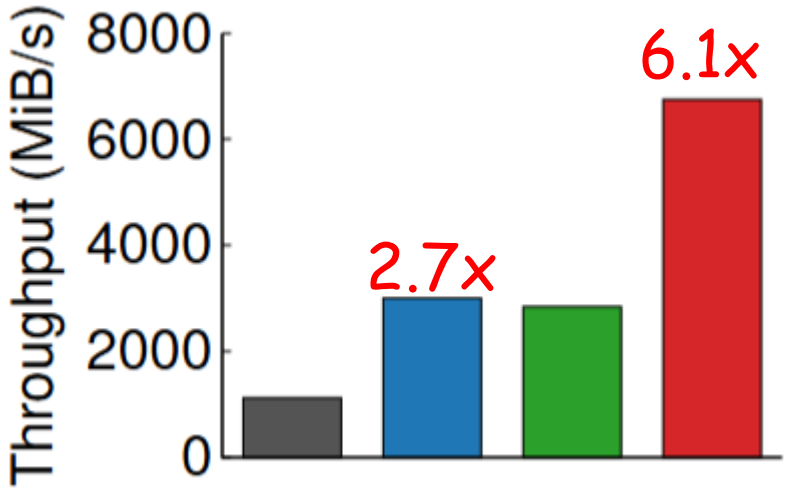




Ablation study

F2FS + Split Transaction + Per segment Range List + Multi block allocation





Fragmented logging

Number of IOs and lock requests, 1 MB Append

	# of Threads	1	2	4	6	8	10
F2FS	# of IOs	1	118	252	238	221	229
	# of Lock Req	256	256	256	256	256	256
F2FS-ScaLog	# of IOs	1.3	1.3	1.3	1.5	1.6	1.6
	# of Lock Req	1.3	1.3	1.3	1.5	1.6	1.6



Conclusion

Problem: LFS hits a CPU bottleneck under direct write on fast SSDs

- Coarse-grained locking
- Fragmented logging

ScaLog: Split Transaction, Per-segment Range List, Multi-block Allocation

- 4 KB random write: 1.7× throughput
- 1 MB append: 6.1× throughput

(PCIe 5.0 SSD, 40 threads)

Future work

- Impact of fragmented logging on the read performance
- Macro benchmarks on real applications
- Applying ScaLog to the buffered write



Thank you!

