SSDFS: flash-friendly file system with highly optimized GC activity, diff-on-write, and deduplication

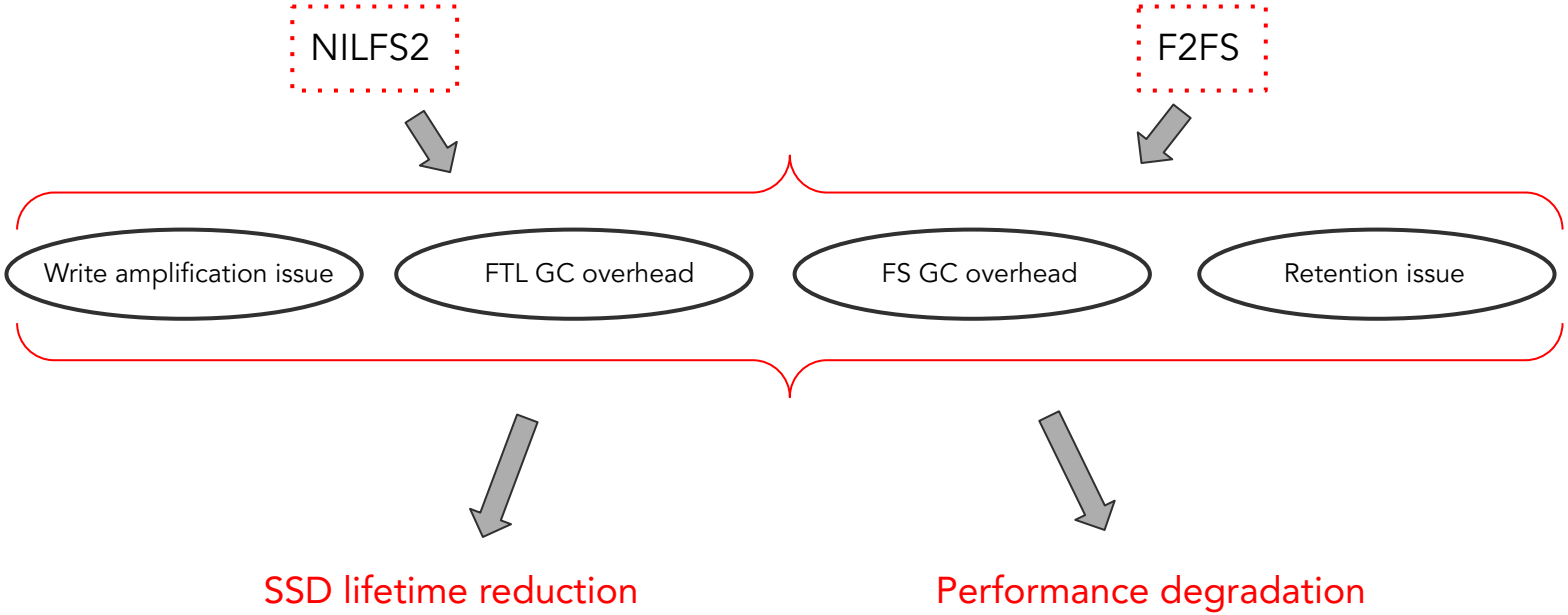
Viacheslav Dubeyko (STE team)
viacheslav.dubeyko@bytedance.com

ByteDance

Content

1. Problem
2. Design goals
3. Testing methodology
4. Benchmarking results
5. Future work
6. Conclusion

Problem



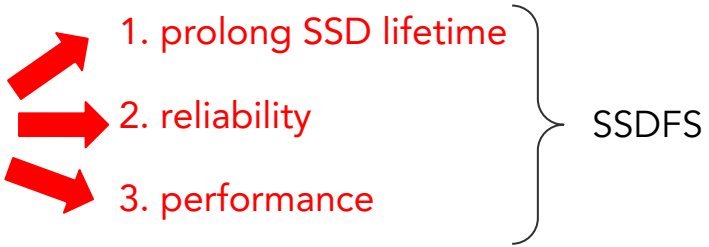
Why yet another file system?

NILFS2 → reliability
- in-place update superblocks
- COW policy (LFS)
- user-space GC
- snapshots

F2FS → performance
- in-place update metadata area
- COW area
- kernel-space GC
- dual checkpoints
- transparent file compression
- file system level encryption

bcachefs → reliability + performance
- Copy on write (COW) - like zfs or btrfs
- COW b-trees + journal
- Copying garbage collection
- Full data and metadata checksumming
- compression
- Multiple devices
- Replication + Erasure coding
- encryption
- snapshots

SSDFS
- Pure LFS (COW policy) + ZNS SSD ready
- compression + delta-encoding + compaction scheme
- migration scheme + migration stimulation + noGC overhead
- deduplication (not fully implemented)
- post-deduplication delta-compression (planned)
- prolong SSD lifetime
- snapshots (not fully implemented)
- recoverfs (reconstruct file system state -> heavily corrupted volume)
- employ parallelism of multiple NAND dies



ByteDance

SSDFS design goals

SSDFS is flash-friendly and ZNS compatible open-source kernel-space file system:

1

Prolong SSD lifetime

Decrease write amplification
- Compression
- Compaction scheme
- Delta-encoding technique
- Deduplication technique
- Post-deduplication delta-compression

Exclude GC overhead
- Exclude FTL GC responsibility
- Minimize FS GC activity

Decrease retention issue
- Smart management of “cold” data
- Efficient TRIM policy

2

Strong reliability

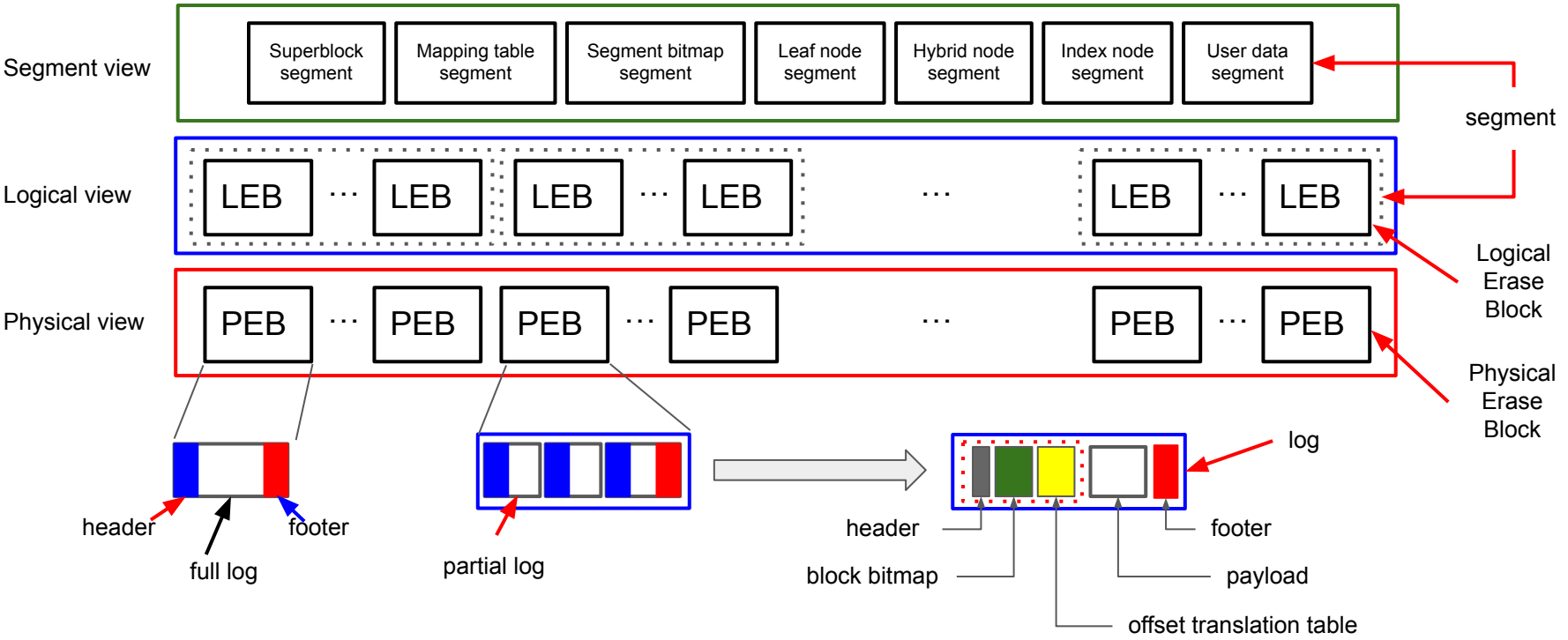
- Checksumming support
- Metadata replication
- Snapshots support
- Erasure coding support
- Reconstruct corrupted file system

3

Stable file system performance

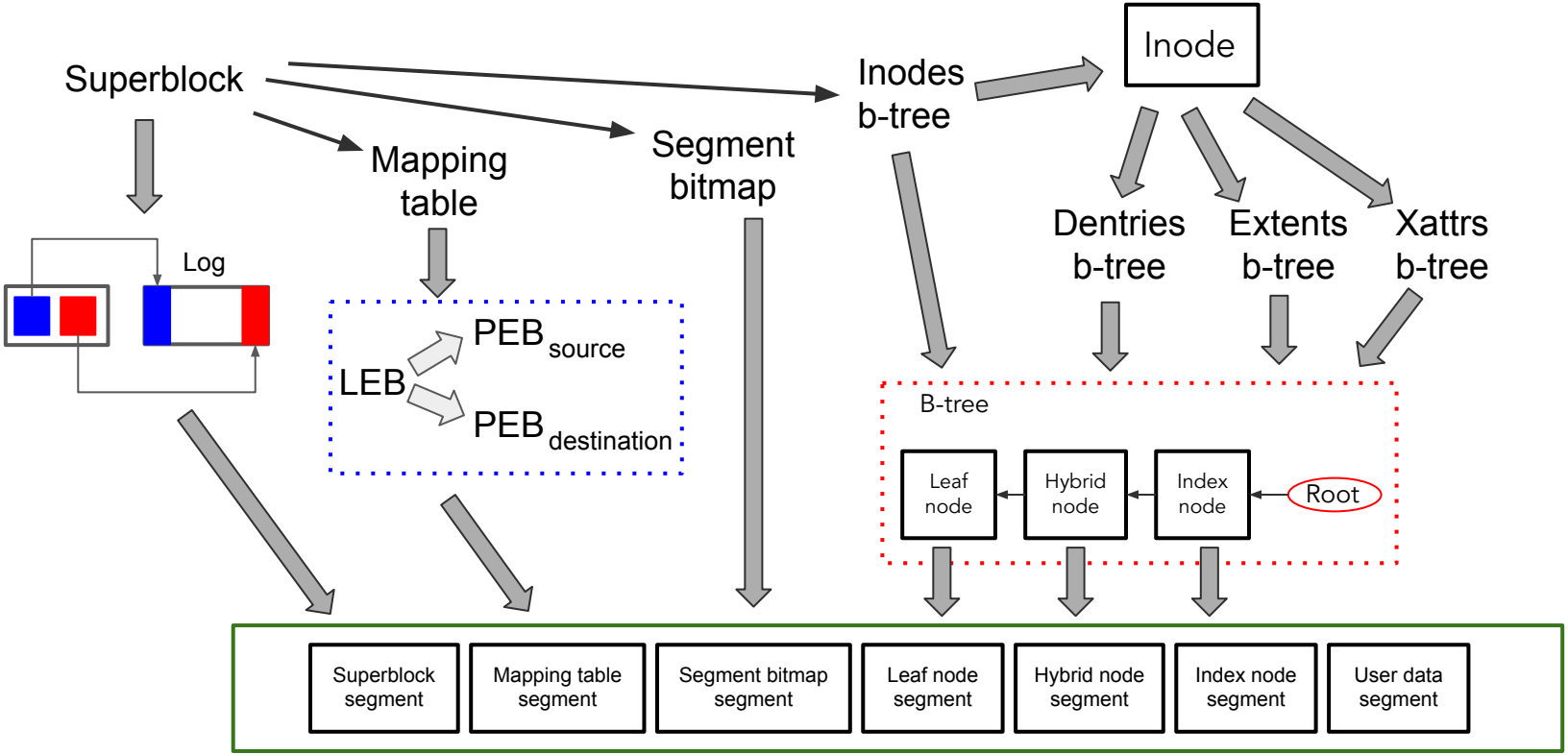
- Employ parallelism of multiple NAND dies
- Multiple PEBs in segment
- ZeroGC overhead
- Minimized write amplification
- B-trees in metadata
- Efficient TRIM policy

SSDFS architecture (logical vs. physical view)



ByteDance

SSDFS architecture (metadata)



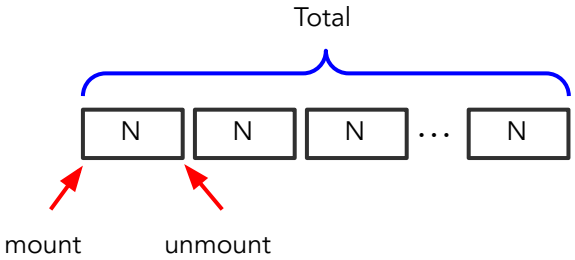
ByteDance

Testing use-case(s)

Metadata	User data	
Create empty file	Create file	64 bytes
		16KB
		100KB
Update empty file	Update file	64 bytes
		16KB
		100KB
Delete empty file	Delete file	64 bytes
		16KB
		100KB

SSDFS	
Erase block size	128KB
	512KB
	8MB

N	Total
10	1000
10	10000
100	1000
100	10000
1000	1000
1000	10000



Testing sequence:
- format partition (mkfs - default settings)
- blktrace <partition>
- while (iterations < (Total/N)) {
  mount();
  while (items < N) {
    execute_use_case();
  }
  unmount();
 }
- stop blktrace

Currently, available results for "Create empty file" use-case only.

Methodology

$$\text{Lifetime} = \frac{\text{Erase}_{\text{limit}}}{\text{Erase}_{\text{total}}}$$

$$\text{Erase}_{\text{limit}} = \text{Capacity}_{\text{EB}} * \text{Erase Block}_{\text{limit}}$$

$$\text{Erase}_{\text{total}} = \text{Erase}_{\text{FTL GC}} + \text{Erase}_{\text{TRIM}} + \text{Erase}_{\text{FS GC}} + \text{Erase}_{\text{read disturbance}} + \text{Erase}_{\text{retention}}$$

$$\text{Erase}_{\text{FTL GC}} = \text{Write}^{\text{I/O}}_{\text{EB}} - \text{Payload}_{\text{EB}}$$

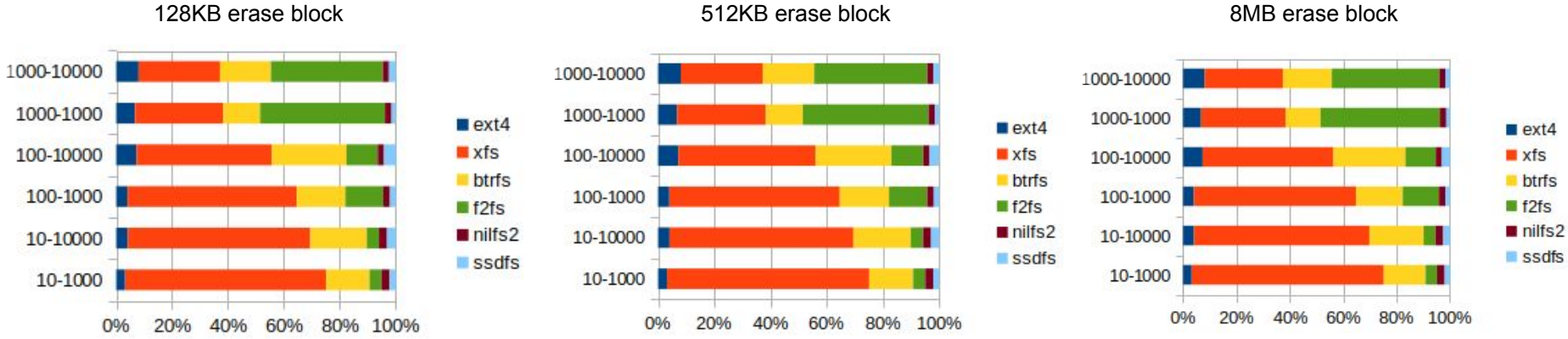
$$\text{Payload}_{\text{EB}} = \text{Erase Block}_{\text{unique}} - \text{TRIM}_{\text{EB}}$$

$$\text{Erase}_{\text{FS GC}} = \text{Payload}_{\text{EB}} - \text{Valid Data}_{\text{EB}}$$

$$\text{Erase}_{\text{read disturbance}} = \frac{\text{Read}^{\text{I/O}}_{\text{EB}}}{\text{Threshold}_{\text{disturbance}}}$$

$$\text{Erase}_{\text{retention}} = \frac{\text{Time}_{\text{use-case}}}{3 \text{ months}} * \text{Payload}_{\text{EB}}$$

Write I/O (erase blocks)



SSDFS generates smaller amount of write I/O requests:

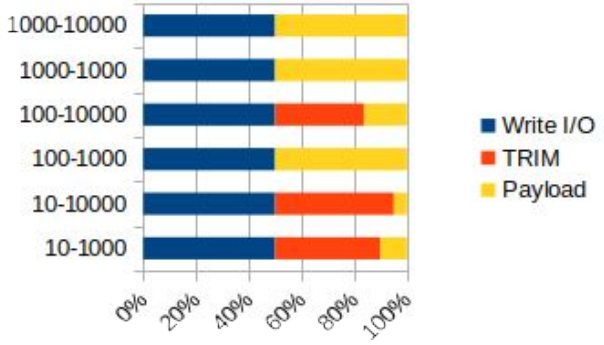
- 1.3x – 4.8x compared with ext4
- 11.4x - 36x compared with xfs
- 6.3x - 10.4x compared with btrfs
- 1.3x – 32x compared with f2fs

NILFS2 competes with SSDFS. However, SSDFS can be 1.1x - 1.6x more efficient for real-life use-cases. Moreover, SSDFS testing took place without using delta-encoding and deduplication (there is room for improvement).

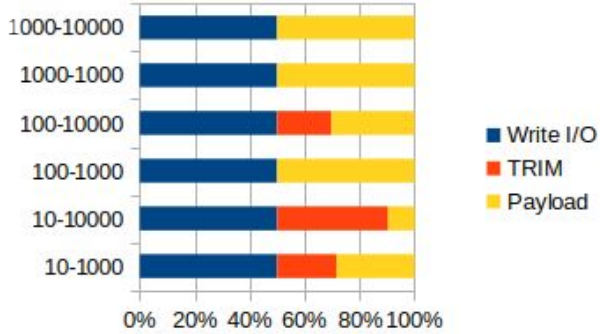
ByteDance

TRIM (erase blocks)

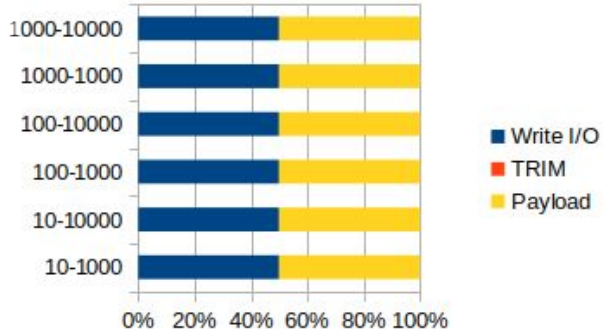
128KB erase block



512KB erase block



8MB erase block



128KB	Write I/O	TRIM	Payload
10-1000	51.375	41	10.375
10-10000	812.25	733	79.25
100-10000	153.40625	104	49.40625

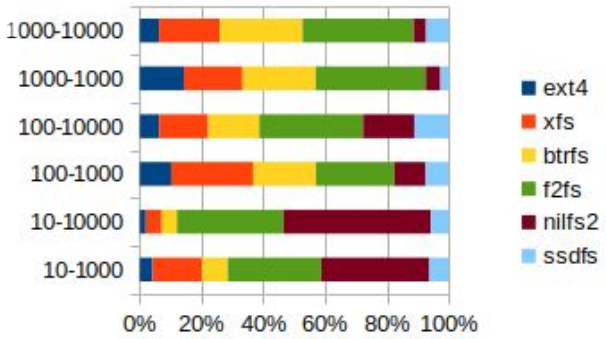
512KB	Write I/O	TRIM	Payload
10-1000	11.59375	5	6.59375
10-10000	180.296875	146	34.296875
100-10000	30.25	12	18.25

SSDFS introduces highly efficient TRIM policy that: (1) eliminate FTL GC activity, (2) decrease retention issue. Migration scheme builds the TRIM efficiency and eliminates the necessity of FS GC activity. Even multiple mount/unmount operations cannot affect the efficiency of TRIM policy.

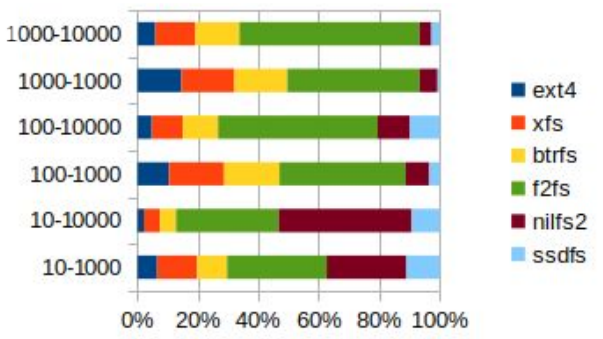
ByteDance

Payload (erase blocks)

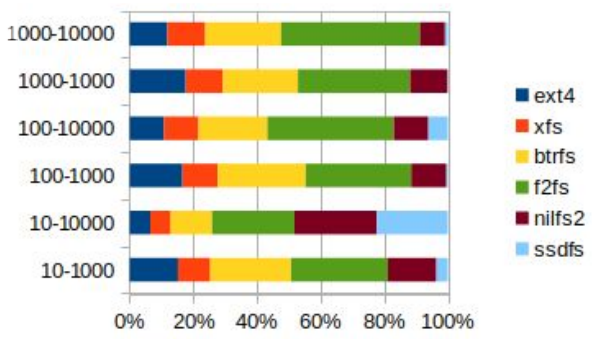
128KB erase block



512KB erase block



8MB erase block

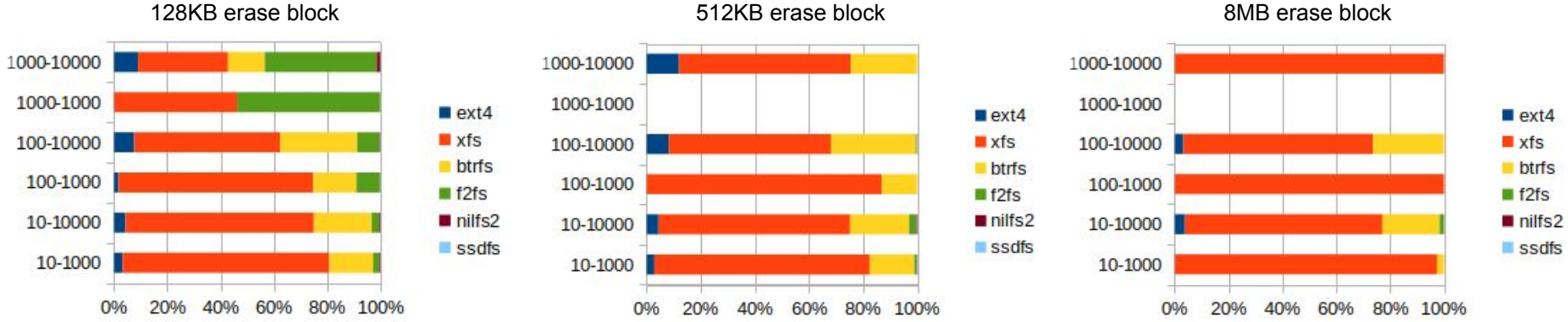


$$Payload_{ratio} = \frac{FS_{payload}}{SSDFS_{payload}}$$

	ext4	xfs	btrfs	f2fs	nilfs2
128KB	0.3x – 5x	0.8x – 7x	0.8x – 9x	3x – 13x	0.5x – 8x
512KB	0.2x – 20x	0.5x – 24x	0.5x – 24x	3x – 60x	1x – 8x
8MB	0.2x – 192x	0.2x – 128x	0.5x – 256x	1.1x – 384x	1.1x – 128x

SSDFS is capable to create smaller (2x - 200x) payload. However, SSDFS can generate bigger payload for some use-cases (for example, 10-10000, 100-10000) compared with ext4, xfs, btrfs.

FTL GC (erase blocks)



FTL responsibility (number of erase blocks)

	ext4	xfs	btrfs	f2fs	nilfs2	ssdfs
128KB	3 – 1037	14 – 16861	0 – 5175	17 – 660	0 – 91	0 – 0
512KB	0 – 258	0 – 4213	0 – 1290	0 – 156	0 – 21	0 – 0
8MB	0 – 13	0 – 261	0 – 75	0 – 5	0 – 0	0 – 0

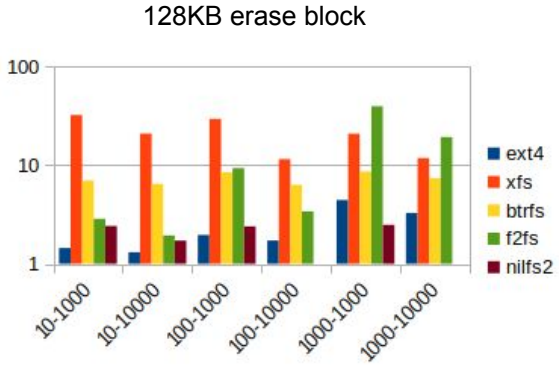
F2FS creates significant FTL GC activity. SSDFS doesn't create FTL GC responsibilities because it's pure LFS file system without any in-place update area.

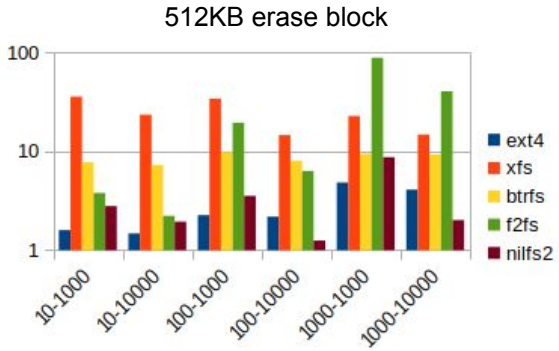
FS GC (erase blocks)

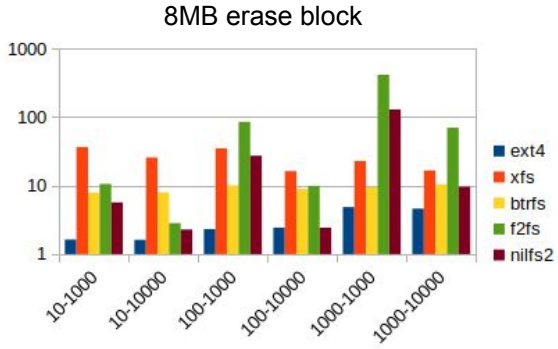
SSDFS: GC I/O is absent because of migration scheme and efficient TRIM policy.

F2FS introduces more FS GC responsibility (1.2x - 48x) compared with NILFS2.
However, NILFS2 introduces more FS GC responsibility (1x - 1.4x) compared with F2FS for 10-10000 use-case.

Write amplification (write I/O + FS GC I/O)







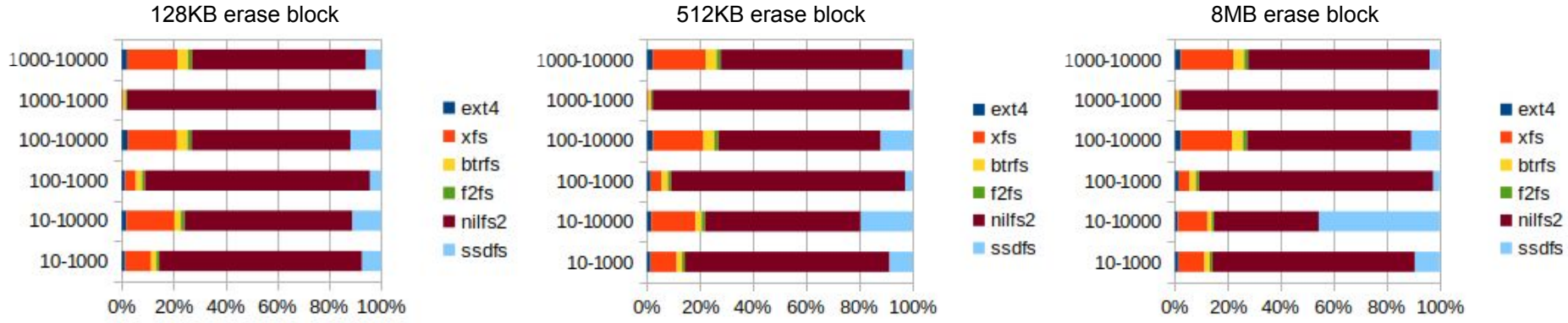
SSDFS decreases write amplification issue:

- 1.3x – 4.8x comparing with ext4
- 11.4x - 36x comparing with xfs
- 6.3x - 10.4x comparing with btrfs
- 1.9x – 412x comparing with f2fs
- 1x - 128x comparing with nilfs2

$$\text{Write Amplification}_{\text{ratio}} = \frac{\text{FS(Write I/O + FS GC I/O)}}{\text{SSDFS(Write I/O + FS GC I/O)}}$$

ByteDance

Read I/O (disturbance)

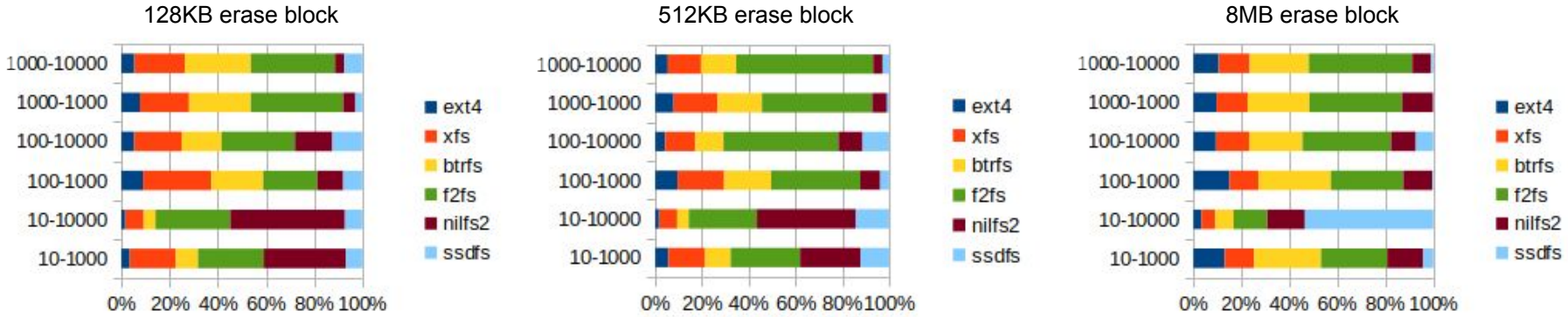


SSDFS generates smaller amount (2x - 150x) of read I/O compared with NILFS2
SSDFS generates comparatively same amount of read I/O compared with XFS
SSDFS generates bigger amount of read I/O:

- (2x - 40x) compared with ext4
- (1x - 29x) compared with btrfs
- (1x - 50x) compared with f2fs

SSDFS generates more read I/O for bigger erase blocks with smaller partial logs. Offsets translation table is the main contributor to this issue. Solution: store full offset translation table in every log + compress offset translation table.

Retention issue (estimation)

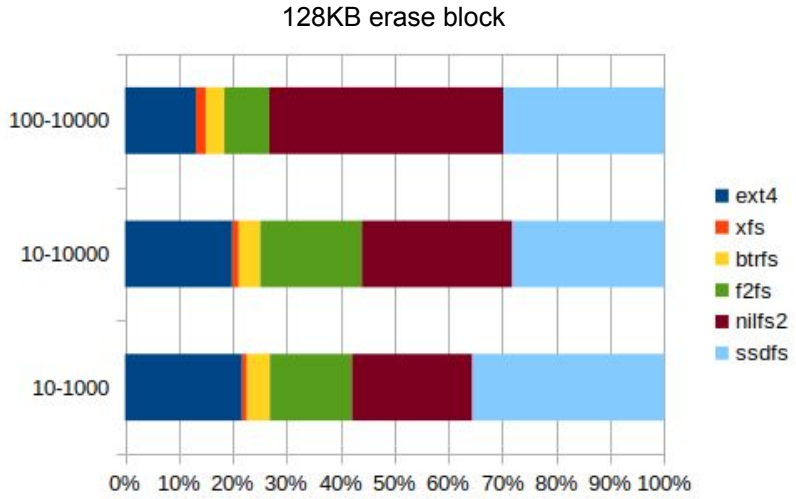


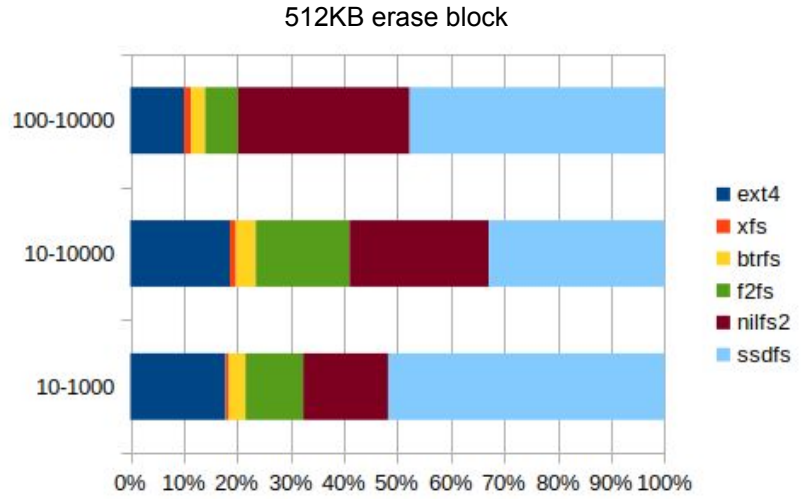
SSDFS is capable to introduce smaller retention issue (in average):

- (1x - 96x) compared with ext4
- (1x - 128x) compared with xfs
- (1x - 256x) compared with btrfs
- (2x - 384x) compared with f2fs
- (1x - 128x) compared with nilfs2

However, SSDFS can introduce bigger retention issue for some use-cases (for example, 10-10000) - big erase blocks with small partial logs. But this estimation has been determined by use-case duration that is defined by bigger amount of read I/O requests. This issue can be fixed by offsets translation table optimization.

SSD lifetime (estimation)





SSDFS is capable to prolong SSD lifetime:

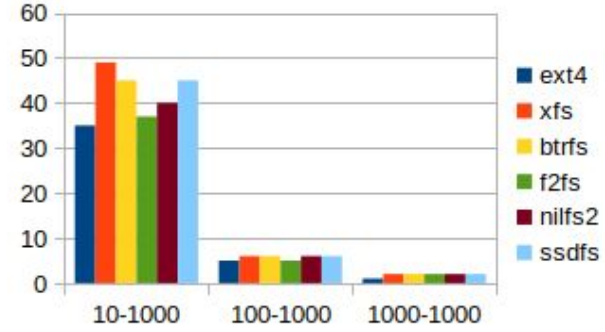
- (1.4x - 4.7x) compared with ext4
- (16x - 81x) compared with xfs
- (7x - 18x) compared with btrfs
- (1.4x - 7.7x) compared with f2fs
- (1x - 3.2x) compared with nilfs2

SSDFS can prolong SSD lifetime 2x - 10x for real-life use-cases

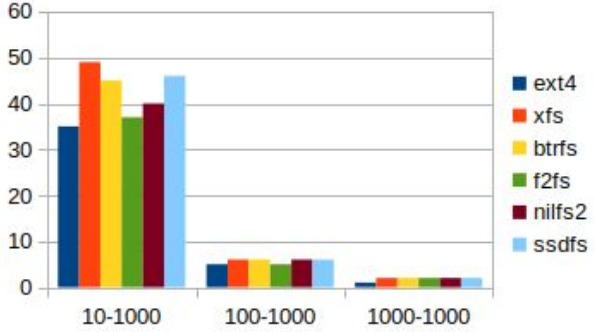
ByteDance

Duration (seconds)

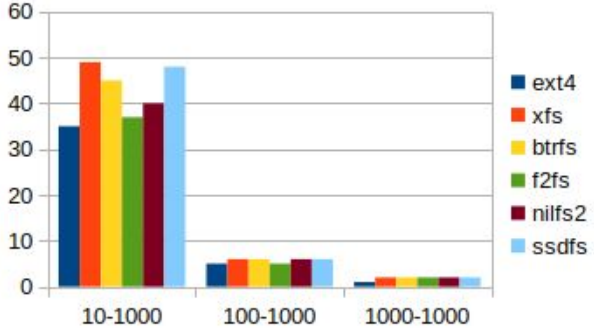
128KB erase block



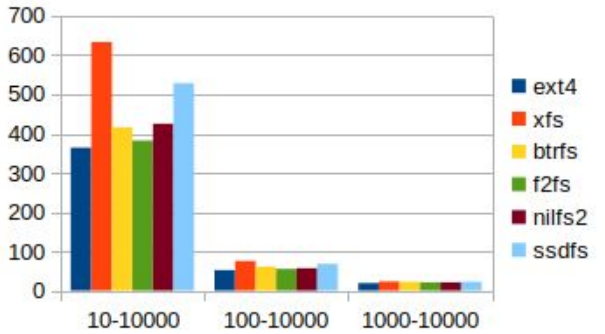
512KB erase block



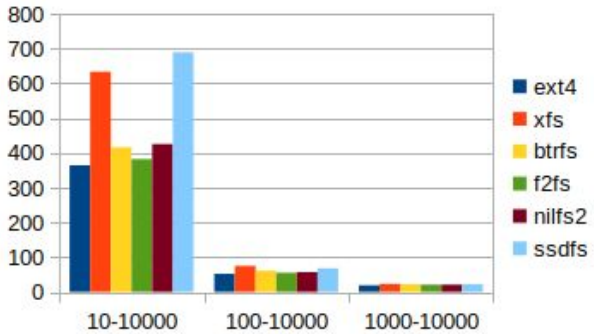
8MB erase block



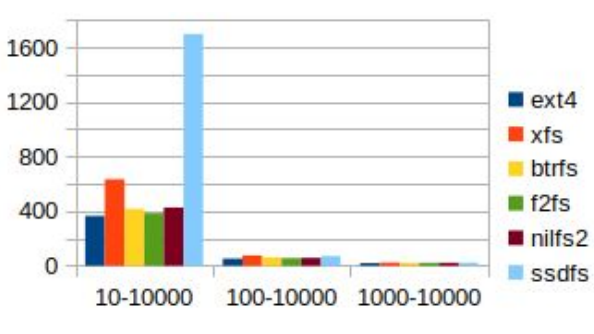
128KB erase block



512KB erase block



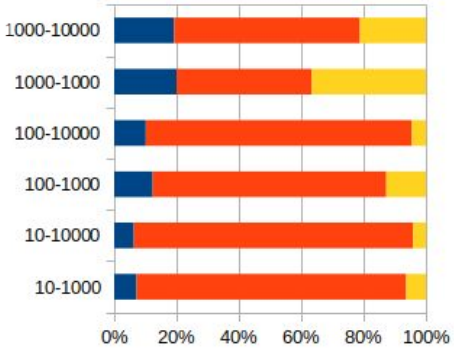
8MB erase block



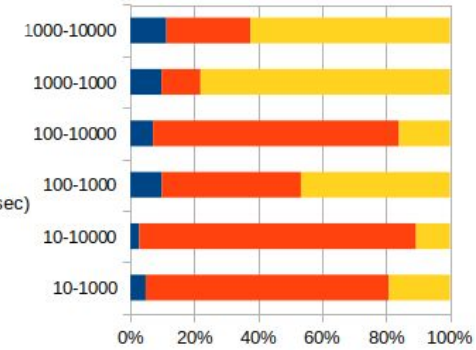
ByteDance

Performance analysis (SSDFS)

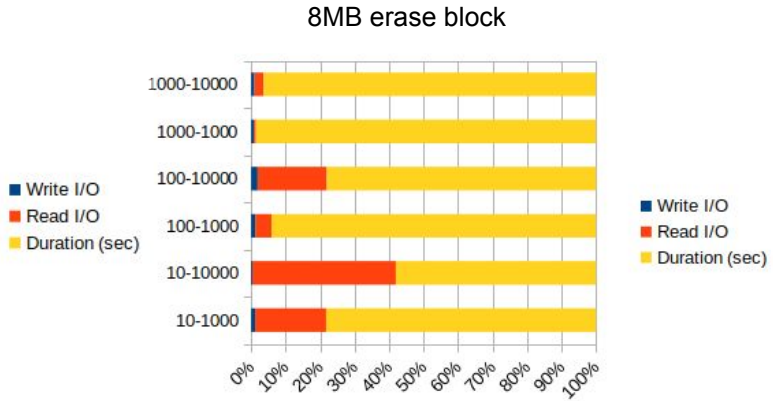
128KB erase block



512KB erase block

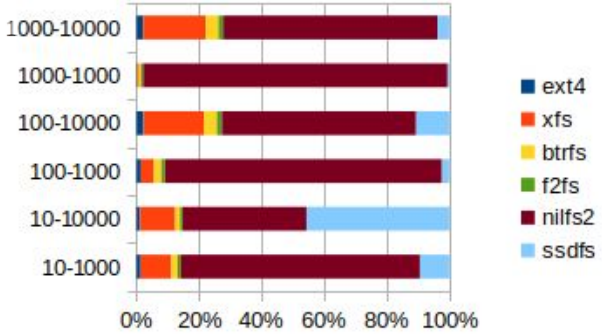


8MB erase block



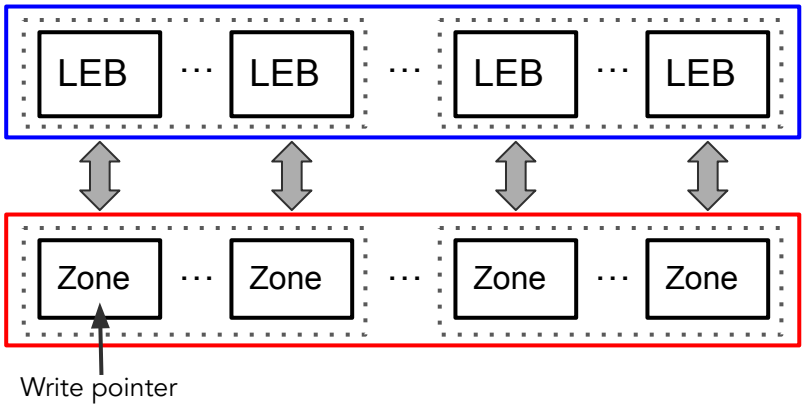
- SSDFS has been tested in debug mode.
- SSDFS still has not fully optimized code.
- Even now SSDFS performance looks comparable with other file systems.
- Currently, SSDFS looks like read dominant.
- The main contributor of read-dominant nature is offset translation table.
- Solution:
  - Store full offset translation table in every log.
  - Compress offset translation table.
  - Employ binary search to find the latest log in a PEB.

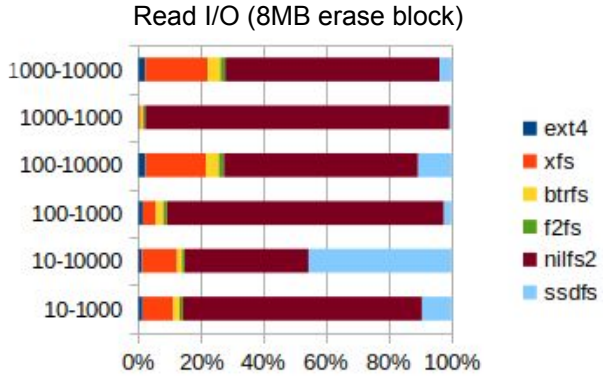
Read I/O (8MB erase block)



ByteDance

Is SSDFS ZNS ready?





- SSDFS is pure LFS file system => zone-aware file system.
- LEB can be mapped into zone (PEB == Zone).
- SSDFS needs some implementation efforts to fully support ZNS SSD.
- Potentially, zone size (256MB, for example) can increase log's metadata size.

Future work

Metadata	User data	
Create empty file	Create file	64 bytes
		16KB
		100KB
Update empty file	Update file	64 bytes
		16KB
		100KB
Delete empty file	Delete file	64 bytes
		16KB
		100KB

Analyze benchmark results

- Fix read I/O performance degradation
- Solution:
  - Store full offset translation table in every log.
  - Compress offset translation table.
  - Employ binary search to find the latest log in a PEB.

- Bug fix
- Finish deduplication support implementation
- Finish snapshot support implementation
- Post-deduplication delta-compression implementation
- fsck implementation
- recoverfs implementation
- ZNS SSD support

SSDFS tools: https://github.com/dubeyko/ssdfs-tools.git
SSDFS driver: https://github.com/dubeyko/ssdfs-driver.git
Linux kernel: https://github.com/dubeyko/linux.git

Conclusion

- SSDFS generates smaller amount of write I/O requests - (1.3x - 5x) in average.
- SSDFS introduces highly efficient TRIM policy. Even multiple mount/unmount operations cannot affect the efficiency of TRIM policy.
- SSDFS is capable to create smaller (2x - 200x) payload.
- SSDFS doesn't create FTL GC responsibilities because it's pure LFS file system without any in-place update area.
- GC I/O is absent because of migration scheme and efficient TRIM policy.
- SSDFS decreases write amplification issue - (1.3x - 10x) in average.
- SSDFS is capable to introduce smaller retention issue.
- SSDFS can prolong SSD lifetime 2x - 10x for real-life use-cases.
- SSDFS looks like read dominant. SSDFS generates more read I/O for bigger erase blocks with smaller partial logs. However, there is a way to fix this issue.

Thank You

Questions???

ByteDance