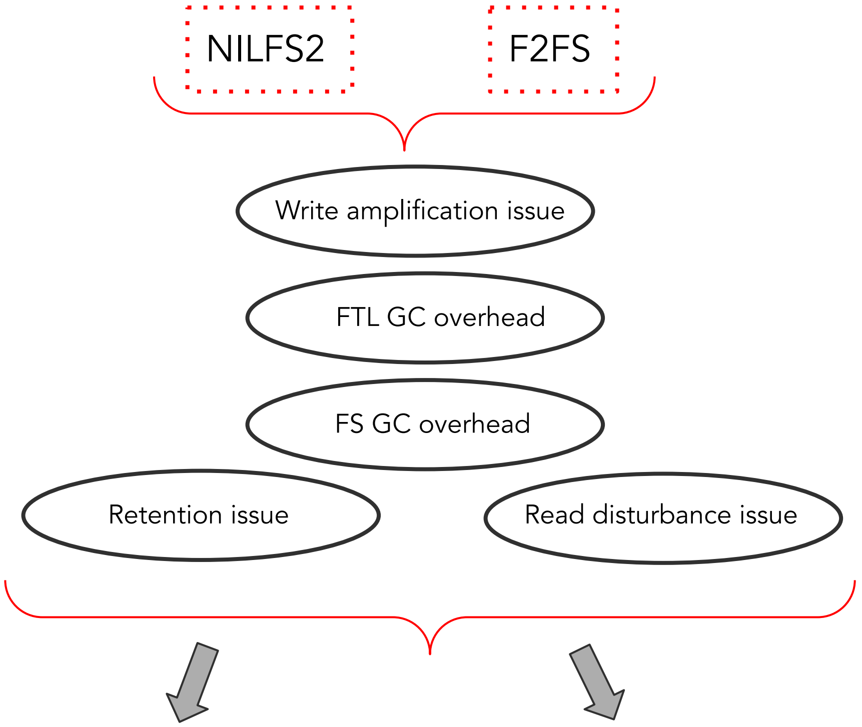
SSDFS: ZNS SSD ready file system with zero GC overhead

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

ByteDance

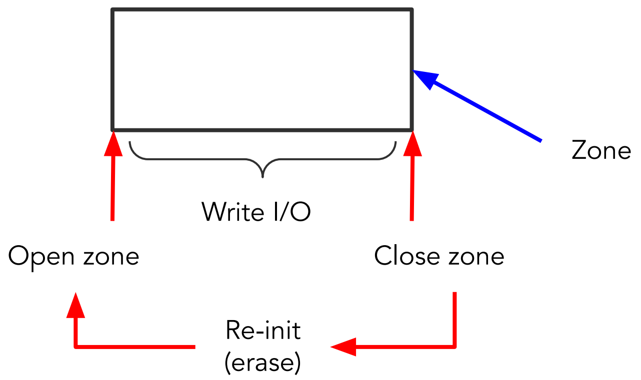
Content

Problem



SSD lifetime reduction   Performance degradation

ZNS SSD model



Limited number of open/active zones

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

1. prolong SSD lifetime
2. reliability
3. performance

} SSDFS

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

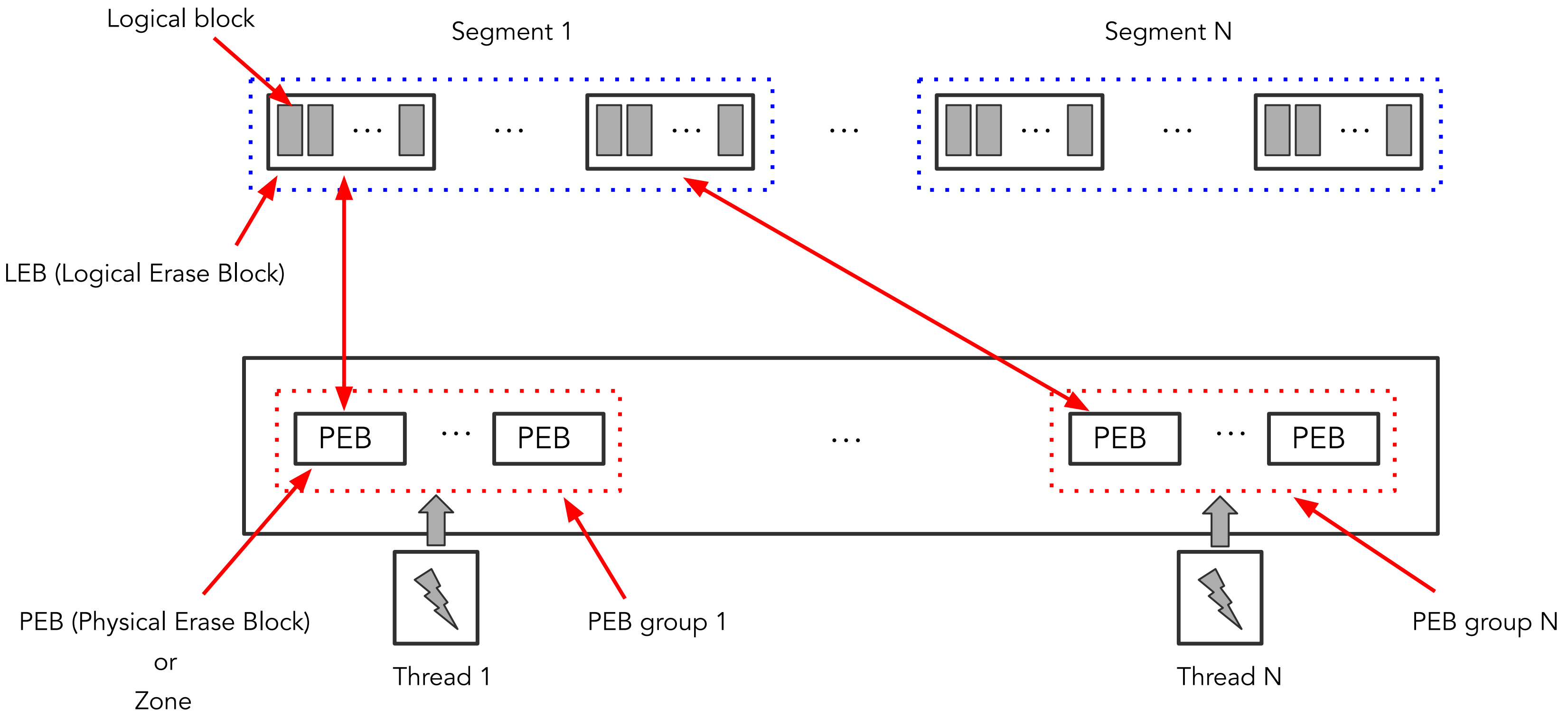
- Checksumming support
- Metadata replication
- Snapshots support
- Erasure coding support
- Reconstruct corrupted file system

③

Stable file system performance

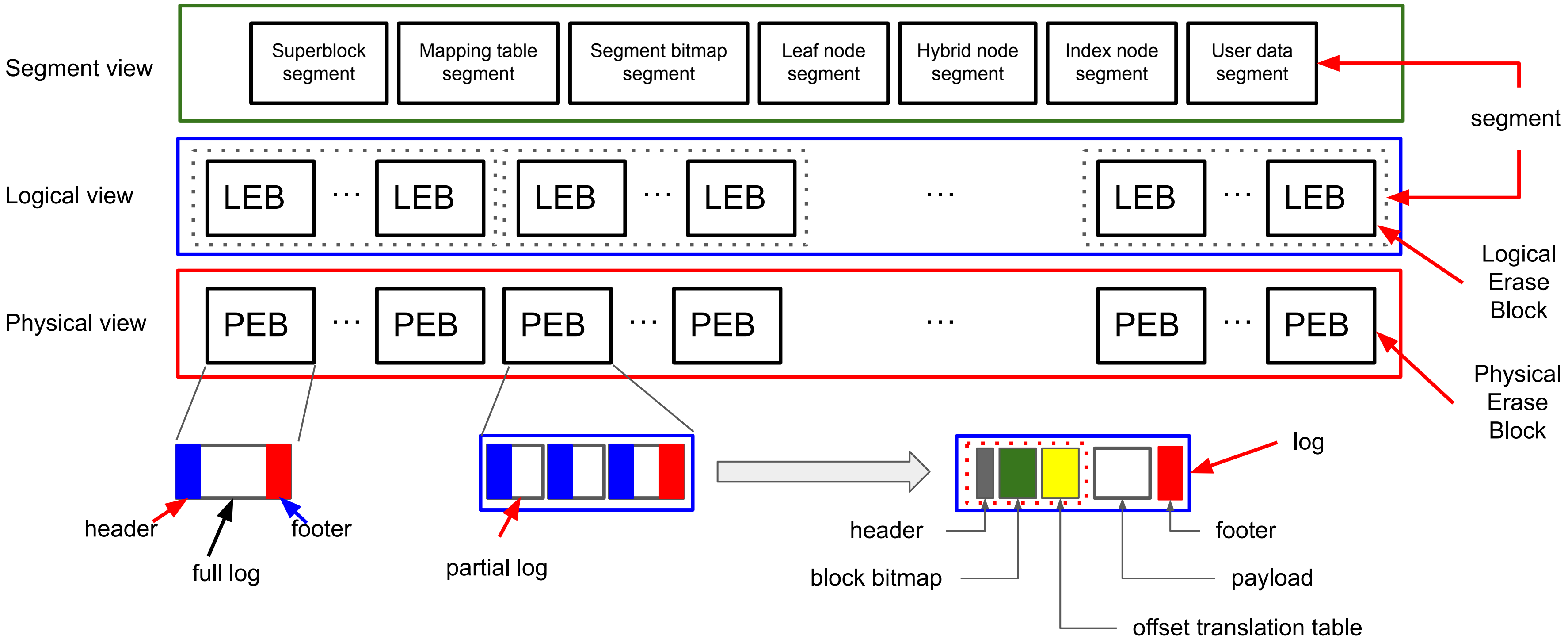
- Employ parallelism of multiple NAND dies
- Multiple PEBs in segment
- ZeroGC overhead
- Minimized write amplification
- B-trees in metadata
- Efficient TRIM policy

SSDFS configuration model



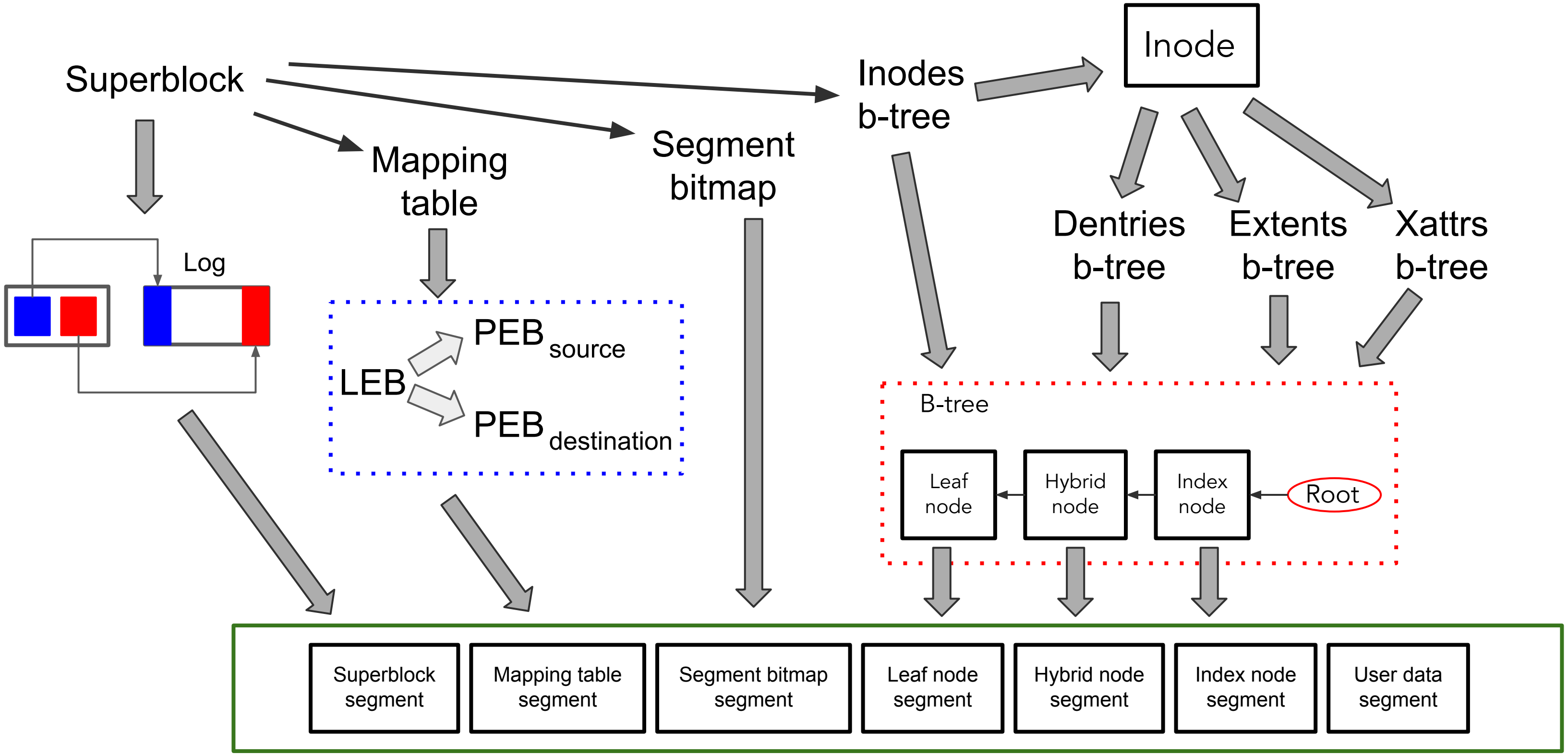
ByteDance

SSDFS architecture (logical vs. physical view)

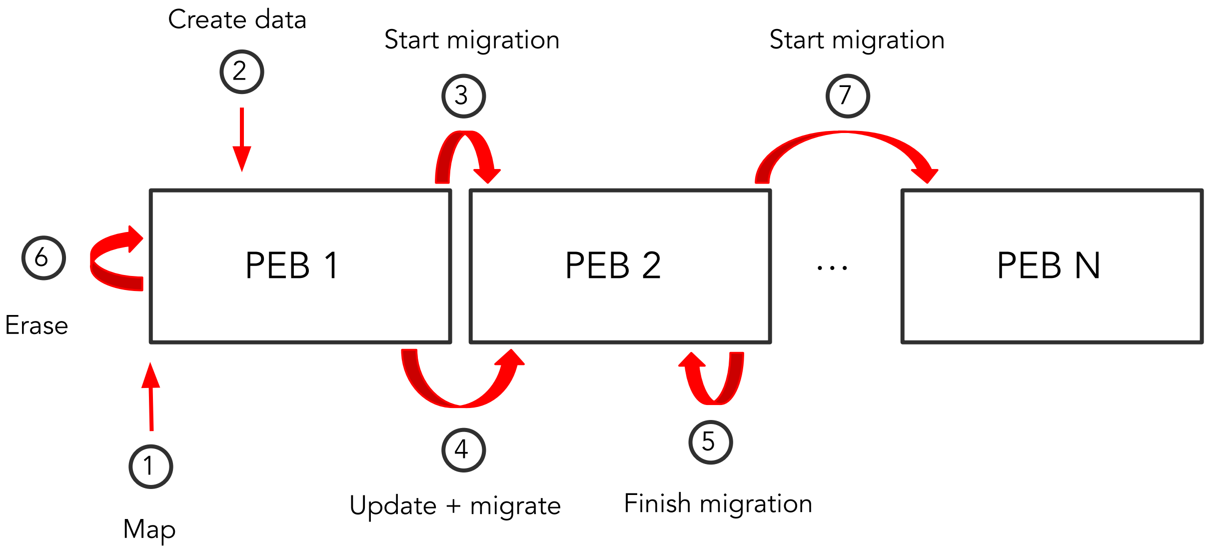


ByteDance

SSDFS architecture (metadata)



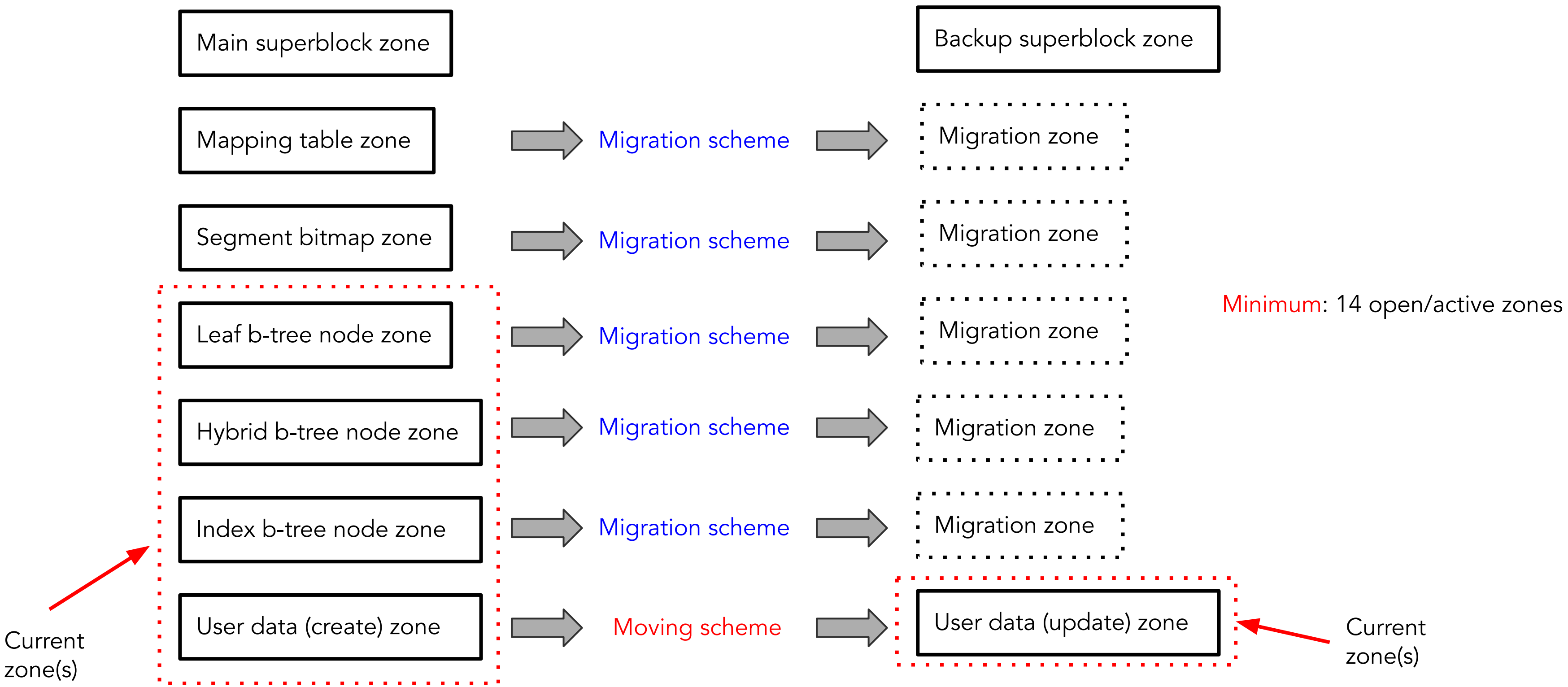
Migration scheme (PEB lifetime)



Migration scheme:

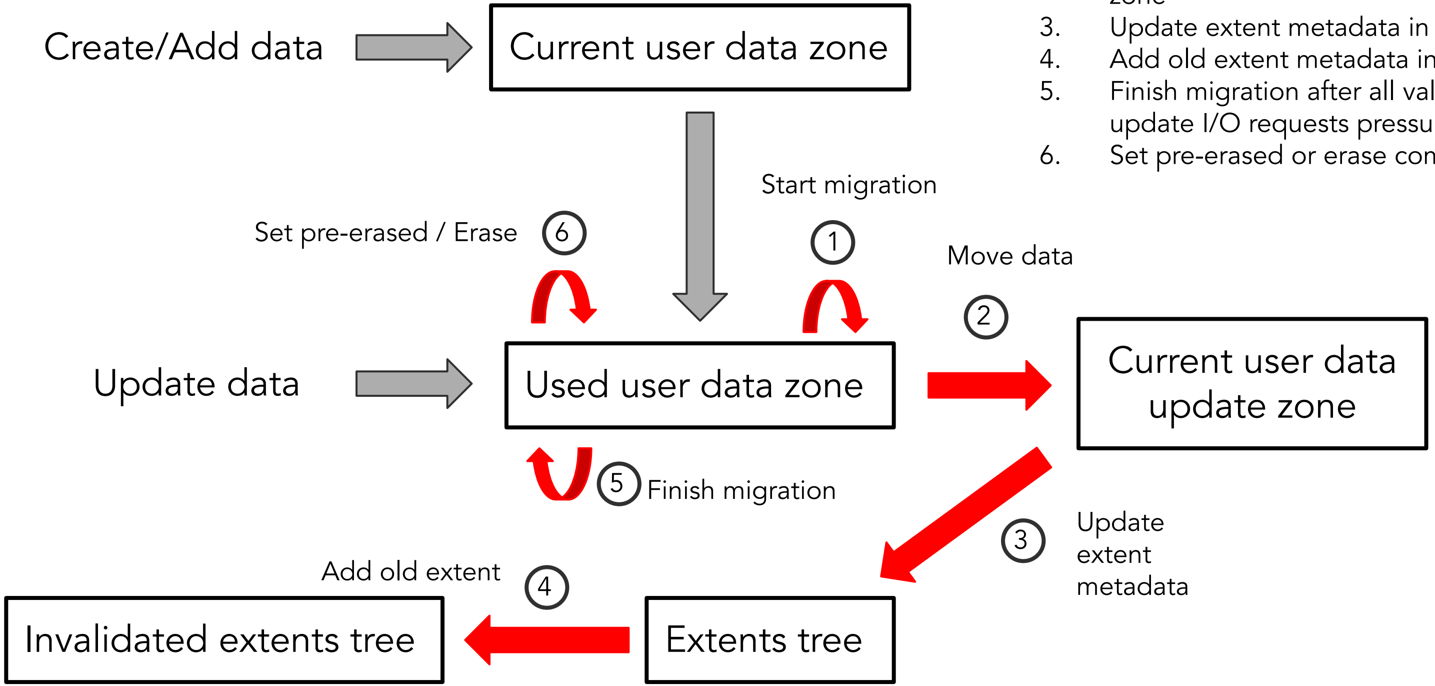
1. Map LEB to PEB
2. Create/Fill by data until PEB exhaustion
3. Start migration (PEB1 -> PEB2)
4. Update data + migrate valid data until PEB1 complete invalidation
5. Finish migration
6. Set PEB1 pre-erased or TRIM/erase PEB1
7. Go to step 3 if PEB2 is exhausted

Current zones



Moving scheme (ZNS SSD only)

1. Start migration
2. Move valid/updated data to current user data update zone
3. Update extent metadata in extents tree
4. Add old extent metadata into invalidated extents tree
5. Finish migration after all valid data will be moved under update I/O requests pressure
6. Set pre-erased or erase completely invalidated zone



ByteDance

Testing use-case(s)

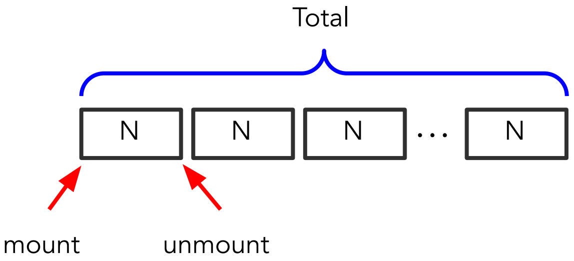
Metadata	User data	
Create empty file	Create file	64 bytes
		16KB
		100KB
Update empty file	Update file	64 bytes
		16KB
		100KB
Delete empty file	Delete file	64 bytes
		16KB
		100KB

SSDFS	
Erase block size	128KB
	512KB
	8MB

Testing sequence:

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

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



Methodology

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

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

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

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

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

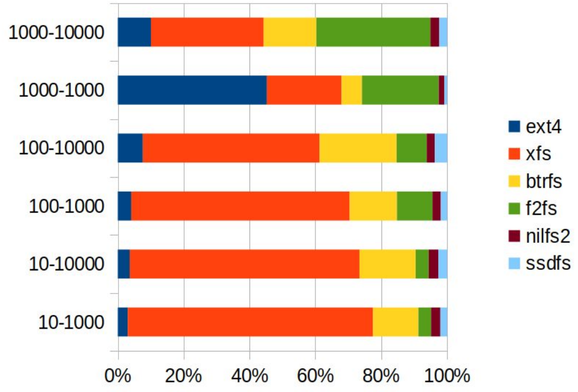
$$\text{Erase}_{\text{FS GC}} = \text{Payload}_{\text{EB}} - \text{Valid Data}_{\text{EB}}$$

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

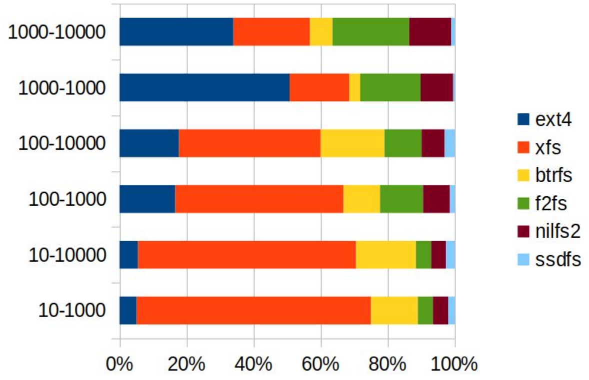
$$\text{Erase}_{\text{retention}} = \frac{\text{Time}_{\text{use-case}}}{\text{3 months}} * \text{Payload}_{\text{EB}}$$

Write I/O (create + update + delete)

Metadata case



64 bytes file case



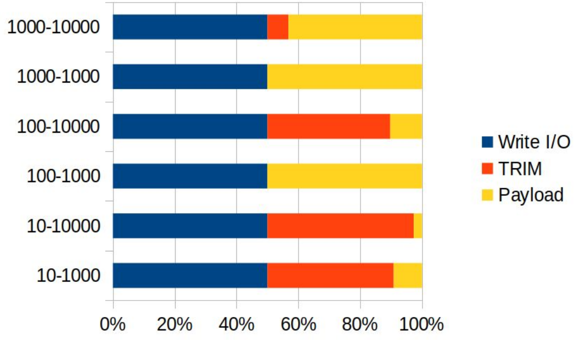
	ext4	xfs	btrfs	f2fs	nilfs2
128KB	1.4x – 56x	14x – 36x	6.2x – 7.7x	1.5x – 29x	0.6x – 2.1x
512KB	1.5x – 61x	18x – 41x	7.4x – 8.7x	1.7x – 31x	0.8x – 2.3x
8MB	1.6x – 61x	16x – 42x	7.3x – 9.8x	1.8x – 31x	0.7x – 2.3x

	ext4	xfs	btrfs	f2fs	nilfs2
128KB	2x – 107x	14x – 37x	6.3x – 7.5x	1.7x – 38x	1.7x – 20x
512KB	2.4x – 116x	18x – 40x	7.4x – 8.7x	2x – 41x	1.9x – 22x
8MB	2.5x – 116x	17x – 40x	7.4x – 8.9x	2x – 41x	2x – 22x

SSDFS is capable to generate smaller amount (1.5x - 20x) of write I/O requests comparing with other file systems.

TRIM (create + update + delete) - erase blocks

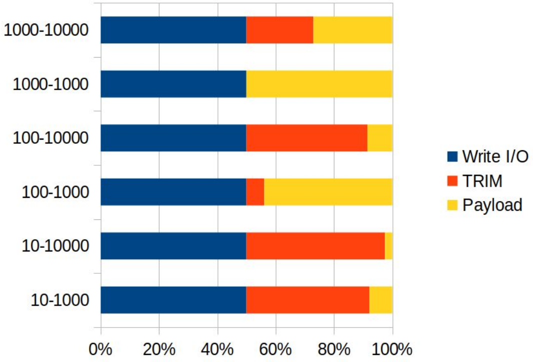
Metadata case



128KB	Write I/O	TRIM	Payload
10-1000	134.65625	110	24.65625
10-10000	1891.375	1791	100.375
100-10000	359.71875	286	73.71875

512KB	Write I/O	TRIM	Payload
10-1000	29.59375	18	11.59375
10-10000	406.4921875	351	55.4921875
100-10000	69.0859375	41	28.0859375

64 bytes file case



128KB	Write I/O	TRIM	Payload
10-1000	141.09375	119	22.09375
10-10000	2022.53125	1922	100.53125
100-10000	410.84375	341	69.84375

512KB	Write I/O	TRIM	Payload
10-1000	31.3984375	20	11.3984375
10-10000	437.84375	388	49.84375
100-10000	77.1328125	51	26.1328125

SSDFS introduces highly efficient TRIM policy that:
(1) eliminate FTL GC activity,
(2) decrease retention issue.

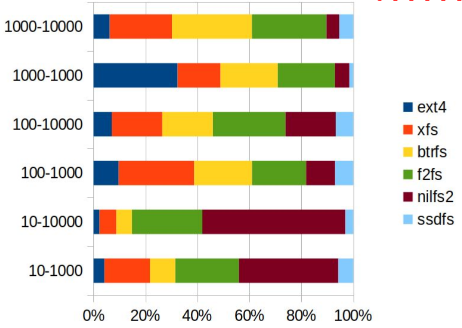
Migration scheme builds the TRIM efficiency and eliminates the necessity of FS GC activity. Even multiple mount/unmount operations cannot affect the efficiency of TRIM policy.

ByteDance

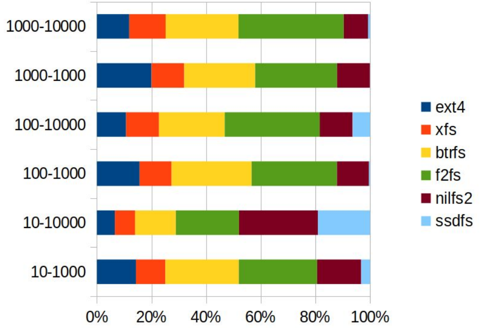
Payload (create + update + delete) - erase blocks

Metadata case

128KB erase block

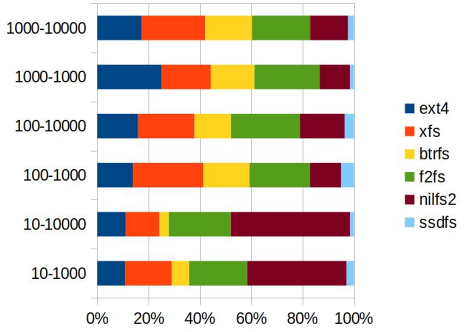


8MB erase block

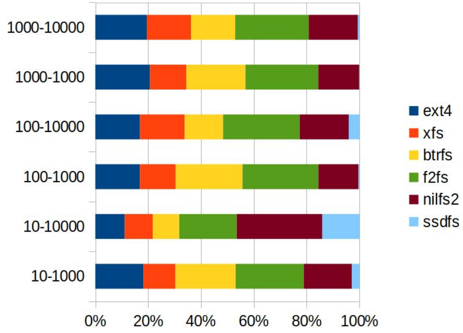


64 bytes file case

128KB erase block



8MB erase block



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

Metadata case

	ext4	xfs	btrfs	f2fs	nilfs2
128KB	0.7x – 21x	2.1x – 10x	1.7x – 14x	2.9x – 14x	0.9x – 18x
512KB	0.4x – 80x	1x – 33x	1x – 39x	3.7x – 63x	2x – 11x
8MB	0.3x – 315x	0.3x – 189x	0.7x – 409x	1.2x – 472x	1.5x - 189x

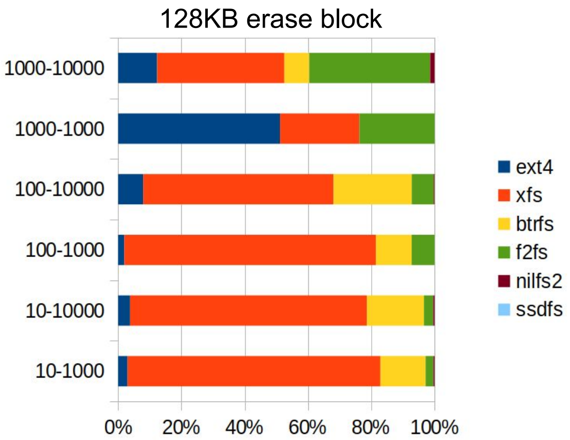
64 bytes file case

	ext4	xfs	btrfs	f2fs	nilfs2
128KB	2.7x – 15x	5.4x – 12x	2.3x – 10x	4.7x – 16x	2.4x – 29x
512KB	2.5x – 64x	3.4x – 40x	1.4x – 27x	5.4x – 68x	8.4x – 31x
8MB	0.7x – 248x	0.7x – 165x	0.7x – 268x	1.5x – 330x	2.2x – 186x

SSDFS is capable to create smaller (2x - 20x) payload. However, SSDFS can generate more payload for some use-cases (for example, 10-10000, 100-10000) compared with ext4, xfs, btrfs.

FTL GC (create + update + delete) - erase blocks

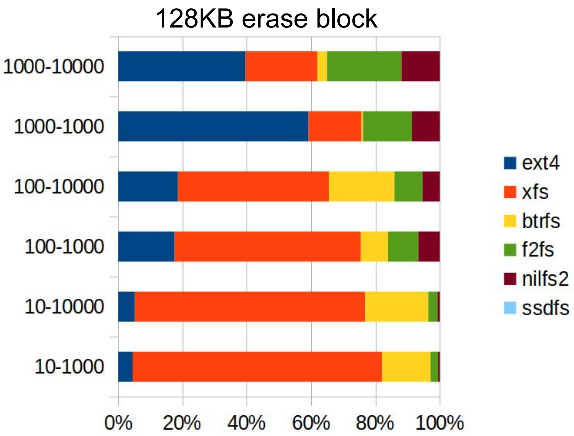
Metadata case



FTL responsibility (number of erase blocks) - metadata

	ext4	xfs	btrfs	f2fs	nilfs2	ssdfs
128KB	11 – 2521	39 – 49701	0 – 11990	37 – 1959	0 – 274	0 - 0
512KB	0 – 626	0 – 12418	0 – 2989	0 – 468	0 – 64	0 - 0
8MB	0 – 32	0 – 770	0 – 172	0 – 16	0 - 0	0 - 0

64 bytes file case



FTL responsibility (number of erase blocks) - 64 bytes file

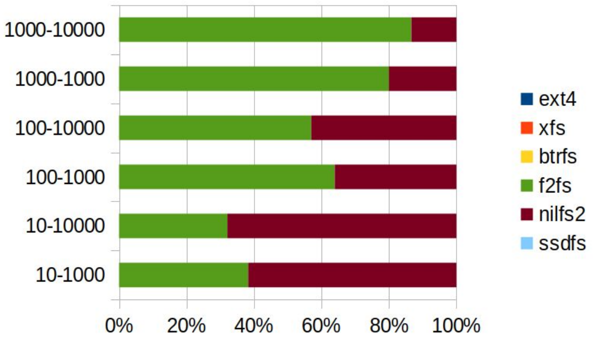
	ext4	xfs	btrfs	f2fs	nilfs2	ssdfs
128KB	149 – 3605	85 – 49817	2 – 13657	79 – 2004	45 – 612	0 - 0
512KB	22 – 892	0.8 – 12448	0 – 3403	0 – 479	0 – 63	0 - 0
8MB	0 – 47	0 – 772	0 – 199	0 – 16	0 - 0	0 - 0

SSDFS doesn’t create FTL GC responsibilities because it’s pure LFS file system without any in-place update area.

ByteDance

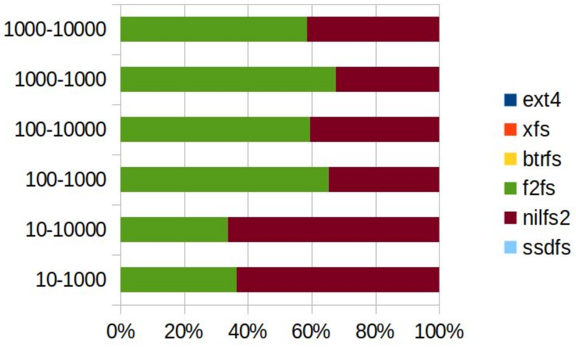
FS GC (create + update + delete) - erase blocks

Metadata case



	f2fs	nilfs2
128KB	39 – 853	7 – 1804
512KB	31 – 235	5 – 455
8MB	14 – 27	5 – 34

64 bytes file case



	f2fs	nilfs2
128KB	50 – 1502	24 – 2957
512KB	52 – 397	23 – 797
8MB	15 – 38	8 – 57

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

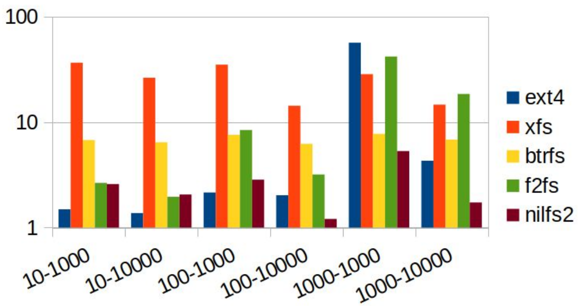
F2FS introduces more FS GC responsibility (1.2x - 5x) compared with NILFS2.
However, NILFS2 introduces more FS GC responsibility (1.3x - 2x) compared with F2FS for 10-10000 use-case.

ByteDance

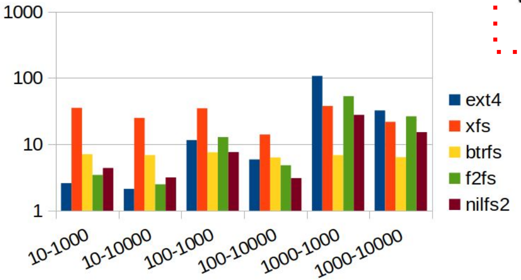
Write amplification (create + update + delete)

Metadata case

128KB erase block

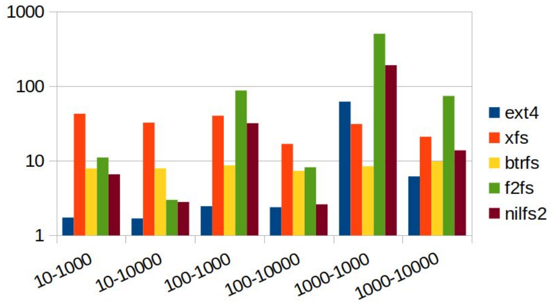


8MB erase block

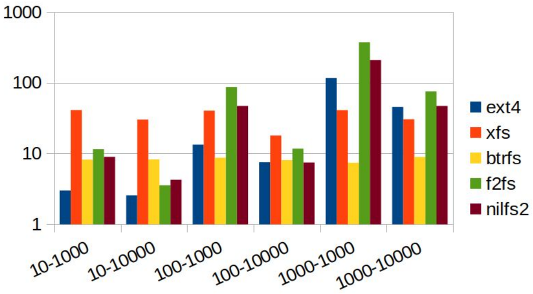


64 bytes file case

128KB erase block



8MB erase block



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

Metadata case

	ext4	xfs	btrfs	f2fs	nilfs2
128KB	1.3x – 56x	14x – 35x	6x – 7.7x	1.9x – 41x	1.2x – 5.3x
512KB	1.5x – 61x	18x – 41	7.4x – 8.7x	2.3x – 93x	1.6x – 13x
8MB	1.6x – 61x	16x – 42x	7.3x – 9.8x	2.9x – 502x	2.6x – 190x

w/o GC I/O

	f2fs	nilfs2
128KB	1.5x – 29x	0.6x – 2.1x
512KB	1.7x – 31x	0.8x – 2.3x
8MB	1.8x – 31x	0.7x – 2.3x

64 bytes file case

	ext4	xfs	btrfs	f2fs	nilfs2
128KB	2x – 107x	14x – 37x	6.3x – 7.5x	2.4x – 53x	3x – 27x
512KB	2.4x – 116x	18x – 40x	7.4x – 8.7x	2.9x – 108x	3.7x – 52x
8MB	2.5x – 116x	17x – 41x	7.4x – 8.9x	3.5x – 371x	4.2x – 208x

w/o GC I/O

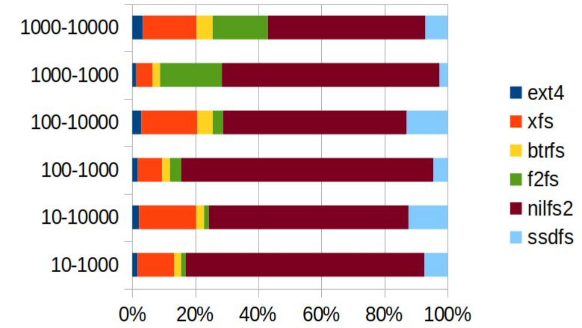
	f2fs	nilfs2
128KB	1.7x – 38x	1.7x – 20x
512KB	2x – 41x	1.9x – 22x
8MB	2.1x – 41x	2x – 22x

SSDFS is capable to decrease a write amplification issue 1.5x - 20x comparing with other file systems.

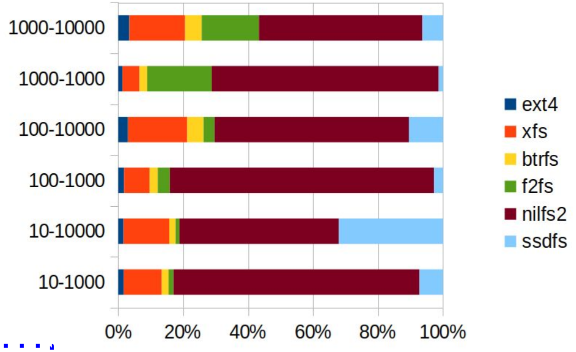
Read disturbance (create + update + delete)

Metadata case

128KB erase block

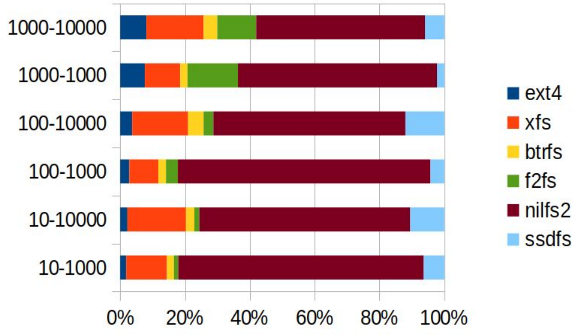


8MB erase block

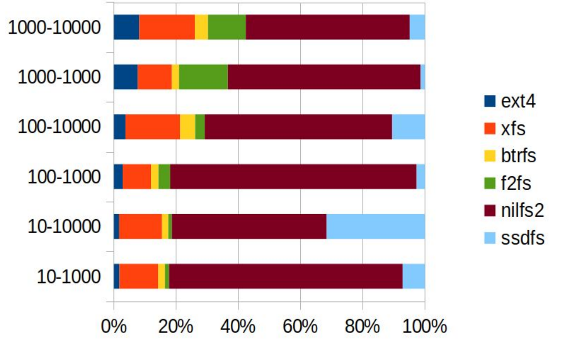


64 bytes file case

128KB erase block



8MB erase block



Metadata case

	ext4	xfs	btrfs	f2fs	nilfs2
128KB	0.1x – 0.5x	1.3x – 2.4x	0.2x – 0.9x	0.1x – 7.8x	4.5x – 27x
512KB	0.1x – 0.8x	0.8x – 4.3x	0.1x – 1.4x	0.06x – 12x	2.8x – 44x
8MB	0.05x – 0.9x	0.4x – 4x	0.06x – 1.8x	0.03x – 15x	1.5x – 53x

64 bytes file case

	ext4	xfs	btrfs	f2fs	nilfs2
128KB	0.2x – 3.4x	1.7x – 4.8x	0.2x – 1x	0.1x – 6.9x	4.9x – 27x
512KB	0.1x – 5x	0.9x – 7.3x	0.1x – 1.5x	0.08x – 10x	3.5x – 41x
8MB	0.05x – 5.6x	0.4x – 8x	0.06x – 1.6x	0.03x – 11x	1.5x – 45x

SSDFS generates smaller amount of read I/O
- (1.5x - 50x) compared with nilfs2
- (1x - 8x) compared with xfs

SSDFS generates bigger amount of read I/O:
- (1x - 20x) compared with ext4
- (1x - 16x) compared with btrfs
- (1x - 26x) compared with f2fs

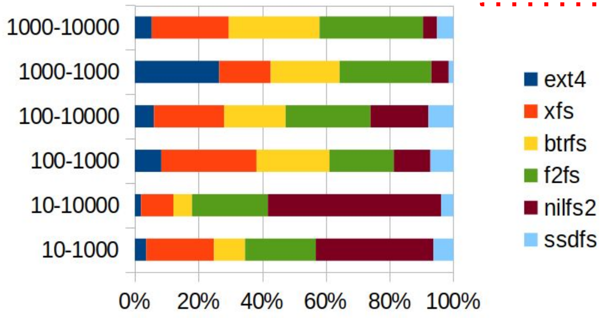
SSDFS generates more read I/O for bigger erase blocks with smaller partial logs. Offsets translation table is the main contributor to this issue.
Solution: store full offset translation table in every log + compress offset translation table.

ByteDance

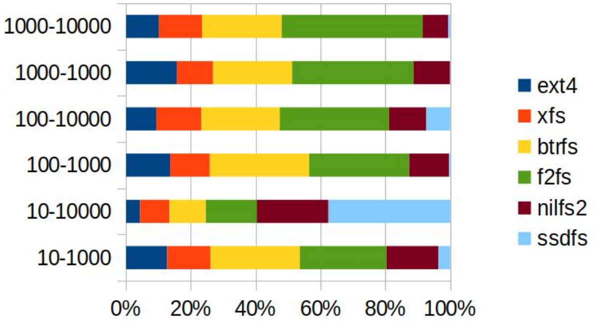
Retention issue (create + update + delete)

Metadata case

128KB erase block

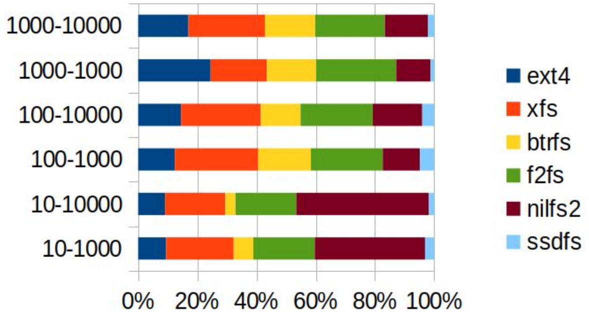


8MB erase block

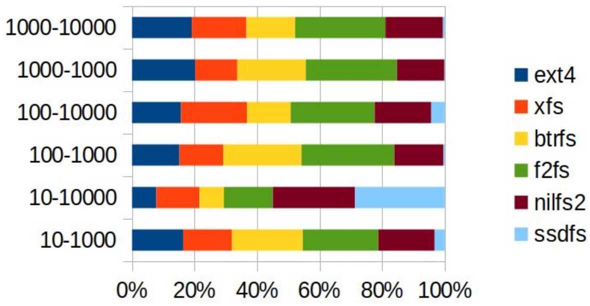


64 bytes file case

128KB erase block



8MB erase block



Metadata case

	ext4	xfs	btrfs	f2fs	nilfs2
128KB	0.5x – 17x	2.6x – 10x	1.5x – 14x	2.8x – 19x	1.5x – 13x
512KB	0.2x – 67x	1x- 33x	0.6x – 39x	2.4x – 84x	1.7x – 11x
8MB	0.1x – 262x	0.2x – 189x	0.2x – 409x	0.4x – 630x	0.5x – 189x

64 bytes file case

	ext4	xfs	btrfs	f2fs	nilfs2
128KB	2.5x – 19x	5.7x – 14x	1.8x – 12x	5x – 20x	2.5 x- 24x
512KB	2x – 77x	4x – 48x	0.9x – 32x	4.6x – 89x	6.5x – 37x
8MB	0.2x – 297x	0.4x – 198x	0.2x – 322x	0.5x – 430x	0.9x – 223x

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

- (1x - 200x) compared with ext4
- (1x - 200x) compared with xfs
- (1x - 400x) compared with btrfs
- (2x - 600x) compared with f2fs
- (1x - 200x) compared with nilfs2

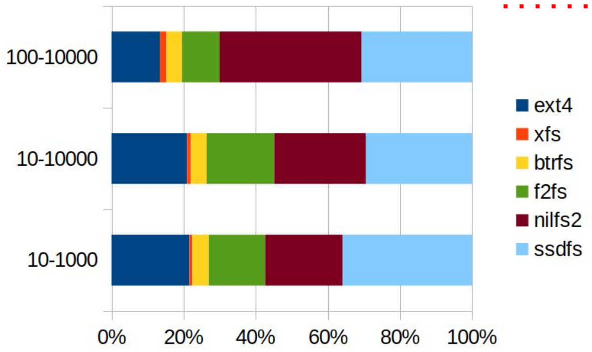
However, SSDFS can introduce bigger retention issue for some use-cases (for example, 10-10000) - big erase blocks with small partial logs. This issue can be fixed by offsets translation table optimization.

ByteDance

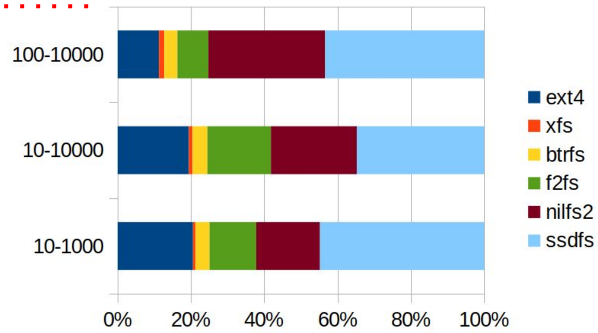
SSD lifetime (create + update + delete)

Metadata case

128KB erase block

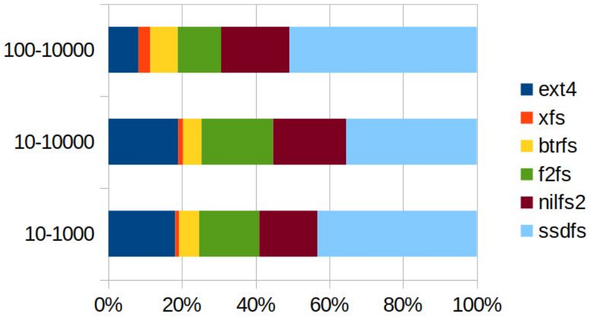


512KB erase block

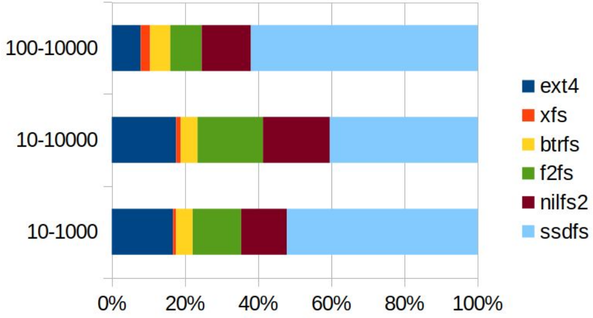


64 bytes file case

128KB erase block



512KB erase block



Metadata case

	ext4	xfs	btrfs	f2fs	nilfs2
128KB	1.4x – 2.2x	17x – 44x	6.6x – 7.8x	1.5x – 2.9x	0.7x – 1.6x
512KB	1.7x – 3.8x	30x – 67x	8.5x – 12x	2x – 5x	1.3x – 2.5x

64 bytes file case

	ext4	xfs	btrfs	f2fs	nilfs2
128KB	1.8x – 6.2x	15x – 40x	6.8x – 7.9x	1.8x – 4.3x	1.7x – 2.7x
512KB	2.3x – 7.8x	24x – 60x	8.7x – 11x	2.2x – 7.2x	2.2x – 4.6x

SSDFS is capable to prolong SSD lifetime:

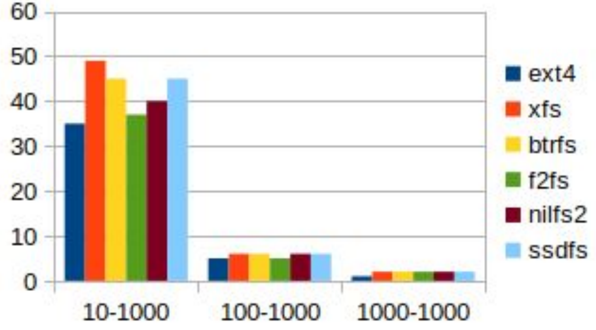
- (1.4x - 7.8x) compared with ext4
- (15x - 60x) compared with xfs
- (6x - 12x) compared with btrfs
- (1.5x - 7x) compared with f2fs
- (1x - 4.6x) compared with nilfs2

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

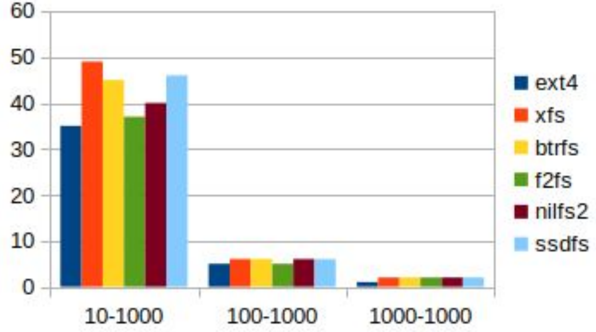
ByteDance

Duration (seconds)

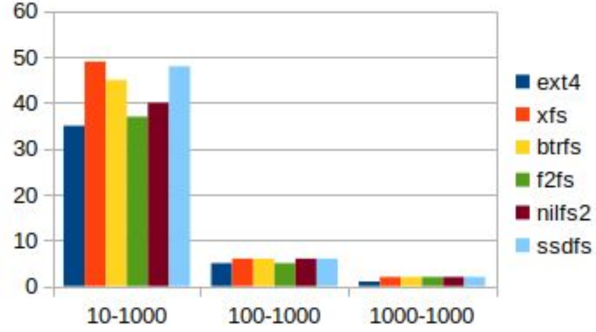
128KB erase block



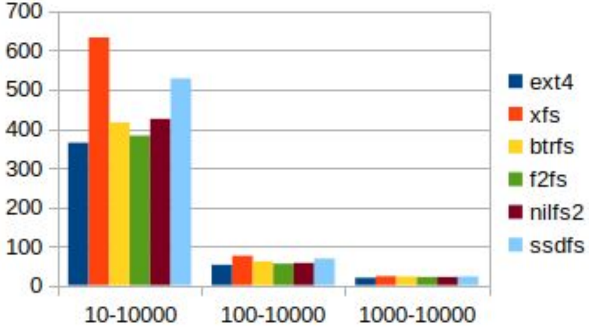
512KB erase block



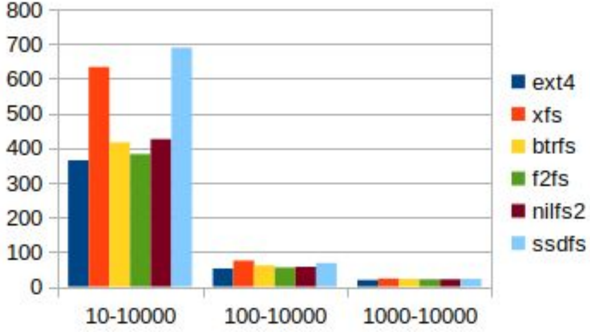
8MB erase block



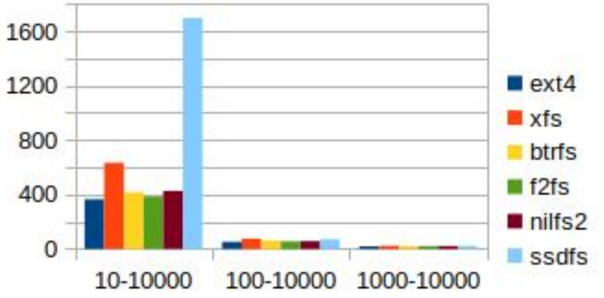
128KB erase block



512KB erase block

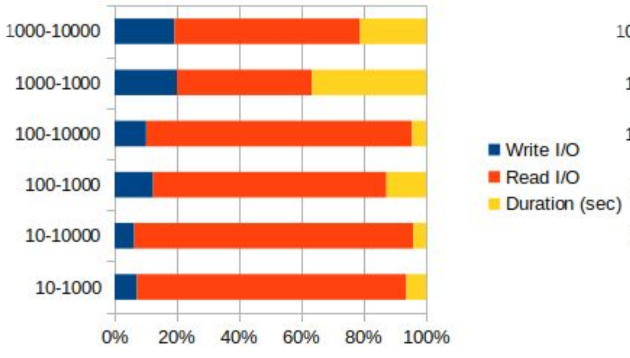


8MB erase block

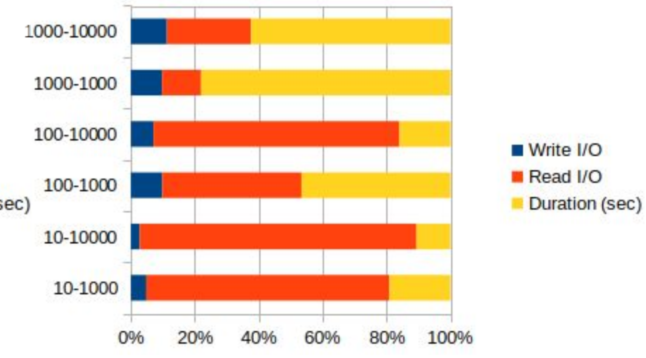


Performance analysis (SSDFS)

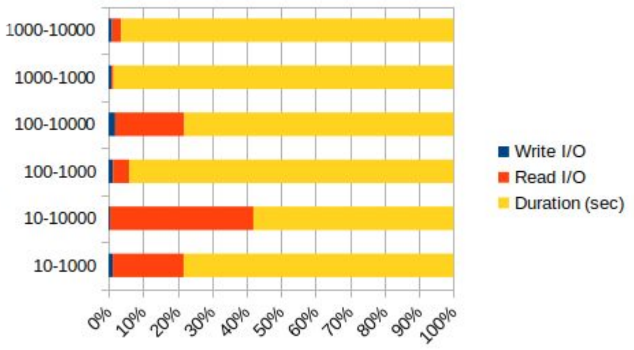
128KB erase block



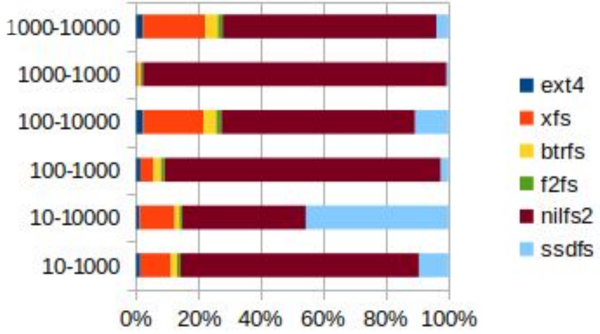
512KB erase block



8MB erase block



Read I/O (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.

ByteDance

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

- 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.

- Analyze benchmark results + btrfs compression + bcachefs
- Bug fix
- Finish deduplication support implementation
- Finish snapshot support implementation
- Post-deduplication delta-compression implementation
- fsck implementation
- recoverfs implementation
- ZNS SSD support code stabilization

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

ZNS SSD support -> ssdfs-zns-support branch (ssdfs-driver.git)

ByteDance

Conclusion

- SSDFS is natively compatible with ZNS SSD model. However, it will be good to have number of open/active zones equals to zone capacity of storage device.
- SSDFS generates smaller amount of write I/O requests - (1.5x - 20x) 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 - 20x) 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 - (1.5x - 20x) 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