Determinizing Crash Behavior with a Verified Snapshot-Consistent Flash Translation Layer

Yun-Sheng Chang Yao Hsiao Tzu-Chi Lin Che-Wei Tsao Chun-Feng Wu
Yuan-Hao Chang Hsiang-Shang Ko Yu-Fang Chen

Institute of Information Science, Academia Sinica, Taiwan

SCFTL overview

Simple and useful specification: the snapshot-consistent disk model

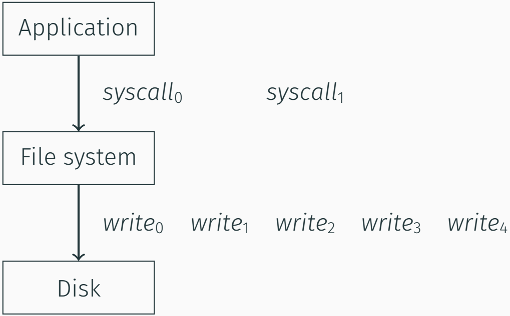
- guarantee snapshot consistency
- expose the standard read-write-flush interface

Good performance

- exploit the out-of-place update nature of FTLs
- use an efficient checkpointing algorithm

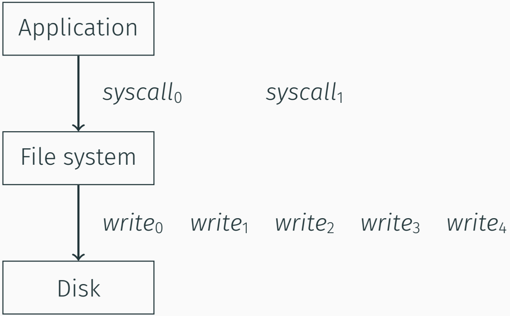
Formally verified against its specification

Storage stack



Storage stack

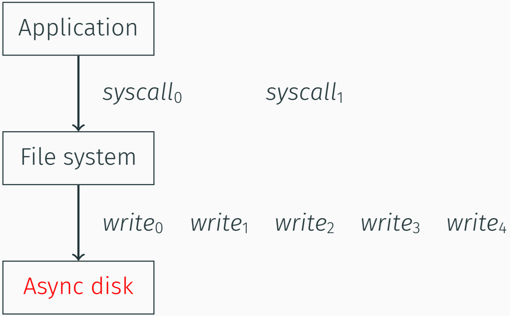
Correctness and performance of uppers layers are often determined by the guarantee provided by their lower layers. However, upper layers often have to make minimal assumptions about the lower layers.



Storage stack

Correctness and performance of uppers layers are often determined by the guarantee provided by their lower layers. However, upper layers often have to make minimal assumptions about the lower layers.

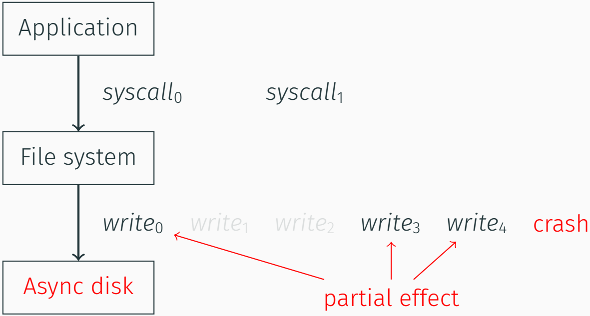
- For example, file systems usually assume the underlying disk follows the asynchronous disk model, which has no guarantees about writes after the last flush.



Storage stack

Correctness and performance of uppers layers are often determined by the guarantee provided by their lower layers. However, upper layers often have to make minimal assumptions about the lower layers.

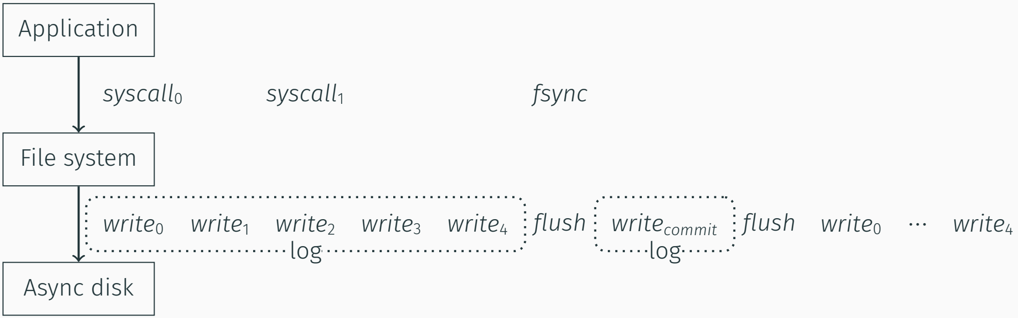
- For example, file systems usually assume the underlying disk follows the asynchronous disk model, which has no guarantees about writes after the last flush.



Crash recovery mechanism

File systems usually use a crash recovery mechanism (e.g., a write-ahead log).

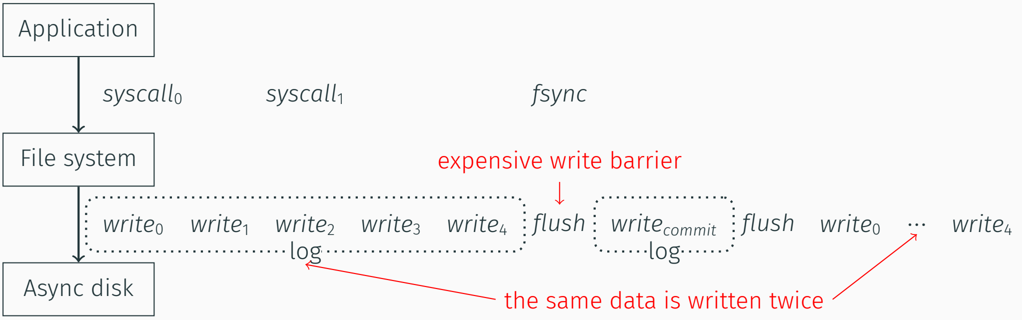
- Hard to get it right because the crash behavior is complicated: B3 (OSDI '18)
- Verified crash-safe file systems: FSCQ (SOSP '15), Yxv6 (OSDI '16), BilbyFS (ASPLOS '16), and DFSCQ (SOSP '17)



Crash recovery mechanism

File systems usually use a crash recovery mechanism (e.g., a write-ahead log).

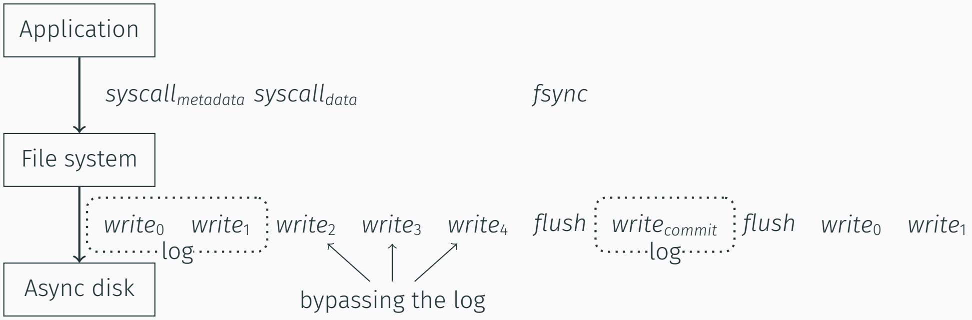
- Hard to get it right because the crash behavior is complicated: B3 (OSDI '18)
- Verified crash-safe file systems: FSCQ (SOSP '15), Yxv6 (OSDI '16), BilbyFS (ASPLOS '16), and DFSCQ (SOSP '17)
- Performance overhead due to data replication and write barriers



Reducing the overhead through optimizations

Some optimizations can reduce the overhead, but they often compromise the crash guarantee file systems are able to provide:

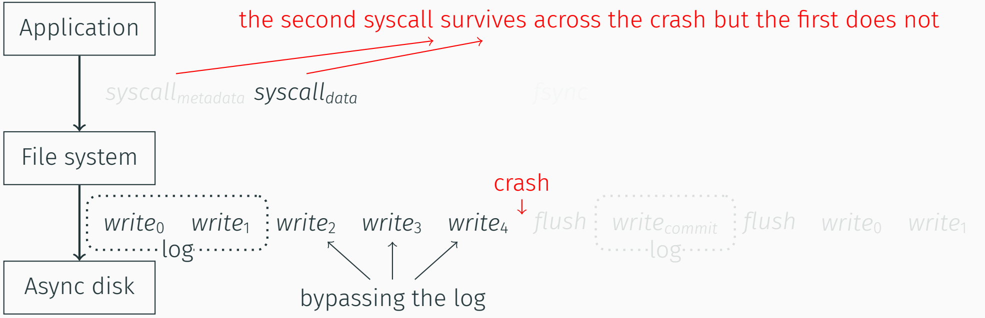
- bypassing the log for data breaks the order between metadata and data updates



Reducing the overhead through optimizations

Some optimizations can reduce the overhead, but they often compromise the crash guarantee file systems are able to provide:

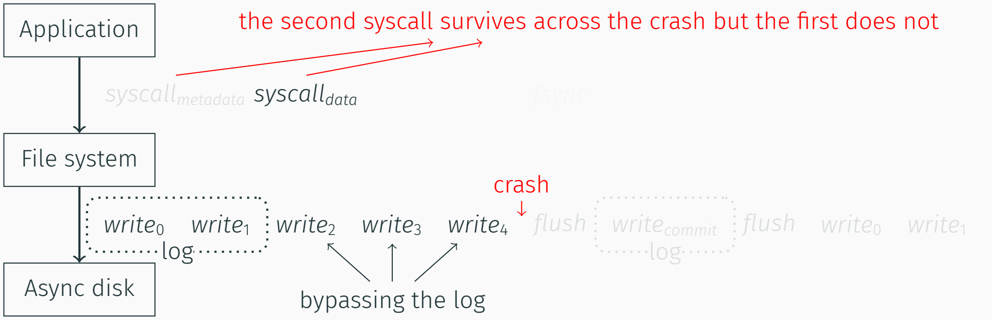
- bypassing the log for data breaks the order between metadata and data updates



Reducing the overhead through optimizations

Some optimizations can reduce the overhead, but they often compromise the crash guarantee file systems are able to provide:

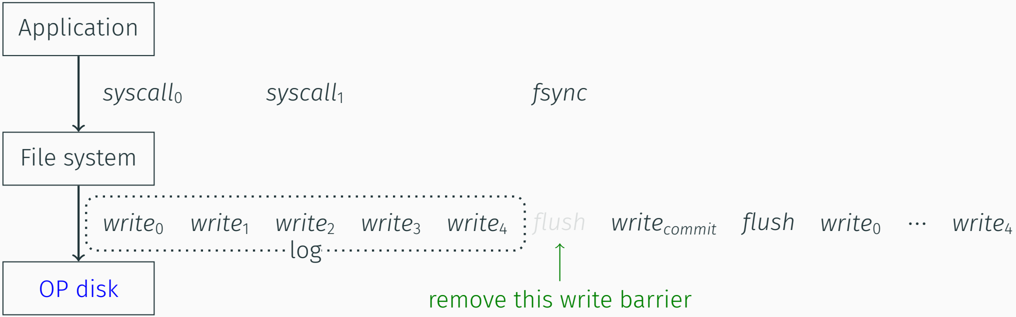
- bypassing the log for data breaks the order between metadata and data updates
- crash vulnerabilities in widely used applications (Pillai et al., OSDI '14) and ACID violations in database systems (Zheng et al., OSDI '14)



Prior work: Providing a stronger crash guarantee at the disk level

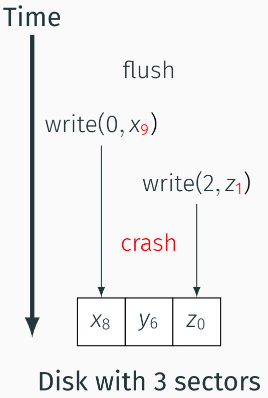
OPTR (ATC '19) and BarrierFS (FAST '18) propose order-preserving disk models, which preserve the order of disk operations across crashes.

- We can remove some flushes that are used to enforce ordering constraints

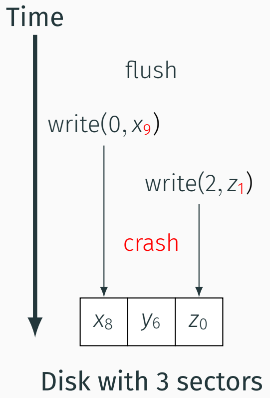


OP = order-preserving

Comparison of disk models



Comparison of disk models



Allowed crash behavior

(n is the number of writes after the last flush)

Asynchronous disk model: 2^n post-crash states

x_8	y_6	z_0
x_9	y_6	z_0
x_8	y_6	z_1
x_9	y_6	z_1

Order-preserving disk model (ATC '19): $n + 1$ post-crash states

x_8	y_6	z_0
x_9	y_6	z_0
x_8	y_6	z_1
x_9	y_6	z_1

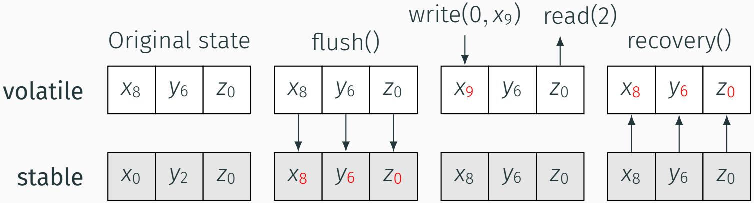
Snapshot-consistent disk model (this work): 1 post-crash state

x_8	y_6	z_0
x_9	y_6	z_0
x_8	y_6	z_1
x_9	y_6	z_1

Snapshot-consistent disk model

Moreover, we can easily model this behavior with two arrays:

- **volatile** represents the disk state observable to the users, and
- **stable** captures the disk state right before the last flush.

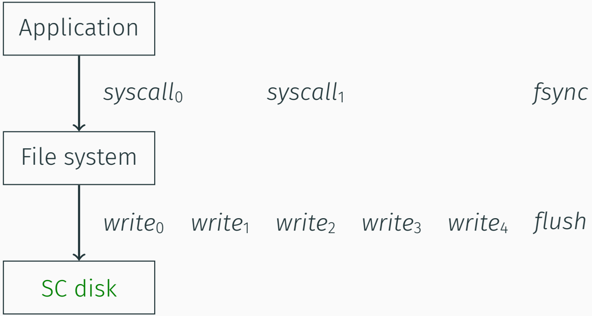


See the paper for the formal specification!

Snapshot consistency

Snapshot consistency ensures the **atomicity of multiple disk writes between two consecutive flushes**.

- File systems can simply call a flush to commit a transaction.
- Data are only written once and no write barriers are required.



SC = snapshot-consistent

SCFTL design highlight

Goals

- guarantee snapshot consistency
- maintain a good performance

Main techniques

- Checkpointing: remember the disk state right before the last flush
- Two-phase garbage collection (2PGC): prevent premature erasure

Flash memory and FTL basics

Flash memory has a few intrinsic device characteristics:

- a page has to be erased before being written to.
- the basic unit for write is a page,
- but the basic unit for erase is a block (multiple pages).

Most flash disks come with a flash translation layer (FTL), which

- implements the disk interface using flash operations, and
- performs out-of-place update with a logical-to-physical table (L2P).

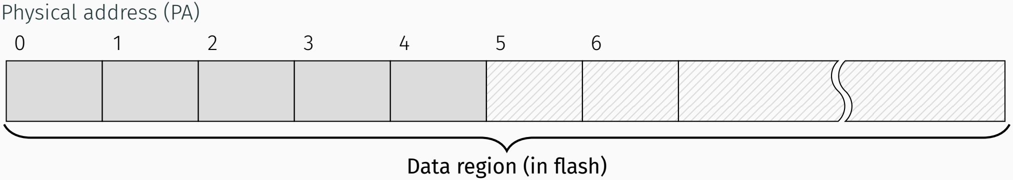
Out-of-place update with L2P

Handling request:

write(0,d)
write(1,d')
flush()

LA	PA
0	3
1	4
2	0

L2P (in memory)



Out-of-place update with L2P

Handling request:

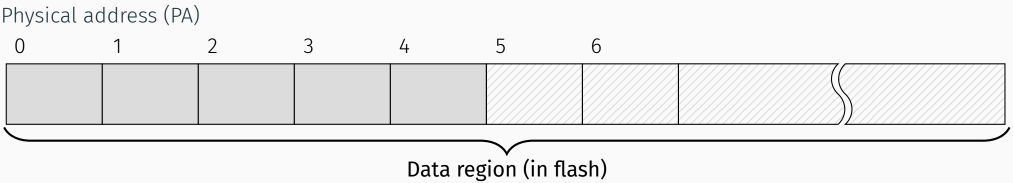
write(0,d)

write(1,d')

flush()

LA	PA
0	3
1	4
2	0

L2P (in memory)



Out-of-place update with L2P

Handling request:

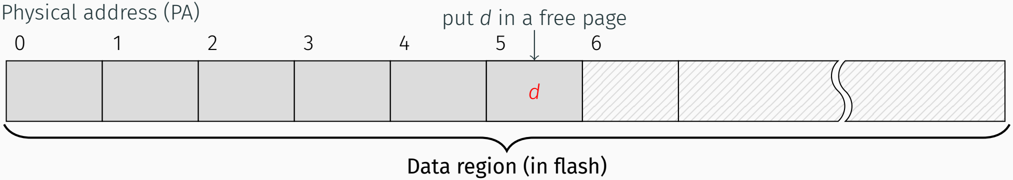
write(0,d)

write(1,d')

flush()

LA	PA
0	3
1	4
2	0

L2P (in memory)



Out-of-place update with L2P

Handling request:

write(0,d)

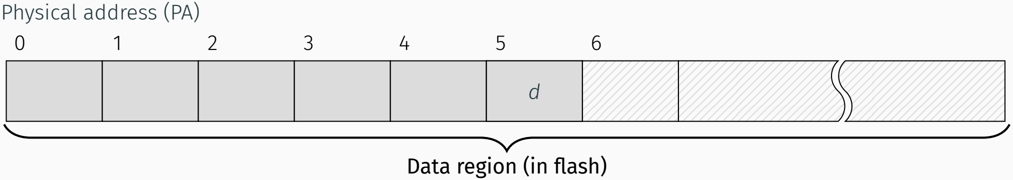
write(1,d')

flush()

LA	PA
0	5
1	4
2	0

←update the L2P entry to point to the page

L2P (in memory)



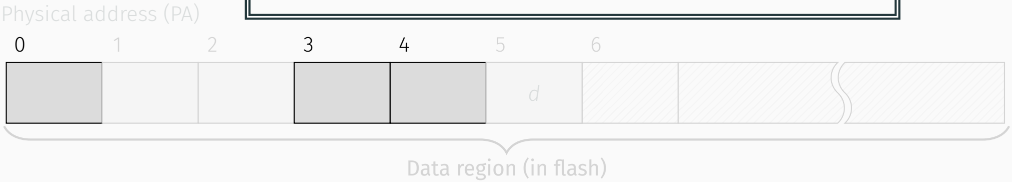
Out-of-place update with L2P

Handling request:
write(0,*d*)
write(1,*d′*)
flush()

LA	PA
0	5
1	4
2	0

L2P (in memory)

Notice that all the old data remain intact at their previous locations, so if SCFTL can remember where the old data are, then it can rollback to the previous disk state after recovery.



Remembering the disk state with the stable L2P

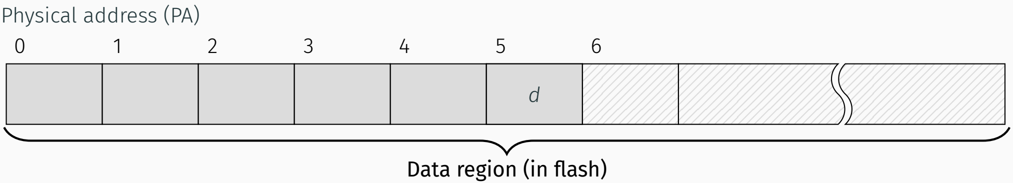
Handling request:
write(0,d)
write(1,d')
flush()

LA	PA
0	5
1	4
2	0

L2P (in memory)

LA	PA
0	3
1	4
2	0

Stable L2P (in flash)



Remembering the disk state with the stable L2P

Handling request:

write(0,d)

write(1,d')

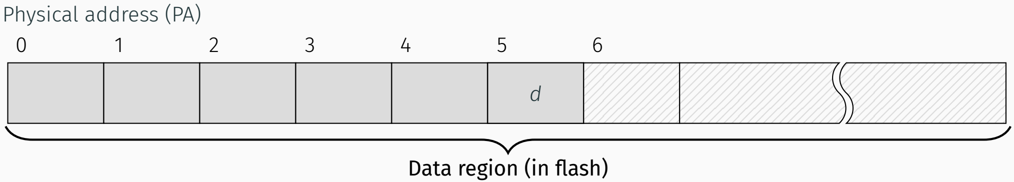
flush()

LA	PA
0	5
1	4
2	0

Volatile L2P (in memory)

LA	PA
0	3
1	4
2	0

Stable L2P (in flash)



Handling request:

write(0,d)

write(1,d')

flush()

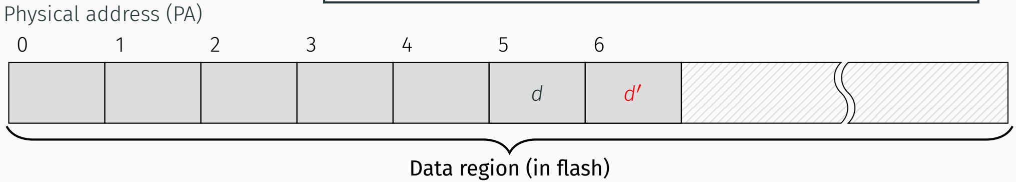
LA	PA
0	5
1	6
2	0

Volatile L2P (in memory)

LA	PA
0	3
1	4
2	0

Stable L2P (in flash)

> A write operation only modifies the **volatile L2P**, but not the **stable L2P**.



Remembering the disk state with the stable L2P

Handling request:
write(0,d)
write(1,d')
flush()

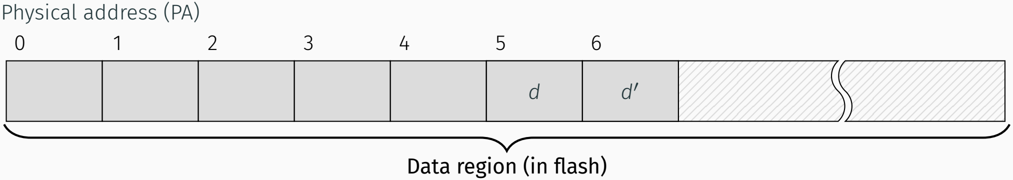
LA	PA
0	5
1	6
2	0

Volatile L2P (in memory)

LA	PA
0	5
1	6
2	0

Stable L2P (in flash)

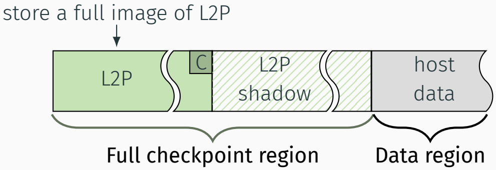
A flush operation copies the **volatile L2P** to the **stable L2P**.



Full checkpointing

LA	PA
0	5
1	6
2	0

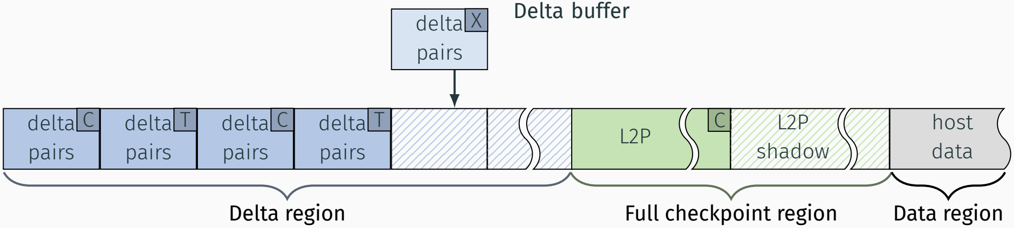
Stable L2P (in flash)



Delta checkpointing

LA	PA
0	5
1	6
2	0

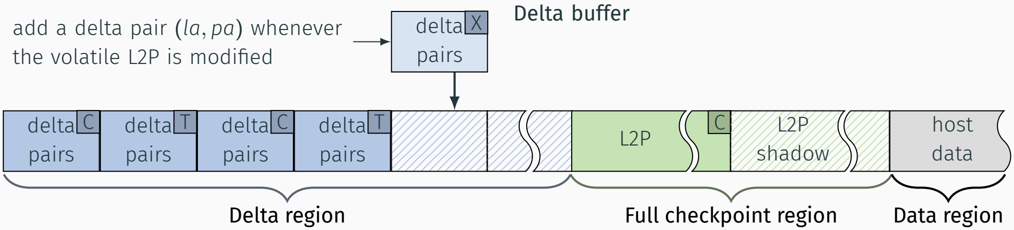
Stable L2P (in flash)



Delta checkpointing

LA	PA
0	5
1	6
2	0

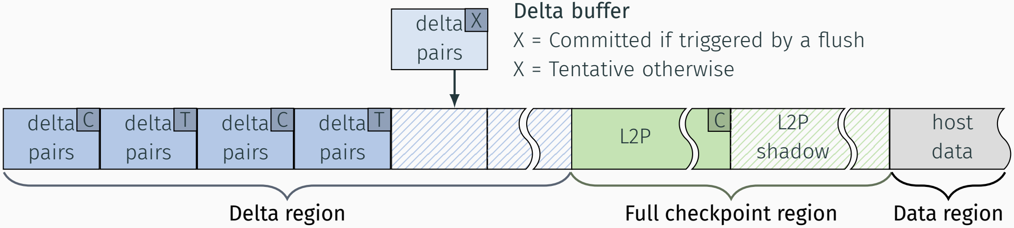
Stable L2P (in flash)



Delta checkpointing

LA	PA
0	5
1	6
2	0

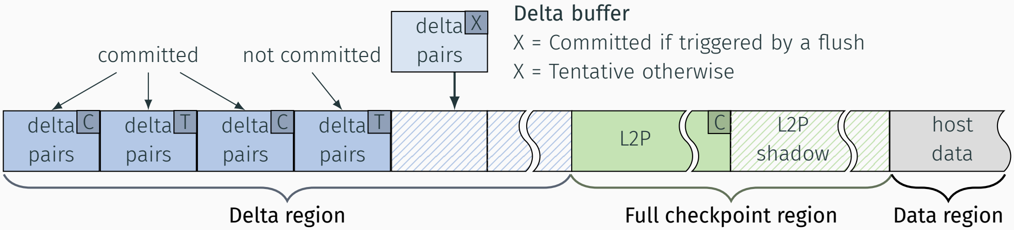
Stable L2P (in flash)



Delta checkpointing

LA	PA
0	5
1	6
2	0

Stable L2P (in flash)



Standard garbage collection (GC)

Standard GC workflow

- choose a victim block
- relocate valid data (those referred by the volatile L2P) in the victim block
- erase the victim block

Standard garbage collection (GC)

Standard GC workflow

- choose a victim block
- relocate valid data (those referred by the volatile L2P) in the victim block
- erase the victim block

Problem with standard GC: Data referred by the stable L2P may be deleted by the garbage collector.

Standard garbage collection (GC)

Standard GC workflow

- choose a victim block
- relocate valid data (those referred by the volatile L2P) in the victim block
- erase the victim block

Problem with standard GC: Data referred by the stable L2P may be deleted by the garbage collector.

Naïve solution: Also relocate data referred by the stable L2P.

- incur performance and memory overhead
- complicate the recovery procedure

Two-phase garbage collection (2PGC)

2PGC workflow (relocation phase)

- choose a victim block
- relocate valid data (those referred by the volatile L2P) in the victim block

2PGC workflow (erasure phase)

- erase the victim block

The erasure phase is **delayed until a flush is invoked**.

- the old data are no longer referred by the stable L2P after a flush.

Verifying SCFTL implementation against its specification

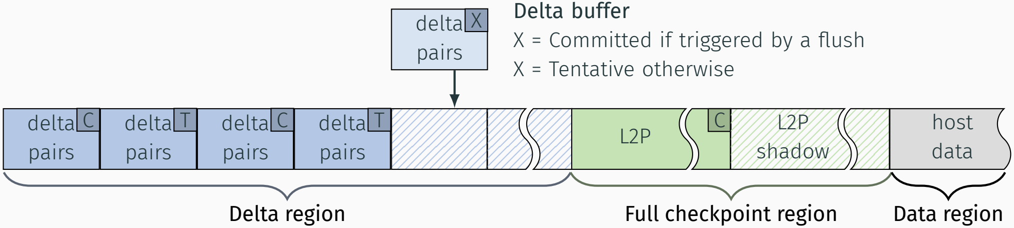
Implementation

LA	PA
0	3
1	4
2	0

L2P (in memory)

Specification

volatile	x_8	y_6	z_0
stable	x_0	y_2	z_0



Verifying SCFTL implementation against its specification

Implementation

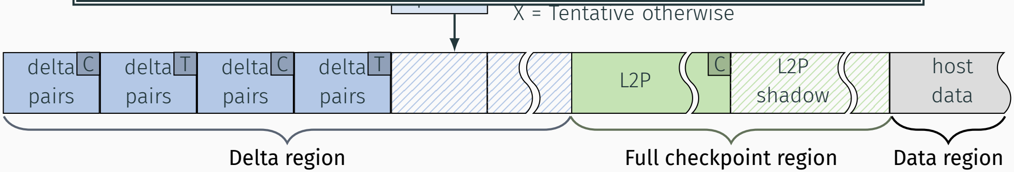
LA	PA
0	3
1	4
2	0

L2P (in memory)

Specification

volatile	x_8	y_6	z_0
stable	x_0	y_2	z_0

Abstraction relations capture how a state in the implementation is interpreted as a state in the specification.



Verifying SCFTL implementation against its specification

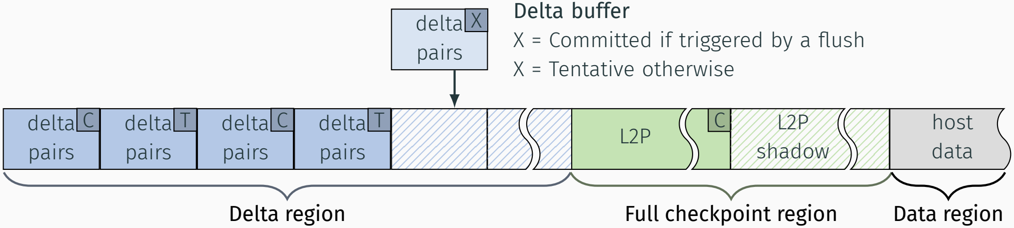
Implementation

LA	PA
0	3
1	4
2	0

L2P (in memory)

Specification

> **Representation invariants** describe properties that should be satisfied by various data structures in an implementation state.



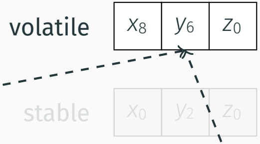
Examples: Abstration relations and representation invariants

Implementation

LA	PA
0	3
1	4
2	0

L2P (in memory)

Specification



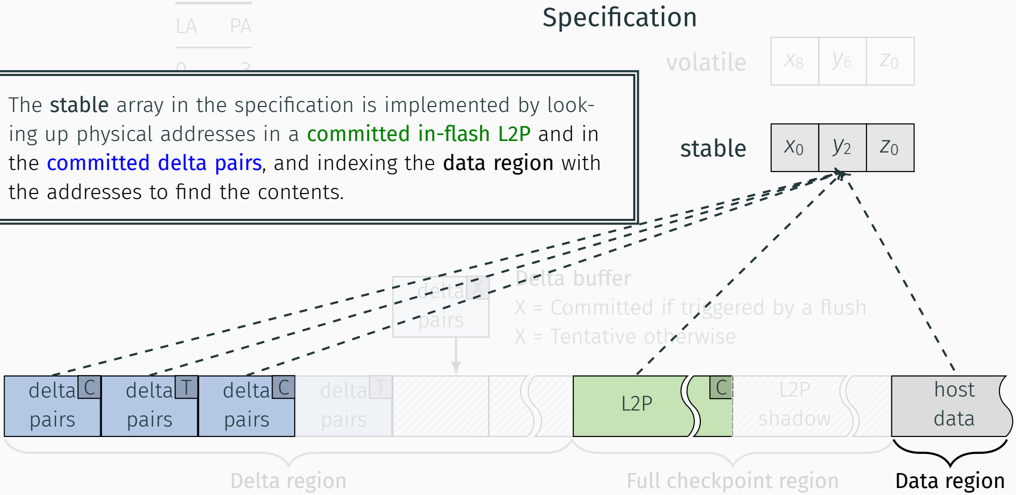
The **volatile** array in the specification is implemented by looking up physical addresses in the **in-memory L2P**, and indexing the **data region** with the addresses to find the contents.

Examples: Abstration relations and representation invariants

Implementation

Specification

The **stable** array in the specification is implemented by looking up physical addresses in a committed in-flash L2P and in the committed delta pairs, and indexing the **data region** with the addresses to find the contents.



Examples: Abstration relations and representation invariants

Implementation

LA	PA
0	3
1	4
2	0

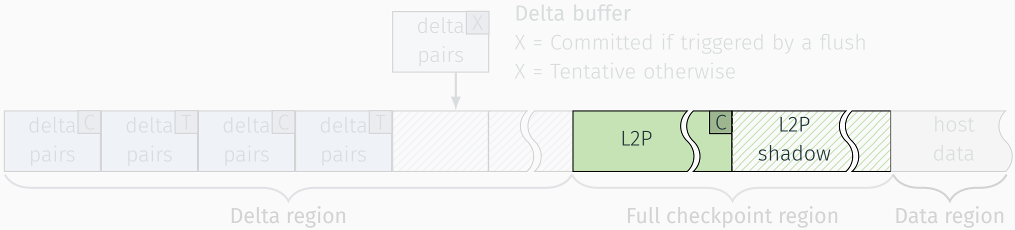
L2P (in

Specification

volatile x_8 y_6 z_0

stable x_0 y_2 z_0

At least one in-flash L2P is committed.



Examples: Abstration relations and representation invariants

Implementation

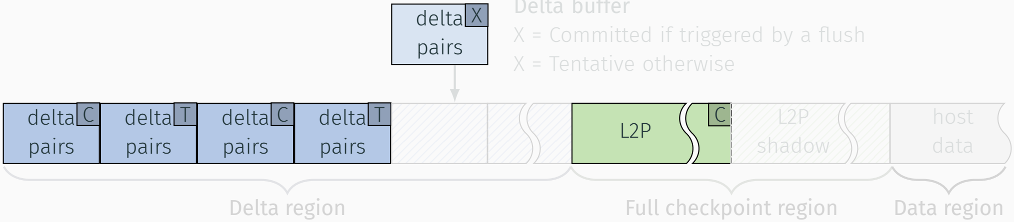
LA	PA
0	3
1	4
2	0

L2P (in memory)

Specification

volatile x_8 y_6 z_0

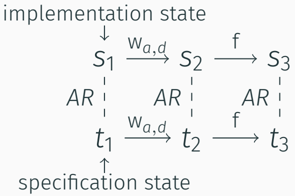
Starting with a **committed in-flash L2P** and applying all delta pairs in the **delta region** and in the **delta buffer** should yield the **in-memory L2P**.



Crash-safety simulation

With proper abstraction relations and representation invariants, we then use the symbolic executor Serval [SOSP '19] and the SMT solver Z3 to prove that the implementation and the specification establishes a forward simulation:

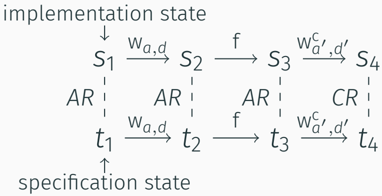
- successful operations preserve *AR* and *RI*



Crash-safety simulation

With proper abstraction relations and representation invariants, we then use the symbolic executor Serval [SOSP '19] and the SMT solver Z3 to prove that the implementation and the specification establishes a forward simulation:

- successful operations preserve *AR* and *RI*
- the relationship deteriorates to *CR* and *CI* on crashed operations

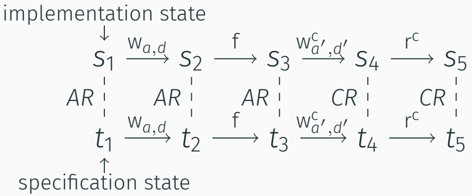


> The weaker abstraction relation *CR* and representation invariant *CI* should:
>
> - describe only the properties about flash states
> - be weak enough to hold even with crash reordering, but strong enough to allow a successful recovery

Crash-safety simulation

With proper abstraction relations and representation invariants, we then use the symbolic executor Serval [SOSP ’19] and the SMT solver Z3 to prove that the implementation and the specification establishes a forward simulation:

- successful operations preserve *AR* and *RI*
- the relationship deteriorates to *CR* and *CI* on crashed operations
- crashed recovery preserves *CR* and *CI*



Crash-safety simulation

With proper abstraction relations and representation invariants, we then use the symbolic executor Serval [SOSP ’19] and the SMT solver Z3 to prove that the implementation and the specification establishes a forward simulation:

- successful operations preserve *AR* and *RI*
- the relationship deteriorates to *CR* and *CI* on crashed operations
- crashed recovery preserves *CR* and *CI*
- the relationship restores to *AR* and *RI* on successful recovery

implementation state

$$
\begin{array}{ccccccccccccc}
 & s_1 & \xrightarrow{\mathrm{w}_{a,d}} & s_2 & \xrightarrow{\mathrm{f}} & s_3 & \xrightarrow{\mathrm{w}^{\mathrm{c}}_{a',d'}} & s_4 & \xrightarrow{\mathrm{r}^{\mathrm{c}}} & s_5 & \xrightarrow{\mathrm{r}} & s_6 & \longrightarrow \\
AR & \vdots & AR & \vdots & AR & \vdots & CR & \vdots & CR & \vdots & AR & \vdots & \cdots \\
 & t_1 & \xrightarrow{\mathrm{w}_{a,d}} & t_2 & \xrightarrow{\mathrm{f}} & t_3 & \xrightarrow{\mathrm{w}^{\mathrm{c}}_{a',d'}} & t_4 & \xrightarrow{\mathrm{r}^{\mathrm{c}}} & t_5 & \xrightarrow{\mathrm{r}} & t_6 & \longrightarrow
\end{array}
$$

specification state

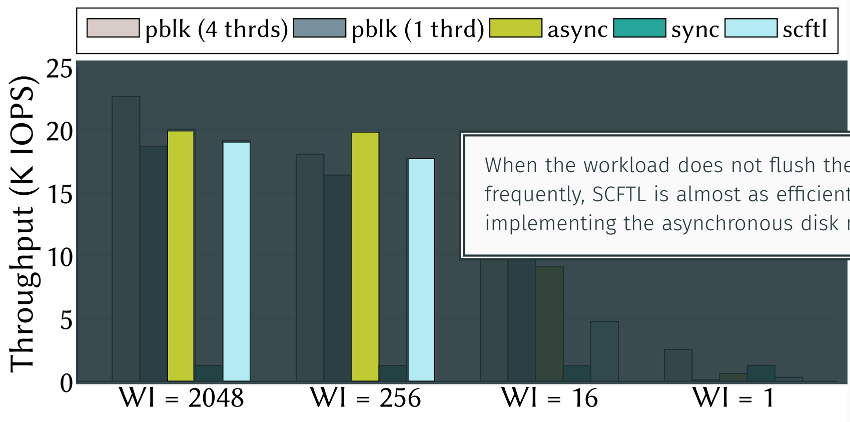
Evaluation: 4-KB random writes

We used the Linux LightNVM (FAST ’17) module to host our SCFTL and FEMU (FAST ’18) to emulate flash memory.

We used a disk workload that issues random write requests and invokes a flush for every given number of writes.

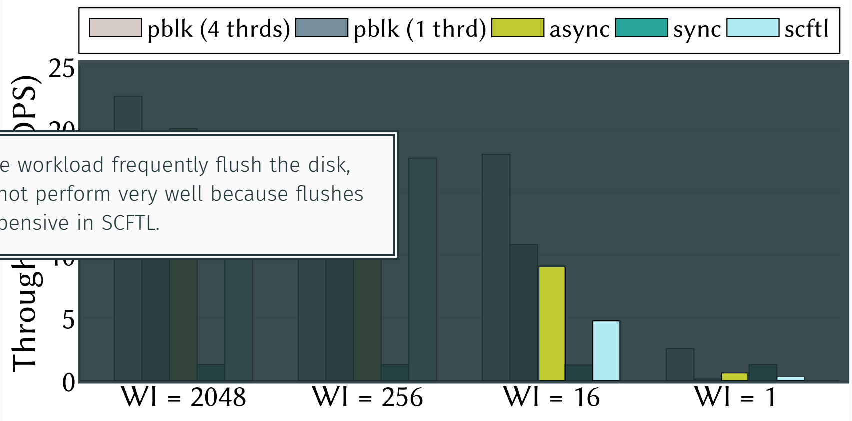
We compared SCFTL with another FTL whose design is similar to SCFTL, except the FTL follows the asynchronous disk model.

Evaluation: 4-KB random writes



When the workload does not flush the disk too frequently, SCFTL is almost as efficient as the FTL implementing the asynchronous disk model.

Evaluation: 4-KB random writes



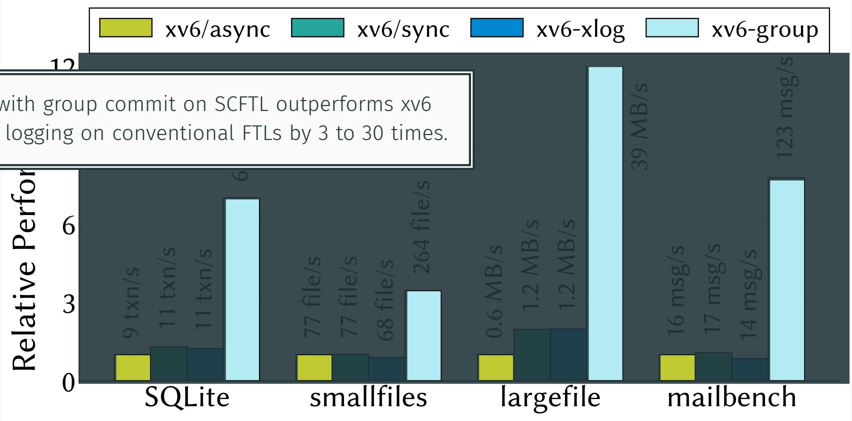
But when the workload frequently flush the disk, SCFTL does not perform very well because flushes are more expensive in SCFTL.

Evaluation: Modifying xv6 with SCFTL

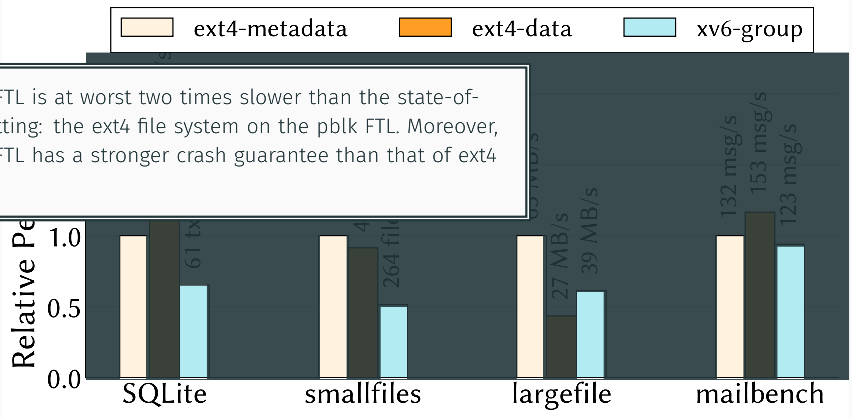
We modified the xv6 file system to support a standard optimization called **group commit** and to exploit the strong crash guarantee granted by SCFTL.

We used file system benchmarks to evalute this modification.

Evaluation: Modifying xv6 with SCFTL



xv6 with group commit on SCFTL outperforms xv6 with logging on conventional FTLs by 3 to 30 times.



xv6 on SCFTL is at worst two times slower than the state-of-the-art setting: the ext4 file system on the pblk FTL. Moreover, xv6 on SCFTL has a stronger crash guarantee than that of ext4 on pblk.

Conclusion

SCFTL provides a strong crash guarantee while maintaining a good performance, and its implementation is formally verified against its specification.

We demonstrate that starting at a lower-level of abstraction can make verifying crash safety easier while still resulting in an efficient system.

We plan to build an efficient storage stack by exploiting the benefits brought by SCFTL, and to extend SCFTL with common FTL optimizations, such as wear leveling and hot-cold data separation.

SCFTL is available at:

`https://github.com/yunshengtw/scftl`