Application Crash Consistency and Performance with CCFS

Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, Lanyue Lu,
Vijay Chidambaram, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

Application-Level Crash Consistency

Storage must be robust even with system crashes

- Power loss (2016 UPS issues: Github outage, Internet outage across UK)
  [source:www.datacenterknowledge.com]
- Kernel bugs [Lu et al., OSDI 2014, Palix et al., ASPLOS 2011, Chou et al., SOSP 2001]

Application-Level Crash Consistency

Storage must be robust even with system crashes

- Power loss (2016 UPS issues: Github outage, Internet outage across UK)
  [source:www.datacenterknowledge.com]
- Kernel bugs [Lu et al., OSDI 2014, Palix et al., ASPLOS 2011, Chou et al., SOSP 2001]

Applications need to implement crash consistency

- E.g., Database applications ensure transactions are atomic

Application-Level Crash Consistency

Storage must be robust even with system crashes

- Power loss (2016 UPS issues: Github outage, Internet outage across UK)
  [source:www.datacenterknowledge.com]
- Kernel bugs [Lu et al., OSDI 2014, Palix et al., ASPLOS 2011, Chou et al., SOSP 2001]

Applications need to implement crash consistency

- E.g., Database applications ensure transactions are atomic

Applications implement crash consistency wrongly

- Pillai et al., OSDI 2014 (11 applications) and Zhou et al., OSDI 2014 (8 databases)
- Conclusion: All applications had some form of incorrectness

Ordering and Application Consistency

App crash consistency depends on FS behavior

[Pillai et al., OSDI 2014]

- E.g., Bad FS behavior: 60 vulnerabilities in 11 applications
- Good FS behavior: 10 vulnerabilities in 11 applications

Ordering and Application Consistency

App crash consistency depends on FS behavior

[Pillai et al., OSDI 2014]

- E.g., Bad FS behavior: 60 vulnerabilities in 11 applications
- Good FS behavior: 10 vulnerabilities in 11 applications

FS-level ordering is important for applications

- All writes should (logically) be persisted in their issued order
- Major factor affecting application crash consistency

Ordering and Application Consistency

App crash consistency depends on FS behavior

[Pillai et al., OSDI 2014]

- E.g., Bad FS behavior: 60 vulnerabilities in 11 applications
- Good FS behavior: 10 vulnerabilities in 11 applications

FS-level ordering is important for applications

- All writes should (logically) be persisted in their issued order
- Major factor affecting application crash consistency

Few FS configurations provide FS-level ordering

- Ordering is considered bad for performance

In this paper ...

Stream abstraction

- Allows FS-level ordering with little performance overhead
- Needs a single, backward-compatible change to user code
- Flexible: More code changes improve performance

In this paper ...

Stream abstraction

- Allows FS-level ordering with little performance overhead
- Needs a single, backward-compatible change to user code
- Flexible: More code changes improve performance

Crash-Consistent File System (CCFS)

- Efficient implementation of stream abstraction on ext4
- High performance similar to ext4
- Noticeably higher crash consistency for applications

Outline

Introduction

Background

Stream API

Crash-Consistent File System

Evaluation

Conclusion

File-System Behavior

Each file system behaves differently across a crash

- Little standardization of behavior across crashes

File-System Behavior

Each file system behaves differently across a crash

- Little standardization of behavior across crashes

File-System Behavior

Each file system behaves differently across a crash

- Little standardization of behavior across crashes

FS Crash Behavior

Atomicity

Effects of a `write()` system call atomic on a system crash?

Ordering

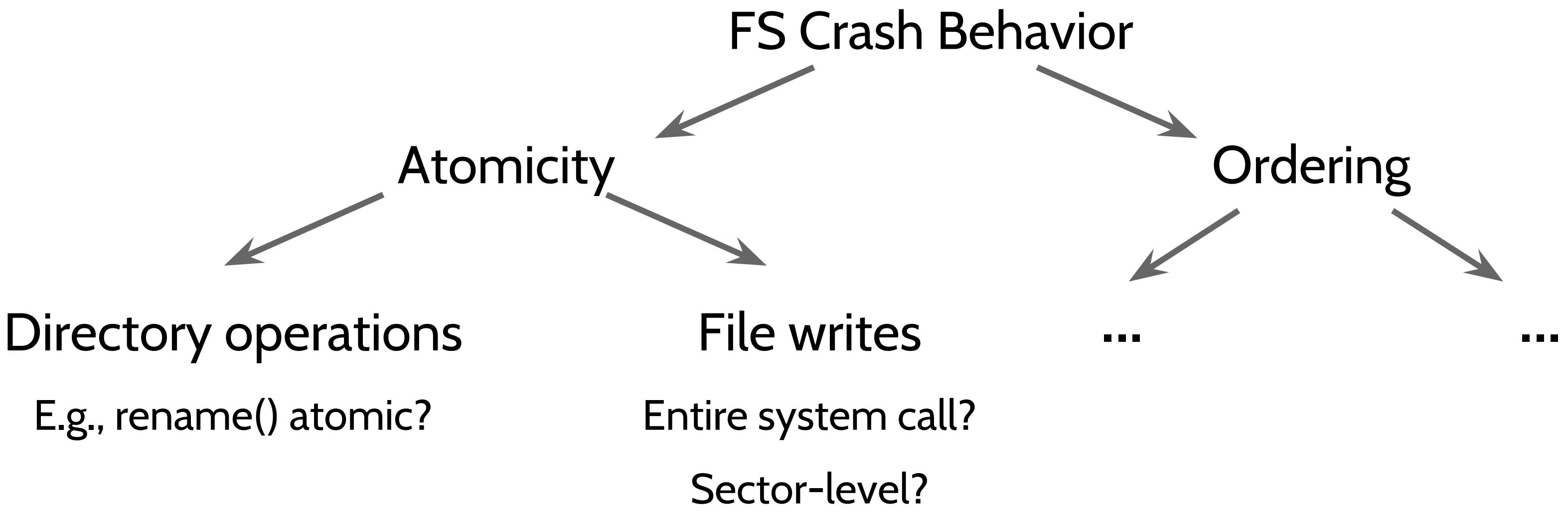
```
creat(A);
creat(B);
```

Possible after crash that B exists, but A does not?

File-System Behavior

Each file system behaves differently across a crash

- Little standardization of behavior across crashes



Vulnerabilities Study

Previous work: App crash consistency vs FS behavior

[Pillai et al., OSDI 2014]

Vulnerabilities Study

Previous work: App crash consistency vs FS behavior

[Pillai et al., OSDI 2014]

"*Vulnerability*": Place in application source code that can lead to inconsistency, depending on FS behavior

Vulnerabilities Study: Results

	Ext2-like FS	Btrfs	Ext3-DJ
LevelDB-1.10	10	4	1
LevelDB-1.15	6	3	1
LMDB	1		
GDBM	5	4	2
HSQLDB	10	4	
SQLite-Roll	1	1	1
SQLite-WAL	0		
PostgreSQL	1		
Git	9	5	2
Mercurial	10	8	3
VMWare	1		
HDFS	2	1	
ZooKeeper	4	1	
Total	60	31	10

Vulnerabilities Study: Results

File systems

Applications

	Ext2-like FS	Btrfs	Ext3-DJ
LevelDB-1.10	10	4	1
LevelDB-1.15	6	3	1
LMDB	1		
GDBM	5	4	2
HSQLDB	10	4	
SQLite-Roll	1	1	1
SQLite-WAL	0		
PostgreSQL	1		
Git	9	5	2
Mercurial	10	8	3
VMWare	1		
HDFS	2	1	
ZooKeeper	4	1	
Total	60	31	10

Vulnerabilities under safest application configuration

Vulnerabilities Study: Results

	Ext2-like FS	Btrfs	Ext3-DJ
Ordering	✗	✗	✔
Atomicity	✗	✔	✔

← File-system behavior

	Ext2-like FS	Btrfs	Ext3-DJ
LevelDB-1.10	10	4	1
LevelDB-1.15	6	3	1
LMDB	1		
GDBM	5	4	2
HSQLDB	10	4	
SQLite-Roll	1	1	1
SQLite-WAL	0		
PostgreSQL	1		
Git	9	5	2
Mercurial	10	8	3
VMWare	1		
HDFS	2	1	
ZooKeeper	4	1	
Total	60	31	10

Vulnerabilities Study: Results

	Ext2-like FS	Btrfs	Ext3-DJ
Ordering	✗	✗	✔
Atomicity	✗	✔	✔
LevelDB-1.10	10	4	1
LevelDB-1.15	6	3	1
LMDB	1		
GDBM	5	4	2
HSQLDB	10	4	
SQLite-Roll	1	1	1
SQLite-WAL	0		
PostgreSQL	1		
Git	9	5	2
Mercurial	10	8	3
VMWare	1		
HDFS	2	1	
ZooKeeper	4	1	
Total	60	31	10

Under FS with few guarantees of atomicity and ordering, 60 vulnerabilities are exposed

- Serious consequences: unavailability, data loss

Vulnerabilities Study: Results

	Ext2-like FS	Btrfs	Ext3-DJ
Ordering	✗	✗	✔
Atomicity	✗	✔	✔
LevelDB-1.10	10	4	1
LevelDB-1.15	6	3	1
LMDB	1		
GDBM	5	4	2
HSQLDB	10	4	
SQLite-Roll	1	1	1
SQLite-WAL	0		
PostgreSQL	1		
Git	9	5	2
Mercurial	10	8	3
VMWare	1		
HDFS	2	1	
ZooKeeper	4	1	
Total	60	31	10

Mercurial (Btrfs, 8) → Repository corruption

ZooKeeper (Btrfs, 1) → Unavailability

Under btrfs, with atomicity but lots of re-ordering, 31 vulnerabilities

- Serious consequences

Vulnerabilities Study: Results

	Ext2-like FS	Btrfs	Ext3-DJ
Ordering	✗	✗	✔
Atomicity	✗	✔	✔
LevelDB-1.10	10	4	1
LevelDB-1.15	6	3	1
LMDB	1		
GDBM	5	4	2
HSQLDB	10	4	
SQLite-Roll	1	1	1
SQLite-WAL	0		
PostgreSQL	1		
Git	9	5	2
Mercurial	10	8	3
VMWare	1		
HDFS	2	1	
ZooKeeper	4	1	
Total	60	31	10

SQLite-Roll (Ext3-DJ: 1) → Documentation error

Mercurial (Ext3-DJ: 3) → Dirstate corruption

Under data-journaled ext3, with both atomicity and ordering, 10 vulnerabilities

- Minor consequences

Real-world vs Ideal FS behavior

Ideal behavior: Ordering, “weak atomicity”

- *All* file system updates should be persisted in-order
- Writes can split at sector boundary; everything else atomic

Real-world vs Ideal FS behavior

Ideal behavior: Ordering, “weak atomicity”

- *All* file system updates should be persisted in-order
- Writes can split at sector boundary; everything else atomic

Modern file systems already provide weak atomicity

- E.g.: Default modes of ext4, btrfs, xfs

Real-world vs Ideal FS behavior

Ideal behavior: Ordering, “weak atomicity”

- *All* file system updates should be persisted in-order
- Writes can split at sector boundary; everything else atomic

Modern file systems already provide weak atomicity

- E.g.: Default modes of ext4, btrfs, xfs

Only rarely used FS configurations provide ordering

- E.g.: Data-journaling mode of ext4, ext3

Background: Summary

File-system behavior affects application consistency

- Behavior is not standardized
- 60 vulnerabilities with ext2-like FS; 10 with well-behaved FS

Desired behavior: Ordering and weak atomicity

- Weak atomicity already provided by modern file systems
- Ordering provided only by rarely-used FS configurations

Outline

Introduction

Background

Stream API

Crash-Consistent File System

Evaluation

Conclusion

Why not use an order-preserving FS?

Some existing file systems preserve order

- Example: ext3 and ext4 under data-journaling mode
- Performance overhead?

Why not use an order-preserving FS?

Some existing file systems preserve order

- Example: ext3 and ext4 under data-journaling mode
- Performance overhead?

New techniques are efficient in maintaining order

- CoW, optimized forms of journaling
- Ordering doesn’t require disk-level seeks

Why not use an order-preserving FS?

Some existing file systems preserve order

- Example: ext3 and ext4 under data-journaling mode
- Performance overhead?

New techniques are efficient in maintaining order

- CoW, optimized forms of journaling
- Ordering doesn't require disk-level seeks

Reason: False ordering dependencies

- Inherent overhead of ordering, irrespective of technique used

False Ordering Dependencies

Application A

Application B

False Ordering Dependencies

Time	Application A	Application B
1	`pwrite(f1, 0, 150 MB);`	

False Ordering Dependencies

Time	Application A	Application B
1	`pwrite(f1, 0, 150 MB);`	
2		`write(f2, “hello”);`
3		`write(f3, “world”);`

False Ordering Dependencies

Time	Application A	Application B
1	`pwrite(f1, 0, 150 MB);`	
2		`write(f2, "hello");`
3		`write(f3, "world");`
4		`fsync(f3);`

False Ordering Dependencies

In a globally ordered file system …

Time	Application A	Application B
1	`pwrite(f1, 0, 150 MB);`	
2		`write(f2, "hello");`
3		`write(f3, "world");`
4		`fsync(f3);`

write(f1) ***has*** to be sent to disk before write(f2)

False Ordering Dependencies

In a globally ordered file system …

Time	Application A	Application B
1	`pwrite(f1, 0, 150 MB);`	
2		`write(f2, “hello”);`
3		`write(f3, “world”);`
4		`fsync(f3);`

2 seconds, irrespective of implementation used to get ordering!

False Ordering Dependencies

Problem: Ordering between independent applications

In a globally ordered file system …

Time	Application A	Application B
1	`pwrite(f1, 0, 150 MB);`	
2		`write(f2, "hello");`
3		`write(f3, "world");`
4		`fsync(f3);` ← 2 seconds, irrespective of implementation used to get ordering!

False Ordering Dependencies

Problem: Ordering between independent applications

Solution: Order only within each application

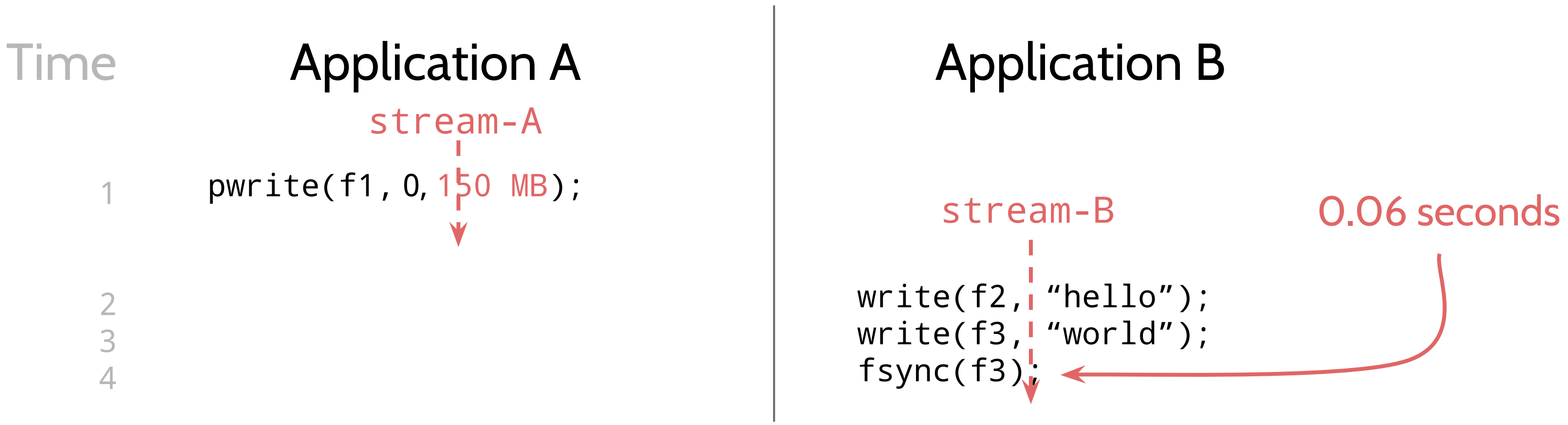
- Avoids performance overhead, provides app consistency

Time	Application A	Application B
1	`pwrite(f1, 0, 150 MB);`	
2		`write(f2, “hello”);`
3		`write(f3, “world”);`
4		`fsync(f3);`

Stream Abstraction

New abstraction: Order only within a *"stream"*

- Each application is usually put into a separate stream



Stream API: Normal Usage

New set_stream() call

- All updates after set_stream(X) associated with stream X
- When process forks, previous stream is adopted

Time	Application A	Application B
	`set_stream(A)`	`set_stream(B)`
1	`pwrite(f1, 0, 150 MB);`	
2		`write(f2, “hello”);`
3		`write(f3, “world”);`
4		`fsync(f3);`

Stream API: Normal Usage

New set_stream() call

- All updates after set_stream(X) associated with stream X
- When process forks, previous stream is adopted

Using streams is easy

- Add a single set_stream() call in beginning of application
- Backward-compatible: set_stream() is no-op in older FSes

Stream API: Extended Usage

set_stream() is versatile

- Many applications can be assigned the same stream
- Threads within an application can use different streams
- Single thread can keep switching between streams

Stream API: Extended Usage

set_stream() is versatile

- Many applications can be assigned the same stream
- Threads within an application can use different streams
- Single thread can keep switching between streams

Ordering vs durability: stream_sync(), IGNORE_FSYNC flag

- Applications use fsync() for both ordering and durability [Chidambaram et al., SOSP2013]
- IGNORE_FSYNC ignores fsync(), respects stream_sync()

Streams: Summary

In an ordered FS, false dependencies cause overhead

- Inherent overhead, independent of technique used

Streams provide order only within application

- Writes across applications can be re-ordered for performance
- For consistency, ordering required only within application

Easy to use!

Outline

Introduction

Background

Stream API

Crash-Consistent File System

Evaluation

Conclusion

CCFS: Design

"Crash consistent file system"

- Efficient implementation of stream abstraction

CCFS: Design

“Crash consistent file system”

- Efficient implementation of stream abstraction

Basic design: Based on ext4 with data-journaling

- Ext4 data-journaling guarantees global ordering
- Ordering across all applications: false dependencies
- CCFS uses separate transactions for each stream

CCFS: Design

“Crash consistent file system”

- Efficient implementation of stream abstraction

Basic design: Based on ext4 with data-journaling

- Ext4 data-journaling guarantees global ordering
- Ordering across all applications: false dependencies
- CCFS uses separate transactions for each stream

Multiple challenges

Ext4 Journaling: Global Order

Ext4 has 1) main-memory structure, “running transaction”,

2) on-disk journal structure

Running transaction

Main memory

On-disk journal

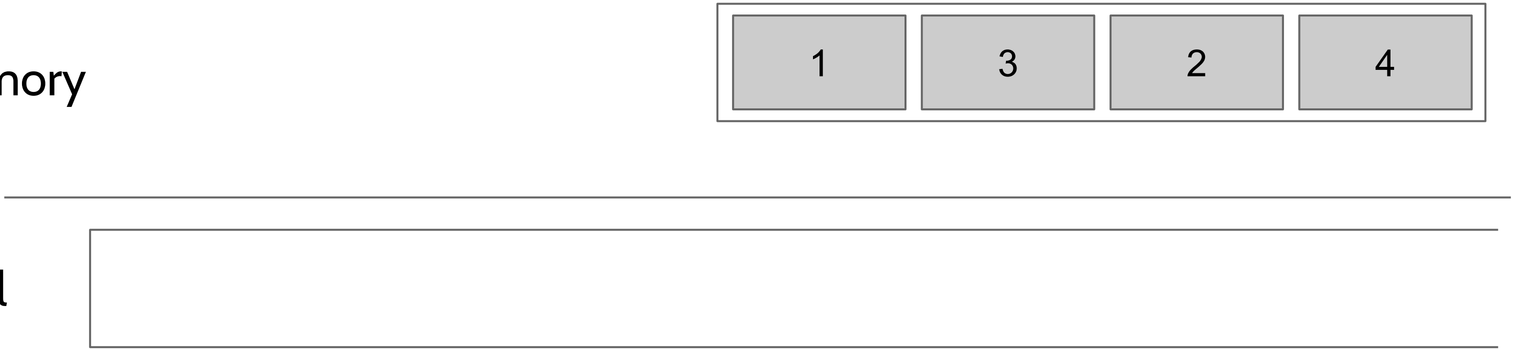
Ext4 Journaling: Global Order

Application modifications recorded in main-memory running transaction

Application A	Application B
`Modify blocks #1,#3`	
	`Modify blocks #2,#4`

Running transaction

Main memory

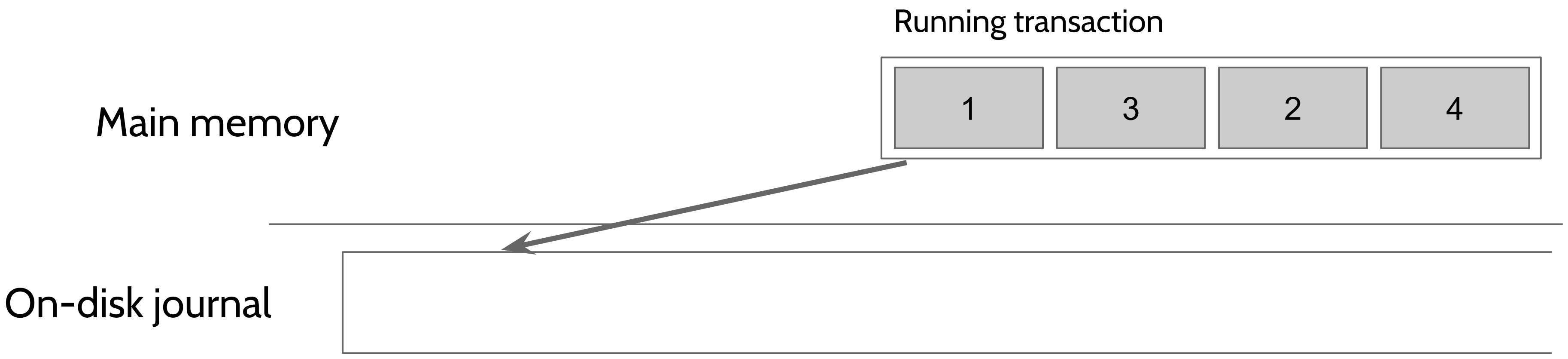


On-disk journal

Ext4 Journaling: Global Order

On fsync() call, running transaction "committed" to on-disk journal

Application A	Application B
`Modify blocks #1,#3`	
	`Modify blocks #2,#4` `fsync()`



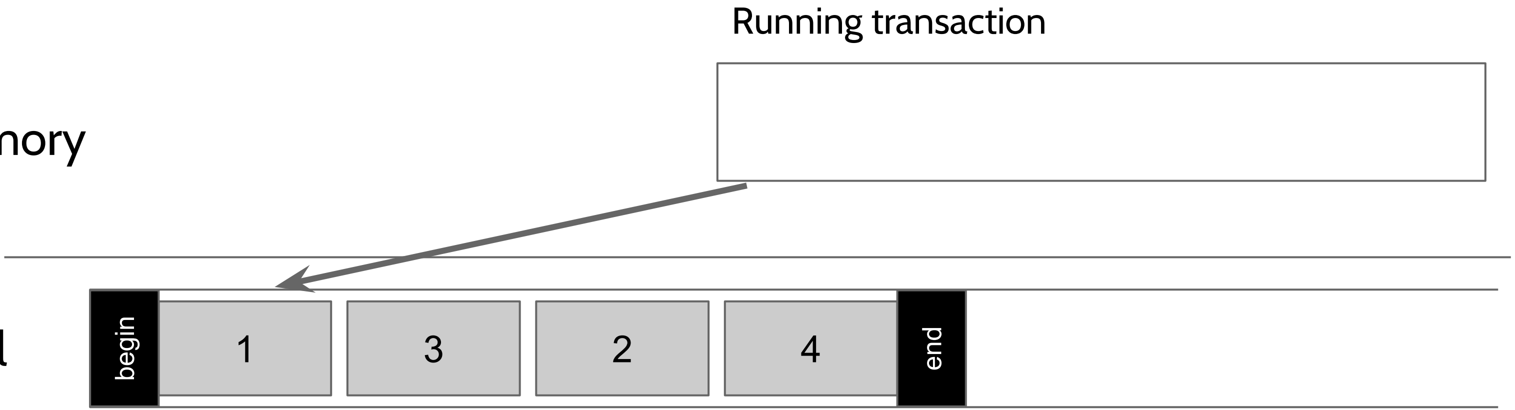
Ext4 Journaling: Global Order

On fsync() call, running transaction “committed” to on-disk journal

Application A	Application B
`Modify blocks #1,#3`	
	`Modify blocks #2,#4` `fsync()`

Main memory

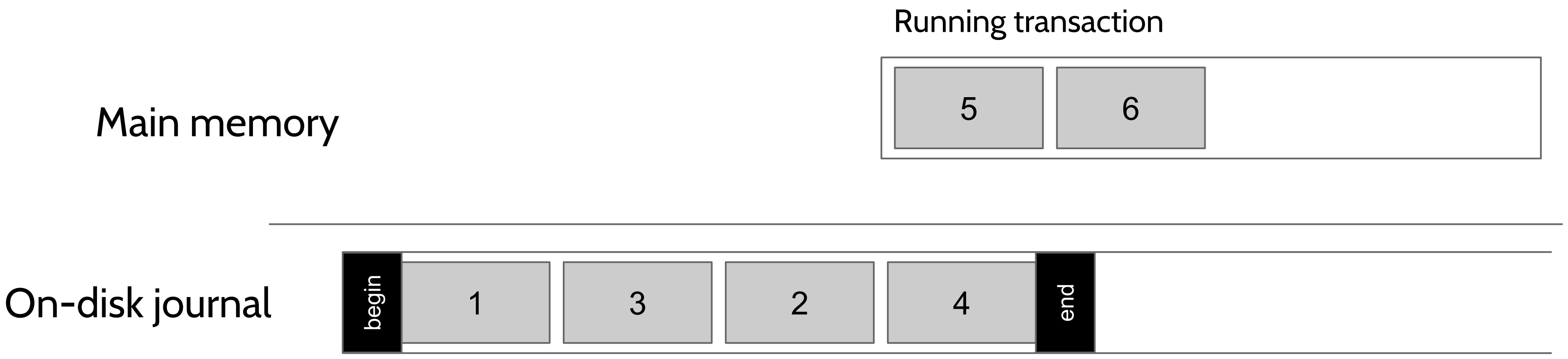
On-disk journal



Ext4 Journaling: Global Order

Further application writes recorded in new running transaction and committed

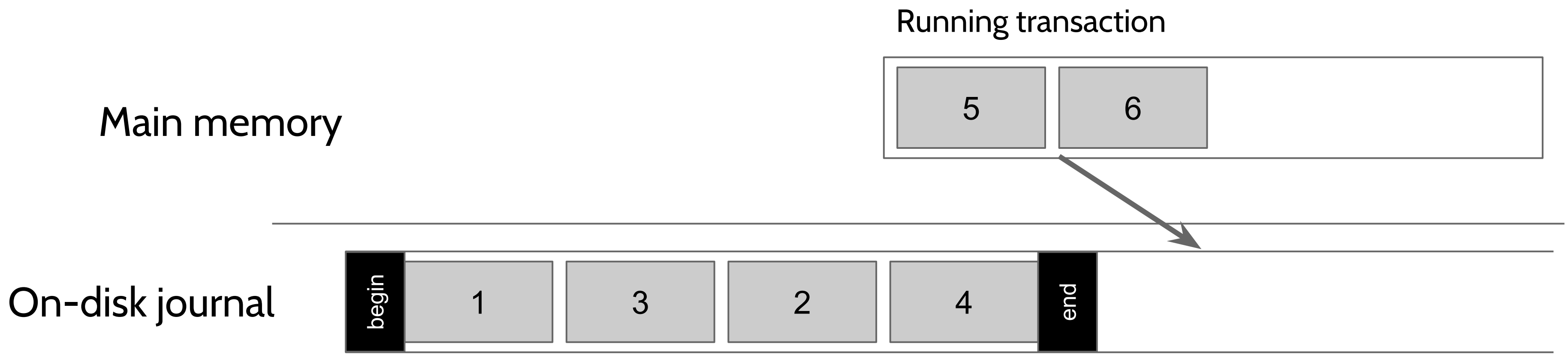
Application A	Application B
`Modify blocks #1,#3`	
	`Modify blocks #2,#4` `fsync()`
`Modify blocks #5,#6`	



Ext4 Journaling: Global Order

Further application writes recorded in new running transaction and committed

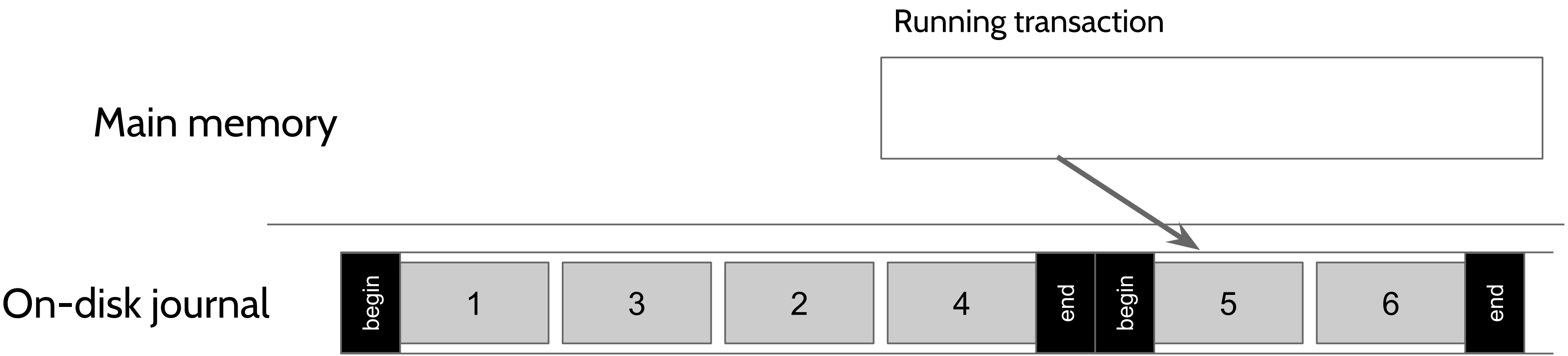
Application A	Application B
`Modify blocks #1,#3`	
	`Modify blocks #2,#4` `fsync()`
`Modify blocks #5,#6`	



Ext4 Journaling: Global Order

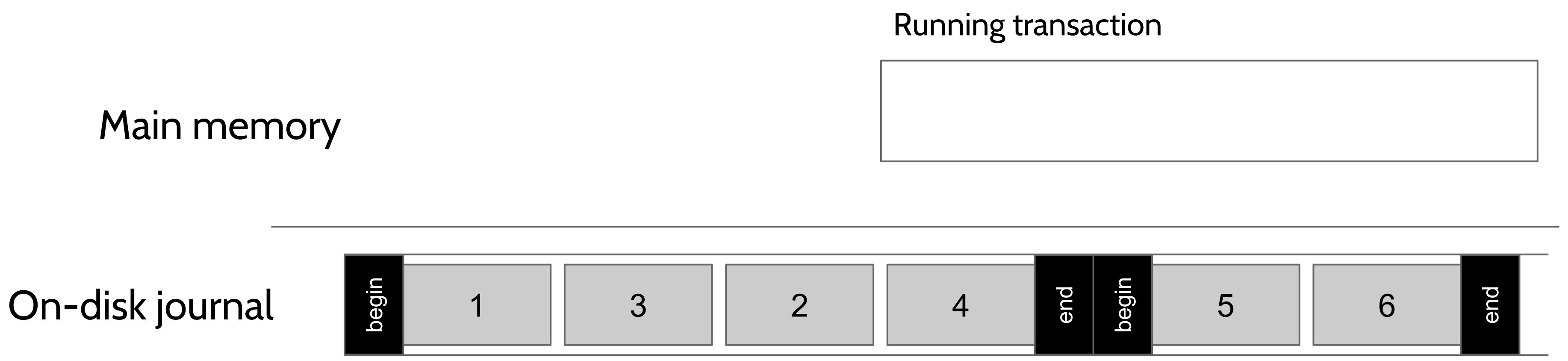
Further application writes recorded in new running transaction and committed

Application A	Application B
`Modify blocks #1,#3`	
	`Modify blocks #2,#4` `fsync()`
`Modify blocks #5,#6`	



Ext4 Journaling: Global Order

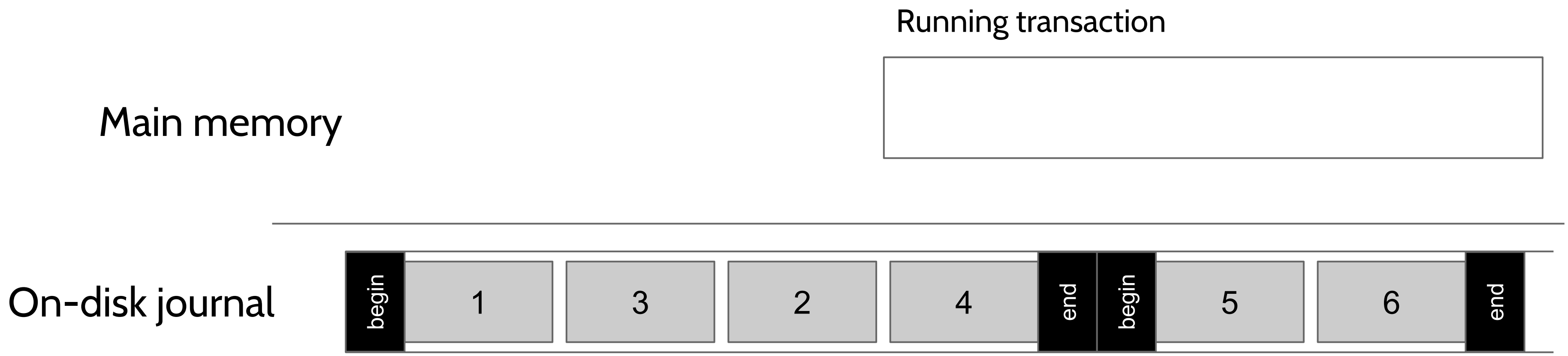
On system crash, on-disk journal transactions recovered atomically, in sequential order



Ext4 Journaling: Global Order

On system crash, on-disk journal transactions recovered atomically, in sequential order

Global ordering is maintained!



CCFS: Stream Order

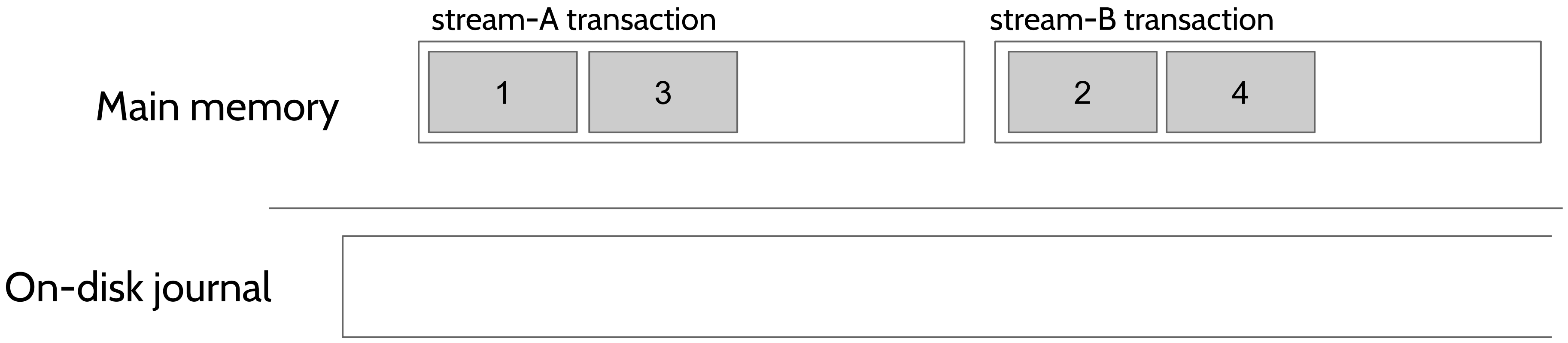
CCFS maintains separate running transaction per stream

Application A

```
set_stream(A)
Modify blocks #1,#3
```

Application B

```
set_stream(B)

Modify blocks #2,#4
```



CCFS: Stream Order

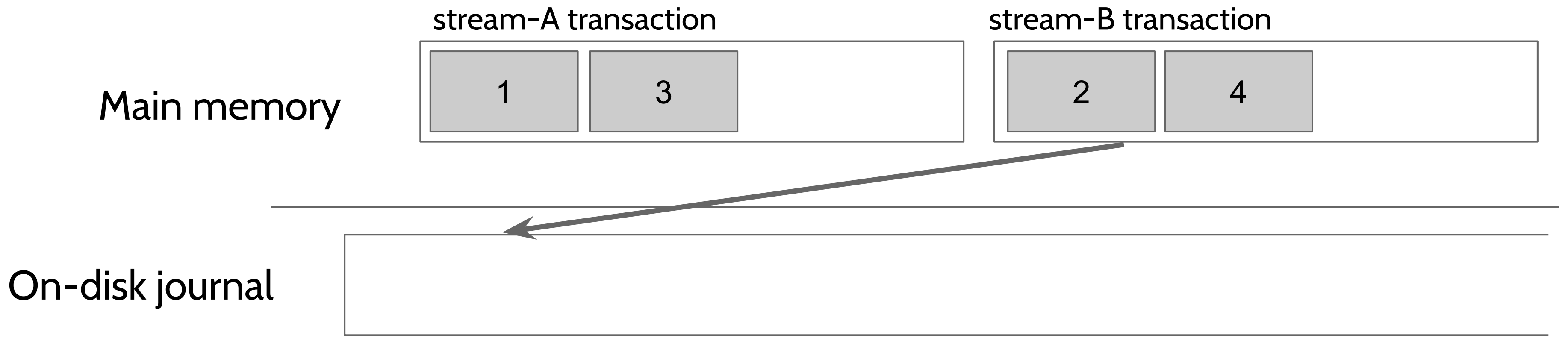
On fsync(), only that stream is committed

Application A

```
set_stream(A)
Modify blocks #1,#3
```

Application B

```
set_stream(B)

Modify blocks #2,#4
fsync()
```



CCFS: Stream Order

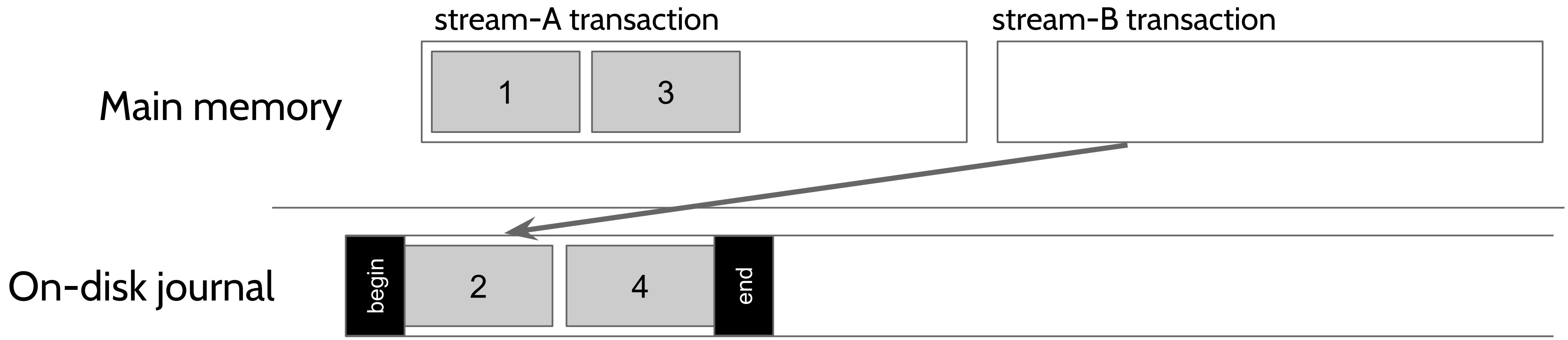
On fsync(), only that stream is committed

Application A

```
set_stream(A)
Modify blocks #1,#3
```

Application B

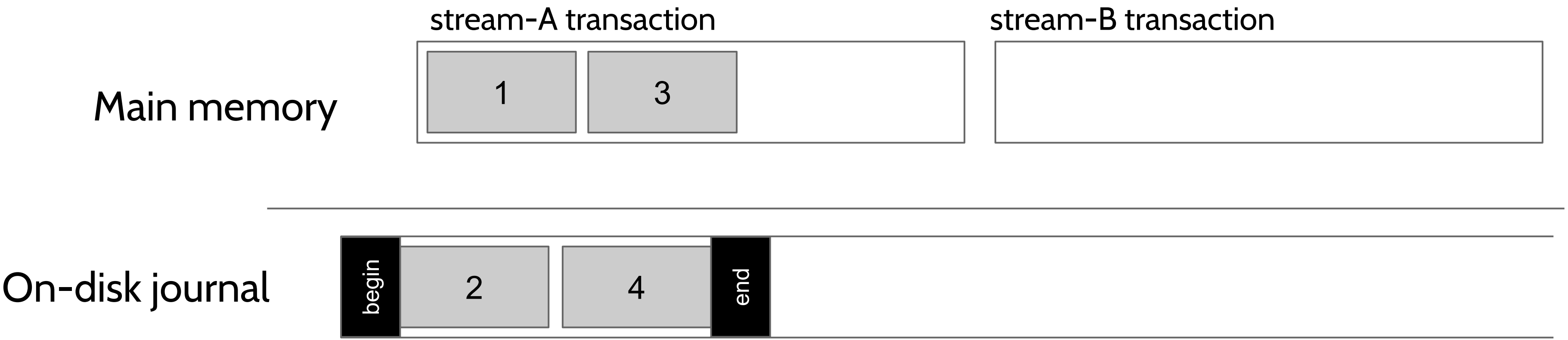
```
set_stream(B)

Modify blocks #2,#4
fsync()
```



CCFS: Stream Order

Ordering maintained within stream, re-order across streams!

Application A	Application B
`set_stream(A)`	`set_stream(B)`
`Modify blocks #1,#3`	
	`Modify blocks #2,#4`
	`fsync()`



CCFS: Multiple Challenges

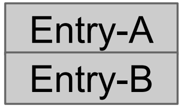
Example: Two streams updating adjoining dir-entries

Application A	Application B
`set_stream(A)`	`set_stream(B)`
`create(/X/A)`	
	`create(/X/B)`

CCFS: Multiple Challenges

Example: Two streams updating adjoining dir-entries

Block-1 (belonging to directory X)



Application A

```
set_stream(A)
create(/X/A)
```

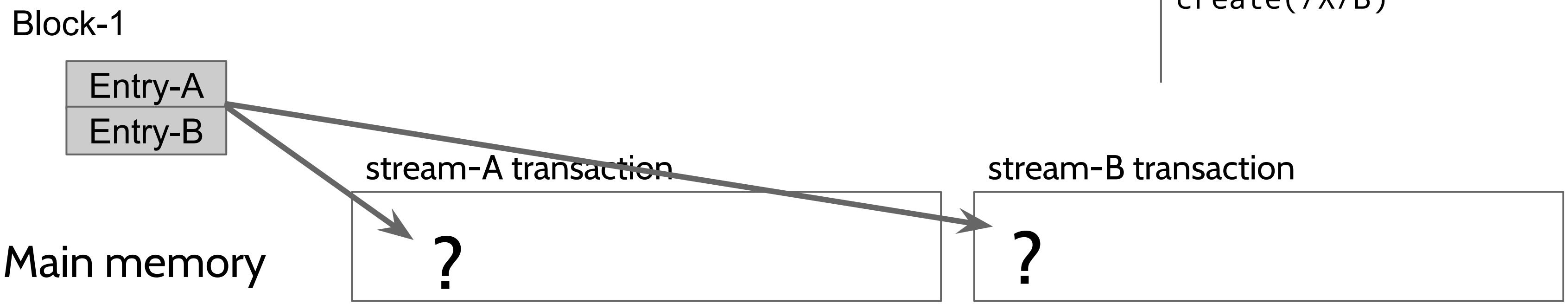
Application B

```
set_stream(B)

create(/X/B)
```

Challenge #1: Block-Level Journaling

Two independent streams can update same block!

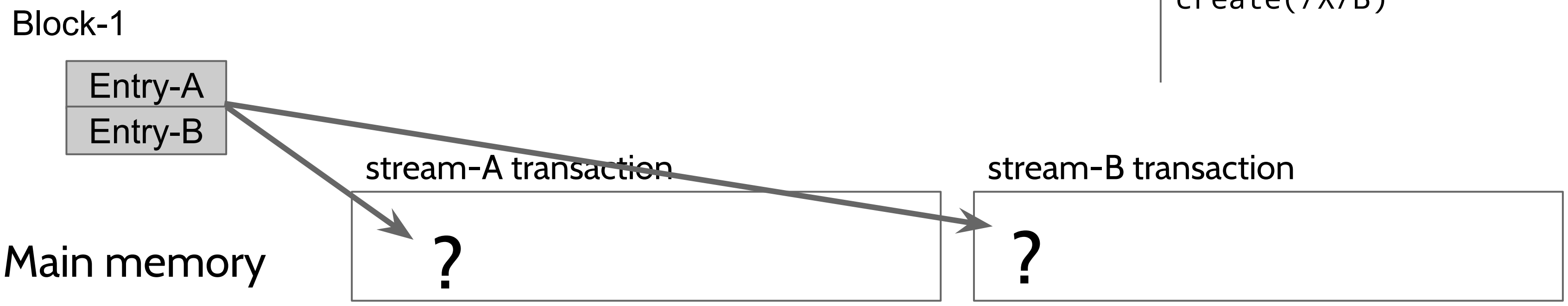
Application A	Application B
`set_stream(A)`	`set_stream(B)`
`create(/X/A)`	
	`create(/X/B)`



Challenge #1: Block-Level Journaling

Two independent streams can update same block!

Application A	Application B
`set_stream(A)`	`set_stream(B)`
`create(/X/A)`	
	`create(/X/B)`



Faulty solution: Perform journaling at byte-granularity

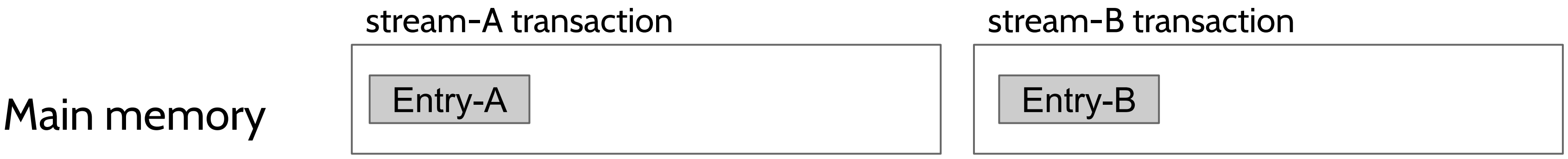
- Disables optimizations, complicates disk updates

Challenge #1: Block-Level Journaling

CCFS solution:

Record running transactions at byte granularity

Application A	Application B
`set_stream(A)`	`set_stream(B)`
`create(/X/A)`	
	`create(/X/B)`



Challenge #1: Block-Level Journaling

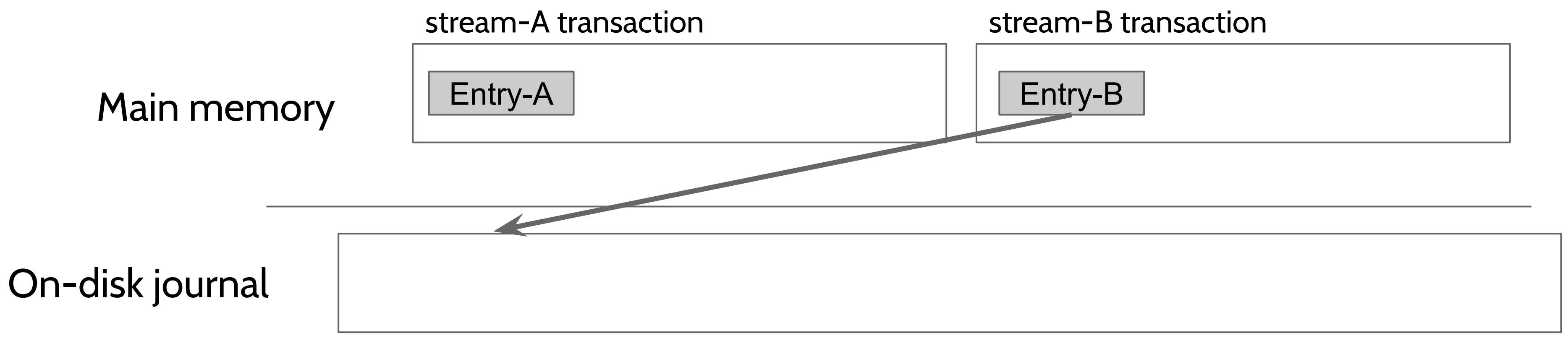
CCFS solution:

Record running transactions at byte granularity

Commit at block granularity

Application A	Application B

`set_stream(A)`
`create(/X/A)`
	`set_stream(B)`
	
	`create(/X/B)`



Challenge #1: Block-Level Journaling

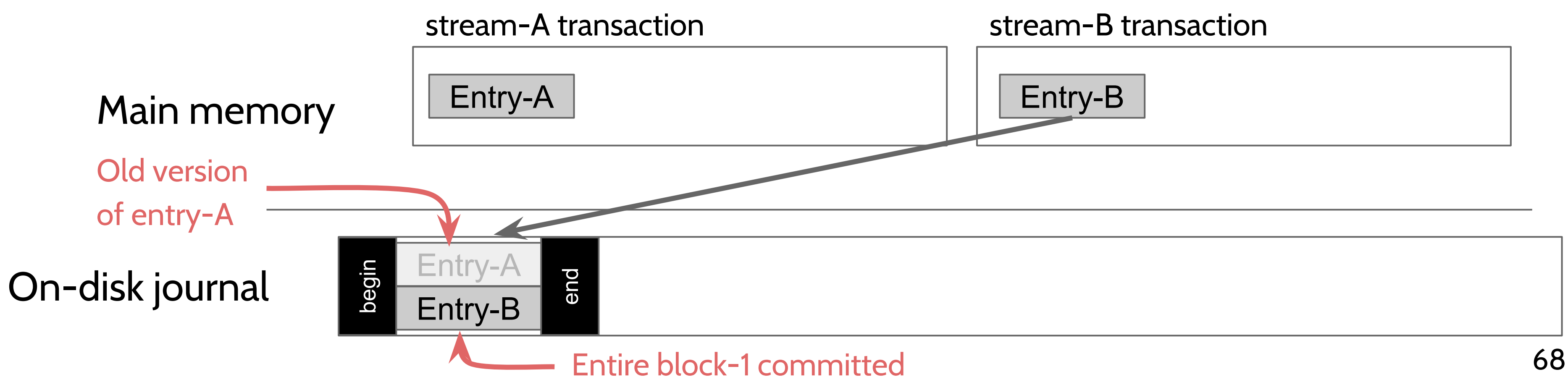
CCFS solution:

Record running transactions at byte granularity

Commit at block granularity

Application A	Application B

`set_stream(A)`
`create(/X/A)`
	`set_stream(B)`
	
	`create(/X/B)`



More Challenges ...

1. Both streams update directory's modification date
    - Solution: Delta journaling

More Challenges …

1. Both streams update directory's modification date
   - Solution: Delta journaling
2. Directory entries contain pointers to adjoining entry
   - Solution: Pointer-less data structures

More Challenges ...

1. Both streams update directory's modification date
    - Solution: Delta journaling
2. Directory entries contain pointers to adjoining entry
    - Solution: Pointer-less data structures
3. Directory entry freed by stream A can be reused by stream B
    - Solution: Order-less space reuse

More Challenges ...

1. Both streams update directory's modification date
    - Solution: Delta journaling
2. Directory entries contain pointers to adjoining entry
    - Solution: Pointer-less data structures
3. Directory entry freed by stream A can be reused by stream B
    - Solution: Order-less space reuse
4. Ordering technique: Data journaling cost
    - Solution: Selective data journaling [Chidambaram et al., SOSP 2013]

More Challenges …

1. Both streams update directory's modification date
    - Solution: Delta journaling
2. Directory entries contain pointers to adjoining entry
    - Solution: Pointer-less data structures
3. Directory entry freed by stream A can be reused by stream B
    - Solution: Order-less space reuse
4. Ordering technique: Data journaling cost
    - Solution: Selective data journaling [Chidambaram et al., SOSP 2013]
5. Ordering technique: Delayed allocation requires re-ordering
    - Solution: Order-preserving delayed allocation

More Challenges ...

1. Both streams update directory's modification date
    - Solution: Delta journaling
2. Directory entries contain pointers to adjoining entry
    - Solution: Pointer-less data structures
3. Directory entry freed by stream A can be reused by stream B
    - Solution: Order-less space reuse
4. Ordering technique: Data journaling cost
    - Solution: Selective data journaling [Chidambaram et al., SOSP 2013]
5. Ordering technique: Delayed allocation requires re-ordering
    - Solution: Order-preserving delayed allocation

Details in the paper!

Outline

Introduction

Background

Stream API

Crash-Consistent File System

Evaluation

Conclusion

Evaluation

1. Does CCFS solve application vulnerabilities?

Evaluation

1. Does CCFS solve application vulnerabilities?

- Tested five applications: LevelDB, SQLite, Git, Mercurial, ZooKeeper
- Method similar to previous study (ALICE tool) [Pillai et al., OSDI 2014]
- New versions of applications
- Default configuration, instead of safe configuration

Evaluation

1. Does CCFS solve application vulnerabilities?

	Vulnerabilities	
Application	**ext4**	**ccfs**
LevelDB	1	0
SQLite-Roll	0	0
Git	2	0
Mercurial	5	2
ZooKeeper	1	0

Evaluation

1. Does CCFS solve application vulnerabilities?

	Vulnerabilities	
Application	**ext4**	**ccfs**
LevelDB	1	0
SQLite-Roll	0	0
Git	2	0
Mercurial	5	2
ZooKeeper	1	0

Ext4: 9 Vulnerabilities

- Consistency lost in LevelDB
- Repository corrupted in Git, Mercurial
- ZooKeeper becomes unavailable

Evaluation

1. Does CCFS solve application vulnerabilities?

	Vulnerabilities	
Application	**ext4**	**ccfs**
LevelDB	1	0
SQLite-Roll	0	0
Git	2	0
Mercurial	5	2
ZooKeeper	1	0

Ext4: 9 Vulnerabilities

- Consistency lost in LevelDB
- Repository corrupted in Git, Mercurial
- ZooKeeper becomes unavailable

CCFS: 2 vulnerabilities in Mercurial

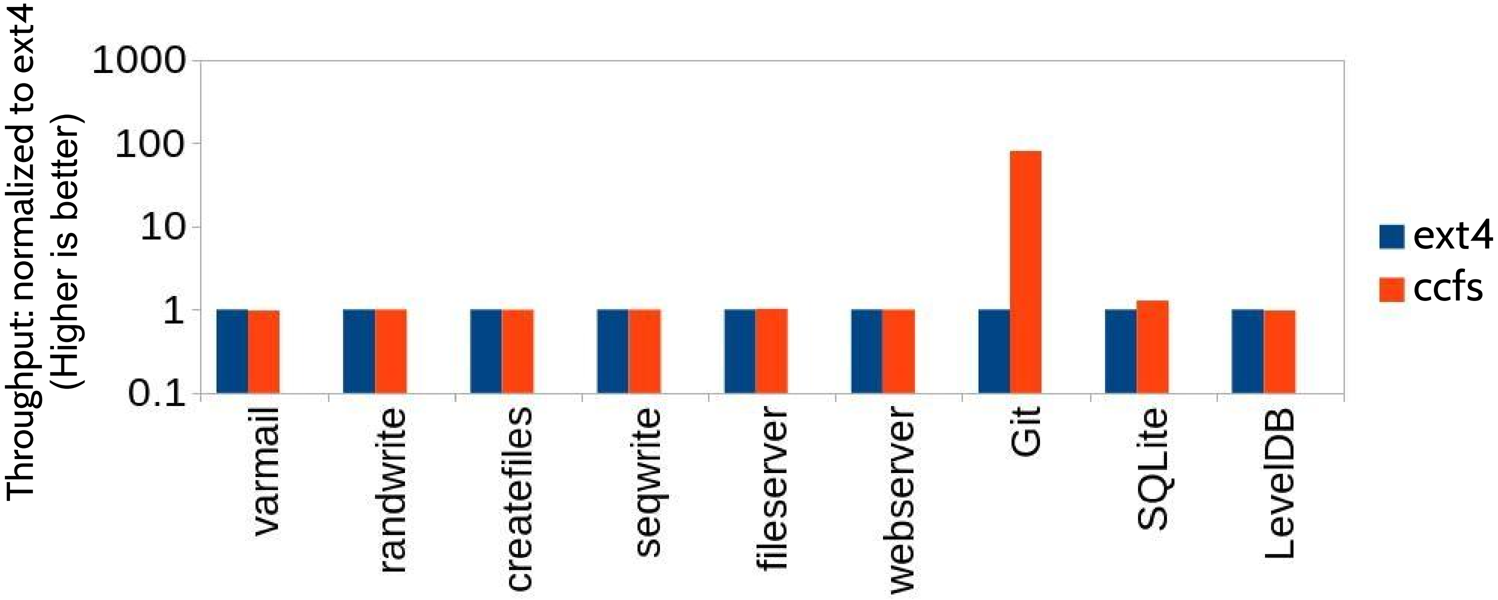
- Dirstate corruption

Evaluation

2. Performance within an application
   - Do false dependencies reduce performance inside application?
   - Or, do we need more than one stream per application?

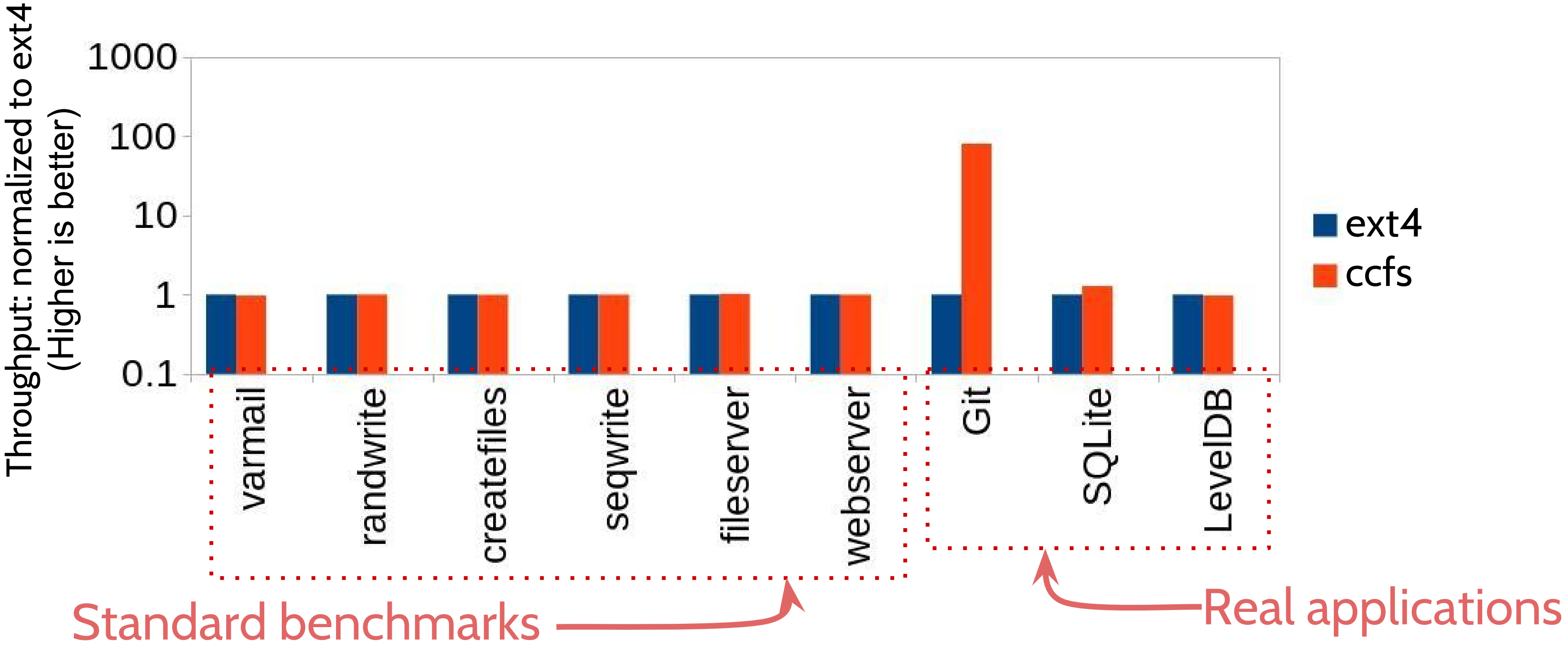
Evaluation

2. Performance within an application



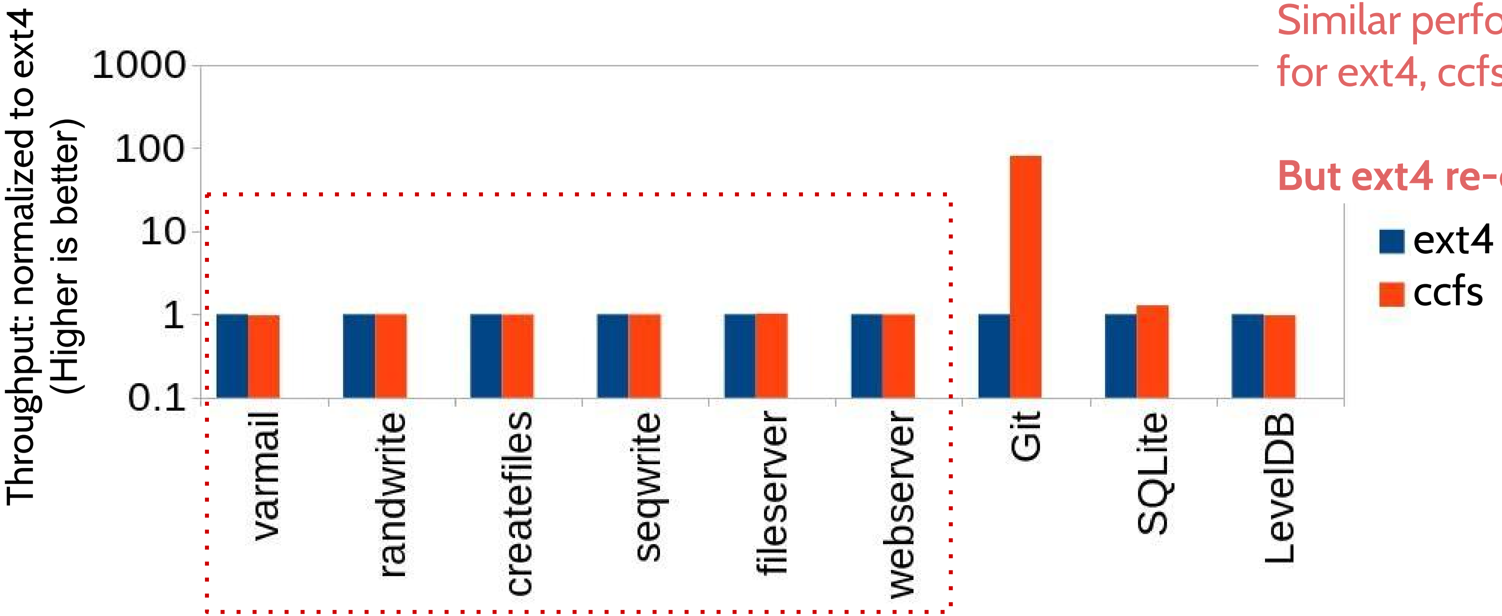
Evaluation

2. Performance within an application



Evaluation

2. Performance within an application

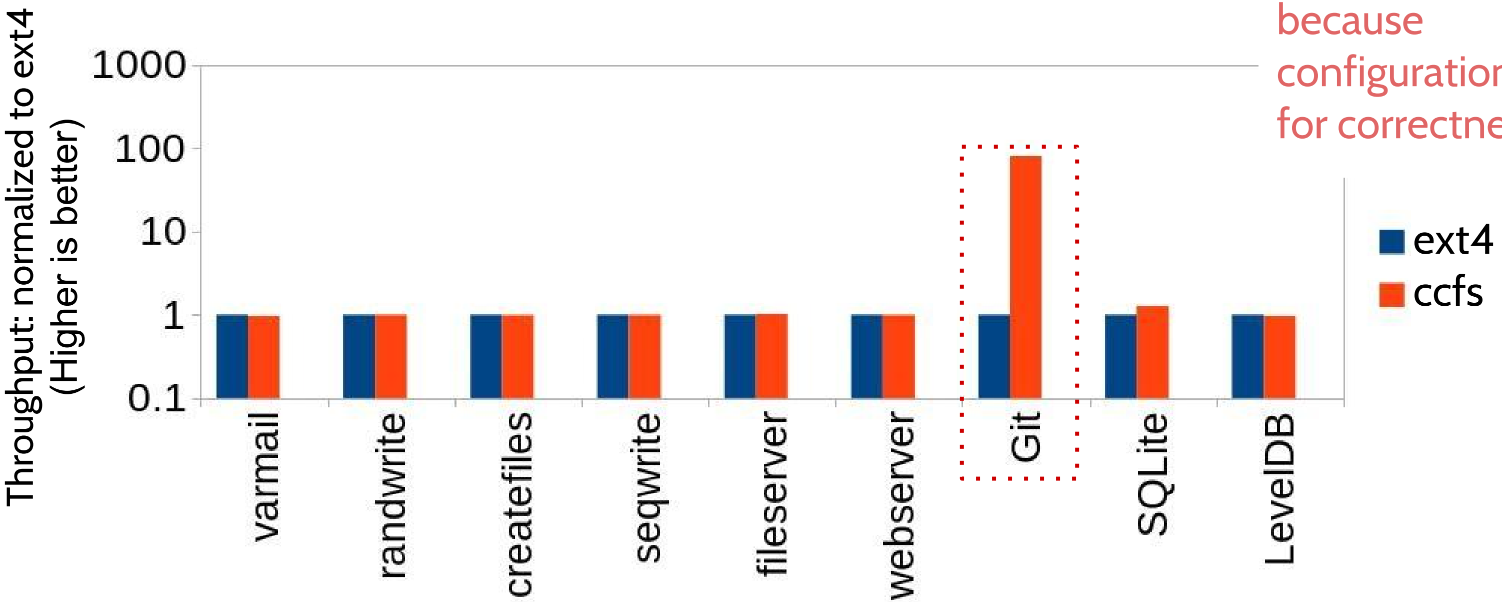


Standard workloads:
Similar performance
for ext4, ccfs

But ext4 re-orders!

Evaluation

2. Performance within an application

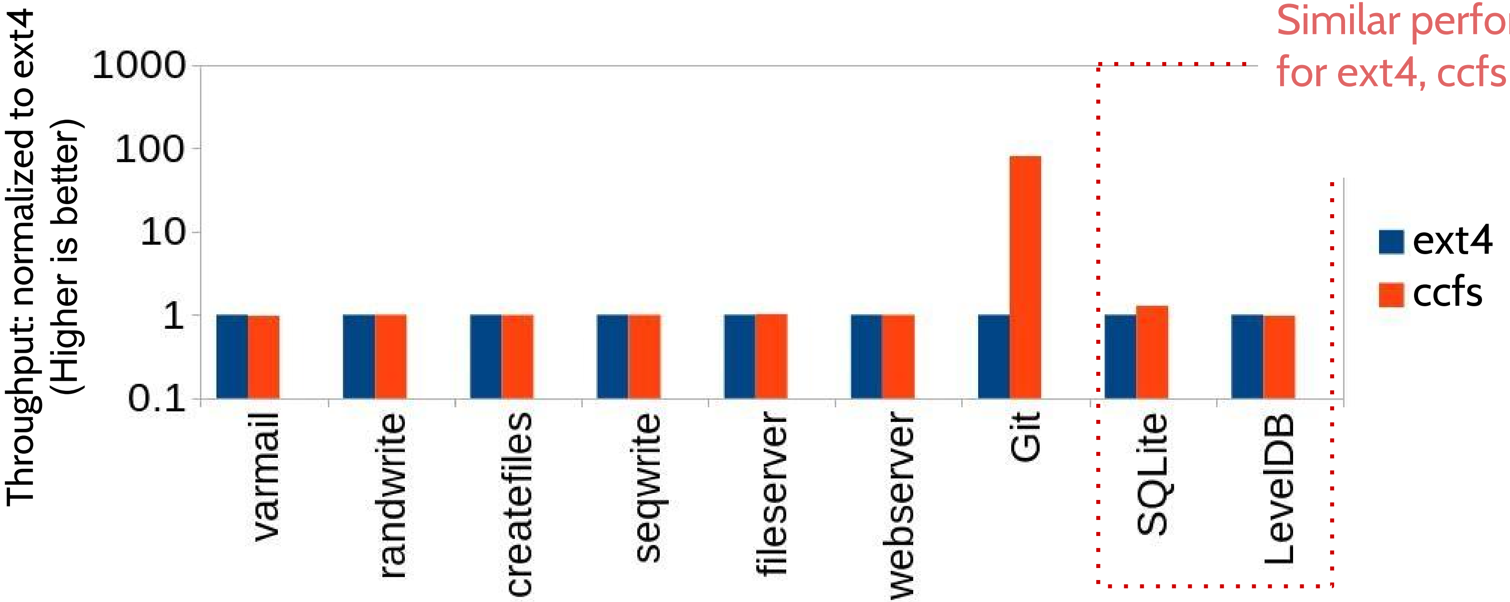


Git under ext4 is slow because of safer configuration needed for correctness

Evaluation

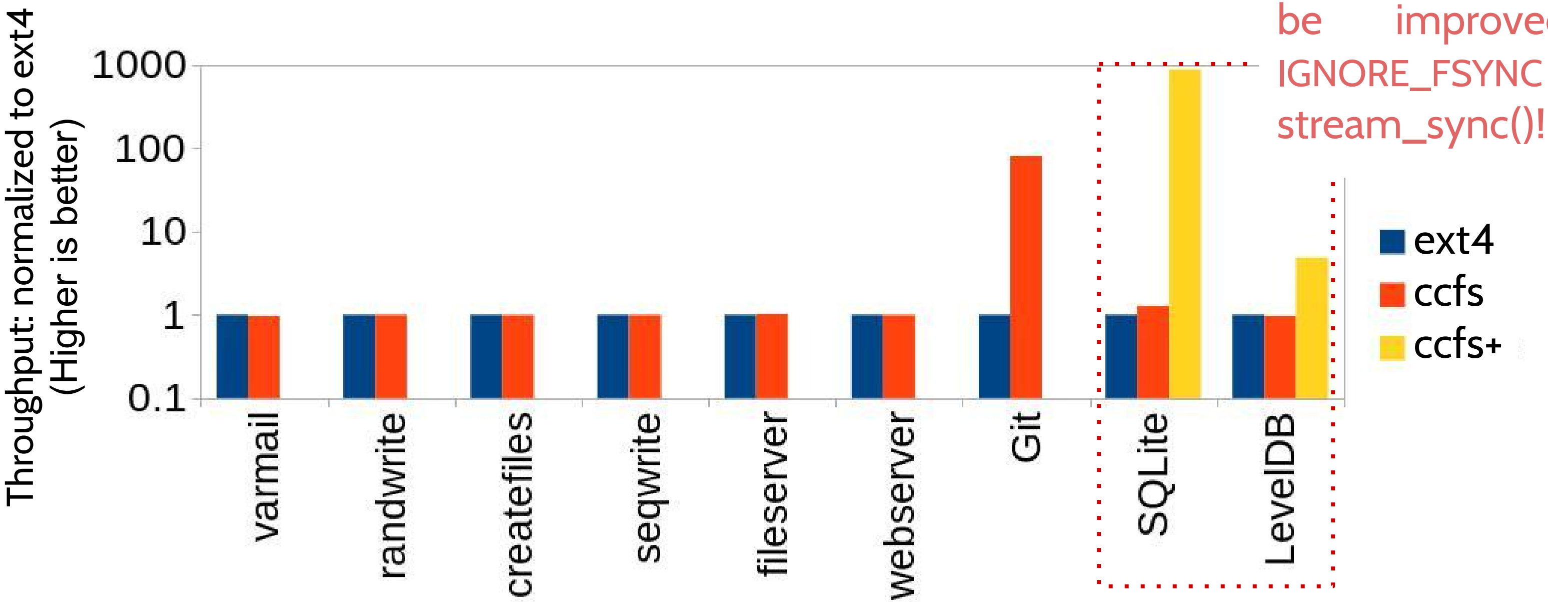
2. Performance within an application

SQLite and LevelDB : Similar performance for ext4, ccfs



Evaluation

2. Performance within an application



But, performance can be improved with IGNORE_FSYNC and stream_sync()!

Evaluation: Summary

Crash consistency: Better than ext4

- 9 vulnerabilities in ext4, 2 minor in CCFS

Performance: Like ext4 with little programmer overhead

- Much better with additional programmer effort

More results in paper!

Conclusion

FS crash behavior is currently not standardized

Conclusion

FS crash behavior is currently not standardized

Ideal FS behavior can improve application consistency

Conclusion

FS crash behavior is currently not standardized

Ideal FS behavior can improve application consistency

Ideal FS behavior is considered bad for performance

Conclusion

FS crash behavior is currently not standardized

Ideal FS behavior can improve application consistency

Ideal FS behavior is considered bad for performance

Stream abstraction and CCFS solve this dilemma

Conclusion

FS crash behavior is currently not standardized

Ideal FS behavior can improve application consistency

Ideal FS behavior is considered bad for performance

Stream abstraction and CCFS solve this dilemma

Thank you! Questions?

Examples

1. LevelDB:
    a. creat(tmp); write(tmp); fsync(tmp); rename(tmp, CURRENT); --> unlink(MANIFEST-old);
        i. Unable to open the database
    b. write(file1, kv1); write(file1, kv2); --> creat(file2, kv3);
        i. kv1 and kv2 might disappear, while kv3 still exists
2. Git:
    a. append(index.lock) --> rename(index.lock, index)
        i. "Corruption " returned by various Git commands
    b. write(tmp); link(tmp, object) --> rename(master.lock, master)
        i. "Corruption " returned by various Git commands
3. HDFS:
    a. creat(ckpt); append(ckpt); fsync(ckpt); creat(md5.tmp); append(md5.tmp); fsync(md5.tmp); rename(md5.tmp, md5); --> rename(ckpt, fsimage);
        i. Unable to boot the server and use the data

File System Study: Results

One sector overwrite: Atomic because of device characteristics

Appends: Garbage in some file systems

File systems do not usually provide atomicity for big writes

File system configuration		Atomicity: *One sector overwrite*	Atomicity: *One sector append*	Atomicity: *Many sector write*	Atomicity: *Directory operation*
ext2	*async*		✘	✘	✘
	sync		✘	✘	✘
ext3	*writeback*		✘	✘	
	ordered			✘	
	data-journal			✘	
ext4	*writeback*		✘	✘	
	ordered			✘	
	no-delalloc			✘	
	data-journal			✘	
btrfs				✘	
xfs	*default*			✘	
	wsync			✘	

File System Study: Results

One sector overwrite: Atomic because of device characteristics

Appends: Garbage in some file systems

File systems do not usually provide atomicity for big writes

Directory operations are usually atomic

File system configuration		Atomicity			
		One sector overwrite	*One sector append*	*Many sector write*	*Directory operation*
ext2	*async*		✗	✗	✗
	sync		✗	✗	✗
ext3	*writeback*		✗	✗	
	ordered			✗	
	data-journal			✗	
ext4	*writeback*		✗	✗	
	ordered			✗	
	no-delalloc			✗	
	data-journal			✗	
btrfs				✗	
xfs	*default*			✗	
	wsync			✗	

Collecting System Call Trace

`git add file1` Application Workload

Record strace, memory accesses (for mmap writes), initial state of datastore

Initial state

```
.git/...
```

Trace

```
creat(index.lock)
creat(tmp)
append(tmp, data, 4K)
fsync(tmp)
link(tmp, permanent)
append(index.lock)
rename(index.lock, index)
```

Calculating Intermediate States

a. Convert system calls into atomic modifications

```
creat(index.lock)
creat(tmp)
append(tmp, 4K)

fsync(tmp)
link(tmp, permanent)
...
```

```
creat(inode=1, dentry=index.lock)
creat(inode=2, dentry=tmp)
truncate(inode=2, 1)
truncate(inode=2, 2)
...
truncate(inode=2, 4K)
write(inode=2, garbage)
write(inode=2, actual data)
...

link(inode=2, dentry=permanent)
...
```

Calculating Intermediate States

b. Find ordering dependencies

```
creat(index.lock)
creat(tmp)
append(tmp, 4K)

fsync(tmp)
link(tmp, permanent)
...
```

```
creat(inode=1, dentry=index.lock)
creat(inode=2, dentry=tmp)
truncate(inode=2, 1)
truncate(inode=2, 2)
...
truncate(inode=2, 4K)
write(inode=2, garbage)
write(inode=2, actual data)
...

link(inode=2, dentry=permanent)
...
```

Calculating Intermediate States

c. Choose *a few* sets of modifications obeying dependencies

```
creat(inode=1, dentry=index.lock)
creat(inode=2, dentry=tmp)
truncate(inode=2, 1)
truncate(inode=2, 2)
...
truncate(inode=2, 4K)
write(inode=2, garbage)
write(inode=2, actual data)
...

link(inode=2, dentry=permanent)
...
```

Set 1:

```
creat(inode=1, dentry=index.lock)
<all truncates and writes to inode 2>
```

Set 2:

```
creat(inode=1, dentry=index.lock)
<all truncates and writes to inode 2>
link(inode=2, dentry=permanent)
```

Set 3:

```
creat(inode=1, dentry=index.lock)
creat(inode=2, dentry=tmp)
truncate(inode=2, 1)
```

... more sets

Calculating Crash States from a Trace

d. Reconstruct states from sets of modifications

Set 1:

```
creat(inode=1, dentry=index.lock)
<all truncates and writes to inode 2>
```

→

```
.git/index.lock (0)
```

Set 2:

```
creat(inode=1, dentry=index.lock)
<all truncates and writes to inode 2>
link(inode=2, dentry=permanent)
```

→

```
.git/index.lock (0)
.git/permanent (4K)
```

Set 3:

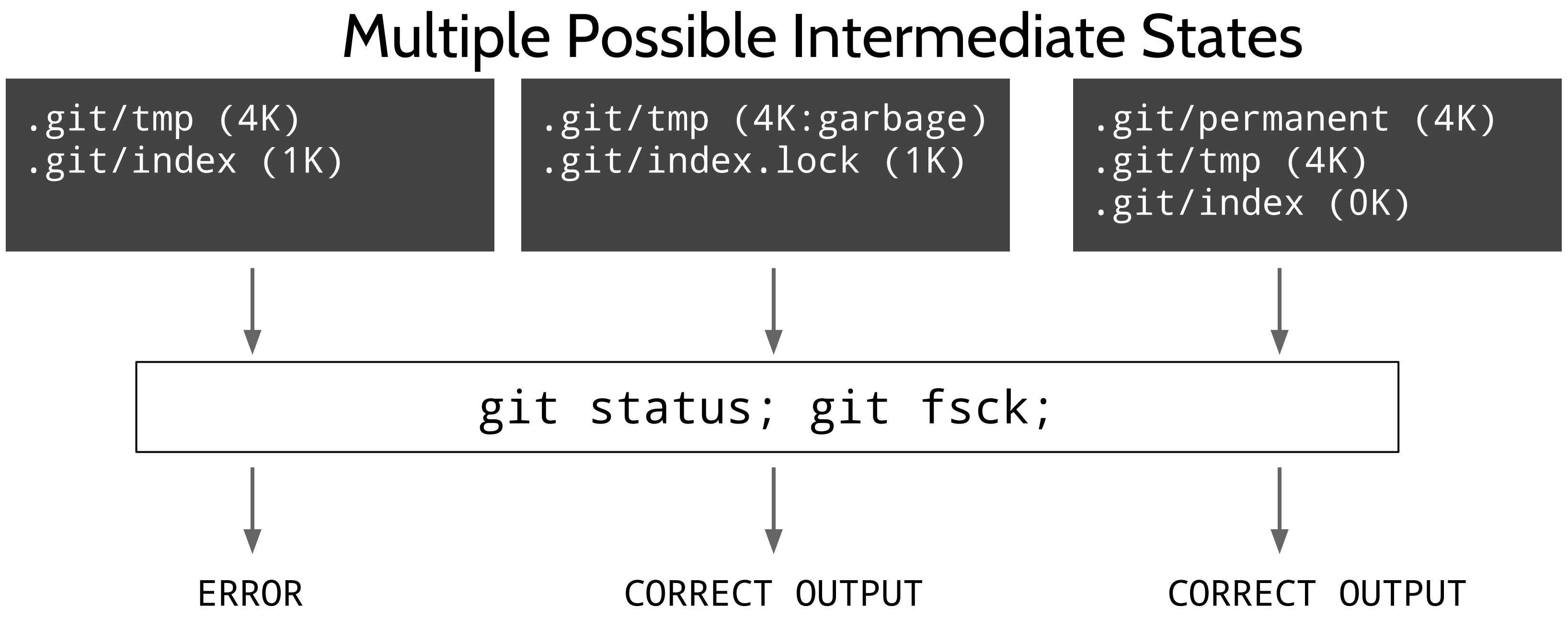
```
creat(inode=1, dentry=index.lock)
creat(inode=2, dentry=tmp)
truncate(inode=2, 1)
```

→

```
.git/index.lock (0)
.git/tmp (1)
```

... more sets

Checking ALC on Intermediate States



Why is ALC problematic?

Applications implement complex *update protocols*

- Aiming for both correctness and performance
- Each protocol is different

Update protocols hard to implement and test

Applications many and varied

- Little effort to test each

<u>Unfortunately, file systems make ALC more difficult</u>

Persistence Models: Too Complex

Persistence models used by us to find vulnerabilites

But, persistence models can be complex

- Example: `write()` ordered before `unlink()` iff they act on the same directory and `write()` is more than 4KB
- Useful for *verifying* ALC atop a file system

Persistence models not suitable to *discuss* ALC

- Is `fsync()` required after writes to log file in *ext3*?
- Or, do `write()` calls persist in-order?

Persistence Properties

Does FS obey a particular interesting behavior?

- Example: Do `write()` calls persist in-order?
- Are `write()` calls atomic?

Applications typically *depend* on some properties

- Forgot an `fsync()`: depends on ordering properties
- Forgot checksum verification: depends on atomic `write()`

Persistence Properties: Example #1

Content-Atomicity of Appends

Does an append result in garbage?

System call sequence

lseek(`file1`, *End of file*)

write(`file1`, "hello")

Impossible
Intermediate State

`/file1` "he#@!" ✗

Allowed
Intermediate State

`/file1` "he" ✓

Persistence Properties: Example #2

Ordered Writes

Are the effects of write() sent to disk in-order?

System call sequence

write(file1, "hello")

write(file2, "world")

Impossible Intermediate State

```
/file1  ""
/file2  "world"
```

Allowed Intermediate State

```
/file1  "hello"
/file2  ""
```

Example: Git

0 mkdir(o/x)
1 creat(o/x/tmp_y)
2 append(o/x/tmp_y)
3 *fsync(o/x/tmp_y)*
4 link(o/x/tmp_y, o/x/y)
5 unlink(o/x/tmp_y)

(i) store object

creat(index.lock)
(i) store object
append(index.lock)
rename(index.lock,index)
stdout(finished add)

(ii) git add

(i) store object
creat(branch.lock)
append(branch.lock)
append(branch.lock)
append(logs/branch)
append(logs/HEAD)
rename(branch.lock,x/branch)
stdout(finished commit)

(iii) git commit

Example: Git

Atomicity

0 mkdir(o/x)

1 creat(o/x/tmp_y)

2 append(o/x/tmp_y)

3 *fsync(o/x/tmp_y)*

4 link(o/x/tmp_y, o/x/y)

5 unlink(o/x/tmp_y)

(i) store object

creat(index.lock)

(i) store object

append(index.lock)

rename(index.lock,index)

stdout(finished add)

(ii) git add

(i) store object

creat(branch.lock)

append(branch.lock)

append(branch.lock)

append(logs/branch)

append(logs/HEAD)

rename(branch.lock,x/branch)

stdout(finished commit)

(iii) git commit

Example: Git

Ordering

0 mkdir(o/x)
1 creat(o/x/tmp_y)
2 append(o/x/tmp_y)
3 *fsync(o/x/tmp_y)*
4 link(o/x/tmp_y, o/x/y)
5 unlink(o/x/tmp_y)

(i) store object

creat(index.lock)
(i) store object
append(index.lock)
rename(index.lock,index)
stdout(finished add)

(ii) git add

(i) store object
creat(branch.lock)
append(branch.lock)
append(branch.lock)
append(logs/branch)
append(logs/HEAD)
rename(branch.lock,x/branch)
stdout(finished commit)

(iii) git commit

Example: Git

Durability

0 mkdir(o/x)
1 creat(o/x/tmp_y)
2 append(o/x/tmp_y)
3 *fsync(o/x/tmp_y)*
4 link(o/x/tmp_y, o/x/y)
5 unlink(o/x/tmp_y)

(i) store object

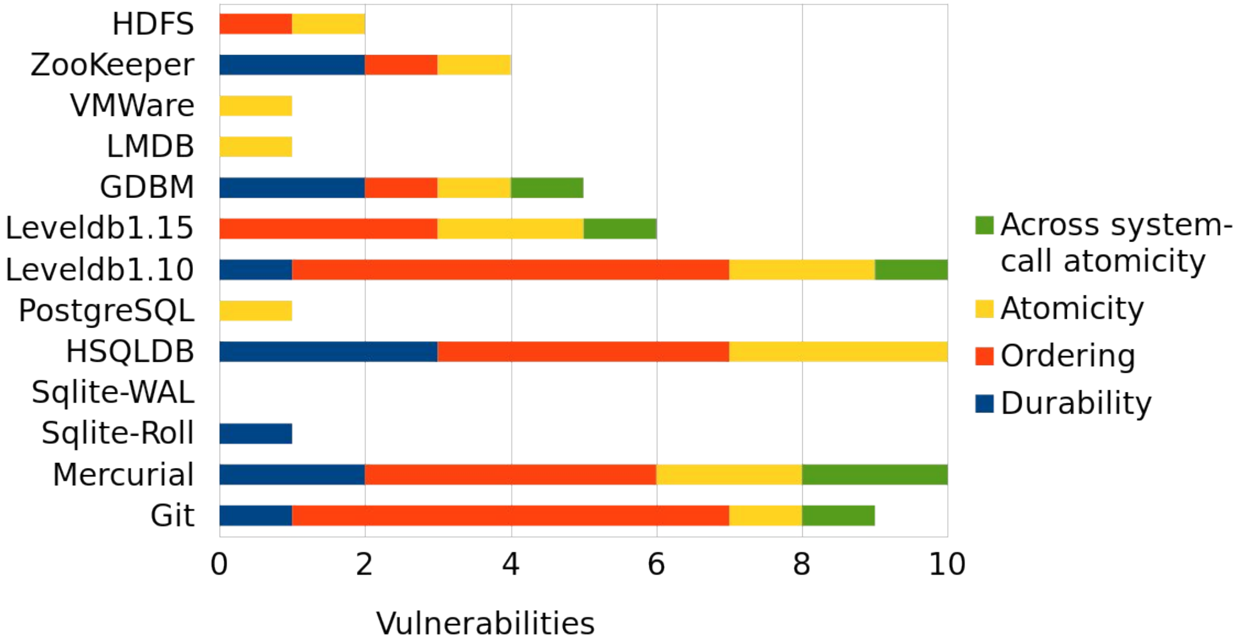
creat(index.lock)
(i) store object
append(index.lock)
rename(index.lock,index)
stdout(finished add)

(ii) git add

(i) store object
creat(branch.lock)
append(branch.lock)
append(branch.lock)
append(logs/branch)
append(logs/HEAD)
rename(branch.lock,x/branch)
stdout(finished commit)

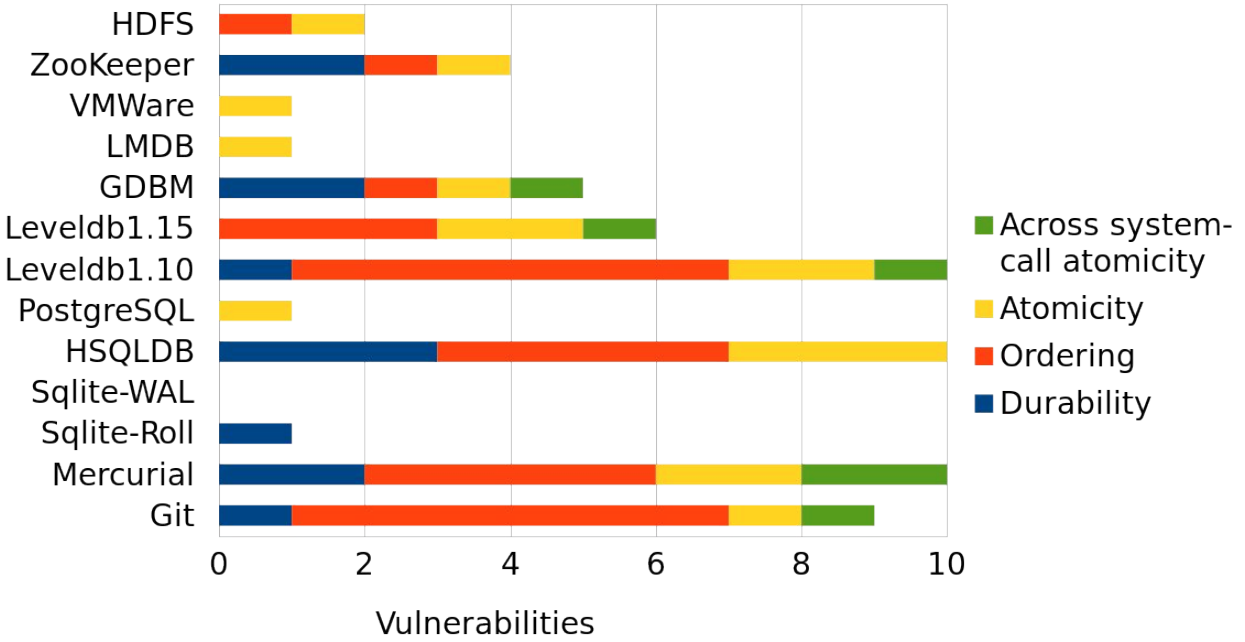
(iii) git commit

Vulnerability Study: Patterns



Vulnerability Study: Patterns

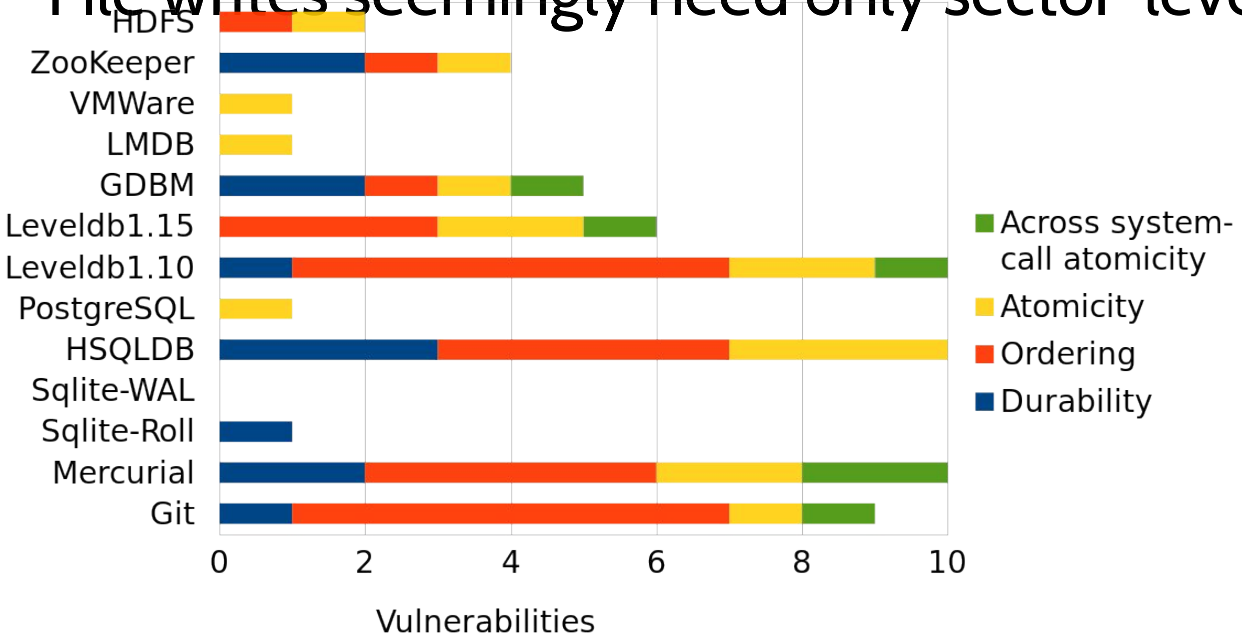
Across syscall atomicity: Few, minor consequences



Vulnerability Study: Patterns

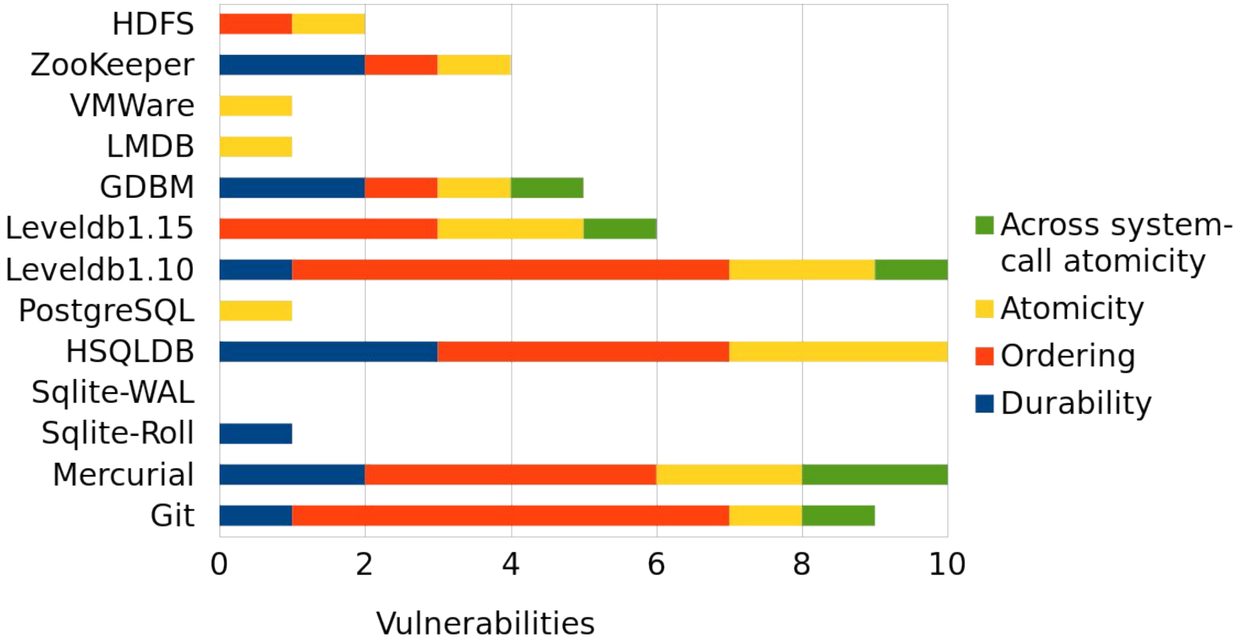
Garbage during appends cause 4 vulnerabilities

File writes seemingly need only sector-level atomicity



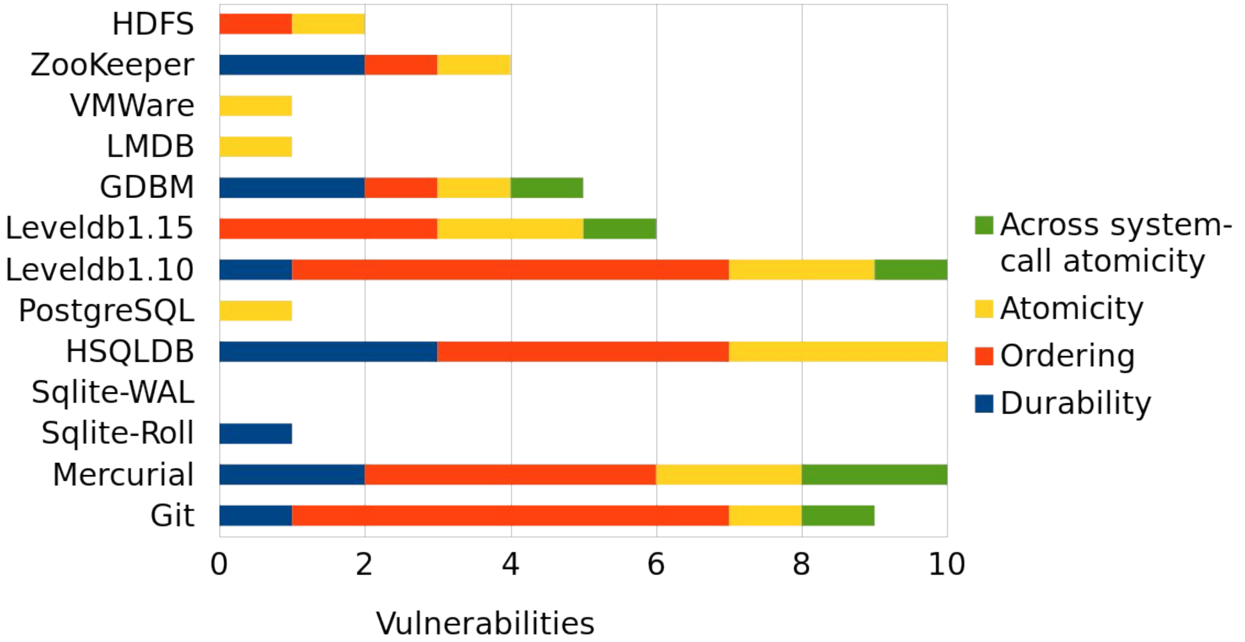
Vulnerability Study: Patterns

A separate fsync() on parent directory: 6 vulnerabilities



Vulnerability Study: Patterns

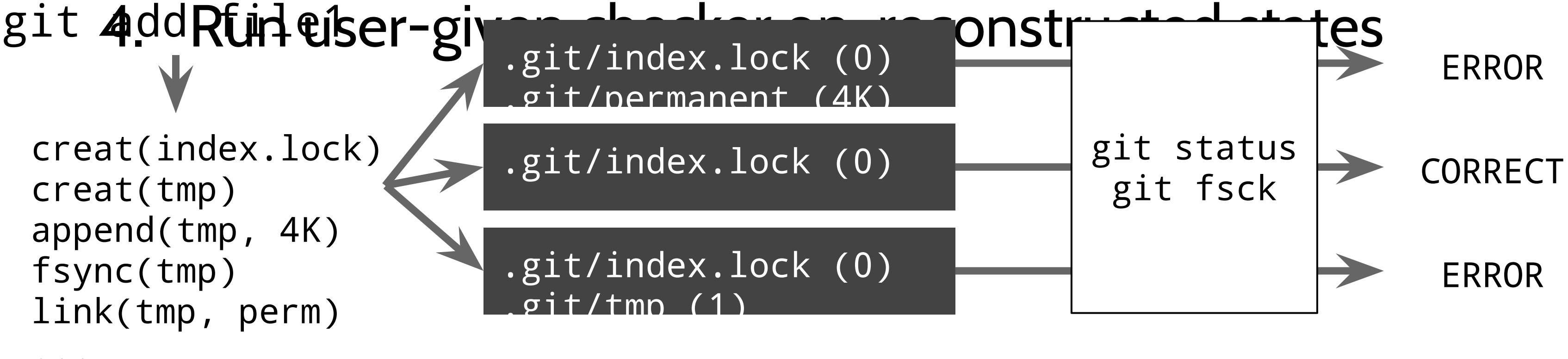
Six applications do not fsync() directory operations



ALICE: Solution

Solution:

1. User supplies application workload
2. Record a system-call trace from workload
3. Use "Abstract Persistence Model" and reconstruct targeted intermediate states
4. Run user-given checker on reconstructed states



ALICE: Intermediate States #1

Does application need atomicity across system calls?

Method: Crash after each system call

```
creat(index.lock)
creat(tmp)
append(tmp, 4K)
fsync(tmp)
link(tmp, perm)
...
```

ALICE: Intermediate States #1

Does application need atomicity across system calls?

Method: Crash after each system call

```
creat(index.lock)
creat(tmp)
append(tmp, 4K)
fsync(tmp)
link(tmp, perm)
...
```

Crash here

ALICE: Intermediate States #1

Does application need atomicity across system calls?

Method: Crash after each system call

```
creat(index.lock)
creat(tmp)
append(tmp, 4K)
fsync(tmp)
link(tmp, perm)
...
```

Crash here

...

ALICE: Intermediate States #2

Does application need atomicity of an individual system call?

Method:

1. Apply all system calls until examined call
2. Apply various partial effects of examined call

System call examined →

```
creat(index.lock)
creat(tmp)
append(tmp, 4K)
fsync(tmp)
link(tmp, perm)
...
```

ALICE: Intermediate States #2

Does application need atomicity of an individual system call?

Method:

1. Apply all system calls until examined call
2. Ap tial effects of examined call

```
creat(index.lock)
creat(tmp)
append(tmp, 4K)
fsync(tmp)
link(tmp, perm)
...
```

System call examined → creat(tmp)

Apply these calls] creat(index.lock), creat(tmp)

ALICE: Intermediate States #2

Does application need atomicity of an individual system call?

Method:

1. Apply all system calls until examined call
2. Ap… tial effects of examined call

ALICE: Intermediate States #3

Does application need ordering of a system call?

Method:

1. Apply all system calls *except* examined call ...
2. Crash at different points in trace

System call examined →

```
creat(index.lock)
creat(tmp)
append(tmp, 4K)
fsync(tmp)
link(tmp, perm)
...
```

ALICE: Intermediate States #3

Does application need ordering of a system call?

Method:

1. Apply all system calls *except* examined call ...
2. Crash at different points in trace

System call examined

```
creat(index.lock)
creat(tmp)
append(tmp, 4K)
fsync(tmp)
link(tmp, perm)
...
```

Ordering examined

ALICE: Intermediate States #3

Does application need ordering of a system call?

Method:

1. Apply all system calls *except* examined call ...
2. Crash at different points in trace

System call examined →

```
creat(index.lock)
creat(tmp)
append(tmp, 4K)
fsync(tmp)
link(tmp, perm)
...
```

Ordering examined

File System Study: Results

File system configuration		Atomicity				Ordering			
		One sector overwrite	*Append content*	*Many sector overwrite*	*Directory operation*	*Overwrite → Any op*	*Append → Any op*	*Dir-op → Any op*	*Append → Rename*
ext2	*async*	✓							
	sync	✓				✓	✓	✓	✓
ext3	*writeback*	✓			✓			✓	
	ordered	✓	✓		✓		✓	✓	✓
	data-journal	✓	✓		✓	✓	✓	✓	✓
ext4	*writeback*	✓			✓			✓	
	ordered	✓	✓		✓			✓	✓
	no-delalloc	✓	✓		✓		✓	✓	✓
	data-journal	✓	✓		✓	✓	✓	✓	✓
btrfs		✓	✓		✓	✓			✓
xfs	*default*	✓	✓		✓			✓	✓
	wsync	✓	✓		✓		✓	✓	✓

One-sector-overwrite atomicity is due to current hardware, might change with NVMs

File System Study: Results

File system configuration		Atomicity				Ordering			
		One sector overwrite	*Append content*	*Many sector overwrite*	*Directory operation*	*Overwrite → Any op*	*Append → Any op*	*Dir-op → Any op*	*Append → Rename*
ext2	*async*	✓							
	sync	✓				✓	✓	✓	✓
ext3	*writeback*	✓			✓			✓	
	ordered	✓	✓		✓		✓	✓	✓
	data-journal	✓	✓		✓	✓	✓	✓	✓
ext4	*writeback*	✓			✓			✓	
	ordered	✓	✓		✓			✓	✓
	no-delalloc	✓	✓		✓		✓	✓	✓
	data-journal	✓	✓		✓	✓	✓	✓	✓
btrfs		✓	✓		✓	✓			✓
xfs	*default*	✓	✓		✓			✓	✓
	wsync	✓	✓		✓		✓	✓	✓

File systems patched to obey a particular property

Vulnerability Study: Goals

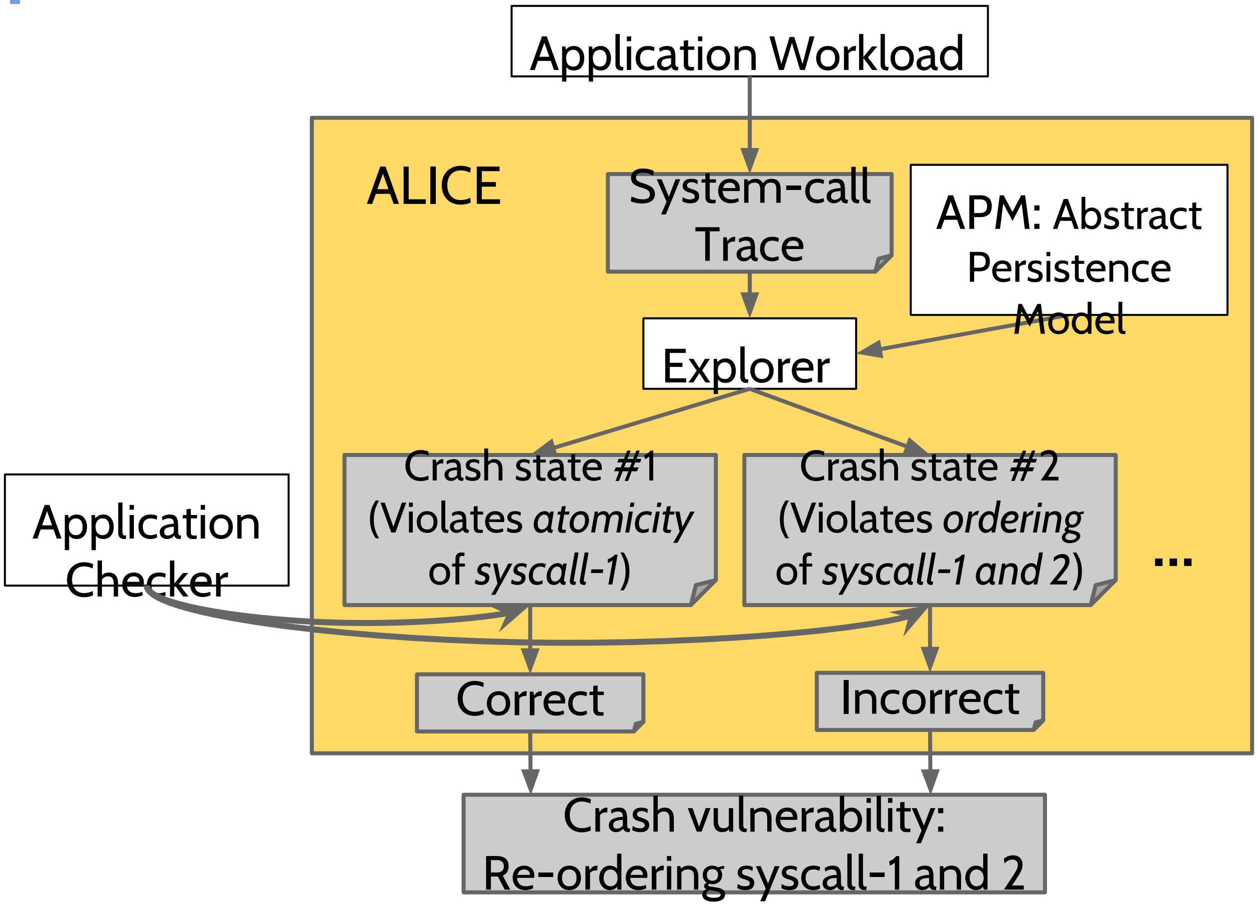
Does FS behavior affect applications?

What FS behaviors are important?

Is testing for crash vulnerabilities generally helpful?

Not a goal: Comparing correctness among applications

ALICE: Technique



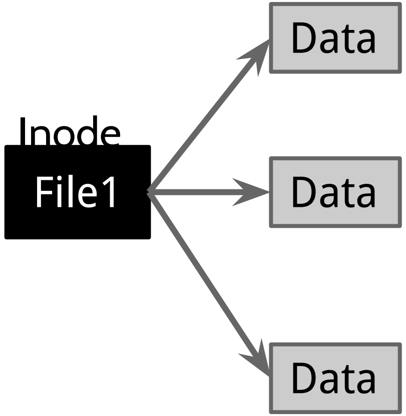
File System Study: Conclusion

File systems vary in persistence properties

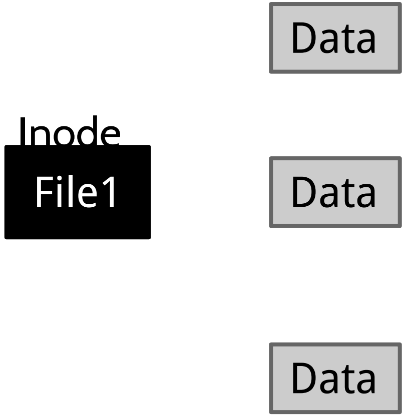
Application correctness can vary among file systems!

Challenge: Validating application correctness without assuming a particular underlying file system

Challenge #2: Space Reuse

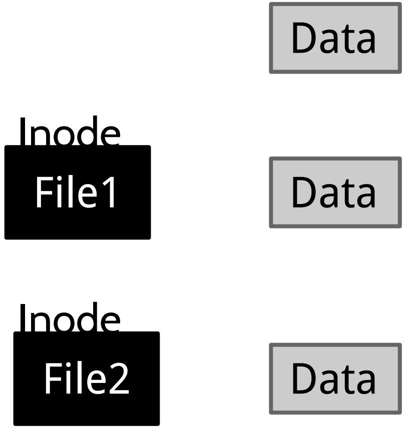


Challenge #2: Space Reuse




```
truncate(file1);
```

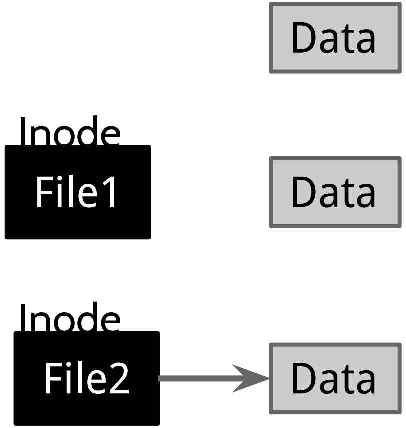
Challenge #2: Space Reuse




```
truncate(file1);
```

```
creat(file2);
```

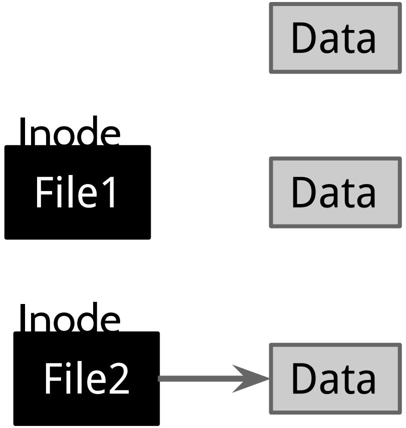
Challenge #2: Space Reuse




```
truncate(file1);
```

```
creat(file2);
write(file2, “hello”);
```

Challenge #2: Space Reuse

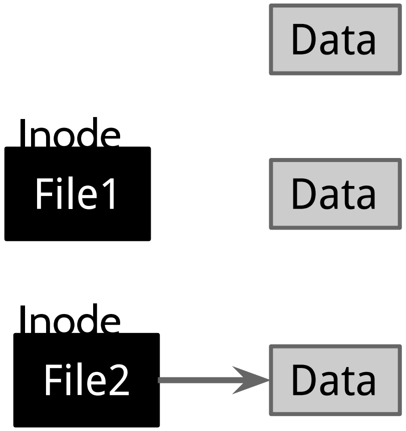


Block pointer manipulation shown so far occurs in memory

```
truncate(file1);
```

```
creat(file2);
write(file2, “hello”);
```

Challenge #2: Space Reuse



What if pointer manipulation occurs in different streams?

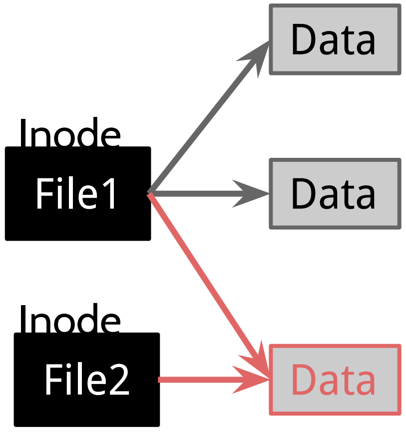
Stream 1 (Application 1)

```
truncate(file1);
```

Stream 2 (Application 2)

```
creat(file2);
write(file2, “hello”);
```

Challenge #2: Space Reuse



Possible crash state

If only one stream commits,
FS consistency will be affected

Stream 1
(Application 1)

```
truncate(file1);
```

Stream 2
(Application 2)

```
creat(file2);
write(file2, “hello”);
```

File-System Behavior

Each file system behaves differently across a crash

- Behavior across crashes are not standardized
- Behavior can be divided into atomicity and ordering

Atomicity of updates might not be maintained

- Atomicity of file writes
- Other operations: Renaming a file, deleting a file etc.

Ordering of updates might not be maintained