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)
- Kernel bugs

Applications need to implement crash consistency
- E.g., Database applications ensure transactions are atomic

[Source: www.datacenterknowledge.com, 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)
- Kernel bugs

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
App crash consistency depends on FS behavior

- 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

- 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
- 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

[Pillai et al., OSDI 2014]
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

Atomicity  FS Crash Behavior  Ordering
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?
Each file system behaves differently across a crash

- Little standardization of behavior across crashes

**File-System Behavior**

- Atomicity
  - Directory operations: E.g., rename() atomic?
  - File writes: Entire system call? Sector-level?

- Ordering
  - FS Crash Behavior
  - ...
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

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

[Pillai et al., OSDI 2014]
## Vulnerabilities Study: Results

<table>
<thead>
<tr>
<th></th>
<th>Ext2-like FS</th>
<th>Btrfs</th>
<th>Ext3-DJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>LevelDB-1.10</td>
<td>10</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>LevelDB-1.15</td>
<td>6</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>LMDB</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GDBM</td>
<td>5</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>HSQLDB</td>
<td>10</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>SQLite-Roll</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SQLite-WAL</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Git</td>
<td>9</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Mercurial</td>
<td>10</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>VMWare</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HDFS</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>ZooKeeper</td>
<td>4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>60</strong></td>
<td><strong>31</strong></td>
<td><strong>10</strong></td>
</tr>
</tbody>
</table>
## Vulnerabilities Study: Results

### File systems

<table>
<thead>
<tr>
<th>Applications</th>
<th>Ext2-like FS</th>
<th>Btrfs</th>
<th>Ext3-DJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>LevelDB-1.10</td>
<td>10</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>LevelDB-1.15</td>
<td>6</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>LMDB</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GDBM</td>
<td>5</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>HSQLDB</td>
<td>10</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>SQLite-Roll</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SQLite-WAL</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Git</td>
<td>9</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Mercurial</td>
<td>10</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>VMWare</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HDFS</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>ZooKeeper</td>
<td>4</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**Total**: 60, 31, 10
## Vulnerabilities Study: Results

### File-system behavior

<table>
<thead>
<tr>
<th></th>
<th>Ext2-like FS</th>
<th>Btrfs</th>
<th>Ext3-DJ</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Ordering</strong></td>
<td>✗</td>
<td>✗</td>
<td>✔️</td>
</tr>
<tr>
<td><strong>Atomicity</strong></td>
<td>✗</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>

### System Vulnerabilities

<table>
<thead>
<tr>
<th>System</th>
<th>Ext2-like FS</th>
<th>Btrfs</th>
<th>Ext3-DJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>LevelDB-1.10</td>
<td>10</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>LevelDB-1.15</td>
<td>6</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>LMDB</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GDBM</td>
<td>5</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>HSQLDB</td>
<td>10</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>SQLite-Roll</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SQLite-WAL</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Git</td>
<td>9</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Mercurial</td>
<td>10</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>VMware</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HDFS</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>ZooKeeper</td>
<td>4</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**Total**

- Ext2-like FS: 60
- Btrfs: 31
- Ext3-DJ: 10
## Vulnerabilities Study: Results

<table>
<thead>
<tr>
<th>Ordering</th>
<th>Ext2-like FS</th>
<th>Btrfs</th>
<th>Ext3-DJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomicity</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Database</th>
<th>Ext2-like FS</th>
<th>Btrfs</th>
<th>Ext3-DJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>LevelDB-1.10</td>
<td>10</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>LevelDB-1.15</td>
<td>6</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>LMDB</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GDBM</td>
<td>5</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>HSQLDB</td>
<td>10</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>SQLite-Roll</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SQLite-WAL</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Git</td>
<td>9</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Mercurial</td>
<td>10</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>VMWare</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HDFS</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>ZooKeeper</td>
<td>4</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**Total**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>60</td>
<td>31</td>
<td>10</td>
</tr>
</tbody>
</table>

Under FS with few guarantees of atomicity and ordering, 60 vulnerabilities are exposed
- Serious consequences: unavailability, data loss
### Vulnerabilities Study: Results

<table>
<thead>
<tr>
<th>Ordering</th>
<th>Ext2-like FS</th>
<th>Btrfs</th>
<th>Ext3-DJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomicity</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Repository Corruption</th>
<th>Unavailability</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Database</th>
<th>Ext2-like FS</th>
<th>Btrfs</th>
<th>Ext3-DJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>LevelDB-1.10</td>
<td>10</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>LevelDB-1.15</td>
<td>6</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>LMDB</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GDBM</td>
<td>5</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>HSQLDB</td>
<td>10</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>SQLite-Roll</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SQLite-WAL</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Git</td>
<td>9</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Mercurial</td>
<td>10</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>VMWare</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HDFS</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>ZooKeeper</td>
<td>4</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>60</strong></td>
<td><strong>31</strong></td>
<td><strong>10</strong></td>
</tr>
</tbody>
</table>
## Vulnerabilities Study: Results

<table>
<thead>
<tr>
<th>Tool</th>
<th>Ext2-like FS</th>
<th>Btrfs</th>
<th>Ext3-DJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ordering</td>
<td>X</td>
<td>X</td>
<td>✔</td>
</tr>
<tr>
<td>Atomicity</td>
<td>X</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

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

- Minor consequences

- Documentation error

- Dirstate corruption

<table>
<thead>
<tr>
<th>Tool</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>LevelDB-1.10</td>
<td>10</td>
</tr>
<tr>
<td>LevelDB-1.15</td>
<td>6</td>
</tr>
<tr>
<td>LMDB</td>
<td>1</td>
</tr>
<tr>
<td>GDBM</td>
<td>5</td>
</tr>
<tr>
<td>HSQLDB</td>
<td>10</td>
</tr>
<tr>
<td>SQLite-Roll</td>
<td>1</td>
</tr>
<tr>
<td>SQLite-WAL</td>
<td>0</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>1</td>
</tr>
<tr>
<td>Git</td>
<td>9</td>
</tr>
<tr>
<td>Mercurial</td>
<td>10</td>
</tr>
<tr>
<td>VMware</td>
<td>1</td>
</tr>
<tr>
<td>HDFS</td>
<td>2</td>
</tr>
<tr>
<td>ZooKeeper</td>
<td>4</td>
</tr>
</tbody>
</table>

**Total**

- **60**
- **31**
- **10**
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
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

<table>
<thead>
<tr>
<th>Time</th>
<th>Application A</th>
<th>Application B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>pwrite(f1, 0, 150 MB);</code></td>
<td></td>
</tr>
</tbody>
</table>
# False Ordering Dependencies

<table>
<thead>
<tr>
<th>Time</th>
<th>Application A</th>
<th>Application B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>pwrite(f1, 0, 150 MB);</code></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td><code>write(f2, &quot;hello&quot;);</code></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td><code>write(f3, &quot;world&quot;);</code></td>
</tr>
</tbody>
</table>
False Ordering Dependencies

<table>
<thead>
<tr>
<th>Time</th>
<th>Application A</th>
<th>Application B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>pwrite(f1, 0, 150 MB);</td>
<td>write(f2, “hello”);</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>write(f3, “world”);</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>fsync(f3);</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## False Ordering Dependencies

In a globally ordered file system ...

<table>
<thead>
<tr>
<th>Time</th>
<th>Application A</th>
<th>Application B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>pwrite(f1, 0, 150 MB);</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>write(f2, “hello”);</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>write(f3, “world”);</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>fsync(f3);</td>
</tr>
</tbody>
</table>

write(f1) has to be sent to disk before write(f2)
## False Ordering Dependencies

In a globally ordered file system ...

<table>
<thead>
<tr>
<th>Time</th>
<th>Application A</th>
<th>Application B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>pwrite(f1, 0, 150 MB);</code></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td><code>write(f2, “hello”);</code></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td><code>write(f3, “world”);</code></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td><code>fsync(f3);</code></td>
</tr>
</tbody>
</table>

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 | 
3 | 
4 | 

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

<table>
<thead>
<tr>
<th>Time</th>
<th>Application A</th>
<th>Application B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>pwrite(f1, 0, 150 MB);</td>
<td>write(f2, “hello”);</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>write(f3, “world”);</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>fsync(f3);</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Stream Abstraction

New abstraction: Order only within a "stream"

- Each application is usually put into a separate stream

<table>
<thead>
<tr>
<th>Time</th>
<th>Application A</th>
<th>Application B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>pwrite(f1, 0, 150 MB);</td>
<td>write(f2, &quot;hello&quot;);</td>
</tr>
<tr>
<td>2</td>
<td>stream-A</td>
<td>write(f3, &quot;world&quot;);</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>fsync(f3);</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>0.06 seconds</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Time</th>
<th>Application A</th>
<th>Application B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>set_stream(A)</code></td>
<td><code>set_stream(B)</code></td>
</tr>
<tr>
<td></td>
<td><code>pwrite(f1, 0, 150 MB);</code></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td><code>write(f2, &quot;hello&quot;);</code></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td><code>write(f3, &quot;world&quot;);</code></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td><code>fsync(f3);</code></td>
</tr>
</tbody>
</table>
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
Ext4 Journaling: Global Order

Application modifications recorded in main-memory running transaction

Main memory

Application A
Modify blocks #1,#3

Application B
Modify blocks #2,#4

Running transaction

1 3 2 4

On-disk journal
Ext4 Journaling: Global Order

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

Application A
Modify blocks #1,#3

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

Main memory

On-disk journal

Running transaction

1 3 2 4
Ext4 Journaling: Global Order

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

Application A
Modify blocks #1,#3

Application B
Modify blocks #2,#4

fsync()
Further application writes recorded in new running transaction and committed.
Ext4 Journaling: Global Order

Further application writes recorded in new running transaction and committed

Application A
- Modify blocks #1,#3
- Modify blocks #5,#6

Application B
- Modify blocks #2,#4
- fsync()

Running transaction

Main memory

On-disk journal
Further application writes recorded in new running transaction and committed

Application A
Modify blocks #1,#3
Modify blocks #5,#6

Application B
Modify blocks #2,#4
fsync()

Running transaction

Main memory

On-disk journal
Ext4 Journaling: Global Order

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

Main memory

Running transaction

On-disk journal
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

<table>
<thead>
<tr>
<th>Application A</th>
<th>Application B</th>
</tr>
</thead>
</table>
| `set_stream(A)`
 Modify blocks #1,#3 | `set_stream(B)`
 Modify blocks #2,#4 |

Main memory

<table>
<thead>
<tr>
<th>1</th>
<th>3</th>
</tr>
</thead>
</table>

| 2 | 4 |

On-disk journal
CCFS: Stream Order

On fsync(), only that stream is committed

Application A

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

Stream A Transaction

Main memory

1 3

On-disk journal

Application B

- set_stream(B)
- Modify blocks #2,#4
- fsync()

Stream B Transaction

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()`

Main memory

On-disk journal
CCFS: Stream Order

Ordering maintained within stream, re-order across streams!

Application A
- set_stream(A)
- Modify blocks #1,#3

Application B
- set_stream(B)
- Modify blocks #2,#4
- fsync()

Main memory
- stream-A transaction: [1, 3]

On-disk journal
- stream-B transaction: [2, 4]
**CCFS: Multiple Challenges**

Example: Two streams updating adjoining dir-entries

<table>
<thead>
<tr>
<th>Application A</th>
<th>Application B</th>
</tr>
</thead>
<tbody>
<tr>
<td>set_stream(A)</td>
<td></td>
</tr>
<tr>
<td>create(/X/A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>set_stream(B)</td>
<td></td>
</tr>
<tr>
<td>create(/X/B)</td>
<td></td>
</tr>
</tbody>
</table>
CCFS: Multiple Challenges

Example: Two streams updating adjoining dir-entries

<table>
<thead>
<tr>
<th>Block-1 (belonging to directory X)</th>
<th>Application A</th>
<th>Application B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry-A</td>
<td>set_stream(A) create(/X/A)</td>
<td>set_stream(B) create(/X/B)</td>
</tr>
<tr>
<td>Entry-B</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Challenge #1: Block-Level Journaling

Two independent streams can update same block!

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

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

Block-1
Entry-A
Entry-B

Main memory

stream-A transaction

stream-B transaction

?
Challenge #1: Block-Level Journaling

Two independent streams can update same block!

Application A
- set_stream(A)
- create(/X/A)

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

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

Main memory

stream-A transaction
Entry-A

stream-B transaction
Entry-B
Challenge #1: Block-Level Journaling

CCFS solution:
- Record running transactions at byte granularity
- Commit at block granularity

Application A
- set_stream(A)
- create(/X/A)

Application B
- set_stream(B)
- create(/X/B)

Main memory
- Entry-A
- Entry-B

On-disk journal
Challenge #1: Block-Level Journaling

CCFS solution:
Record running transactions at byte granularity
Commit at block granularity

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

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

Main memory

On-disk journal

stream-A transaction

stream-B transaction

begin
Entry-A
Entry-B
end

Entire block-1 committed

Old version of entry-A
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?
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
1. Does CCFS solve application vulnerabilities?

<table>
<thead>
<tr>
<th>Application</th>
<th>Vulnerabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>ext4</td>
<td>1</td>
</tr>
<tr>
<td>ccfs</td>
<td>0</td>
</tr>
<tr>
<td>LevelDB</td>
<td>1</td>
</tr>
<tr>
<td>SQLite-Roll</td>
<td>0</td>
</tr>
<tr>
<td>Git</td>
<td>2</td>
</tr>
<tr>
<td>Mercurial</td>
<td>5</td>
</tr>
<tr>
<td>ZooKeeper</td>
<td>1</td>
</tr>
</tbody>
</table>
Evaluation

1. Does CCFS solve application vulnerabilities?

<table>
<thead>
<tr>
<th>Application</th>
<th>ext4</th>
<th>ccfS</th>
</tr>
</thead>
<tbody>
<tr>
<td>LevelDB</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>SQLite-Roll</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Git</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Mercurial</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>ZooKeeper</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Ext4: 9 Vulnerabilities
- Consistency lost in LevelDB
- Repository corrupted in Git, Mercurial
- ZooKeeper becomes unavailable
Evaluation

1. Does CCFS solve application vulnerabilities?

<table>
<thead>
<tr>
<th>Application</th>
<th>ext4</th>
<th>ccfs</th>
</tr>
</thead>
<tbody>
<tr>
<td>LevelDB</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>SQLite-Roll</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Git</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Mercurial</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>ZooKeeper</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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

![Graph showing throughput normalized to ext4 for various applications]

- Standard benchmarks:
  - varread, randwrite, createfiles, seqwrite, fileserver, webserv
- Real applications:
  - Git, SQLite, LevelDB

*Higher is better*
Evaluation

2. Performance within an application

Throughput normalized to ext4 (Higher is better)

Standard workloads:
Similar performance for ext4, ccfs

But ext4 re-orders!

- varmail
- randwrite
- createfiles
- seqwrite
- fileserver
- webserver
- Git
- SQLite
- LevelDB

Throughput: normalized to ext4 (Higher is better)
Evaluation

2. Performance within an application

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

2. Performance within an application

Throughput normalized to ext4 (Higher is better)

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

<table>
<thead>
<tr>
<th>File system configuration</th>
<th>Atomicity</th>
<th>One sector overwrite</th>
<th>One sector append</th>
<th>Many sector write</th>
<th>Directory operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ext2 async</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
</tr>
<tr>
<td>ext2 sync</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
</tr>
<tr>
<td>ext3 writeback</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
</tr>
<tr>
<td>ext3 ordered</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
</tr>
<tr>
<td>ext3 data-journal</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
</tr>
<tr>
<td>ext4 writeback</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
</tr>
<tr>
<td>ext4 ordered</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
</tr>
<tr>
<td>ext4 no-delalloc</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
</tr>
<tr>
<td>ext4 data-journal</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
</tr>
<tr>
<td>btrfs</td>
<td>❌</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xfs default</td>
<td>❌</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xfs wsync</td>
<td>❌</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>File system configuration</th>
<th>Atomicity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>One sector overwrite</td>
</tr>
<tr>
<td>ext2</td>
<td>async</td>
</tr>
<tr>
<td></td>
<td>sync</td>
</tr>
<tr>
<td>ext3</td>
<td>writeback</td>
</tr>
<tr>
<td></td>
<td>ordered</td>
</tr>
<tr>
<td></td>
<td>data-journal</td>
</tr>
<tr>
<td>ext4</td>
<td>writeback</td>
</tr>
<tr>
<td></td>
<td>ordered</td>
</tr>
<tr>
<td></td>
<td>no-delalloc</td>
</tr>
<tr>
<td></td>
<td>data-journal</td>
</tr>
<tr>
<td>btrfs</td>
<td></td>
</tr>
<tr>
<td>xfs</td>
<td>default</td>
</tr>
<tr>
<td></td>
<td>wsync</td>
</tr>
</tbody>
</table>
Collecting System Call Trace

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

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

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

```plaintext
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

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>

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
Checking ALC on Intermediate States

Multiple Possible Intermediate States

- `.git/tmp (4K)`
- `.git/index (1K)`

- `.git/tmp (4K:garbage)`
- `.git/index.lock (1K)`

- `.git/permanent (4K)`
- `.git/tmp (4K)`
- `.git/index (0K)`

- `git status; git fsck;`

  - ERROR
  - CORRECT OUTPUT
  - CORRECT OUTPUT
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

Unfortunately, file systems make ALC more difficult
Persistence Models: Too Complex

Persistence models used by us to find vulnerabilities

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?
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

ioctl(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

```plaintext
write(file1, "hello")
write(file2, "world")
```

Impossible Intermediate State

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

Allowed Intermediate State

```
/file1 "hello"
/file2 ""
```
Example: Git

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

(ii) git add

(iii) git commit

mkdir(o/x)
creat(o/x/tmp_y)
append(o/x/tmp_y)
fsync(o/x/tmp_y)
link(o/x/tmp_y, o/x/y)
unlink(o/x/tmp_y)
stdout(finished commit)

(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)
Example: Git

Atomicity

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

(ii) git add
creat(o/x/tmp_y)
append(o/x/tmp_y)
fsync(o/x/tmp_y)
link(o/x/tmp_y, o/x/y)
unlink(o/x/tmp_y)

(iii) git commit
mkdir(o/x)
creat(o/x/tmp_y)
append(o/x/tmp_y)
rename(branch.lock,x/branch)
stdout(finished commit)
Example: Git

Ordering

mkdir(o/x)
creat(o/x/tmp_y)
append(o/x/tmp_y)
fsync(o/x/tmp_y)
link(o/x/tmp_y, o/x/y)
unlink(o/x/tmp_y)

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

(iii) git commit
creat(branch.lock)
append(branch.lock)
append(branch.lock)
append(logs/branch)
append(logs/HEAD)
rename(branch.lock,x/branch)
stdout(finished commit)
Example: Git

Durability

1. creat(index.lock)
2. append(index.lock)
3. rename(index.lock,index)
4. stdout(finished add)

(i) store object

1. creat(o/x)
2. creat(o/x/tmp_y)
3. append(o/x/tmp_y)
4. fsync(o/x/tmp_y)
5. link(o/x/tmp_y, o/x/y)
6. unlink(o/x/tmp_y)
7. stdout(finished commit)

(i) store object

1. creat(branch.lock)
2. append(branch.lock)
3. append(branch.lock)
4. append(logs/branch)
5. append(logs/HEAD)
6. rename(branch.lock,x/branch)
7. stdout(finished commit)
Vulnerability Study: Patterns

- HDFS
- ZooKeeper
- VMWare
- LMDB
- GDBM
- LevelDB 1.15
- LevelDB 1.10
- PostgreSQL
- HSQLDB
- Sqlite-WAL
- Sqlite-Roll
- Mercurial
- Git

Legend:
- Green: Across system-call atomicity
- Yellow: Atomicity
- Red: Ordering
- Blue: Durability

Vulnerabilities
Vulnerability Study: Patterns

Across syscall atomicity: Few, minor consequences

[Bar chart showing the comparison of different systems based on their vulnerabilities in terms of atomicity, ordering, and durability.]
Vulnerability Study: Patterns

Garbage during appends cause 4 vulnerabilities

File writes seemingly need only sector-level atomicity

Vulnerabilities

Across system-call atomicity
Atomicty
Ordering
Durability
A separate fsync() on parent directory: 6 vulnerabilities
Vulnerability Study: Patterns

Six applications do not fsync() directory operations

- HDFS
- ZooKeeper
- VMWare
- LMDB
- GDBM
- LevelDB 1.15
- LevelDB 1.10
- PostgreSQL
- HSQLDB
- SQLite-WAL
- SQLite-Roll
- Mercurial
- Git

Vulnerabilities

- Across system-call atomicity
- Atomicsity
- Ordering
- Durability
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

```
- git add file
- creat(index.lock)
- creat(tmp)
- append(tmp, 4K)
- fsync(tmp)
- link(tmp, perm)

.git/index.lock (0)
.git/permanent (4K)
.git/index.lock (0)
.git/index.lock (0)
.git/index.lock (0)
.git/tmp (1)
```

```
git status
git fsck
```

ERROR  CORRECT  ERROR
ALICE: Intermediate States #1

Does application need atomicity across system calls?

Method: Crash after each system call

```plaintext
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

```plaintext
creat(index.lock) ← Crash here
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

```c
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

- creat(figlock)
- 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. Apply various partial effects of examined call

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

Apply these calls
ALICE: Intermediate States #2

Does application need atomicity of an individual system call?

Method:

1. Apply all system calls until examined call

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

2. Apply partial effects of examined call

   (or)
   append(tmp, 2K)
   (or)
   append(tmp, "#@!%^")
   (or)
   append(tmp, 1K)
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
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

```
creat(index.lock)
creat(tmp)
append(tmp, 4K)
fsync(tmp)
link(tmp, perm)
...```
**File System Study: Results**

<table>
<thead>
<tr>
<th>File system configuration</th>
<th>Atomicity</th>
<th>Ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>One sector overwrite</td>
<td>Append content</td>
</tr>
<tr>
<td>ext2</td>
<td>async</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>sync</td>
<td>✓</td>
</tr>
<tr>
<td>ext3</td>
<td>writeback</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>ordered</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>data-journal</td>
<td>✓</td>
</tr>
<tr>
<td>ext4</td>
<td>writeback</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>ordered</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>no-delalloc</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>data-journal</td>
<td>✓</td>
</tr>
<tr>
<td>btrfs</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>xfs</td>
<td>default</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>wsync</td>
<td>✓</td>
</tr>
</tbody>
</table>

One-sector-overwrite atomicity is due to current hardware, might change with NVMs
## File System Study: Results

<table>
<thead>
<tr>
<th>File system configuration</th>
<th>Atomicity</th>
<th>Ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>One sector overwrite</td>
<td>Append content</td>
</tr>
<tr>
<td>ext2</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>async</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>sync</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ext3</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>writeback</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ordered</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>data-journal</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ext4</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>writeback</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ordered</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>no-delalloc</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>data-journal</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>btrfs</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>xfs</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>default</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>wsync</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

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

Application Workload

ALICE

System-call Trace

Explorer

APM: Abstract Persistence Model

Crash state #1 (Violates atomicity of syscall-1)

Crash state #2 (Violates ordering of syscall-1 and 2)

Correct

Incorrect

Application Checker

Crash vulnerability: Re-ordering syscall-1 and 2
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

```
creat(file2);
write(file2, "hello");
fsync(file2)
```
Challenge #2: Space Reuse

```c
write(file3,150MB);
truncate(file1);
```

Challenge #2: Space Reuse

Inode
File1
Data

Inode
File2
Data

trim(file1);

create(file2);
Challenge #2: Space Reuse

```plaintext
write(file1, 150MB);
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)

write(file3, 150MB);
truncat(file1);

Stream 2
(Application 2)

creat(file2);
write(file2, "hello");
Challenge #2: Space Reuse

If only one stream commits, FS consistency will be affected

Possible crash state

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