Understanding and Dealing with Hard Faults in Persistent Memory Systems

Brian Choi, Randal Burns, Ryan Huang

EuroSys ‘21
Summary

- We introduce a new class of faults in PM systems: **Soft-to-Hard Faults**

- In-depth study: 28 **real-world examples** in 7 PM systems

- Solution: Arthas, tool to **recover PM systems** that suffer from Soft-to-Hard Faults to a correct working state
  - minimal data loss
Errors Frequently Occur in Systems
First Thing to Try? Restart!
Soft and Hard Faults

- **“Soft” faults** -> Restart **fixes** the system
  - Bad volatile states: go away upon restart
  - Many production failures are soft/transient (Gray, 1985)

- **“Hard” faults** -> Faults are **recurring** even after restart
  - Bad states written to storage (ie. disk) permanently exist after restart
Soft and Hard Faults

- **“Soft” faults** -> Restart fixes the system
  - Bad volatile states: go away upon restart
  - Many production failures are soft/transient (Gray, 1985)

- **“Hard” faults** -> Faults are *recurring* even after restart
  - Bad states written to storage (ie. disk) permanently exist after restart

- **Hard faults are particularly prominent in persistent memory**
  - Introduces new hard faults in your system
  - Restart not effective anymore!!
Persistent Memory = More Options

- Fast new storage technology called Persistent Memory (PM)
  - Latency comparable to DRAM
  - Persistently store data
- Developers now have more options to persist data
  - PM much faster than previous storage
  - More states become persistent (i.e., cache)

<table>
<thead>
<tr>
<th></th>
<th>Sequential Read</th>
<th>Random Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAM</td>
<td>81.4 ns</td>
<td>83.2 ns</td>
<td>157.7 ns</td>
</tr>
<tr>
<td>PMEM</td>
<td>179.0 ns</td>
<td>317.6 ns</td>
<td>160.4 ns</td>
</tr>
</tbody>
</table>
Soft-to-Hard Faults

- PM persists many states
  - "Soft" states move to PM more frequently, become "hard" states
  - Restart can’t fix any issues with these "Soft-to-Hard" states

[Diagram showing memory leaks, indexing structures, and PM integration.]

- Soft Good States in Volatile Memory after Restart
- Hard States in Persistent Memory
A Real Soft-to-Hard Fault

- Soft Fault in Memcached
- Items and client connection data all in volatile memory
- Two bad states: refcount and h_next corrupted

https://github.com/memcached/memcached/issues/271
A Real Soft-to-Hard Fault

The two bad soft states go away after restart
A Real Soft-to-Hard Fault

Traditional, Disk-based System

Restart

Persistent Memory Integration

Restart
Our Contributions

- Hard Fault Study where we examine 28 real-world bugs in PM systems and analyze the bugs.

- Arthas: tool to recover PM systems that suffer from Soft-to-Hard Faults
  - Dependency-based rollback - minimize data loss
  - Static Analysis - to aid dependency formulation
Hard Fault Study

- We found 28 bugs from 7 PM systems that demonstrate the soft-to-hard fault problem.

<table>
<thead>
<tr>
<th>System</th>
<th>Type</th>
<th>Cases</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Redis</td>
<td>Port</td>
<td>11</td>
<td>in-memory key-value store</td>
</tr>
<tr>
<td>Memcached</td>
<td>Port</td>
<td>9</td>
<td>in-memory key-value store</td>
</tr>
<tr>
<td>Recipe</td>
<td>Port</td>
<td>2</td>
<td>ported concurrent indexes to PM</td>
</tr>
<tr>
<td>LevelHash</td>
<td>New</td>
<td>2</td>
<td>PM hashing index scheme</td>
</tr>
<tr>
<td>PMEMKV</td>
<td>New</td>
<td>2</td>
<td>PM key-value store</td>
</tr>
<tr>
<td>Dash</td>
<td>New</td>
<td>1</td>
<td>scalable PM hashtable</td>
</tr>
<tr>
<td>CCEH</td>
<td>New</td>
<td>1</td>
<td>PM cacheline-conscious indexing structure</td>
</tr>
</tbody>
</table>
Finding 1: Root Causes Are Diverse

- PM Hard Faults can be caused by **many different types** of root causes.
- Logic Errors are most prevalent (46%), but other types of root causes are relatively **evenly distributed**.
Finding 2: Consequences Are Severe

- PM Hard Faults cause **severe problems**, such as **repeated crashes** and **wrong results**, not just minor issues.
Finding 3: Over Half Propagate Bad State

Type I: PM variable has a bad value that directly causes the problem

Type II: PM variable has a bad value that propagates bad state and indirectly causes the problem

<table>
<thead>
<tr>
<th>Type</th>
<th>Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type I (direct)</td>
<td>18%</td>
</tr>
<tr>
<td>Type II (indirect propagation)</td>
<td>68%</td>
</tr>
<tr>
<td>Type III (miscellaneous)</td>
<td>14%</td>
</tr>
</tbody>
</table>

- 68% of the bugs propagate bad state among volatile and persistent variables throughout the system.
PM Hard Faults Lead to Bad State

- PM hard faults are a diverse set of bugs
  - Challenging to statically find all bugs with one solution
- Insight: At runtime, PM hard faults eventually cause bad state to be persisted
  - Revert bad states to good states

![Diagram showing Logic Error, Integer Overflow, Memory Leak, and Bad state eventually gets persisted]
We Must Revert the Root Cause

- 68% of the bugs propagate bad state
  - Design Principle: Even when one bad PM state is rolled back, the PM system could still quickly hit the same failure if the root cause of the bad state is not reverted.

Diagram:
- Bad state
- Good state
- Restart system
- Reoccurs
- Revert bad state
- Crash
How to Revert Bad States?

Goal of effective mitigation: hard fault disappears and doesn’t reappear again

- Rollback bad states to a previous, older version that is a good state
- Checkpoint data: we keep multiple versions of state
- Revert all propagated bad states including the root cause
Standard Checkpoint/Rollback

```
{  
  {key1: value1},  
  {key2: value2}  
}
```

SNAPSHOT
Standard Checkpoint/Rollback

SNAPSHOT

\[
\{ \\
{\text{key1}}: \text{value1}, \\
{\text{key2}}: \text{value2} \\
\}
\]

time: \(t\)

crash

\[
\{ \\
{\text{key1}}: \text{value1}, \\
{\text{key2}}: \text{value2}, \\
{\text{key3}}: \text{value3}, \\
{\text{key4}}: \text{value4}, \\
\vdots \\
{\text{key10000}}: \text{ERROR}, \\
{\text{key10001}}: \text{value6}, \\
\}
\]

time: \(t+n\)
Standard Checkpoint/Rollback

At time $t$:

```
{
  {key1: value1},
  {key2: value2}
}
```

At time $t+n$:

```
{
  {key1: value1},
  {key2: value2},
  {key3: value3},
  {key4: value4},
  ...
  {key10000: ERROR},
  {key10001: value6},
}
```

**SNAPSHOT**

**crash**

**root cause**
Standard Checkpoint/Rollback

Loses keys 3 to 10,001 when only key 10,000 was a bad state
Standard Checkpoint/Rollback

Standard Checkpointing loses an unnecessary amount of data

Loses keys 3 to 10,001 when only key 10,000 was a bad state
Design Goal - Minimal Data Loss

- **Standard Checkpointing** approaches lose too much data
- **Design Principle**: Use static analysis on PM system to find dependencies of the **bad PM variables** and revert only the **bad PM states** using these dependencies

```plaintext
{  
  {key1: value1},  
  {key2: value2},  
  {key3: value3},  
  {key4: value4},  
  ...  
  {key10000: ERROR},  
  {key10001: value6},  
}
```

Use Static Analysis

```plaintext
{  
  {key1: value1},  
  {key2: value2},  
  {key3: value3},  
  {key4: value4},  
  ...  
  {key10001: value6},  
}
```

**root cause**

```plaintext
{key10000: ERROR}
```

**crash**
Arthas: Overview

- Arthas: tool that recovers PM systems from PM hard faults

**Techniques**
- Checkpoint old versions
- Static analysis and dynamic tracing
- Dependency-based rollback

**Goals**
- Recover PM system quickly
- Minimize data loss
- Small runtime overhead
Arthas Workflow

PM System Code

Arthas Analyzer

Program Dependency Graph

Instrumented PM System

Checkpoint Library

Detector

Reactor

PM Checkpoint File
Checkpoint Library

- Checkpoint multiple versions of PM state to later revert
- Implementation: Intercepts PM framework API calls
- Global sequence numbers assigned to each PM update
  - ensure order when reverting

```c
pmem_ptr = pmem_alloc();
```

<table>
<thead>
<tr>
<th>cp_entry</th>
<th>0xf4a000</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence num</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sizes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Checkpoint Library

- Checkpoint multiple versions of PM state to later revert
- Implementation: Intercepts PM framework API calls
- Global sequence numbers assigned to each PM update
  - ensure order when reverting

```c
pmem_ptr = pmem_alloc();
*pmem_ptr = 7;
pmem_flush(pmem_ptr, sizeof(int));
```

<table>
<thead>
<tr>
<th>cp_entry</th>
<th>0xf4a000</th>
<th>data</th>
<th>sequence num</th>
<th>sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sizes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cp_entry</th>
<th>0xf4a000</th>
<th>data</th>
<th>sequence num</th>
<th>sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence</td>
<td></td>
<td>1</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>sizes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Checkpoint Library

- **Checkpoint multiple versions of PM state to later revert**
- Implementation: Intercepts PM framework API calls
- Global sequence numbers assigned to each PM update
  - Ensure order when reverting

```c
pmem_ptr = pmem_alloc();
*pmem_ptr = 7;
pmem_flush(pmem_ptr, sizeof(int));

pmem_ptr = pmem_alloc();
*pmem_ptr = 4;
pmem_flush(pmem_ptr, sizeof(int));
```

<table>
<thead>
<tr>
<th>cp_entry</th>
<th>0xf4a000</th>
<th>data</th>
<th>sequence num</th>
<th>sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cp_entry</th>
<th>0xf4a000</th>
<th>data</th>
<th>sequence num</th>
<th>sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>7</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cp_entry</th>
<th>0xf4a000</th>
<th>data</th>
<th>sequence num</th>
<th>sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>7</td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cp_entry</th>
<th>0xf4a000</th>
<th>data</th>
<th>sequence num</th>
<th>sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cp_entry</th>
<th>0xf4a000</th>
<th>data</th>
<th>sequence num</th>
<th>sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cp_entry</th>
<th>0xf4a000</th>
<th>data</th>
<th>sequence num</th>
<th>sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>7</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cp_entry</th>
<th>0xf4a000</th>
<th>data</th>
<th>sequence num</th>
<th>sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
Analyzer: Dependency-based Rollback

- Dependency-based Rollback: More targeted reversion
- Use dependencies to only revert necessary bad states.
- Program Dependency Graph of PM System
  - Fault instruction -> starting point of dependency analysis
  - Slice: see what instructions influence this fault
Analyzer: Dependency-based Rollback

- This is the Persistent Memory Write Timeline of a PM system

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>3</td>
</tr>
<tr>
<td>x</td>
<td>-5</td>
</tr>
<tr>
<td>b</td>
<td>7</td>
</tr>
<tr>
<td>y</td>
<td>x+5</td>
</tr>
<tr>
<td>z</td>
<td>2/y</td>
</tr>
</tbody>
</table>

```
a = 3;
x = -5;
b = 7;
y = x+5;
z = 2/y;
```
Analyzer: Dependency-based Rollback

- Crash at z, root cause due to $x = -5$;

```plaintext
a = 3;
x = -5;
b = 7;
y = x + 5;
z = 2 / y;
```

- volatile write
- PM write
- dependency

root cause

crash
Arthas Analyzer: Slicing

Slice: to preserve data dependencies during reversion

- a = 3;
- x = -5;
- b = 7;
- y = x + 5;
- z = 2/y;

Slice:
- x = -5;
- y = x + 5;
- z = 2/y;

root cause

crash
Arthas Analyzer: Purge Mode

- Minimizes data loss, but may not lead to a perfectly consistent system

```
a = 3;

Revert;

b = 7;

y Revert;

z Revert;
```

Slice:

```
x = -5;

y = x+5;

z = 2/y;
```

Root cause:

crash
Arthas Analyzer: Rollback Mode

- More conservative approach: reverts between dependent updates
- Captures dependencies of the system

<table>
<thead>
<tr>
<th>a = 3;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = -5;</td>
</tr>
<tr>
<td>Revert</td>
</tr>
<tr>
<td>b = 7;</td>
</tr>
<tr>
<td>y Revert;</td>
</tr>
<tr>
<td>z = 2/y;</td>
</tr>
<tr>
<td>Revert</td>
</tr>
</tbody>
</table>

Slice:

- x = -5;
- y = x+5;
- z = 2/y;

Root cause:

- x = -5;

Crash:
Evaluation

- One 8-core CPU (2.50GHz) and two 128 GB Intel Optane DC Persistent Memory DIMMs.
- We test on 12 bugs both from our study and other existing bugs.
- We run Arthas against two baselines:
  - pmCRIU: A state-of-the-art checkpoint and rollback system.
  - ArCkpt: Alternate version of Arthas that reverts without the analyzer component, time-based.

<table>
<thead>
<tr>
<th>Type</th>
<th>Memcached</th>
<th>Redis</th>
<th>Pelikan</th>
<th>PMEMKV</th>
<th>CCEH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bugs</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
12 Real-World Bugs

<table>
<thead>
<tr>
<th>No.</th>
<th>System</th>
<th>Fault</th>
<th>Consequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1</td>
<td>Memcached</td>
<td>Refcount Overflow</td>
<td>Data loss</td>
</tr>
<tr>
<td>f2</td>
<td>Memcached</td>
<td>flush_all logic bug</td>
<td>Data loss</td>
</tr>
<tr>
<td>f3</td>
<td>Memcached</td>
<td>Hashtable lock data race</td>
<td>Data loss</td>
</tr>
<tr>
<td>f4</td>
<td>Memcached</td>
<td>Integer overflow in append</td>
<td>Segfault</td>
</tr>
<tr>
<td>f5</td>
<td>Memcached</td>
<td>Rehashing flag bit flip</td>
<td>Data loss</td>
</tr>
<tr>
<td>f6</td>
<td>Redis</td>
<td>Listpack buffer overflow</td>
<td>Segfault</td>
</tr>
<tr>
<td>f7</td>
<td>Redis</td>
<td>Logic bug in refcount</td>
<td>Server panic</td>
</tr>
<tr>
<td>f8</td>
<td>Redis</td>
<td>slowlogEntry leak</td>
<td>Persistent leak</td>
</tr>
<tr>
<td>f9</td>
<td>CCEH</td>
<td>Directory doubling bug</td>
<td>Infinite loop</td>
</tr>
<tr>
<td>f10</td>
<td>Pelikan</td>
<td>Value Length overflow</td>
<td>Segfault</td>
</tr>
<tr>
<td>f11</td>
<td>Pelikan</td>
<td>Null stats response</td>
<td>Segfault</td>
</tr>
<tr>
<td>f12</td>
<td>PMEMKV</td>
<td>Asynchronous lazy free</td>
<td>Persistent leak</td>
</tr>
</tbody>
</table>

We evaluate on a diverse set of bugs with varying consequences
Effectiveness

- Arthas is able to resolve 12 out of the 12 bugs
- ArCkpt suffers from timeouts: 2 out of the 12 bugs
- pmCRIU is only able to reliably mitigate 9 out of the 12 bugs.
Arthas discards 10x less data than pmCRIU
Arthas is slightly slower, but is still in an acceptable range of approximately one minute more than pmCRIU while also minimizing data loss.
Conclusion

- Soft-to-Hard Faults are an underexplored, yet significant problem for new PM systems.
- Arthas reliably detects and mitigates Hard Faults in PM systems
  - Dependency-based rollback - minimize data loss
- Mitigate 12 Hard Faults from 5 PM systems with 10x less data loss than pmCRIU with a reasonable performance overhead.
- Our tool is publicly available at https://github.com/OrderLab/Arthas