Pangolin: A Fault-tolerant Persistent Memory Programming Library

Lu Zhang and Steven Swanson

Non-Volatile Systems Laboratory
Department of Computer Science & Engineering
University of California, San Diego
Persistent memory (PMEM) finally arrives

- Working alongside DRAM
- New programming model
  - Byte addressability
  - Memory semantics
  - Direct access (DAX)
Challenges with PMEM programming

• Crash consistency
  – Volatile CPU caches
  – 8-byte store atomicity

• Fault tolerance
  – Media errors
  – Software bugs
Persistent memory error types

- Persistent memory and its controller implement ECC
  - ECC-detectable & correctable errors do not need software intervention
Persistent memory error types

- Persistent memory and its controller implement ECC
  - ECC-detectable & correctable errors do not need software intervention
  - ECC-detectable but uncorrectable ones require signal handling
Persistent memory error types

• Persistent memory and its controller implement ECC
  – ECC-detectable & correctable errors do not need software intervention
  – ECC-detectable but uncorrectable ones require signal handling
  – ECC-undetectable errors demand software detection and correction
Handle uncorrectable & undetectable errors

- Prepare some redundancy for recovery
- Implement software-based error detection and correction

![Diagram]

Application ➔ SIGBUS ➔ Receiving SIGBUS
PMEM Controller ➔ Data X ➔ Error Detected but uncorrectable
PMEM ➔ Data X

Application ➔ Data X ➔ Receiving bad data
PMEM Controller ➔ Data X
PMEM ➔ Data X ➔ Error undetectable
DAX-filesystem cannot protect mmap’ed data

- Some filesystems (e.g. NOVA) provide protection only via read()/write()
- No known filesystem can protect DAX-mmap’ed PMEM data
DAX-filesystem cannot protect mmap’ed data

- Some filesystems (e.g. NOVA) provide protection only via read()/write()
- No known filesystem can protect DAX-mmap’ed PMEM data
Pangolin design goals

• Ensure crash consistency
• Protect application data against media and software errors
• Require very low storage overhead (1%) for fault tolerance
Pangolin – Replication, parity, and checksums

• Combines replication and parity as redundancy
  – Similar performance compared to replication
  – Low space overhead (1% of gigabyte-sized object store)

• Checksums all metadata and object data
Pangolin – Transactions with micro-buffering

• Provides micro-buffering-based transactions
  – Buffers application changes in DRAM
  – Atomically updates objects, checksums, and parity
Pangolin – Transactions with micro-buffering

- Provides micro-buffering-based transactions
  - Buffers application changes in DRAM
  - Atomically updates objects, checksums, and parity
Pangolin’s data redundancy

- Reserve space for metadata replication and object parity
- Organize object data pages into “rows”

Row size: default 160 MB (1% of a data “zone”)

Row 0 | Row 1 | Row 2 | Row 3 | Row p
Pangolin’s parity coding

- Compute a parity page vertically across all rows
- Afford losing one whole row of data
- By default, Pangolin implements 100 rows per data zone

\[
\begin{align*}
\text{Row 0} & : \quad \text{Page 0} \oplus \text{Page 1} \\
\text{Row 1} & : \quad \text{Page 2} \oplus \text{Page 3} \\
\text{Row 2} & : \quad \text{Page 4} \oplus \text{Page 5} \\
\text{Row 3} & : \quad \text{Page 6} \oplus \text{Page 7} \\
\text{Row p} & : \quad \text{Page 8} \oplus \text{Page 9}
\end{align*}
\]
Micro-buffering provides transactions

- Move object data in DRAM and perform data integrity check
- Buffer writes to objects and write back to PMEM on commit
- Guarantee consistency with redo logging (replicated)
Updating parity using only modified ranges

\[ \text{obj 1} \oplus \Delta 1 = P' \]

Row 0

Row 1

Row 2

Row 3

Row p
Parity’s crash consistency depends on object logs

- Apply all redo-logs (if exist) and then re-compute parity
Parity’s crash consistency depends on object logs

- Apply all redo-logs (if exist) and then re-compute parity
Multithreaded update – Lock parity ranges

• Lock a range of parity and serialize parity updates
Multithreaded update – Atomic XORs

- Parity range can update, lock-free, with atomic XORs

**Diagram:**

- **Thread1:**
  - D1’ → ⊕ = Δ1

- **Thread2:**
  - D7’ → ⊕ = Δ7

**Diagram Elements:**

- **DRAM:**
  - D1'
  - D7'

- **PMEM:**
  - D1'
  - D7'

**Data Matrix:**

<table>
<thead>
<tr>
<th></th>
<th>obj 1</th>
<th>D1</th>
<th>obj 2</th>
<th>obj 3</th>
<th>obj 4</th>
<th>obj 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row 2</td>
<td>obj 6</td>
<td></td>
<td>obj 7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row 3</td>
<td>obj 7</td>
<td>D7</td>
<td>obj 8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row p</td>
<td>obj 9</td>
<td></td>
<td></td>
<td>unused (zero bytes)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parity</td>
<td>Parity</td>
<td>P17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Mathematical Equations:**

- \( D1' \oplus \Delta_1 = \Delta_1 
- \( D7' \oplus \Delta_7 = \Delta_7 \)
Multithreaded update – Hybrid scheme

- Atomic XORs can be slower than vectorized ones
- Use shared mutex to coordinate both methods
- Small updates (< 8KB)
  - Take shared lock of a parity range (8 KB)
  - Update parity concurrently with atomic XORs
- Large updates (≥ 8KB)
  - Take exclusive locks of parity ranges (8 KB each)
  - Update parity using vectorized XORs (non-atomic)
Performance – Single-object transactions

- Evaluation based on Intel’s Optane DC persistent memory
- On average, Pangolin’s latency is 11% lower than libpmemobj with replication.
Performance – Multi-object transactions

- Performance of Pangolin is 90% of libpmemobj’s with replication
- Pangolin incurs about $100 \times$ less space overhead

### Average Insertion Latencies

- **libpmemobj**
- **libpmemobj-replication**
- **pangolin**

### Average Removal Latencies

- **libpmemobj**
- **libpmemobj-replication**
- **pangolin**
Conclusion

• PMEM programming libraries should also consider fault tolerance for critical applications.
• Parity-based redundancy provides similar performance compared to replication and significantly reduces space overhead.
• Micro-buffering-based transactions can both support crash consistency and provide fault tolerance.