Optimizing Persistent Memory Transactions

PanteA Zardoshti, Tingzhe Zhou, Yujie Liu* and Michael Spear
Lehigh University
*Google

PACT 2019
24 September
Persistent Transactional Memory

**Note.** All experiments use a single Intel Xeon Platinum 8160 processor, and assume the “PDOM-1” model of persistence. Flush latencies are accurate, memory latencies are not.
Why?

The fundamental difference between STM and PTM is that STM assumes that concurrency is a dynamic property, but PTM can assert that persistence is a static property.
Outline

• Background
  • Persistent Domains
  • Programing Models
  • Logging Techniques

• Optimizations and Performance
  • All PTM Algorithms
  • Specific PTM Algorithms
Non-Volatile Memory Technologies

Low-Level ISA for Ordering to PDOM

- **clwb**
  - write back cache line to MC
- **sfence**
  - ensure previous flushes have completed
- **pcommit (deprecated)**
  - persist all accepted stores at MC

Our focus: PDOM-1

Source: Persistent Memory Transactions, Marathe et al., arXiv
Data Accessed by Transactions

- **General Persistence Model (GP)**
  - A transaction can access **both** DRAM and NVM
  - Reads, writes of DRAM, reads of NVM from **outside** of transaction are allowed

- **Ideal Persistence Model (IP)**
  - A transaction only accesses **one type** of Memory: DRAM or NVM
  - Every access to NVM is performed from within a transaction
Naïve Concurrent Persistence

• Begin with software transactional memory algorithm
  • Metadata: per-thread logs plus global metadata (e.g., lock table)
  • Instrumentation:
    • BEGIN: take a checkpoint, initialize per-thread metadata
    • READ: add a location to the transaction if it can be read consistently (validate)
    • WRITE: speculatively update a location (undo or redo log)
    • COMMIT: finish the two-phase locking routine, perform optimistic validation, ...
    • ABORT: roll back writes, clean up local metadata, restore checkpoint
  • Eager or lazy data versioning, variety of conflict detection mechanisms
• Add fences and flushes
## Undo and Redo Logging

### Undo Algorithms

<table>
<thead>
<tr>
<th>Memory</th>
<th>Undo-Log</th>
<th>Memory</th>
<th>Undo-Log</th>
<th>Memory</th>
<th>Undo-Log</th>
<th>Memory</th>
<th>Undo-Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>X = 0</td>
<td>X = 5</td>
<td>X = 5</td>
<td>X = 5</td>
<td>X = 5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y = 0</td>
<td>Y = 0</td>
<td>Y = 0</td>
<td>Y = 5</td>
<td>Y = 5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Commit

- Write-back

#### Abort

- Write-back

### Redo Algorithms

<table>
<thead>
<tr>
<th>Memory</th>
<th>Redo-Log</th>
<th>Memory</th>
<th>Redo-Log</th>
<th>Memory</th>
<th>Redo-Log</th>
<th>Memory</th>
<th>Redo-Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>X = 0</td>
<td>X = 0</td>
<td>X = 0</td>
<td>X = 0</td>
<td>X = 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y = 0</td>
<td>Y = 0</td>
<td>Y = 0</td>
<td>Y = 0</td>
<td>Y = 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Commit

- Write-back

#### Abort

- Write-back

#### Persistent

```c
global int x = 0, y = 0;
...
persistent {
    x = 5; y = x;
}
```

- Update memory directly
- Buffer old values in undo log
- Read in place

#### Faster Commit

- Faster read
- Faster abort

#### Slower Commit

- Slower abort
- Slower read

- Buffer updates in redo log
- Indirect reads through redo-log
- Replay redo-log during commit
Undo and Redo Logging: PDOM-1 Additions

Undo Algorithms

Redo Algorithms
Experimental Setup

- **Platform**
  - LLVM TransMem – works with LLVM 5+

- **Environment**
  - 24 cores/48 threads 2.1GHz Intel Xeon Platinum 8160
  - Red Hat Linux 7.4, LLVM-Clang 6.0, -O3 optimization.
  - 192G RAM (not persistent, however, `clwb` incurs accurate latencies.)

- **Algorithms**
  - 7 STM
  - 8 PTM

- **Benchmark**
  - real-world application Memcached
  - write-only: TPCC (New Order), TATP (Location Update), B+-Tree
  - Vacation (travel reservation)
STM vs. Naïve PTM – TPCC-B+-Tree (GP)

[ Felber et al. PPoPP 2008] (“Orec-eager/lazy”)
[ Volos et al. ASPLOS 2011] (“Orec-mixed”)
[ Dalessandro et al. PPoPP 2010] (“NOrec”)
[ Spear et al. SPAA 2008] (“Ring”)
[ Dice et al. SPAA 2010] (“TLRW”)

[Image: Graph showing throughput versus number of threads for different transaction types.]
Dynamic Captured Persistent Memory

**Purpose:** Avoid memory access instrumentation for newly allocated memory in transactions

**Implementation Details:**

1. Track the last allocated memory chunk with addr and size. Record all allocated memory with \(<\text{addr}, \text{size}\>\) in set

2. Access directly in memory if the address within range of most recent \([\text{addr}, \text{addr + size}]\)

3. clwb the set before PTM commits or becomes recoverable

---

### Optimizing Persistent Memory Transactions

<table>
<thead>
<tr>
<th></th>
<th>TPCC-HashTable</th>
<th>TPCC-B+Tree</th>
<th>B+Tree (Insert)</th>
</tr>
</thead>
<tbody>
<tr>
<td>p-lock-eager</td>
<td>1.674</td>
<td>1.543</td>
<td>1.235</td>
</tr>
<tr>
<td>p-lock-lazy</td>
<td>1.055</td>
<td>1.045</td>
<td>1.041</td>
</tr>
<tr>
<td>p-orec-eager</td>
<td>1.674</td>
<td>1.443</td>
<td>1.126</td>
</tr>
<tr>
<td>p-orec-lazy</td>
<td>1.115</td>
<td>1.101</td>
<td>1.048</td>
</tr>
<tr>
<td>p-norec</td>
<td>1.107</td>
<td>1.081</td>
<td>1.061</td>
</tr>
<tr>
<td>p-ring</td>
<td>1.068</td>
<td>1.093</td>
<td>1.011</td>
</tr>
<tr>
<td>p-tlw</td>
<td>1.626</td>
<td>1.358</td>
<td>1.127</td>
</tr>
<tr>
<td>p-orec-mixed</td>
<td>1.118</td>
<td>1.125</td>
<td>1.088</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Vacation (low)</th>
<th>Vacation (high)</th>
<th>Memcached</th>
</tr>
</thead>
<tbody>
<tr>
<td>p-lock-eager</td>
<td>1.190</td>
<td>1.139</td>
<td>1.01</td>
</tr>
<tr>
<td>p-lock-lazy</td>
<td>1.026</td>
<td>1.021</td>
<td>1.01</td>
</tr>
<tr>
<td>p-orec-eager</td>
<td>1.179</td>
<td>1.177</td>
<td>1.272</td>
</tr>
<tr>
<td>p-orec-lazy</td>
<td>1.067</td>
<td>1.069</td>
<td>1.12</td>
</tr>
<tr>
<td>p-norec</td>
<td>1.052</td>
<td>1.024</td>
<td>0.999</td>
</tr>
<tr>
<td>p-ring</td>
<td>1.003</td>
<td>1.092</td>
<td>1.031</td>
</tr>
<tr>
<td>p-tlw</td>
<td>1.173</td>
<td>1.162</td>
<td>1.264</td>
</tr>
<tr>
<td>p-orec-mixed</td>
<td>1.099</td>
<td>1.058</td>
<td>1.026</td>
</tr>
</tbody>
</table>

Eager algorithms up to **67%** speedup

Lazy algorithms up to **12%** speedup
Coarse-Grained Logging + Alignment

Initially:

shared global char X = ‘\0’;
private global char Y = ‘\0’;

Thread 1: persist {
    X = ‘a’;
}

Thread 2: Y = ‘b’;

Granularity problems occur if STM manages data on a per-word or per-cache-line basis.

STM Solutions: (a) log at the granularity of individual bytes; or (b) accompany each coarse log entry with a bitmap

PTM Solutions: Persistence is not a dynamic property. X and Y must both be accessed via persistent transactions.
Coarse-Grained Logging + Alignment: performance

Only suitable for IP model
(IP model requires all NVM accesses to happen inside of transactions)

- 32-byte granularity provided most stable performance
  - Accelerated and simplified log checking and write-back
  - Up to 51% speedup for eager algorithms
  - Up to 34% speedup for lazy algorithms
  - Slowdowns for NOrec: granularity affects other parts of algorithm

<table>
<thead>
<tr>
<th></th>
<th>TPCC-HashTable</th>
<th>TPCC-B+Tree</th>
<th>TATP</th>
<th>B+Tree (Insert)</th>
<th>B+Tree (Mix)</th>
</tr>
</thead>
<tbody>
<tr>
<td>p-lock-eager</td>
<td>1.229</td>
<td>1.519</td>
<td>1.049</td>
<td>1.366</td>
<td>1.167</td>
</tr>
<tr>
<td>p-lock-lazy</td>
<td>1.086</td>
<td>1.227</td>
<td>1.296</td>
<td>1.226</td>
<td>1.229</td>
</tr>
<tr>
<td>p-orec-eager</td>
<td>1.185</td>
<td>1.464</td>
<td>1.224</td>
<td>1.406</td>
<td>1.109</td>
</tr>
<tr>
<td>p-orec-lazy</td>
<td>1.041</td>
<td>1.124</td>
<td>1.084</td>
<td>1.113</td>
<td>1.096</td>
</tr>
<tr>
<td>p-no rec</td>
<td>0.423</td>
<td>0.315</td>
<td>0.905</td>
<td>0.750</td>
<td>1.042</td>
</tr>
<tr>
<td>p-ring</td>
<td>1.022</td>
<td>1.121</td>
<td>1.107</td>
<td>1.158</td>
<td>1.075</td>
</tr>
<tr>
<td>p-thr</td>
<td>1.251</td>
<td>1.347</td>
<td>1.116</td>
<td>1.357</td>
<td>1.183</td>
</tr>
<tr>
<td>p-orec-mixed</td>
<td>1.088</td>
<td>1.113</td>
<td>1.052</td>
<td>1.063</td>
<td>1.086</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Vacation (low)</th>
<th>Vacation (high)</th>
<th>Memcached</th>
</tr>
</thead>
<tbody>
<tr>
<td>p-lock-eager</td>
<td>1.231</td>
<td>1.197</td>
<td>1.11</td>
</tr>
<tr>
<td>p-lock-lazy</td>
<td>1.218</td>
<td>1.205</td>
<td>1.03</td>
</tr>
<tr>
<td>p-orec-eager</td>
<td>1.185</td>
<td>1.196</td>
<td>1.16</td>
</tr>
<tr>
<td>p-orec-lazy</td>
<td>1.104</td>
<td>1.120</td>
<td>1.114</td>
</tr>
<tr>
<td>p-no rec</td>
<td>0.651</td>
<td>0.567</td>
<td>1.02</td>
</tr>
<tr>
<td>p-ring</td>
<td>1.076</td>
<td>1.058</td>
<td>1.022</td>
</tr>
<tr>
<td>p-thr</td>
<td>1.155</td>
<td>1.177</td>
<td>1.139</td>
</tr>
<tr>
<td>p-orec-mixed</td>
<td>1.144</td>
<td>1.097</td>
<td>1.055</td>
</tr>
</tbody>
</table>
TM-Specific Fence Pipelining

- **Purpose**: Reduce number of *sfences* by deferring the in-place write

<table>
<thead>
<tr>
<th></th>
<th>TPCC-HashTable</th>
<th>TPCC-B+Tree</th>
<th>TATP</th>
<th>B+Tree (Insert)</th>
<th>B+Tree (Mix)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup</td>
<td>13.3%</td>
<td>14.2%</td>
<td>0.67%</td>
<td>10.91%</td>
<td>2.5%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Vacation (low)</th>
<th>Vacation (high)</th>
<th>Memcached</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup</td>
<td>2.85%</td>
<td>4.05%</td>
<td>9.07%</td>
</tr>
</tbody>
</table>

- TLRW PTM already has one sfence per read and one CAS per write: can we use it to order clwb?

Readers/writer locking in TLRW introduces sfence for every read and write

TxWrite: 1. write to undo log; 2. sfence; 3. in-place update;

Defer until the next instrumentation.

At 48 threads, TLRW performs 1.7x better than other PTM on Vacation, compared to 1.15x without pipelining.
### Performance: Deferred Flushing – Lazy TM

Redo-based PTM performs clwb’s to main memory while holding locks → enlarged critical sections

<table>
<thead>
<tr>
<th></th>
<th>TPCC-HashTable</th>
<th>TATP</th>
<th>B+Tree (Insert)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Threads</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>p-lock-lazy</strong></td>
<td>1.199</td>
<td>1.003</td>
<td>1.258</td>
</tr>
<tr>
<td><strong>p-orec-lazy</strong></td>
<td>1.026</td>
<td>1.052</td>
<td>1.007</td>
</tr>
<tr>
<td><strong>p-norec</strong></td>
<td>1.108</td>
<td>1.167</td>
<td>1.147</td>
</tr>
<tr>
<td><strong>p-ring</strong></td>
<td>1.041</td>
<td>1.028</td>
<td>1.130</td>
</tr>
<tr>
<td><strong>p-orec-mixed</strong></td>
<td>0.998</td>
<td>1.003</td>
<td>1.012</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vacation (low)</th>
<th>Memcached</th>
<th>TX_Commit:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Threads</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>p-lock-lazy</strong></td>
<td>1.035</td>
<td>rl.wb()</td>
</tr>
<tr>
<td><strong>p-orec-lazy</strong></td>
<td>1.012</td>
<td>clwb (rl)</td>
</tr>
<tr>
<td><strong>p-norec</strong></td>
<td>1.061</td>
<td>sfence</td>
</tr>
<tr>
<td><strong>p-ring</strong></td>
<td>1.027</td>
<td>clwb(s ← active)</td>
</tr>
<tr>
<td><strong>p-orec-mixed</strong></td>
<td>0.971</td>
<td>sfence</td>
</tr>
</tbody>
</table>

locks.acquire()  
read_set.validate()  
locks.release()  
clwb(wb items)
Optimized GP Vs Optimized IP (Memcached)

Throughput (Kops/s)

Number of Threads

Optimizing Persistent Memory Transactions
Evaluation Summary

- **GP:**
  - Optimized PTM is from 1.1x (B+Tree) to 1.3x (Memcached) faster than naïve PTM, with a geometric mean of 1.22x across all benchmarks
  - Single redo-based algorithm provides best overall performance (1-48 threads)
  - A case for specialization: fence pipelining improves TLRW by 1.7x on Vacation

- **IP:**
  - Optimized PTM is from 1.23x (TPC-C) to 4.06x (B+Tree) faster than GP, with a geometric mean of 1.92x across all benchmarks
  - Single redo-based algorithm provides best overall performance (1-48 threads)

STM vs. Lock
Latency: 2.7x
Peak Speedup: 1.2x

PTM vs. PLock
Latency: 11%
Peak Speedup: ~5x
Future Work

• Study behavior of Optane DC systems
  • Latency from L3 to Optane is much slower than L3 to DRAM
  • Amplifies impact of our optimizations
  • But calls for more work (at many levels of systems stack) to realize potential of PTM
Thank you

LLVM STM and PTM Source Code
http://github.com/pzardoshti/llvm-transmem