SecPM: a Secure and Persistent Memory System for Non-volatile Memory

Pengfei Zuo, Yu Hua
Huazhong University of Science and Technology, China

10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage’18)
Persistence Issue

- The non-volatility of NVM enables data to be persistently stored into NVM
- Data may be incorrectly persisted due to crash inconsistency
  - Modern processors and caches usually reorder memory writes
  - Volatile caches cause partial update
Consistency Guarantee for Persistence

- Durable transaction: a commonly used solution
  - NV-Heaps (ASPLOS’11), Mnemosyne (ASPLOS’11), DCT (ASPLOS’16), DudeTM (ASPLOS’17), NVML (Intel)
  - Enable a group of memory updates to be performed in an atomic manner
- Enforce write ordering
  - Cache line flush and memory barrier instructions
- Avoid partial update
  - Logging

```c
TX_BEGIN
    do some computation;
    // Prepare stage: backing up the data in log
    write undo log;
    flush log;
    memory_barrier();
    // Mutate stage: updating the data in place
    write data;
    flush data;
    memory_barrier();
    // Commit stage: invalidating the log
    log->valid = false;
    flush log->valid;
    memory_barrier();
TX_END
```
Security Issue

- Traditional DRAM: volatile
  - If a DRAM DIMM is removed from a computer
    - Data are quickly lost

- NVM: non-volatile
  - If an NVM DIMM is removed
    - An attacker can directly stream out the data from the DIMM
    - Unsecure
Memory Encryption for Security

- Counter mode encryption
  - Hide the decryption latency
  - Generate One Time Pad (OTP) using a per-line counter
    - Counters are buffered in an on-chip counter cache

(a) Traditional encryption

(b) Counter mode encryption
The Gap between Persistence and Security

- Ensuring both security and persistence
  - Simply combining existing persistence schemes with memory encryption is inefficient
  - Each write in the secure NVM has to persist two data
    - Including the data itself and the counter

- Crash inconsistency
  - Cache line flush instruction cannot operate the counter cache
  - Memory barrier instruction fails to ensure the ordering of counter writes

- Performance degradation
  - Double write requests
## Durable Transaction in Secure NVM

<table>
<thead>
<tr>
<th>Stage</th>
<th>Log content</th>
<th>Log counter</th>
<th>Data content</th>
<th>Data counter</th>
<th>Recoverable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prepare</td>
<td>Wrong</td>
<td>Wrong</td>
<td>Correct</td>
<td>Correct</td>
<td>Yes</td>
</tr>
<tr>
<td>Mutate</td>
<td>Correct</td>
<td>Unknown</td>
<td>Wrong</td>
<td>Wrong</td>
<td>No</td>
</tr>
<tr>
<td>Commit</td>
<td>Correct</td>
<td>Unknown</td>
<td>Correct</td>
<td>Unknown</td>
<td>No</td>
</tr>
</tbody>
</table>

- Selective counter-atomicity (HPCA’18): modifications in software & hardware layers
  - Programming language
    - Add `CounterAtomic` variable and `counter_cache_writeback()` function
  - Compiler
    - Support the new primitives
  - Memory controller
    - Add a counter write queue

```c
TX_BEGIN
    do some computation;
    // Prepare stage: backing up the data in log
    write undo log;
    flush log;
    memory_barrier();
    // Mutate stage: updating the data in place
    write data;
    flush data;
    memory_barrier();
    // Commit stage: invalidating the log
    log->valid = false;
    flush log->valid;
    memory_barrier();
TX_END
```
SecPM: a Secure and Persistent Memory System

- Perform only slight modifications on the memory controller, being transparent for programmers
  - Programs running on an un-encrypted NVM can be directly executed on a secure NVM with SecPM

- Consistency guarantee
  - A counter cache write-through (CWT) scheme

- Performance improvement
  - A locality-aware counter write reduction (CWR) scheme

Asynchronous DRAM refresh (ADR): cache lines reaching the write queue can be considered durable.
Counter Cache Write-through (CWT) Scheme

- CWT ensures the crash consistency of both data and counter
  - Append the counter of the data in the write queue during encrypting the data
  - Ensure the counter is durable before the data flush completes.
Durable Transaction in SecPM

<table>
<thead>
<tr>
<th>Stage</th>
<th>Log content</th>
<th>Log counter</th>
<th>Data content</th>
<th>Data counter</th>
<th>Recoverable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prepare</td>
<td>Wrong</td>
<td>Wrong</td>
<td>Correct</td>
<td>Correct</td>
<td>Yes</td>
</tr>
<tr>
<td>Mutate</td>
<td>Correct</td>
<td>Correct</td>
<td>Wrong</td>
<td>Wrong</td>
<td>Yes</td>
</tr>
<tr>
<td>Commit</td>
<td>Correct</td>
<td>Correct</td>
<td>Correct</td>
<td>Correct</td>
<td>Yes</td>
</tr>
</tbody>
</table>

At least one of log and data is correct in whichever stage a system failure occurs.

The system can be recoverable in a consistent state in SecPM.

```c
TX_BEGIN
  do some computation;
  // Prepare stage: backing up the data in log
  write undo log;
  flush log;
  memory_barrier();
  // Mutate stage: updating the data in place
  write data;
  flush data;
  memory_barrier();
  // Commit stage: invalidating the log
  log->valid = false;
  flush log->valid;
  memory_barrier();
TX_END
```
Counter Write Reduction (CWR) Scheme

- leveraging the spatial locality of counter storage, log and data writes
  - The spatial locality of counter storage
    - The counters of all memory lines in a page are stored in one memory line
    - Each memory line is encrypted by the major counter concatenated with a minor counter

![Diagram showing memory line with major counter and 64 minor counters]
Counter Write Reduction (CWR) Scheme

leveraging the **spatial locality** of counter storage, log and data writes

- **The spatial locality of counter storage**
  - The counters of all memory lines in a page are stored in one memory line
  - Each memory line is encrypted by the major counter concatenated with a minor counter

- **The spatial locality of log and data writes**
  - A log is stored in a contiguous region
  - Programs usually allocate a contiguous memory region for a transaction
Counter Write Reduction (CWR) Scheme

An illustration of the write queue when writing a log
- The counters $A_c$, $B_c$, $C_c$, and $D_c$ are written into the same memory line
- The latter cache lines contain the updated contents of the former ones ($A_c \in B_c \in C_c \in D_c$)
  - They are evicted from the write-through counter cache

The log contents
The counters of log contents

The write queue
(Each cell is a cache line to be written into NVM)
Counter Write Reduction (CWR) Scheme

- When a new cache line arrives, remove the existing cache line with the same physical address in the write queue
  - Without causing any loss of data
Counter Write Reduction (CWR) Scheme

- When a new cache line arrives, remove the existing cache line with the same physical address in the write queue
  - Without causing any loss of data
Counter Write Reduction (CWR) Scheme

- When a new cache line arrives, remove the existing cache line with the same physical address in the write queue
  - Without causing any loss of data

![The Write Queue Diagram]

The Write Queue
Counter Write Reduction (CWR) Scheme

- When a new cache line arrives, **remove** the existing cache line **with the same physical address** in the write queue
  - Without causing any loss of data

![The Write Queue Diagram]
Counter Write Reduction (CWR) Scheme

- When a new cache line arrives, remove the existing cache line with the same physical address in the write queue
  - Without causing any loss of data
Counter Write Reduction (CWR) Scheme

- When a new cache line arrives, **remove** the existing cache line **with the same physical address** in the write queue
  - Without causing any loss of data
Counter Write Reduction (CWR) Scheme

- When a new cache line arrives, **remove** the existing cache line **with the same physical address** in the write queue
  - Without causing any loss of data
  - Using a flag to distinguish whether a cache line is from CPU caches or the counter cache

![The Write Queue](image)

(1: from CPU caches; 0: from the counter cache)
Counter Write Reduction (CWR) Scheme

- When a new cache line arrives, remove the existing cache line with the same physical address in the write queue
  - Without causing any loss of data
  - Using a flag to distinguish whether a cache line is from CPU caches or the counter cache
Performance Evaluation

- Model NVM using gem5 and NVMain

<table>
<thead>
<tr>
<th>CPU and Caches</th>
<th>Memory Using PCM</th>
</tr>
</thead>
<tbody>
<tr>
<td>X86-64 CPU, at 2 GHz</td>
<td>Capacity: 16GB</td>
</tr>
<tr>
<td>32KB L1 data &amp; instruction caches</td>
<td>Read/write latency: 150/450ns</td>
</tr>
<tr>
<td></td>
<td>Encryption/decryption latency: 40ns</td>
</tr>
<tr>
<td>2MB L2 cache</td>
<td>Counter cache: 1MB, 10ns latency</td>
</tr>
<tr>
<td>8MB shared L3 cache</td>
<td></td>
</tr>
</tbody>
</table>

- Storage benchmarks
  - A hash table based key-value store
  - A B-tree based key-value store
**The Number of NVM Write Requests**

**Hash table based KV store**

<table>
<thead>
<tr>
<th>Request Size (B)</th>
<th>Insec-PM</th>
<th>SecPM w/o CWR</th>
<th>SecPM</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>2.5</td>
<td>1.5</td>
<td>1.0</td>
</tr>
<tr>
<td>256</td>
<td>2.0</td>
<td>1.5</td>
<td>1.0</td>
</tr>
<tr>
<td>1K</td>
<td>1.5</td>
<td>1.2</td>
<td>1.0</td>
</tr>
<tr>
<td>4K</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

**B-tree based KV store**

<table>
<thead>
<tr>
<th>Request Size (B)</th>
<th>Insec-PM</th>
<th>SecPM w/o CWR</th>
<th>SecPM</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>2.5</td>
<td>1.5</td>
<td>1.0</td>
</tr>
<tr>
<td>256</td>
<td>2.0</td>
<td>1.5</td>
<td>1.0</td>
</tr>
<tr>
<td>1K</td>
<td>1.5</td>
<td>1.2</td>
<td>1.0</td>
</tr>
<tr>
<td>4K</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Compared with the SecPM w/o CWR, SecPM significantly reduces NVM writes.

Compared with Insec-PM, SecPM only causes 13%, 5%, and 2% more writes when the request size is 256B, 1KB, and 4KB, respectively.
Transaction Throughput

Compared with the SecPM w/o CWR, SecPM significantly increases the throughput by $1.4 \sim 2.1$ times.

Compared with InsecPM, SecPM incurs a little throughput reduction, due to the more NVM writes and the latency overhead of data encryption.
Conclusion

- Both **security and persistence** of NVM are important
- Simply combining existing persistence schemes with memory encryption is inefficient
  - Crash inconsistency
  - Significant performance degradation
- This paper proposes **SecPM** to bridge the gap between security and persistence
  - Guarantee consistency via a counter cache write-through (CWT) scheme
  - Improve performance via a locality-aware counter write reduction (CWR) scheme
Thanks! Q&A