Encrypted Non-volatile Main Memory Systems

Yu Hua

Huazhong University of Science and Technology
https://csyhua.github.io/
Non-volatile Memory (NVM)

- Non-volatile memory is expected to replace or complement DRAM in memory hierarchy
  - ✓ Non-volatility, low power, high density, large capacity
  - × Limited write cell endurance

<table>
<thead>
<tr>
<th></th>
<th>PCM</th>
<th>ReRAM</th>
<th>DRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read (ns)</td>
<td>20-70</td>
<td>10-50</td>
<td>10</td>
</tr>
<tr>
<td>Write (ns)</td>
<td>150-220</td>
<td>30-100</td>
<td>10</td>
</tr>
<tr>
<td>Non-volatility</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Standby Power</td>
<td>~0</td>
<td>~0</td>
<td>High</td>
</tr>
<tr>
<td>Endurance</td>
<td>$10^7$-$10^9$</td>
<td>$10^8$-$10^{12}$</td>
<td>$10^{15}$</td>
</tr>
<tr>
<td>Density (Gb/cm²)</td>
<td>13.5</td>
<td>24.5</td>
<td>9.1</td>
</tr>
</tbody>
</table>

NVM Security

➢ 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
    • Data are still retained in NVM
  – An attacker can directly read the data
    • Unsecure
Two General attacks to NVM

- Attacks:
  - Stolen NVMM
  - Bus snooping
- Memory encryption is important to NVM

- Encrypt data in CPU side, not in-memory.
  - Direct encryption: AES
  - Counter encryption: OTP
Encryption Increases Bit Writes to NVM

- Diffusion property of encryption
  - The change of one bit in original data has to modify half of bits in the encrypted data.

Old data: 00000000...0000000000
↑ 1 of 512 bits modified
Encryption
New data: 10000000...0000000000
      Encryption
                             01011010...0010110100
                                     ↑ 256 of 512 bits modified

Memory encryption causes 50% bit flips on each write
A large number of entire-line duplicates exist, varying from 18% to 98%.

On average 58% duplicates lines v.s. 16% zero lines.
Motivation

- Eliminating duplicate lines via performing deduplication in line level
  - Improve secure NVM endurance
    - Remove duplicate writes
  - Improve system performance
    - Remove the high write latency of duplicate writes
    - Reduce the wait time of read and non-duplicate write requests
Challenges

- **How to perform in-line deduplication** in NVMM without the decrease of system performance
  - Existing memory deduplication is performed out of line
    - Duplicates are first written into memory and then eliminated
    - Fail to reduce writes
  - Existing in-line deduplication incurs high latency
    - Use cryptographic hash functions, e.g., SHA-1 and MD5
    - > 300ns computation latency that is close to NVM write latency

- **How to integrate deduplication with NVM encryption** while delivering good performance
  - Be executed serially in the critical path of memory writes
  - Both produce metadata overheads
DeWrite

- **Light-weight deduplication** leveraging asymmetric NVM reads and writes
  - Eliminate a write at the cost of a read latency
  - Write latency is much higher than read latency (3~8×)

- **Efficient synergization of deduplication and encryption** via parallelism and metadata colocation
  - Opportunistically perform deduplication and encryption in parallel
  - Co-locate their metadata storage for saving space
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
Prediction-based Parallelism

The direct way

A Write Request

Detect Duplication

Is duplicate? Yes -> Cancel the Write
No -> Encrypt Data

Write to NVM

The parallel way

A Write Request

Detect Duplication

Encrypt Data

Is duplicate? Yes -> Write to NVM
No -> Discard the Ciphertext

Be inefficient for applications where most lines are non-duplicate
- Serial execution latency

Be inefficient for applications where most lines are duplicate
- Unnecessary encryption
Prediction-based Parallelism

- How to know whether a cache line is duplicate beforehand?
- **Observation:** the duplication states of most memory writes are the same as those of their previous ones
  - Rationale: The size of duplicate (non-duplicate) data is usually larger than a cache line

![Diagram](image)
Prediction-based Parallelism

- How to know whether a cache line is duplicate beforehand?
- **Observation:** the duplication states of most memory writes are the same as those of their previous ones
  - Rationale: The size of duplicate (non-duplicate) data is usually larger than a cache line

![Diagram showing CPU and Memory states](image)
Prediction-based Parallelism

- How to know whether a cache line is duplicate beforehand?
  - Observation: the duplication states of most memory writes are the same as those of their previous ones
    - Rationale: The size of duplicate (non-duplicate) data is usually larger than a cache line

- Solution: a simple yet effective prediction scheme
  - Exploiting the duplication states of the most recent memory writes
Light-weight Deduplication for NVMM

- Compute the light-weight hash (CRC-32) of a cache line, instead of the cryptographic hash.

<table>
<thead>
<tr>
<th>Hash Functions</th>
<th>SHA-1</th>
<th>MD5</th>
<th>CRC-32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency</td>
<td>321 ns</td>
<td>312 ns</td>
<td>15 ns</td>
</tr>
<tr>
<td>Size</td>
<td>160 bits</td>
<td>128 bits</td>
<td>32 bits</td>
</tr>
</tbody>
</table>

- If the hash matches the value in an existing line, read the line and compare data byte by byte (\(t_Q\): hash query time).

<table>
<thead>
<tr>
<th>Methods</th>
<th>Traditional</th>
<th>DeWrite</th>
</tr>
</thead>
<tbody>
<tr>
<td>A duplicate line</td>
<td>(\geq 312) ns + (t_Q)</td>
<td>(91) ns + (t'_Q)</td>
</tr>
<tr>
<td>A non-duplicate line</td>
<td>(\geq 312) ns + (t_Q)</td>
<td>(15) ns + (t'_Q)</td>
</tr>
</tbody>
</table>
Evaluation

- **Benchmarks**
  - 12 Benchmarks from SPEC CPU2006: single-threaded
  - 8 benchmarks from m PARSEC 2.1: multiple-threaded
NVM Endurance

- DeWrite reduces 54% writes to secure NVM on average
DeWrite speeds up NVM writes by 4.2X on average
DeWrite speeds up NVM reads by 3.1X on average
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
```
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
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</th>
<th>Data</th>
<th>Data</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 of CWR scheme]

- 64B
- M, m1, m2, m3, ..., m64
- Major counter (64 bit)
- 64 minor counters (each 7 bit)
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
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

![Diagram of the Write Queue]

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
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
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 Diagram]

(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

With CWR

Without CWR
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</td>
<td>Read/write latency: 150/450ns</td>
</tr>
<tr>
<td>2MB L2 cache</td>
<td>Encryption/decryption latency:</td>
</tr>
<tr>
<td>8MB shared L3 cache</td>
<td>Counter cache: 1MB, 10ns latency</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

- 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.
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.
- **DeWrite** is a line-level write reduction technique to enhance the endurance & performance:
  - Lightweight deduplication leveraging read/write asymmetry
  - Efficient synergization of deduplication and encryption via parallelism and metadata colocation
- **SecPM** bridges 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