Logging in Persistent Memory: to Cache, or Not to Cache?

Mengjie Li, Matheus Ogleari, Jishen Zhao
These nonvolatile devices are able to retain the data in a consistent state in case of power loss.
Logging in Persistent Memory

Update persistent memory with transactions

\[ Tx_{\text{begin}} \]
do some reads
do some computation

write C

\[ Tx_{\text{commit}} \]

Micro-ops:
store \( C'_1 \)
store \( C'_2 \)
...

Time

Crash

Memory Barrier

\[ \text{Log}_{C'} \]

\[ \text{Log}_{C'} \]

\[ \text{Log}_{C'} \]
To cache, or not cache? That is the question.

[Mengjie Li+, Memsys 2017]
Experimental Setup

- **Desktop** – Dell OptiPlex 7040 Tower
  - CPU – 4-core 3.4GHz Intel Core-i7
  - Cache – 8 MB last-level cache
- **Measurement Tools** – Perf & rdtsc
- **Micro-benchmarks** – run 20 times and report the average performance without initialization time
  - Various working set sizes
  - Various transaction sizes and write intensity
  - Various data structures: hashtable, rbtree, array, …
Microbenchmarks Example

// initialization
Create an array of strings

// Uncacheable log
for (i = 0; i < array_size; ++i) {
    value = random_string;
    key   = i;
    // Log updates
    // Intrinsic functions to invoke movnti
    _mm_stream_si32(&log[2 * i], key);
    _mm_stream_si32(&log[2 * i + 1], value);
    asm volatile ("sfence");
    array[i] = value;
}

// Cacheable log
for (i = 0; i < array_size; ++i) {
    value = random_string;
    key   = i;
    // Log updates
    log[2 * i] = key;
    log[2 * i + 1] = value;
    asm volatile ("sfence");
    array[i] = value;
}
Issue with Cacheable log

Cache pollution

Core

L1i Cache  L1d Cache

...  

Last-Level Cache

Log

Core

L1i Cache  L1d Cache

Log

Log

Memory Bus

DRAM

NVM
LLC Miss Rate and Execution Time

![Graph showing LLC Miss Rate and Execution Time]

- **LLC Miss Rate**
  - Uncacheable
  - Cacheable

- **Execution Time (Million Cycles)**
  - 0.0
  - 0.2
  - 0.4
  - 0.6
  - 0.8
  - 1.0
  - 1.2
  - 1.4
How about uncacheable log performance?
How do we make log uncacheable?

Example:

x86 processors provide uncacheable write instructions (movnti, movntg, etc)

Instructions can be invoked by

• Inline functions (__asm__)  
• Intrinsic functions(_mm_stream_si32)
Write Combining Buffer (WCB)

- Core
  - L1 Cache
  - WCB
  - L1 Cache
  - WCB
  - Log

- Last-Level Cache
- Memory Bus
- DRAM
- NVM
- Log

4-6 cache lines
Issues with Uncacheable Log

• Existing uncacheable writing schemes are sub-optimal
  o Partial writes in WCB
  o Overhead of uncacheable write instructions
  o Limited WCB size
Partial writes are inefficient, because they underutilize the memory bus bandwidth.
Execution Time vs. Transaction Size

— Partial Writes

Partial writes:
String Size – 4B
Iterations – 2097152
Total Data – 8MB

Full writes:
String Size – 64B
Iterations – 131072
Total Data – 8MB

![Bar chart showing execution time for partial and full writes in cacheable and uncacheable cases.]

- **Uncacheable**
  - Partial Writes: 1.28E09 Cycles
  - Full Writes: 1.15E08 Cycles

- **Cacheable**
  - Partial Writes: 1.15E09 Cycles
  - Full Writes: 1.28E08 Cycles
//Uncacheable log
for (i = 0; i < array_size; ++i) {
    value = random_string;
    key = i;

    // Log updates
    // Intrinsic functions to invoke movnti
    _mm_stream_si32(&log[2 * i], key);
    _mm_stream_si32(&log[2 * i + 1], value);
    asm volatile("sfence");

    array[i] = value;
}

//Cacheable log
for (i = 0; i < array_size; ++i) {
    value = random_string;
    key = i;

    // Log updates
    log[2 * i] = key;
    log[2 * i + 1] = value;
    asm volatile("sfence");

    array[i] = value;
}
Overhead of Uncacheable Write Instructions

More overhead to do type casting, if the type of data written is not integer

```c
void __mm_stream_si32(int *p, int a)
asm("movnti %1, %0" : "=m" (*p) : "r"(v)); // int * p, int v;
```
Issues with Limited WCB Size

Log updates among transactions issued by program

WCB

NVDRAM bus
Inefficiencies of Uncacheable Log

<table>
<thead>
<tr>
<th>String size (Bytes)</th>
<th>iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2097152</td>
</tr>
<tr>
<td>8</td>
<td>1048576</td>
</tr>
<tr>
<td>16</td>
<td>524288</td>
</tr>
<tr>
<td>32</td>
<td>262144</td>
</tr>
<tr>
<td>64</td>
<td>131072</td>
</tr>
<tr>
<td>128</td>
<td>65536</td>
</tr>
<tr>
<td>256</td>
<td>32768</td>
</tr>
</tbody>
</table>

Execution Time (Billion cycles)

- uncacheable
- cacheable
- speedup

Partial writes and sfence

WCB size limit

String size (Bytes)
Summary

• Tradeoff between cacheable and uncacheable log
  o Issues with cacheable log – cache contamination
  o Issues with uncacheable log – sub-optimal design in
    • Uncacheable write instructions and programming interface
    • Hardware components, e.g., write-combining buffer design and the way it is used

• More results
  o Sensitivity study on read/write ratio in transactions
  o Sensitivity study on transaction size
  o Other data structures: hash table, rbtree, b+tree, etc.
Logging in Persistent Memory: to Cache, or Not to Cache?

Mengjie Li, Matheus Ogleari, Jishen Zhao