Lazy Persistency: a High-Performing and Write-Efficient Software Persistency Technique

Mohammad Alshboul, James Tuck, and Yan Solihin

Email: maalshbo@ncsu.edu

ARPERS Research Group
Introduction

• Future systems will likely include Non-Volatile Main Memory (NVMM)

• NVMM can host data persistently across crashes and reboots

• Crash consistent data requires persistency models, which define when stores reach NVMM (i.e. become durable)
  – E.g. Intel PMEM: CLFLUSH, CLFLUSHOPT, CLWB, SFENCE
ARPERS Research Group
We refer to this type of persistency models as **Eager Persistency**

- CLFLUSHOPT flushes a cache block to NVMM
- SFENCE orders CLFLUSHOPT with other stores
Our Solution: Lazy Persistency

- Principle: Make the Common Case Fast
- Software technique
- Code is broken into Lazy Persistency (LP) regions
  - Each LP region protected by a checksum
  - Checksum enables persistency failure detection after a crash
  - On recovery, failed regions are re-executed

- Lazily relies on natural cache evictions
  No persist barriers (CLFLUSHOPT, SFENCE) needed
Lazy Persistency Details

- Programs are divided into associative LP regions
- Programmers choose LP region granularity
- A checksum covers updates in an LP region
  - Stored at the end of the LP region

ARPERS Research Group
Lazy Persistency Details

- Programs are divided into associative LP regions
- Programmers choose LP region granularity
- A checksum covers updates in an LP region
  - Stored at the end of the LP region
for (kk=starting_kk; kk<n; kk+=bsize) {
    for (ii = starting_i; ii < n; ii += bsize) {
        ResetCheckSum();
        for (jj =0; jj <n; jj +=bsize) {
            for (i=ii ; i<(ii+bsize); i++) {
                for (j=jj; j<(jj+bsize); j++) {
                    sum = c[i][j];
                    for (k=kk; k<(kk+bsize); k++)
                        sum += a[i][k] * b[k][j];
                    c[i][j] = sum;
                    UpdateCheckSum(c[i][j]);
                } // end of for j
            } // end of for i
        } // end of for jj
        hashIndex = GetHashIndex(ii,kk);
        HashTable[hashIndex] = GetCheckSum();
    } // end of for ii
} // end of for kk
Lazy Persistency Details

```c
for (kk=starting_kk; kk<n; kk+=bsize) {
    for ( ii = starting_ii ; ii < n; ii +=bsize) {
        ResetChecksum();
        for ( jj =0; jj <n; jj +=bsize) {
            for ( i=ii ; i<(ii+bsize); i++) {
                for ( j=jj ; j<(jj+bsize); j++) {
                    sum = c[i][ j ];
                    for (k=kk; k<(kk+bsize); k++)
                        sum += a[i][k]*b[k][ j ];
                    c[i ][ j ] = sum;
                    UpdateChecksum(c[i][j]);
                } // end of for j
            } // end of for i
        } // end of for jj
        hashIndex = GetHashIndex(ii,kk);
        HashTable[hashIndex] = GetChecksum();
    } // end of for ii
} // end of for kk
```
for (kk=starting_kk; kk<n; kk+=bsize) {
    for ( ii = starting ii ; ii < n; ii +=bsize) {
        ResetCheckSum();
        for ( jj =0; jj <n; jj +=bsize) {
            for ( i=ii ; i<(ii+bsize); i++) {
                for ( j=jj ; j<(jj+bsize); j++) {
                    sum = c[i][ j ];
                    for (k=kk; k<(kk+bsize); k++)
                        sum += a[i][k]*b[k][ j ];
                    c[i ][ j ] = sum;
                    UpdateCheckSum(c[i][j]);
                } // end of for j
            } // end of for i
        } // end of for jj
    } // end of for ii
} // end of for kk

hashIndex = GetHashIndex(ii,kk);
HashTable[hashIndex] = GetCheckSum();
Lazy Persistency Details

```c
for (kk=starting_kk; kk<n; kk+=bsize) {
    for (ii = starting_i; ii < n; ii += bsize) {
        ResetChecksum();
        for (jj = 0; jj < n; jj += bsize) {
            for (i = ii; i < (ii + bsize); i++) {
                for (j = jj; j < (jj + bsize); j++) {
                    sum = c[i][j];
                    for (k = kk; k < (kk + bsize); k++)
                        sum += a[i][k]*b[k][j];
                    c[i][j] = sum;
                }
            }
        }
    }
    hashIndex = GetHashindex(ii, kk);
    HashTable[hashIndex] = GetChecksum();
}
```

Initialize at the beginning of the region
Lazy Persistency Details

```plaintext
for (kk=starting_kk; kk<n; kk+=bsize) {
    for (ji = starting_ji; ji<n; ji+=bsize) {
        ResetCheckSum();

        for (jj=0; jj<n; jj+=bsize) {
            for (i=ji; i<(ji+bsize); i++) {
                for (j=jj; j<(jj+bsize); j++) {
                    sum = c[i][j];
                    for (k=kk; k<(kk+bsize); k++)
                        sum += a[i][k]*b[k][j];
                    c[i][j] = sum;
                }
            }
        }
    }

    hashIndex = GetHashIndex(ii,kk);
    HashTable[hashIndex] = GetCheckSum();
}
```
Lazy Persistency Details

```java
for (kk = starting_kk; kk < n; kk += bsize) {
    for (ii = starting_ii; ii < n; ii += bsize) {
        ResetChecksum();
        for (jj = 0; jj < n; jj += bsize) {
            for (i = ii; i < (ii + bsize); i++) {
                for (j = jj; j < (jj + bsize); j++) {
                    sum = c[i][j];
                    for (k = kk; k < (kk + bsize); k++)
                        sum += a[i][k] * b[k][j];
                    c[i][j] = sum;
                } // end of for j
            } // end of for i
        } // end of for jj
    } // end of for ii
}
hashIndex = GetHashIndex(ii, kk);
HashTable[hashIndex] = GetChecksum();
```
CPU

ST A1
ST B1
ST CHK1

ST A2
ST B2
ST CHK2

ST A3
ST B3
ST CHK3

ST A4
ST B4
ST CHK4

INST
INST ← PC

Cache

A1   B1   CHK1

A2   B2   CHK2

A3   B3   CHK3

A4   B4   CHK4

Eviction

NVMM
Eviction

CPU

ST A1
ST B1
ST CHK1

ST A2
ST B2
ST CHK2

ST A3
ST B3
ST CHK3

ST A4
ST B4
ST CHK4

INST

INST  PC

Cache

NVMM

A1  B1

CHK1

A2  B2

CHK2

A3  B3

CHK3

A4  B4

CHK4
CPU

<table>
<thead>
<tr>
<th>ST A1</th>
<th>ST B1</th>
<th>ST CHK1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST A2</td>
<td>ST B2</td>
<td>ST CHK2</td>
</tr>
<tr>
<td>ST A3</td>
<td>ST B3</td>
<td>ST CHK3</td>
</tr>
<tr>
<td>ST A4</td>
<td>ST B4</td>
<td>ST CHK4</td>
</tr>
</tbody>
</table>

Cache

Eviction

NVMM

A1 | B1 | CHK1
A2 | B2 | CHK2
A3 | B3 | CHK3
A4 | B4 | CHK4
### CPU

<table>
<thead>
<tr>
<th>ST A1</th>
<th>ST B1</th>
<th>ST CHK1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ST A2</th>
<th>ST B2</th>
<th>ST CHK2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ST A3</th>
<th>ST B3</th>
<th>ST CHK3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ST A4</th>
<th>ST B4</th>
<th>ST CHK4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**INST**

**PC**

---

### Cache

---

### NVMM

<table>
<thead>
<tr>
<th>A1</th>
<th>B1</th>
<th>CHK1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A2</th>
<th>B2</th>
<th>CHK2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A3</th>
<th>B3</th>
<th>CHK3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A4</th>
<th>B4</th>
<th>CHK4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Eviction**
<table>
<thead>
<tr>
<th>CPU</th>
<th>Cache</th>
<th>NVMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST A1</td>
<td></td>
<td>A1</td>
</tr>
<tr>
<td>ST B1</td>
<td></td>
<td>B1</td>
</tr>
<tr>
<td>ST CHK1</td>
<td></td>
<td>CHK1</td>
</tr>
<tr>
<td>ST A2</td>
<td></td>
<td>A2</td>
</tr>
<tr>
<td>ST B2</td>
<td>B2</td>
<td>CHK2</td>
</tr>
<tr>
<td>ST CHK2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST A3</td>
<td></td>
<td>A3</td>
</tr>
<tr>
<td>ST B3</td>
<td></td>
<td>B3</td>
</tr>
<tr>
<td>ST CHK3</td>
<td>CHK3</td>
<td></td>
</tr>
<tr>
<td>ST A4</td>
<td>A4</td>
<td></td>
</tr>
<tr>
<td>ST B4</td>
<td>B4</td>
<td></td>
</tr>
<tr>
<td>ST CHK4</td>
<td></td>
<td>CHK4</td>
</tr>
<tr>
<td>INST</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INST</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CPU

ST A1
ST B1
ST CHK1

ST A2
ST B2
ST CHK2

ST A3
ST B3
ST CHK3

ST A4
ST B4
ST CHK4

INST
INST ➔ PC

Cache

Eviction

NVMM

A1 B1
CHK1

A2 B2
CHK2

A3 B3
CHK3

A4 B4
CHK4
Recovering From a Crash

- On a crash, checksums are validated to detect regions that were not persisted
- Failed regions are recomputed
- Finally, program resumes execution in normal mode
Diagram showing the process of data handling between different memory types and storage:

- **P** to **NVMM**
- **NVMM** to **Disk**
- **STORE**
- **CLFLUSHOPT**
- **ACK**
- **SFENCE**

The diagram illustrates the flow of data from the processor (P) to non-volatile memory (NVMM), then to disk, with delays indicated for NVMM and disk operations.
P → cache → NVMM

Create checksum

Disk Delay

NVMM Delay

Disk
Disk Delay

Disk

NVMM Delay

Create checksum

STORE

Eviction

P

cache

NVMM
Limitations of Lazy Persistency

• LP regions need to be associative, i.e. \((R_1, R_2), R_3 = R_1, (R_2, R_3)\)
  – Most HPC kernels contain loop iterations that satisfy this requirement
  – Can be relaxed in some situations (see the paper)

• Recovery code needed for LP regions
  – Solution: Prior work can be exploited [PACT'17]

• Amount of recovery may be unbounded (e.g. due to hot blocks)
  – Solution: Periodic Flushes (Next Slide)
Bounding the Amount of Recovery

• Cache blocks may stay in the cache for a long time (e.g. hot blocks)
  – Getting worse the larger the cache

• Regions with such blocks may fail to persist

• Upper-bound is needed for the time a block might remain dirty in the cache

• This is needed to guarantee forward progress
Solution: Periodic Flushes

- A simple hardware support
- All dirty blocks in the cache are written back periodically, in the background
- Modest increase in the number of writes (see paper for details)
- The periodic flush interval puts an upper bound for recovery work
Evaluation

Methodology

• Simulations on a modified version of gem5. Supports most Intel PMEM instructions (e.g. CLFLUSHOPT)

• Detailed out-of-order CPU model. Ruby memory system. 8 threads is the default for all experiments

• Evaluation was also done on 32-core DRAM-based real hardware machine
Evaluation

Multi-Threaded Benchmarks

- Tiled Matrix Multiplication
- Cholesky Factorization
- 2D convolution
- Fast Fourier Transform
- Gauss Elimination
Evaluation: All Benchmarks

(a) Execution Time Overhead

(b) Number of Writes Overhead
Evaluation: All Benchmarks

(a) Execution Time Overhead

(b) Number of Writes Overhead

- Eager Persistency
- Lazy Persistency

9% vs 1%

21% vs 3%
More Evaluations

We performed other interesting evaluations that can be found in the paper:

- Sensitivity study with varying the read/write latency for NVMM
- Sensitivity study with varying the number of threads
- Evaluating the execution time for all the 5 benchmark on real hardware
- Sensitivity study with varying the Last Level Cache size
- Analysis for the Number of Writes of Periodic Flushes hardware support
- Evaluating the execution time overhead when trying different error detection mechanisms
Summary

• Lazy Persistency is a software persistency technique that relies on natural cache evictions (No stalls on SFENCE)

• It reduces the execution time and write amplification overheads, from 9% and 21%, to only 1% and 3%, respectively.

• A simple hardware support can provide an upper-bound on the recovery work
Questions?