ResPCT: Fast Checkpointing in Non-volatile Memory for Multi-threaded Applications

Ana Khorguani, Thomas Ropars, Noel De Palma
April 15, 2022

Univ. Grenoble Alpes, CNRS, Grenoble INP
Non-volatile Memory

NVMM is a great opportunity to build fast fault-tolerant programs
Memory architecture

- Volatile
  - Cache
  - DRAM
- Persistent
  - Memory controller
  - NVMM
Challenges with volatile cache

Writes issued by the CPU go through:

- Cache
- Memory controller
- NVMM

Updated cache lines persist:

- At cache line evictions
- At the calls of flush (\texttt{clwb}) and memory fence (\texttt{sfence}) instructions
Our Contribution

A checkpointing-based technique called ResPCT:

- An API enabling programmers to build fault-tolerant programs
- Support for lock-based multi-threaded applications
- Generalized usage of In-Cache-Line Logging
  - Originally proposed by Cohen et al. [ASPLOS 2019]
  - Solves the challenges caused by the volatile cache
- Restart points
  - Identify where a checkpoint may be taken
  - Identify the persistent state

Evaluation on real hardware with micro-benchmarks and real applications:

- Overhead as low as 9%
- Up to $2.7\times$ faster than state-of-the-art systems
Efficient consistent checkpoints
Consistency issue

A persistent Linked list:
Consistency issue

Adding a new element:
Consistency issue

Expected persistent state:
Partial updates may reach NVMM in any order:
Consistency issue

What happens in the event of a failure?

The persistent data structure stays in an inconsistent state
Existing approaches

1. **Transaction-based techniques:**
   - Clobber-NVM [ASPLOS 2021]
     - Logs only the variables that belong both to the read set and the write set of a transaction
   - Quadra/Trinity [PPoPP 2021]
     - Undo logging with In-Cache-Line Logging
   
   **Require frequent calls to flush and fence instructions**

2. **Checkpoint-based techniques:**
   - PMThreads [PLDI 2020]
     - Maintains a shadow copy of the persistent data structures in DRAM
     - **Costly modification tracking**
   - Montage [ICPP 2021]
     - Uses Copy-on-Write: any update in NVMM allocates a new object
     - Frequent memory allocation
Checkpoints

Checkpoints with a single copy of the data:

- The program execution is divided into epochs
- At the end of the epoch, modified data are persisted on NVMM
On cache line evictions, updates may reach NVMM at any time in the epoch.
In the event of a failure:
In the event of a failure:
Challenges:

- **Tracking** - identifying modified memory locations to flush them at checkpoint time
- **Consistency** - rolling back any partial updates that reached NVMM during an epoch that crashed
ResPCT in a nutshell

ResPCT is a checkpointing technique based on:

- **In-Cache-Line Logging**
  - Maintains an undo log
  - Efficiently handles the consistency and tracking challenges

- **Restart Points (RPs)**
  - The programmer places RPs in the source code
  - Defines where a checkpoint can occur
  - Dictates which data belong to the persistent state, and which data require logging
  - Can improve performance
In-Cache-Line Logging (InCLL)

In Intel-x86 architectures:

- Writes to the same cache line reach NVMM in the program order

InCLL:

- The undo log of a variable is stored in the same cache line as the variable itself
- Efficient undo logging without using any flush or fence instruction
For every persistent variable, InCLL saves **in one cache line**:

- The value of the variable
- An undo log – *backup*
- The corresponding epoch identifier – *epoch_id*
In every epoch, when modifying the variable for the first time:

1. **backup** logs the value of the variable
In every epoch, when modifying the variable for the first time:

1. **backup** logs the value of variable
2. **epoch_id** stores the corresponding epoch identifier
In every epoch, when modifying the variable for the first time:

1. backup logs the value of the variable
2. epoch_id stores the corresponding epoch identifier
3. The variable is updated
Track modified addresses

In every epoch, when modifying the variable for the first time:

1. `backup` logs the value of the variable
2. `epoch_id` stores the corresponding epoch identifier
3. The variable is updated
4. The address is added in the list of updated memory locations
   - An unique list of modified addresses
   - Without iterating through the list at insertion

```
variable = new value  backup = variable  epoch_id = current epoch
```

![Cache](image1)
![NVMM](image2)
Persistent undo log

In every epoch, when modifying the variable for the first time:

1. **backup** logs the value of the variable
2. **epoch_id** stores the corresponding epoch identifier
3. The variable is updated
4. The address is added in the list of updated memory locations
   - An unique list of modified addresses
   - Without iterating through the list at insertion
Restart Points (RPs)
Restart Points (RPs)

In existing approaches, checkpoints occur at the end of *transactions* or *critical sections*:

- No flexibility
- No control over the persistent state

**Why use Restart points?**

- Allows reducing the size of the persistent state
  - Can improve performance
- Flexible without changing the application logic
Semantics

RPs:
  • Define where the checkpoints can occur in the program code
  • Define the persistent state

Properties of RPs:
  • A checkpoint occurs only when all threads have reached an RP
  • Threads are blocked in $RP()$ as long as the checkpoint is running

Placement constraints of RPs:
  • $RP()$ must not be called inside a critical section

These guarantee that the checkpointed state is consistent
The variable must be persistent:

- If the scope of the variable includes an RP

The persistent variable requires InCLL:

- If it makes a sub-part of the program starting from an RP not idempotent.

<table>
<thead>
<tr>
<th>Sequence</th>
<th>x = 0</th>
<th>x = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 5</td>
<td></td>
<td>y = x</td>
</tr>
<tr>
<td>y = x</td>
<td></td>
<td>x = 8</td>
</tr>
</tbody>
</table>

| Idempotent? | Yes   | No    |
Impact of RPs on the persistent state

The variable `i` must be Persistent

The variable `i` can be Volatile
Evaluation
Experimental setup and Workloads

Hardware and software setup:

- Two Intel Xeon Gold 5218 CPUs (64 logical cores)
- 384 GiB of DRAM and 1.5 TiB of Intel’s Optane PMem
- Prototype of ResPCT in C
- Checkpoint frequency - 64 msec

Evaluated workloads:

- Highly efficient concurrent HashMap and Queue
  - Impact of the checkpoint period
  - Recovery time
- Benchmarks from multi-threaded benchmark suites: Parsec and Phoenix
- Memcached - a popular in-memory key-value store
Performance with the HashMap

HashMap (2M keys)

- The overhead of ResPCT is below 9%
- Up to 2.7× better than state-of-the-art techniques
• Highest overhead of ResPCT: 37%
• ResPCT outperforms all systems except PMThreads
Detailed analysis of ResPCT

- **ResPCT-InCLL** - ResPCT with InCLL, modification tracking, but no checkpoint
- **ResPCT-noFlush** - ResPCT with the complete algorithm except flushing the modified data on NVMM
Impact of the checkpoint period

HashMap (update=90% search=10%)

- Almost no performance overhead with 16-ms epochs
- Still better than any other systems with 1-ms epochs
Performance with compute-intensive applications

- Dedup – heavily lock-based
- Swaptions – lock-less
- LR – machine-learning oriented
- MatMul – compute intensive

Overhead between 17% and 21%
Different placement of RPs in LR could cause up to $9 \times$ slowdown
Applying ResPCT required modifying 2.5% to 7.2% of the lines of code
We presented ResPCT:

- A highly efficient in-NVMM checkpointing technique
- Efficient usage of InCLL to solve the consistency and the tracking challenges
- Flexible Restart Points to reduce the persistent state size and improve performance
- The proof of correctness and the detailed description of the algorithm are presented in the paper

The evaluation of our algorithm on real hardware showed:

- ResPCT overhead can be as low as 9%
- ResPCT can significantly outperform state-of-the-art techniques
Questions?
<table>
<thead>
<tr>
<th>ResPCT API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InCLL_data&lt;T&gt;</td>
<td>Template for InCLL variables</td>
</tr>
<tr>
<td><em>init_InCLL</em></td>
<td>Initializes InCLL for variable ( l )</td>
</tr>
<tr>
<td><em>update_InCLL</em></td>
<td>Updates variable ( l ) and its log</td>
</tr>
<tr>
<td><em>add_modified</em></td>
<td>Registers the address of a modified variable</td>
</tr>
<tr>
<td><em>RP</em></td>
<td>Identifies a Restart Point</td>
</tr>
<tr>
<td><em>checkpoint_allow/prevent</em></td>
<td>Allows/prevents checkpoint occurrence</td>
</tr>
</tbody>
</table>
Recovery time

- Recovery using 32 threads
- With a 4M-buckets HashMap, recovery takes less than 240 ms
Modifying the original code to apply ResPCT

<table>
<thead>
<tr>
<th></th>
<th>Added or modified LoC</th>
<th>Original LoC</th>
<th>Modifications in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hashmap</td>
<td>19</td>
<td>282</td>
<td>6.74%</td>
</tr>
<tr>
<td>Queue</td>
<td>31</td>
<td>607</td>
<td>5.11%</td>
</tr>
<tr>
<td>Dedup</td>
<td>244</td>
<td>3351</td>
<td>7.28%</td>
</tr>
<tr>
<td>Swaptions</td>
<td>29</td>
<td>1137</td>
<td>2.55%</td>
</tr>
<tr>
<td>MatMul</td>
<td>13</td>
<td>196</td>
<td>6.63%</td>
</tr>
<tr>
<td>LR</td>
<td>69</td>
<td>138</td>
<td>50.00%</td>
</tr>
<tr>
<td>Memcached</td>
<td>97</td>
<td>20520</td>
<td>0.47%</td>
</tr>
</tbody>
</table>

• The modifications represent between 2.5 and 7.2% of the LoC