Designing fault-tolerant multi-threaded applications for Non-Volatile Memory using checkpoints

Ana Khorguani

Thomas Ropars, Noel De Palma
LIG - team ERODS

May 27, 2021

What is Non-Volatile Memory?

Historically, memory was divided into two parts:

- Storage (SSD, HDD)
- Volatile memory (DRAM)

Non-Volatile Memory offers best from the both worlds:

- Support for data persistence
- Byte-addressability (accessed by CPU's load and store instructions)
- Performance on par with DRAM
  - Faster than any storage device with orders of magnitude
  - Around 6 times slower than DRAM

Non-Volatile Memory Modules

NVM memory modules became available from 2019

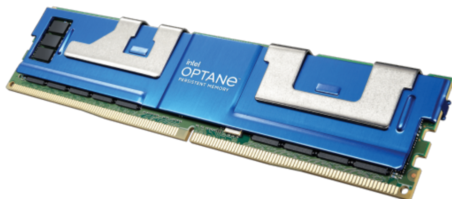
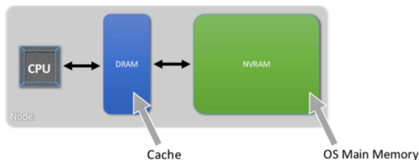


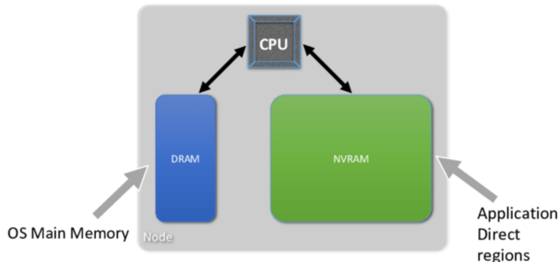
Figure: Intel Optane Persistent Memory

NVM configurations

1. NVM represents the main memory, while DRAM is seen as another layer of the cache



2. DRAM and NVM, both are parts of the main memory

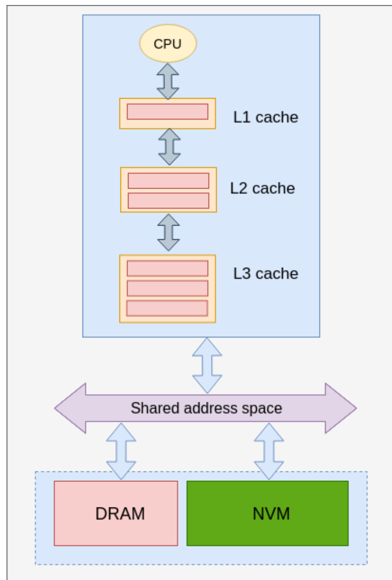


Motivation

The motivation of our work is:

- Using NVM for fault tolerance
  - Handle consistency of the data structures
- Maintaining high performance
  - The frequency of persisting data on NVM will affect performance

Non-Volatile Memory



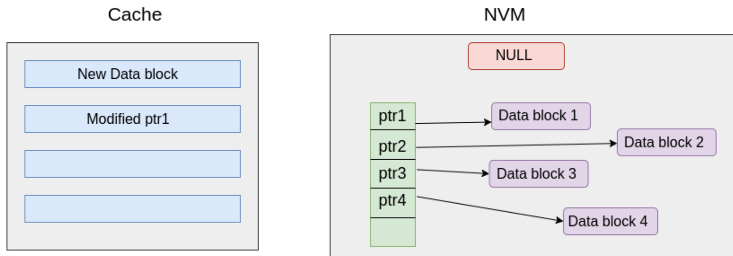
New memory hierarchy:

- Volatile
  - Caches
  - DRAM
- Persistent
  - NVM

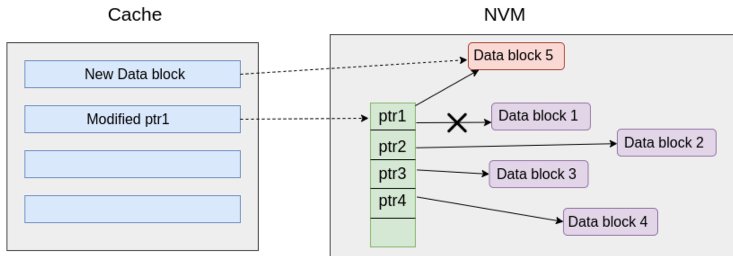
On cache line evictions, updated cache lines are possibly written back out of order to NVM

Consistency issues

Now we care about the data movement from cache to NVM:

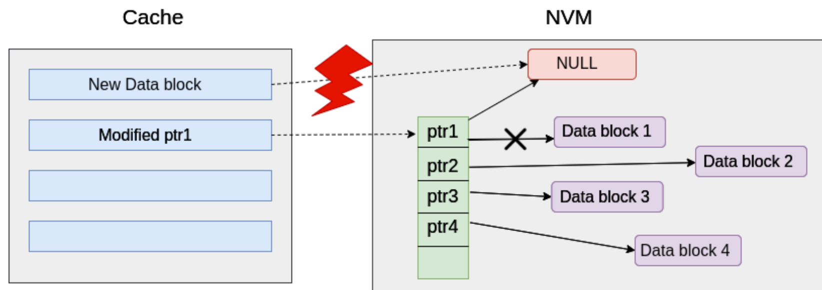


Expected final state of the data structure in NVM:

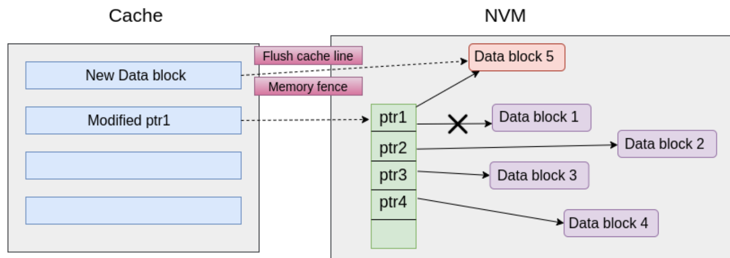


Consistency issues

What if the system crashes after modifying the pointer without updating the data block?



Intel processor flush instructions



Intel explicit flush instruction:

- `Clwb` (@address)

Introduces high overhead due to:

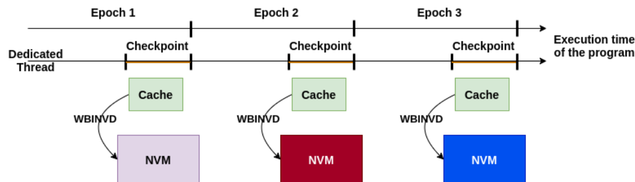
- Invalidating the cache lines
- Putting restrictions on out-of-order executions

Our approach

We propose an algorithm that allows the data structures that are stored in NVM to be restored in a consistent state after a failure, while achieving good performance

Our technique is based on:

- Checkpoints
  - The program execution is divided into epochs
  - At the end of the epoch, the data is persisted by flushing the entire content of the cache



Our approach

Our checkpoint technique is completed with:

- In-Cache-Line Log
  - An efficient undo log, without need of flush instructions
  - The log entries are in the same cache lines as the data fields
  - Relies on Persistent Cache Store Order (PCSO) memory ordering model
- Checkpoint atomic sections (CASEs)
  - Guarantee that the epochs end only when the data is consistent
  - Guarantee that critical sections are executed with respect of the checkpoints

Experimental setup and Workloads

Hardware and software setup:

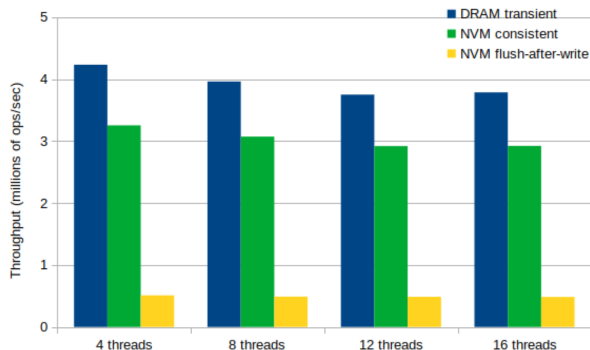
- Two Intel Xeon Gold 5218 CPUs (64 logical cores)
- 384 GiB of DRAM and 1.5 TiB of Intel's Optane DC Persistent Memory
- Checkpoint frequency - 64 msec

We evaluate 2 benchmarks:

- A highly efficient concurrent data structure: MS-Queue
- A data-parallel computation: PARSEC Swaptions app

Performance with MS-Queue

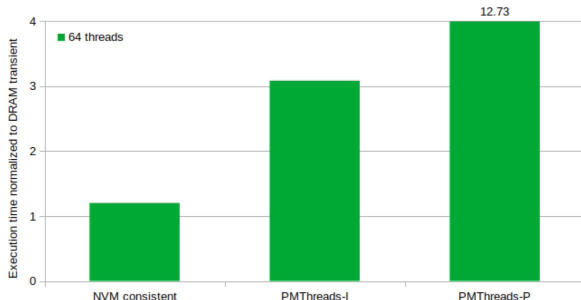
Comparison with solutions based on flush-after-write



- Highest overhead for our solution: 30%
- Up to 6× better than flush-after-write

Performance with Swaptions app

Comparison with PMThreads runtime library (to our knowledge, the best existing checkpoint-based technique)



- The overhead for our solution: 20%
- Up to 2.5× better than the best version of PMThreads

Conclusion

We presented Non-Volatile Memory and described the challenges of designing the fault-tolerant algorithms:

- Consistency issues
- Impact on performance

By evaluating our checkpointing algorithm on real hardware, we illustrated:

- The overhead of our solution is as low as 20-30%
- Our solution outperforms state-of-the-art techniques

Future Work:

- Optimizing persisting data on NVM
- Automatizing the modification of the code

Thank you for your attention