A Persistent Hash Map for Graph Processing Workloads and a Methodology for Persistent Transactional Data Structures

CHRISTINA PETERSON,
KENNETH LAMAR

Overview

Introduction

Introduction

Persistent Memory

- Persistent Memory is positioned as a new tier in the memory hierarchy that delivers capacity of non-volatile storage at speeds close to DRAM
- Benefits:
  - Non-volatile storage
  - Byte addressable
  - Provides higher density than DRAM
  - Has access latencies closer to DRAM than storage
- Persistent memory is commercially available through Intel® Optane™ DC Persistent Memory

Traditional Memory Hierarchy

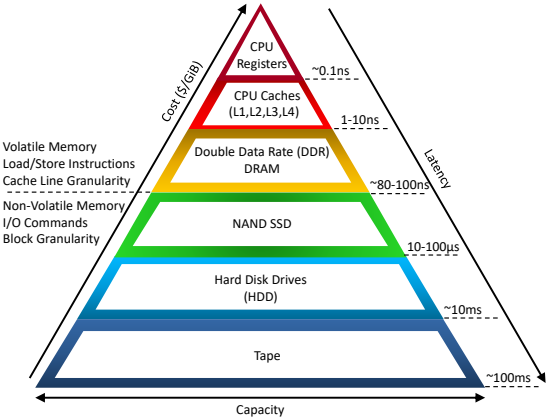


Figure 1: Traditional Memory Hierarchy [1]

New Memory Hierarchy

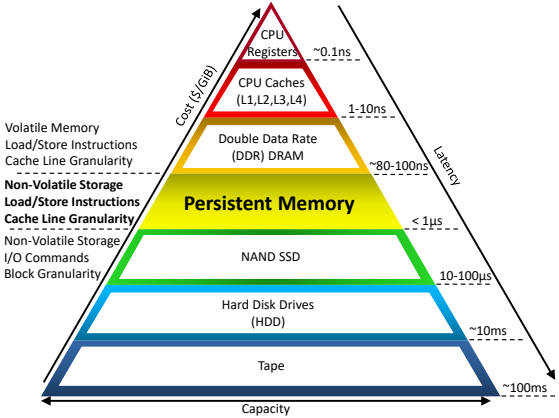


Figure 2: New Memory Hierarchy [1]

Statistics

Feature	DRAM	PMEM	NAND SSD
Volatility	Volatile	Non-Volatile	Non-Volatile
Capacity	16GB-64GB [2]	128GB, 256GB, 512GB [2]	256GB-1TB [3]
Latency	0.1 x 1μ [4]	0.1μ - 1μ [4]	1000 x 0.1μ [4]
Endurance	64E7 PBW [5]	360 Petabytes Written (PBW) [6]	0.3 PBW [7]
Cost*	$5.04/GB (64GB) [8], $7.5/GB (128GB) [9]	$16.30/GB (512GB) [10]	$0.11/GB (1TB) [11]

- *The price per gigabyte increases with increasing density [12]
  - It is difficult to pack a lot of DRAM into a single module [12]

Use-Cases of Persistent Memory

Metagenomics

- Persistent hash table to lookup genome fragments [13]

Astronomy

- Persistent indexing structure to maintain data sets generated by an optical telescope [13]

Graph Analytics

- Maintain graph structure in persistent memory for scalable checkpointing [13]

Use-Cases of Persistent Memory

Common Characteristics of Use-Cases

- Data sets are large, up to trillions of data points [13]
  - High capacity
- Data sets may encounter a large number of updates
  - High endurance
- Data set updates are computationally expensive
  - Non-volatile, low latency

Desirable Properties

- High capacity
- High endurance
- Non-volatile, low latency

Persistent memory provides a happy medium!

Feature	DRAM	PMEM	NAND SSD
Volatility	Volatile	Non-Volatile	Non-Volatile
Capacity	16GB-64GB [2]	128GB, 256GB, 512GB [2]	256GB-1TB [3]
Latency	0.1 x 1μ [4]	0.1μ - 1μ [4]	1000 x 0.1μ [4]
Endurance	64E7 PBW [5]	360 Petabytes Written (PBW) [6]	0.3 PBW [7]
Cost	$5.04/GB (64GB) [8], $7.5/GB (128GB) [9]	$16.30/GB (512GB) [10]	$0.11/GB (1TB) [11]

Pitfalls of Persistent Memory

Architecture Limitations

- Caches and registers are expected to remain volatile
  - Can cause persisted data to be in an inconsistent state if stores prior to the crash were in the cache but not yet written to persistent memory
  - Architecture provides instructions to ensure durability and ordering. Example:
    - `clwb`: x86 ISA cacheline writeback
    - `sfence`: x86 ISA fence

Example

```
1  if(CAS(curr->next, next, node)){
2    clwb(curr->next);
3    sfence();
4  }
```

Figure 3: Persist Ordering Problem

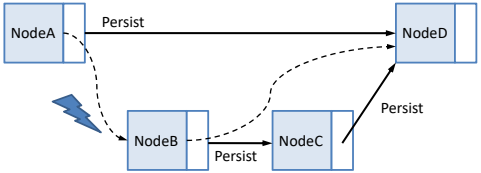


Figure 4: Crash Consistency Violation

Compare-And-Swap (CAS) accepts a memory location, expected value, and new value as parameters. If the dereferenced value of the memory location is equivalent to the expected value, then the memory location value is updated to the new value and true is returned. Otherwise, no change is made and false is returned.

Correctness Properties for Persistent Data Structures

- Crash Consistency
- Durable Linearizability

Persistent Hash Map

Persistent Hash Map

Setting

- Graph analytics
  - Billions of vertices
- Concurrent data structures
  - High performance access to data in shared memory
- **Hash maps**
  - Fundamental data structure
  - Commonly used in graph analytics
  - Few high-performance NVM options

Persistent Hash Map

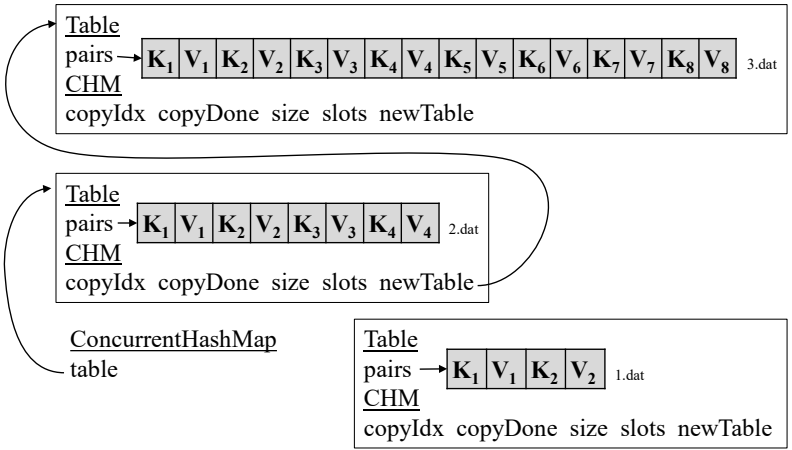
Design Goals

- Read optimized
  - Persistence needs no flush or fence after first read
- Runtime over recovery
  - Persist as little as possible
- Compact representation and few cache misses
  - Arrays
  - Open addressing
- Low memory management overhead
  - Allocate large table chunks

Persistent concurrent hash Map (PMap)

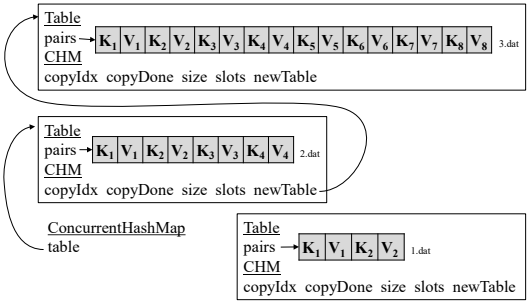
- Non-volatile
- Lock-free
  - Guaranteed system-wide progress
  - Scales up with multiple threads
- Open addressing
  - In-place keys and values
- Resizable
  - Shrink or expand
- Operations
  - `insert()`, `replace()`, `remove()`, `get()`, and `update()`
  - `update()` is an atomic conditional replace

PMap Design Overview



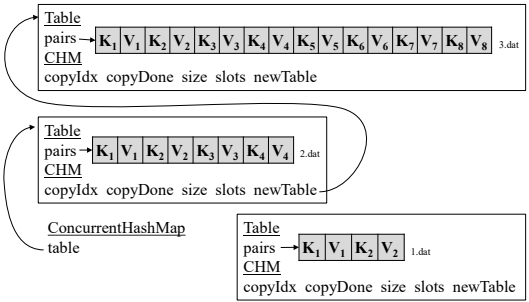
Resizing

- Adapted from Cliff Click's hash map [14]
- Lock-free resizing is challenging
  - Keys and values are separate atomics
    - Partial operations are possible
- Allocate a table twice (or half) the current table size
- Key-value pairs are individually migrated
  - Concurrent
  - Parallel
  - Incremental



Resizing

- Reserves a resize bit for each value
  - Indicates migration in progress
    - Threads must help migrate before returning a value
  - Resize bit cuts into usable bits
    - Limits values to 63 bits
- Once migrated, the old slot is replaced with a migration sentinel
  - Slot cannot be reused
  - Migration is complete when all slots have migration sentinels



Persistence

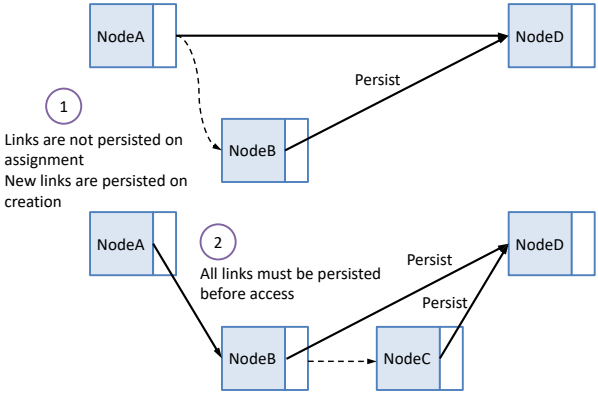
- Goal: Add persistence to concurrent data structures
  - Leverage existing multithreaded synchronization guarantees

Persistence

- Goal: Add persistence to concurrent data structures
  - Leverage existing multithreaded synchronization guarantees
- Naive idea: flush-on-read [15]
  - Flush newly created objects (ex. node and pointer)
  - Flush before each read
  - Simple, but expensive

Flush-on-Read

```
void persist(atomic<uintptr_t> *address) {
    FLUSH(address);
    FENCE;
    return
}
uintptr_t pread(atomic<uintptr_t> *address) {
    persist(address);
    return address->load();
}
bool pcas(atomic<uintptr_t> *address,
uintptr_t &oldVal, uintptr_t newVal) {
    persist(address);
    return address->CAS(oldVal, newVal);
}
```

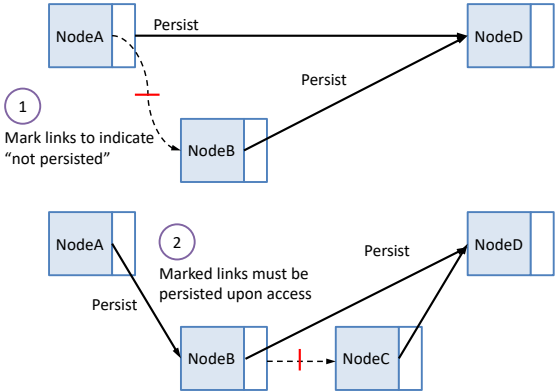


Persistence

- Goal: Add persistence to concurrent data structures
  - Leverage existing multithreaded synchronization guarantees
- Naive idea: flush-on-read [15]
  - Flush newly created objects (ex. node and pointer)
  - Flush before each read
  - Simple, but expensive
- Better idea: link-and-persist [15], [16]
  - Flush newly created objects (just as before)
  - Borrow an unused bit from a pointer
    - Most architectures leave unused bits
  - Mark as dirty on write
  - Flush only on first read, then mark clean
    - Worst case: All threads read at once, see the dirty bit, and all persist (factor of thread count)

Link-and-Persist

```
uintptr_t persist(atomic<uintptr_t> *address,
uintptr_t value) {
    FLUSH(address);
    FENCE;
    address->CAS(value, value & ~DirtyFlag));
    return value;
}
uintptr_t pRead(atomic<uintptr_t> *address) {
    uintptr_t value = address->load();
    if ((value & DirtyFlag) != 0) {
        persist(address, value);
    }
    return value & ~DirtyFlag;
}
bool pCAS(atomic<uintptr_t> *address,
uintptr_t &oldVal, uintptr_t newVal) {
    uintptr_t value = address->load();
    if ((value & DirtyFlag) != 0) {
        persist(address, value);
    }
    return address->CAS(oldVal, newVal | DirtyFlag)
}
```

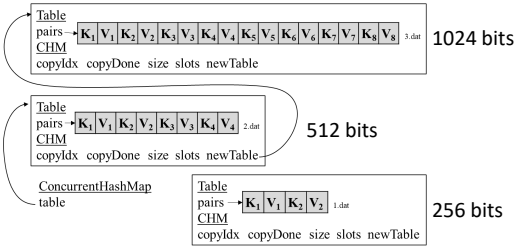


Link-and-Persist in PMap

- Flush new objects: Flush levels to hold default (empty) values.
- Extend from pointers to data.
  - Cost: 1 bit per value
  - Effective limit: 62-bit values (resize bit)
  - Reasonable trade-off for our use cases

Recovery

- Only persist keys and values
- Filesystem data (name, size) infers contents
  - DAX (Direct Access) mode
- Link-and-persist ensures data is consistent
  - Orphans are possible but discarded during recovery



Related Works Compared

- Concurrent level hashing (clevel)
  - Lock-free
  - Open addressing (of pointers)
  - Resize (but only expansion)
- OneFile hash map (OneFile)
  - Wait-free
  - Transactional
  - Node-based
- Standard Template Library hash map (STL)
  - Volatile
  - std::map with global lock
- Persistent Memory Development Kit concurrent_hash_map (PMDK)
  - Based on Intel TBB
  - Reader-writer locks

Testing Environment

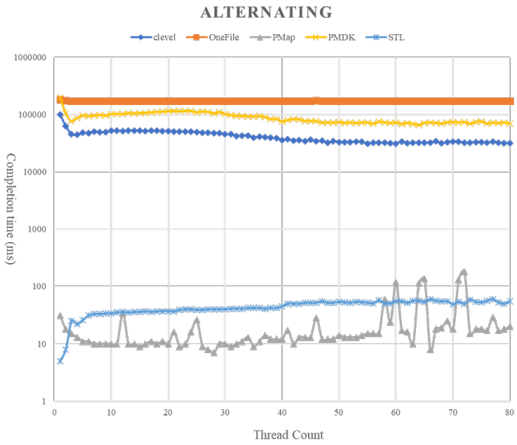
- System:
  - 2x 20 core / 40 thread Intel Xeon Gold 6230
  - 134GB DRAM, 248GB Optane DC

  Optane Configuration:
  - App Direct mode
  - DAX (Direct Access) mode

  Code Configuration:
  - C++
  - GCC 9

  Test Configuration:
  - 62-bit keys and values
  - Table capacity initially 2^{14}
  - No garbage collection

Performance Comparisons



Persistent Transactional Data Structures

Persistent Transactional Systems

Taxonomy

	Transactional	Non-Transactional
High-Level (Data Structure Semantics)		Persistent Data Structures
Low-Level (Reads/writes)	Persistent Transactional Memory (PTM)	`clwb`, `sfence`

Persistent Transactional Data System for Linked Data Structures (PETRA)

Design Goals

- High Performance
  - Low overheads added to achieve durability
- High Scalability
  - Performance scaling well with increasing number of processes
- Non-Blocking
  - There is guaranteed system-wide progress

Persistent Transactional Data System for Linked Data Structures (PETRA)

High Performance

- PETRA keeps the number of cache line flushes and memory fences low by persisting only the *transaction descriptor*:
  - Contains the information needed to execute a transaction
  - Leveraged as redo logs verify consistency after a crash and correct possible inconsistencies
  - Optimized for failure-free execution

Persistent Transactional Data System for Linked Data Structures (PETRA)

High Scalability

- Transactional synchronization for conflicts on nodes
  - Logical rollback when a semantic conflict is detected
- Thread-level synchronization for read/write conflicts
  - Eliminates false aborts

Persistent Transactional Data System for Linked Data Structures (PETRA)

Non-Blocking

- PETRA enforces transactional synchronization using a helping scheme in conjunction with CAS
  - A transaction will help complete another transaction if a conflict is detected
  - The transaction status is updated from Active to either Committed or Aborted using CAS
- PETRA achieves *Obstruction Freedom*:
  - A single process executed in isolation is guaranteed to make progress

PETRA Methodology Example

Algorithm 1 Type Definitions

```
 1: enum TxStatus
 2:     Active
 3:     Committed
 4:     Aborted
 5: enum PersStatus
 6:     Maybe \\Default value
 7:     InProgress
 8:     Persisted
 9: enum OpType
10:     Insert
11:     Delete
12:     Find
13: struct Operation
14:     OpType type
15:     int key
16: struct Desc
17:     int size
18:     int txid
19:     TxStatus status
20:     PersStatus pstatus
21:     Operation ops[ ]
22: struct NodeInfo
23:     Desc* desc
24:     int opid
25: struct Node
26:     NodeInfo* info
27:     int key
28:     ...
```

PETRA Methodology Overview

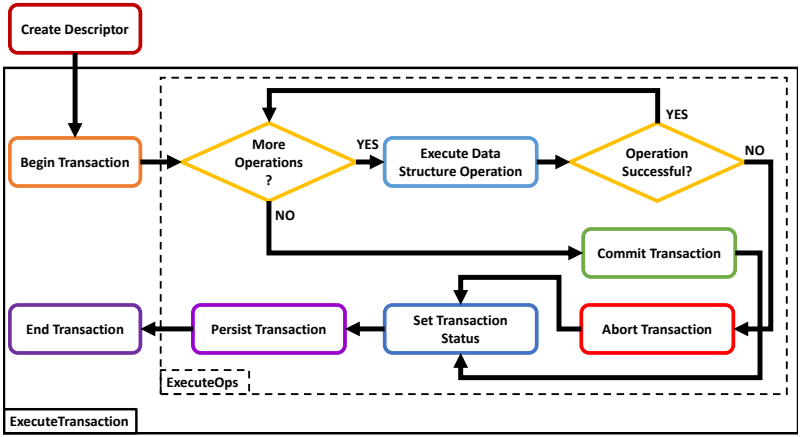


Figure 5: Execute Transaction

PETRA Methodology Overview

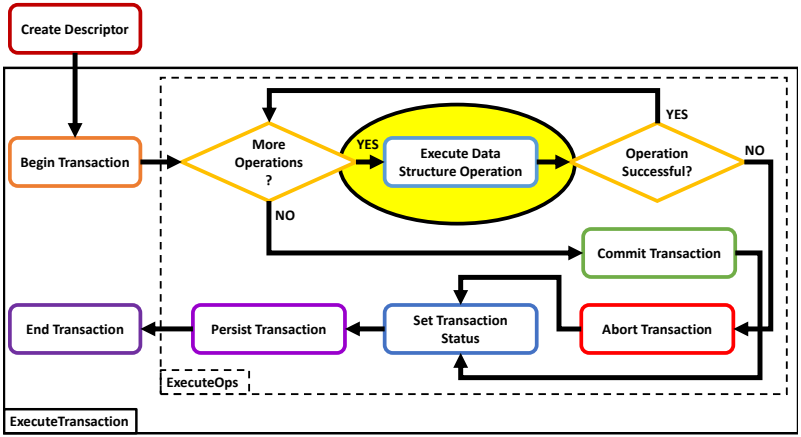


Figure 6: Execute Transaction

PETRA Methodology Overview

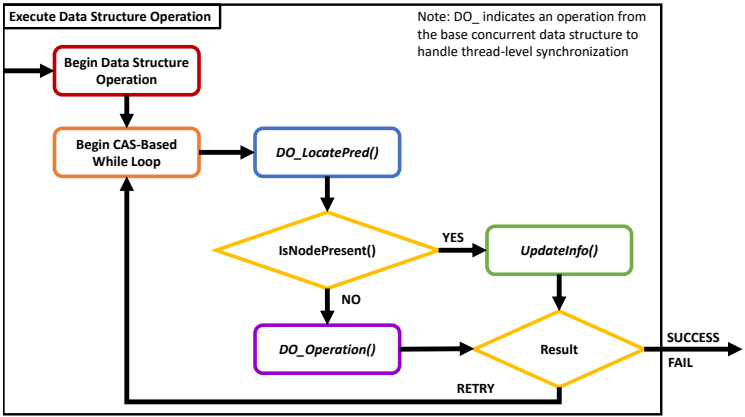


Figure 7: Execute Data Structure Operation

PETRA Methodology Example

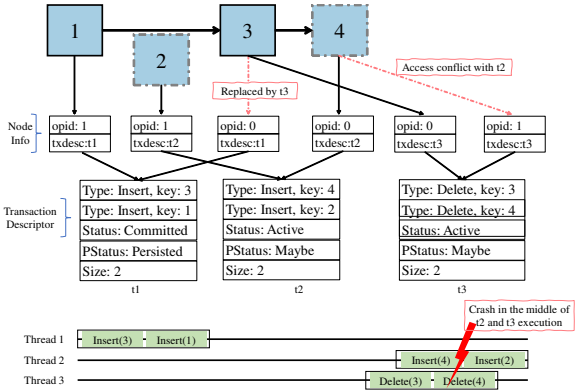


Figure 8: Transaction Descriptors for Conflict Detection, Durability

PETRA Recovery

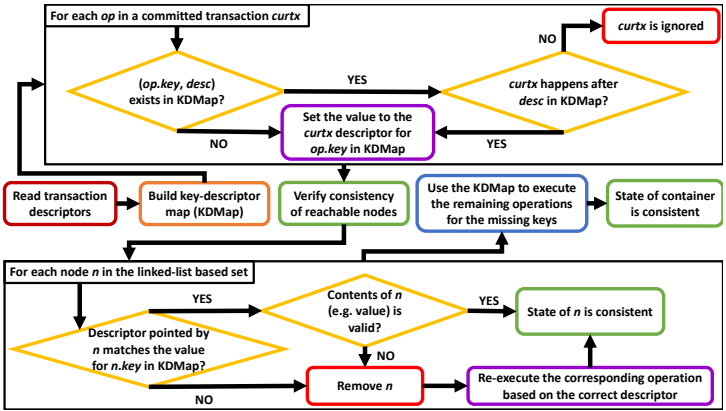


Figure 9: Recovery Steps

Experimental Setup

Machine Testbed

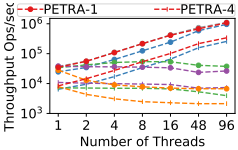
- Intel's second-generation Xeon Scalable processors (Cascade Lake)
  - 48 cores (2 sockets), supporting 96 threads
  - Main memory consists of Intel Optane DC Persistent Memory (DCPM) with 6TB total capacity, plus 768GB DRAM
    - Persistent data structures placed in the DCPM; DRAM is used to store everything else (e.g. code)
- The OS is Ubuntu 18.04 LTS
- The application and micro-benchmarks were compiled using gcc 7.4 with the -O3 optimization flag and C++14 standard flags

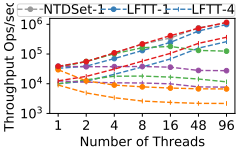
Experimental Setup

Micro-benchmarks

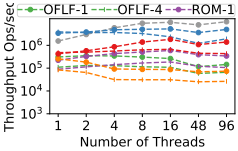
- Operation ratio for write-dominated workload
  - Lists: 40% Insert, 40% Delete, 20% Find
  - Map: 40% Insert, 30% Delete, 10% Update, 20% Find
- Operation ratio for read-dominated workload
  - Lists: 10% Insert, 10% Delete, 80% Find
  - Map: 10% Insert, 10% Delete, 5% Update, 75% Find
- Number of Transactions
  - Linked List: $100K$, Other Data Structures: $1M$
- Key Range
  - Linked List: $10K$, Other Data Structures: $1M$

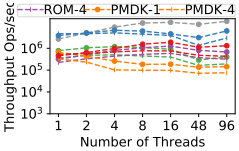
Performance Results





(a) Linked list: write-dominated (b) Linked list: read-dominated

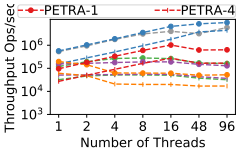


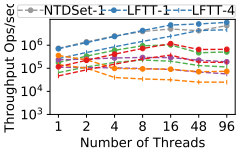


(c) Map: write-dominated (d) Map: read-dominated

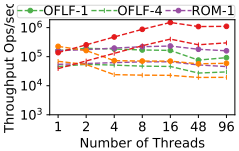
Figure 10: Throughput for transactional data structures for transactions of size 1 and 4.

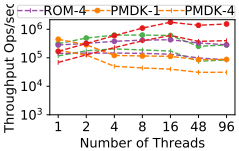
Performance Results





(e) Skiplist: write-dominated (f) Skiplist: read-dominated





(g) MDlist: write-dominated (h) MDlist: read-dominated

Figure 11: Throughput for transactional data structures for transactions of size 1 and 4.

Performance Results

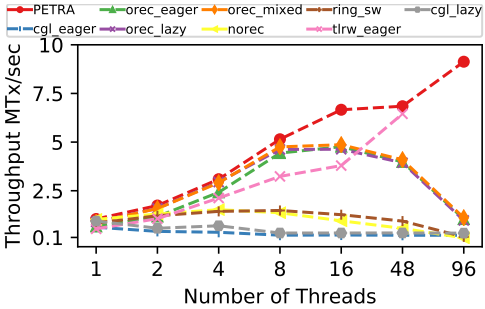


Figure 12: Performance comparison of PETRA with general-purpose PTMs in TATP benchmark.

Live Demonstration

Demonstration Settings

Processor

- AMD EPYC 7501 @ 2 GHz
  - Cores: 32, Logical Processors: 64

Compiler Options

- Use DRAM allocator
- Persistent Write-Back (PWB) is Cacheline Flush (CLFLUSH)

```
1  #ifdef PWB_IS_CLFLUSH
2    #define PWB(addr)      asm volatile("clflush (%0)" :: "r" (addr) : "memory")
3    #define PFENCE()       {}
4    #define PSYNC()        {}
5  #endif
```

Demonstration Settings

Micro-benchmarks

- Operation ratio: 33% Insert, 33% Delete, 34% Find
- Number of Transactions: $10K$
- Key Range: $10K$

Conclusion

Key Take-Aways

- Persistent memory provides a new tier of memory that is non-volatile and high capacity with access latencies close to DRAM
- The persistent hash map uses open addressing
  - Low memory overhead
  - Improved cache locality
- PETRA provides highly scalable durable transactions due to its high-level semantic conflict detection

Source Code

- PMap: https://github.com/ucf-cs/PMap
- PETRA: https://github.com/CLPeterson/PETRA-Non-Pmem

References I

[1] *Flash memory summit*, https://www.flashmemorysummit.com/opt_persistent_memory.html, Accessed: 10-6-2021.

[2] *Intel optane persistent memory*, https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html, Accessed: 10-6-2021.

[3] *3d nand stacking memory cells*, https://www.atpinc.com/blog/3d-nand-ssd-sd-flash-memory-storage-what-is, Accessed: 10-6-2021.

References II

[4] *Researchers scrutinize optane memory performance*, https://www.nextplatform.com/2019/03/18/researchers-scrutinize-optane-memory-performance/, Accessed: 10-6-2021.

[5] *What is dram's future?* https://semiengineering.com/what-is-drams-future/, Accessed: 10-6-2021.

[6] *Is optane dimm endurance good enough?* https://blocksandfiles.com/2019/04/04/enduring-optane-dimm-question-is-its-endurance-good-enough-yes-intel-has-delivered/, Accessed: 10-6-2021.

References III

[7] *Ssd lifespan: How long will your ssd work?* https://www.enterprisestorageforum.com/hardware/ssd-lifespan-how-long-will-your-ssd-work/, Accessed: 10-6-2021.

[8] *Dram 64gb*, https://www.newegg.com/p/pl?d=DRAM+64GB, Accessed: 10-7-2021.

[9] *Dram 128gb*, https://www.newegg.com/p/pl?d=Samsung+128GB+DIMM, Accessed: 10-11-2021.

[10] *Intel optane persistent memory*, https://www.newegg.com/p/pl?d=Optane+Persistent+Memory, Accessed: 10-7-2021.

References IV

[11] *Nand ssd*, https://www.newegg.com/p/pl?d=NAND+SSD, Accessed: 10-7-2021.

[12] *Intel's optane dimm price model*, https://thememoryguy.com/intels-optane-dimm-price-model/, Accessed: 10-7-2021.

[13] Persistent Programming in Real Life 2019 (PIRL 2019), Persistent Memory Evaluation and Experiments (https://www.youtube.com/watch?v=M_kCL1OZjko). Retrieved 3/22/2021.

[14] C. Click, "A lock-free wait-free hash table," *work presented as invited speaker at Stanford*, 2008.

References V

[15] T. Wang, J. Levandoski, and P.-A. Larson, "Easy lock-free indexing in non-volatile memory," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, IEEE, 2018, pp. 461–472.

[16] T. David, A. Dragojević, R. Guerraoui, and I. Zablotchi, "Log-free concurrent data structures," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, Boston, MA: USENIX Association, Jul. 2018, pp. 373–386, ISBN: 978-1-939133-01-4. [Online]. Available: `https://www.usenix.org/conference/atc18/presentation/david`.