SDC 18

September 24-27, 2018
Santa Clara, CA

www.storagedeveloper.org

# Concurrency on Persistent Memory:
## Designing Concurrent Data Structures for Persistent Memory

Sergei Vinogradov
Intel

# Legal Disclaimers and Optimization Notices

# Executive Summary

- Challenges of building concurrent data structures for persistent memory
- Two approaches to design concurrent data structures for persistent memory

# Agenda

- Motivation
- Two approaches for data consistency
  - Transaction approach
  - Atomic approach
- Concurrent hash map for persistent memory
- Integration with PMEMKV
- Summary

# Goal

- Design concurrent data structures for persistent memory.
  - Data structure resides in persistent memory.
  - Operations are thread-safe.
  - Operations are fault-tolerant.
    - Data survive unexpected crashes and power failures

# Motivation

- PMDK provides low-level API for persistent memory programmers.
- Developers think in terms of data structures and algorithms.

# What’s Done

- Evaluated two approaches to support data consistency in concurrent data structures.
- Redesigned Intel® Threading Building Blocks (Intel TBB) **concurrent_hash_map** for persistent memory.
  - Published as part of PMDK
    https://github.com/pmem/libpmemobj-cpp/pull/40
- Integrated our data structures into PMEMKV.

# Two Approaches for Data Consistency

- Transactions
  - Define a set of operations to be done atomically.
- Atomic approach
  - Atomically switch between consistent states.

# Transactions for Data Consistency

```
template<class T>
class list {
    struct node_t {
        T value;
        persistent_ptr<node_t> next;
    };

    persistent_ptr<node_t> head;
    persistent_ptr<node_t> tail;

public:
    void push_back( T v ) {
        transaction::manual( pop );
        auto n = make_persistent<node_t>( v, nullptr );
        if (head == nullptr) {
            head = tail = n;
        } else {
            tail->next = n;
            tail = n;
        }
        transaction::commit();
    }
};
```

Transaction scope

- All modifications are done inside a transaction.
- Write-Ahead Log (WAL) is used to track modifications.
- For each uncommitted transaction, consistent state is restored on restart from WAL.

# Concurrent PMDK Transactions

- PMDK transaction does not support isolation.
  - WAL is per-thread.
- Atomic operations cannot be used inside PMDK transactions.
- Possible solutions:
  - Use critical section.
  - Hold lock until transaction completed.

```
// Variable a is located in persistent memory
std::atomic<int> a = 0;
```

**Thread 1:**

```
tx_begin();
add_to_log(&a);
++a;
…
tx_abort();
```

WAL: a = 0

**Thread 2:**

```
tx_begin();
add_to_log(&a);
++a;
…
tx_commit();
```

WAL: a = 0

Incorrect value of the counter is restored from undo log if thread 1 aborts transaction while thread 2 successfully commits its changes.

# Mutex in Persistent Memory

- Mutex is a volatile entity.
  - Unexpected termination may leave mutex in the locked state.
  - Mutex must be re-initialized to the unlocked state on each process restart.
- PMDK solves this issue:
  - Persistent mutex
    - Unlocked on each process restart.
  - Volatile field in a persistent data structure
    - Re-initialized on each process restart.

```
class Foo {
  pmem::obj::mutex mtx;
};
```

```
class Bar {
  pmem::obj::v<std::mutex> m;
};
```

# Atomic Approach for Data Consistency

```
template<class T>
class list {
    struct node_t {
        T value;
        persistent_ptr<node_t> next;
    };

    persistent_ptr<node_t> head;
    persistent_ptr<node_t> tail;
    persistent_ptr<node_t> new_node; // Hold last allocated node
public:
    void push_back( T v ) {
        make_persistent_atomic<node_t>( pop, new_node, v, nullptr );
        if (head == nullptr) {
            head = tail = new_node;
            pop.persist( head ); // Cache flush
        } else {
            tail->next = new_node;
            pop.persist(tail->next); // Cache flush
            tail = new_node;
        }
        pop.persist( tail ); // Cache flush
    }

    void restore() {
        if (new_node) {
            // Restore logic
        }
    }
};
```

- Alternative approach to PMDK transactions
- Custom restore logic
- PMDK atomic allocator
- Manual cache flushes

# PMDK Atomic Memory Allocator

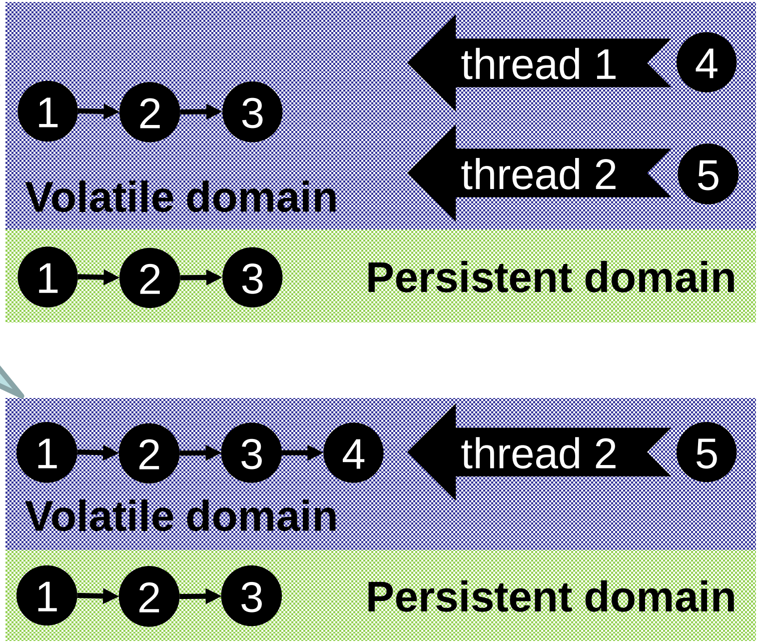
- PMDK atomic allocator atomically does the following:
  - allocates memory;
  - assigns the result to a user-provided persistent pointer.
- Persistent pointer on the stack cannot be used.
  - Persistent memory leak is possible if a process is terminated.

```
int main() {
 pool_base pop = …;
 persistent_ptr<Foo> p = nullptr;
 make_persistent_atomic<Foo>(pop, p);
 return 0;
}
```

# Lock-Free Algorithms on Persistent Memory

- Memory subsystem consists of:
  - **Volatile domain** - registers, caches
  - **Persistent domain** – DIMMs.
- Changes made by one thread are visible to other threads before it is persisted.
  - CMPXCHG + CLWB – not atomic
- Restore after an abnormal termination should take care of such cases.

**Compare-and-swap** is used to insert an element to the tail

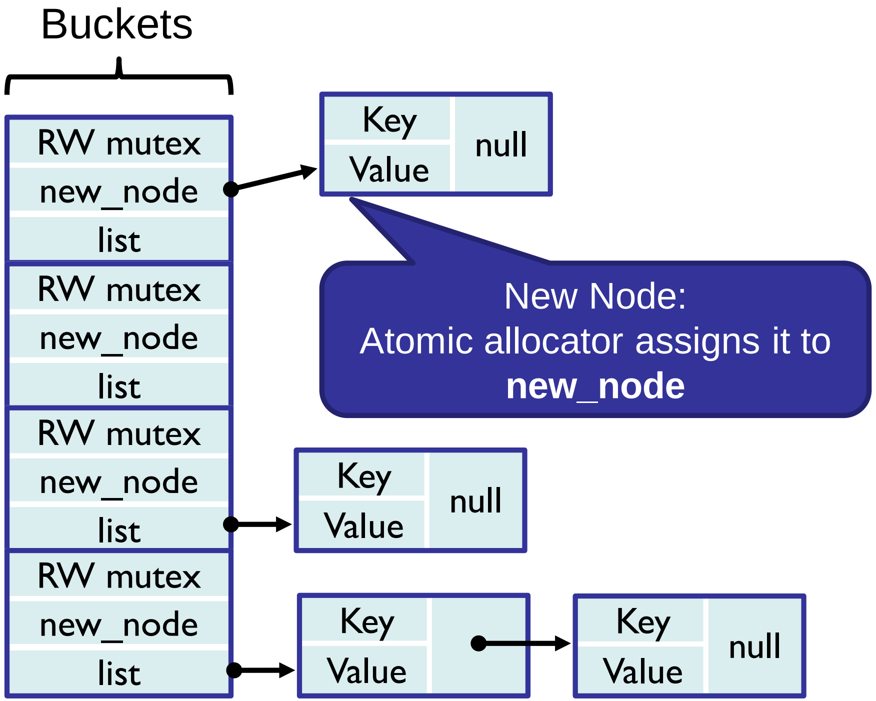


# Transactions vs. Atomic Approach

| | Transactions | Atomic approach |
|---|---|---|
| Data consistency support | • Natural way to support data consistency<br>• PMDK is responsible for data consistency and data restore after crash. | • Developer is responsible for data consistency.<br>• Custom restore logic is required for each particular data structure. |
| Performance overhead | • Performance overhead to handle WAL | • Better performance |
| Concurrency support | • PMDK transaction does not support isolation<br>• Cannot use lock free algorithms | • Suitable for concurrent algorithms |

# Concurrent Hash Map

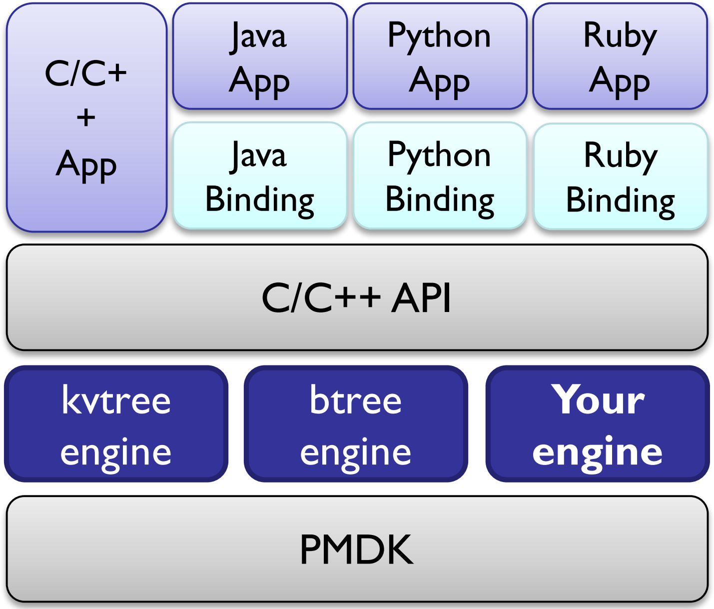
- Per-bucket Read-Write lock
  - Find() acquires read lock.
  - Insert() acquires write lock.
- Insert operation:
  - Finds bucket.
  - Allocates new node.
  - Inserts a new node.
- Lazy restore on bucket access
  - Checks new_node pointer first.



# PMEMKV

https://github.com/pmem/pmemkv

- Embedded Key/Value data store optimized for persistent memory
- An option to create custom storage engines
- Usage of **db_bench** from RocksDB for benchmarking



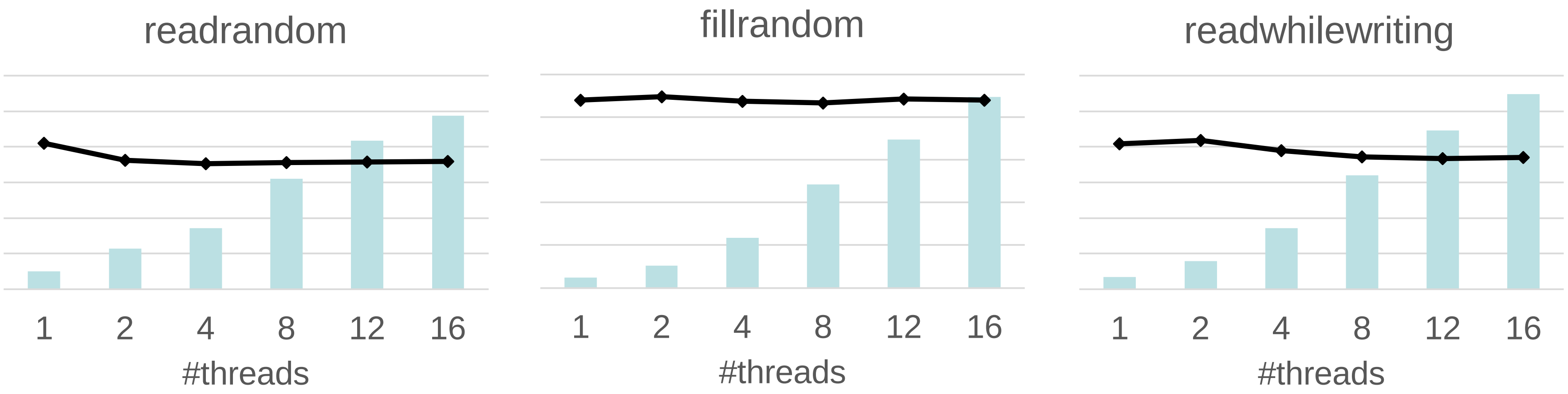
# Concurrent Hash Map and PMEMKV

Concurrent hash map is a new storage engine that:

- enables maximum throughput in multithreaded applications;
- keeps P99 latency flat with a growing number of threads.

# DB_bench Results

Results have been projected or simulated using internal Intel analysis or architecture simulation or modeling, and provided to you for informational purposes. Any differences in your system hardware, software or configuration may affect your actual performance. Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.




Throughput (Ops/sec) – scales with a number of threads

P99 latency (sec/Op) – flat

# Summary

- Compared two approaches to build concurrent data structures for persistent memory:
  - Transactions vs. Atomic approach.
- Designed a concurrent hash map for persistent memory.
- Enabled concurrency in PMEMKV.

# Call to Action

- Try our data structures in your persistent memory workloads: https://github.com/pmem/libpmemobj-cpp
- Try PMEMKV in your C/C++, Java*, Python* apps
  - Customizable Key/Value data storage: https://github.com/pmem/pmemkv
- Provide your feedback.

# Thank You

- Questions?