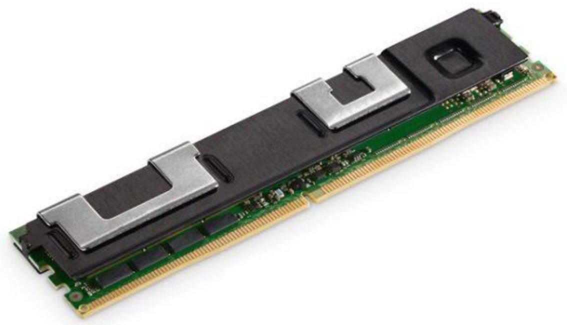
Persistent Memory and the Rise of Universal Constructions

Eurosys 2020

Andreia Correia – *University of Neuchâtel*
Pascal Felber – *University of Neuchâtel*
Pedro Ramalhete – *Cisco Systems*

Persistent Memory



Persistent Memory (or Non-Volatile Main Memory) is a durable media that can be **accessed through load and store instructions**.

Physically, it fits into a DIMM slot

Solutions exist for several years by HPE, Micron and Viking, but all these are battery backed:

https://www.vikingtechnology.com/products/nvdimm/

https://www.hpe.com/nl/en/servers/persistent-memory.html

https://www.micron.com/campaigns/persistent-memory

A year ago, Intel released the **Optane DC Persistent Memory** which does not require a battery. Capacities go up to 512 GiB per module, and 3 TB per CPU socket.

https://arxiv.org/pdf/1903.05714.pdf

Persistent data structures

Some of the reasons that make persistent data structures a difficult topic, are:

- Where to place the **flushes** (CLWBs) and **fences** (SFENCE)
- How to write a correct **recovery procedure**
- How to **allocate and de-allocate** persistent objects efficiently, without leaking
- How to **modify** existing persistent data structures to suit novel business needs

How to make a *concurrent* and *persistent* data structure

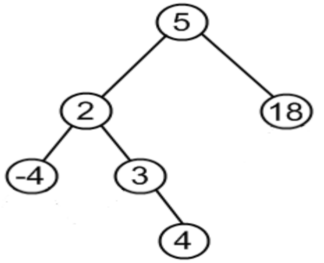
Make a data structure **by hand**	**Locks + cow/undo/redo log** Many different papers		✖ Complex to design and modify Blocking
	Lock-free persistent queue Friedman et al, PPoPP 2018	✔ Lock-Free	✖ Difficult to make other ADTs
Use a technique that transforms existing **Lock-Free** data structures	**pwb/pfence/psync recipe** Izraelevitz et al, DISC 2018	✔ Lock-Free Easy to deploy	✖ Slow
	Capsules Blelloch et al, SPAA 2018	✔ Lock-Free	✖ Difficult to deploy
	NVTraverse Ben-David et al, PLDI 2020	✔ Lock-Free Fast for reads	✖ Difficult to deploy
Use a PTM that transforms **Sequential** data structures	**Mnemosyne** Volos et al, ASPLOS 2016	✔ Scales for writes Easy to deploy	✖ Unstable Blocking
	libpmemobj (PMDK) Intel	✔ Easy to deploy	✖ No concurrency Slow
	OneFile Ramalhete et al, DSN 2019	✔ Wait-Free Easy to deploy	✖ Writes don't scale

How to make a ***concurrent*** and ***persistent*** data structure

Make a data structure **by hand**

Use a technique that transforms existing **Lock-Free** data structures

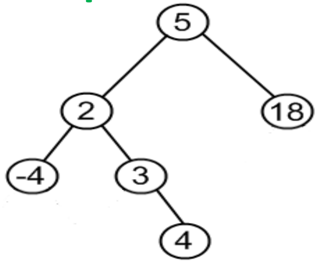
Lock-Free DS



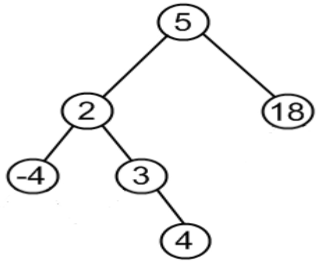
1. Precede every CAS with a flush
2. Flush and fence after every load
3. …

Use a PTM that transforms **Sequential** data structures

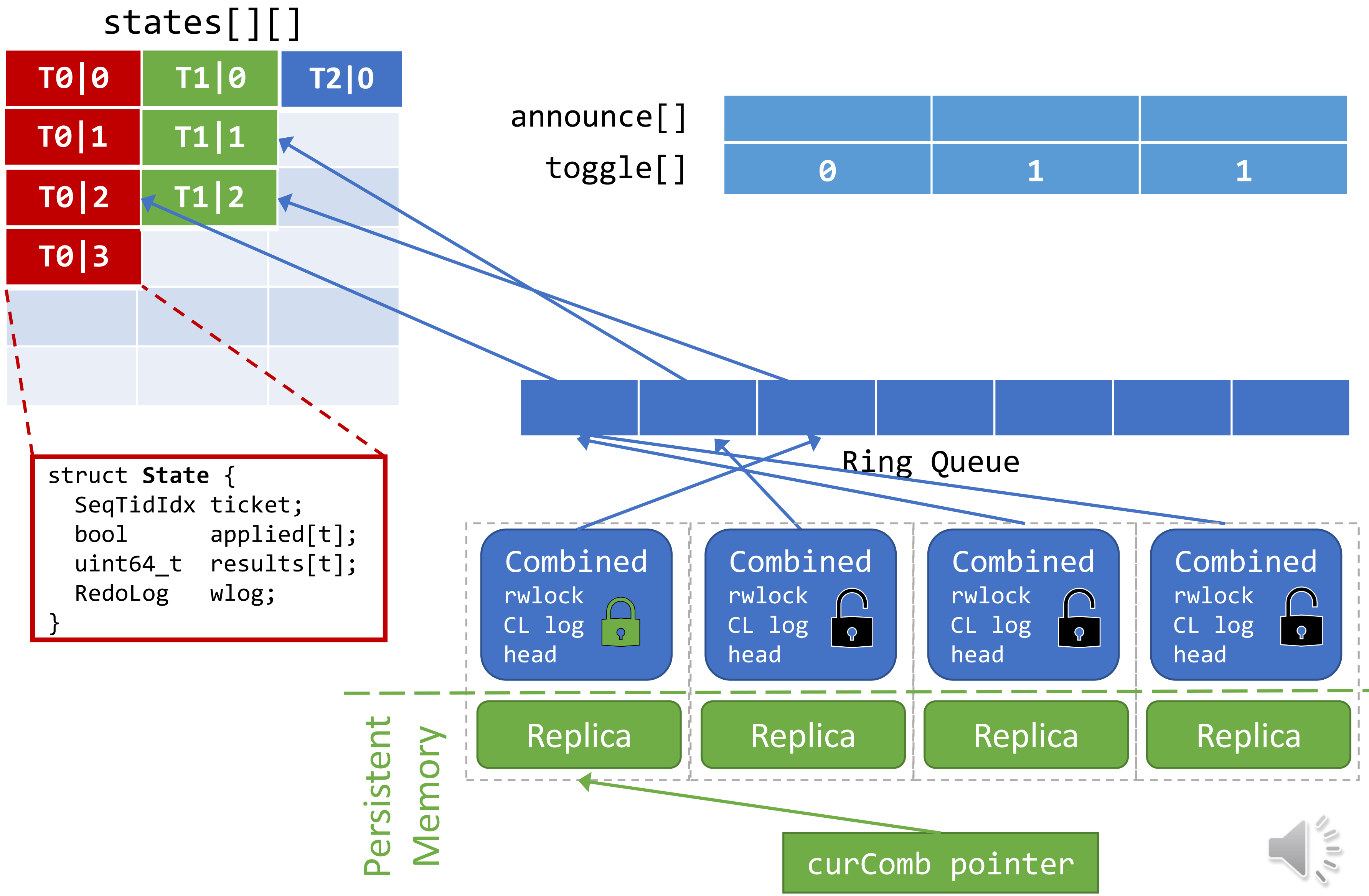
Sequential DS



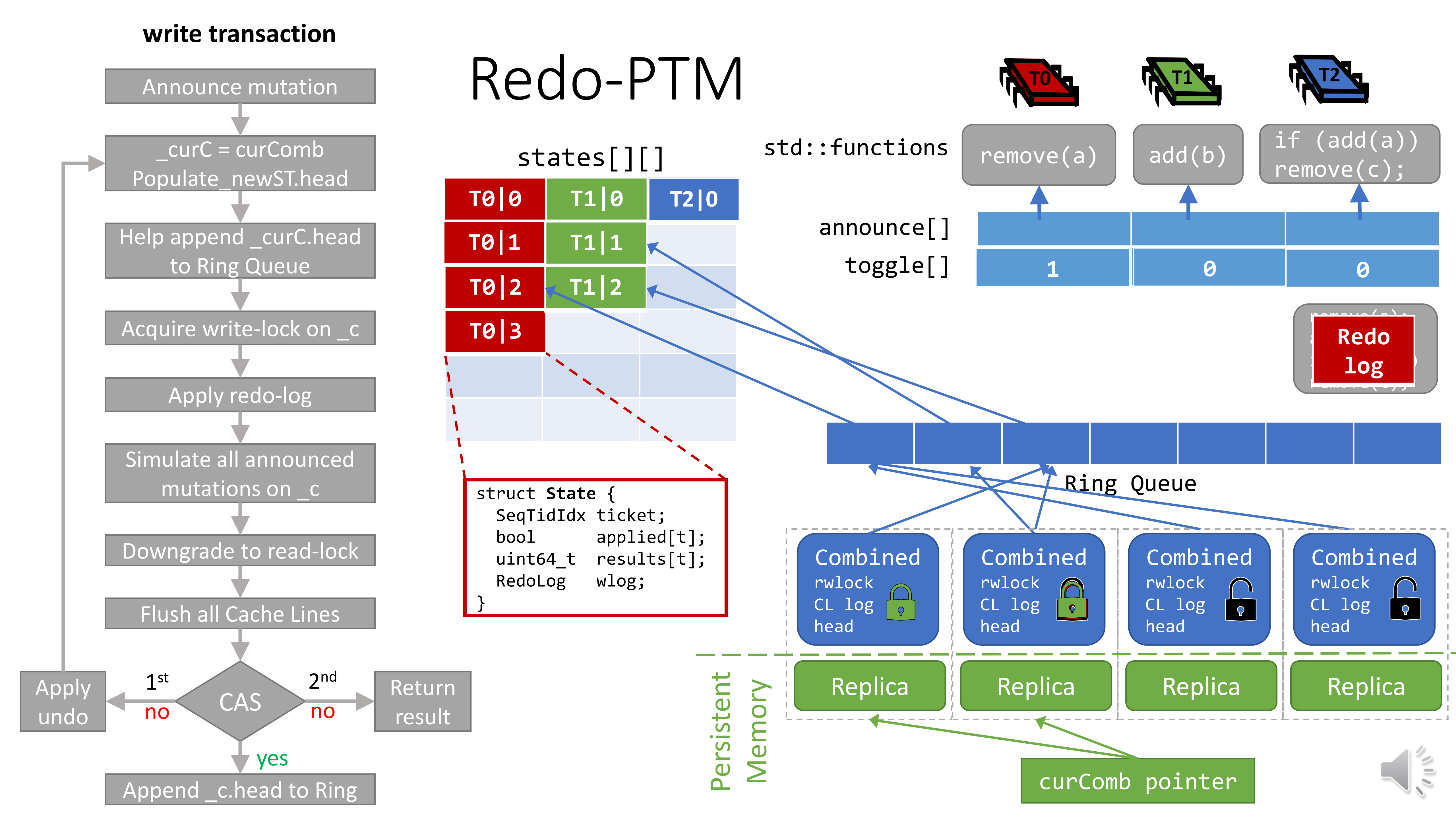
Concurrent and **Persistent** DS



Redo-PTM



states[][]
T0|0
T1|0
T2|0
T0|1
T1|1
T0|2
T1|2
T0|3
announce[]
toggle[]
0
1
1
Ring Queue
struct State {
SeqTidIdx ticket;
bool applied[t];
uint64_t results[t];
RedoLog wlog;
}
Combined
rwlock
CL log
head
Combined
rwlock
CL log
head
Combined
rwlock
CL log
head
Combined
rwlock
CL log
head
Persistent Memory
Replica
Replica
Replica
Replica
curComb pointer


write transaction
Announce mutation
_curC = curComb
Populate _newST.head
Help append _curC.head to Ring Queue
Acquire write-lock on _c
Apply redo-log
Simulate all announced mutations on _c
Downgrade to read-lock
Flush all Cache Lines
CAS
1st
no
2nd
no
yes
Apply undo
Return result
Append _c.head to Ring
Redo-PTM
T0
T1
T2
states[][]
T0|0
T1|0
T2|0
T0|1
T1|1
T0|2
T1|2
T0|3
struct State {
SeqTidIdx ticket;
bool applied[t];
uint64_t results[t];
RedoLog wlog;
}
std::functions
remove(a)
add(b)
if (add(a)) remove(c);
announce[]
toggle[]
1
0
0
Redo log
Ring Queue
Combined
rwlock
CL log
head
Combined
rwlock
CL log
head
Combined
rwlock
CL log
head
Combined
rwlock
CL log
head
Persistent Memory
Replica
Replica
Replica
Replica
curComb pointer

Wait-Free PTM
Comparison table

	OneFile PTM	CX PTM	Redo PTM
Maximum number of instances in use	1	2 t	t + 1

t = total number of threads in the system

What makes Redo-PTM fast

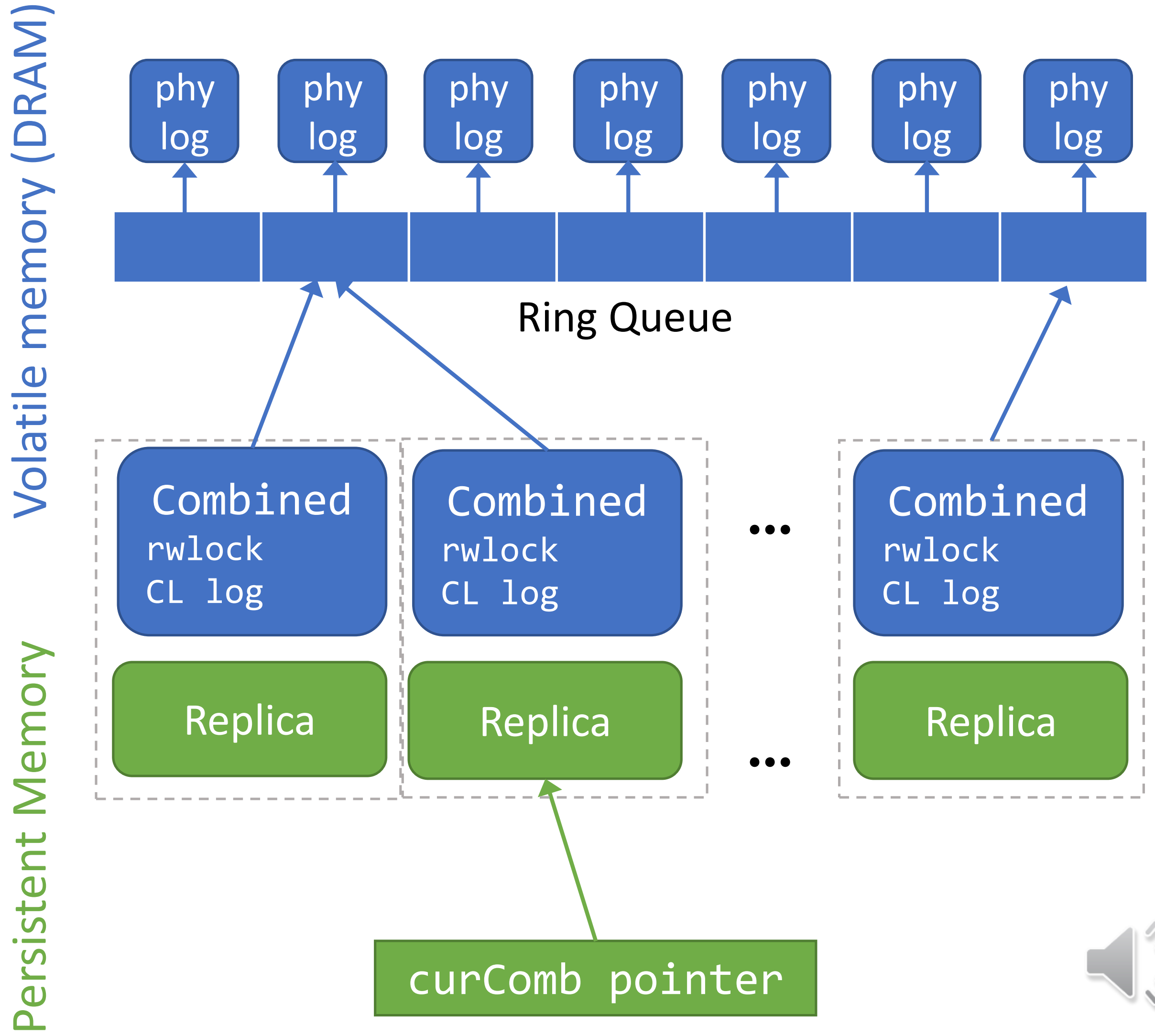
1. Volatile physical logging
2. Store aggregation
3. Flush aggregation
4. Flush deferral
5. Replica copies with non-temporal stores

What makes Redo-PTM fast
1. *Volatile Physical Logging*

In Redo-PTM, the `curComb` variable and the instances (replicas) associated with each `Combined`, are located in **persistent** memory.

All other components are in volatile memory (DRAM) which is much faster than PM:

- Ring Queue and combining consensus
- Physical log of modifications (and intrusive hashmap)
- Combined instances: Log of modified cache lines, reader-writer lock, root pointer, head pointer (which points to an entry in the Ring Queue).



What makes Redo-PTM fast
2. *Store aggregation*

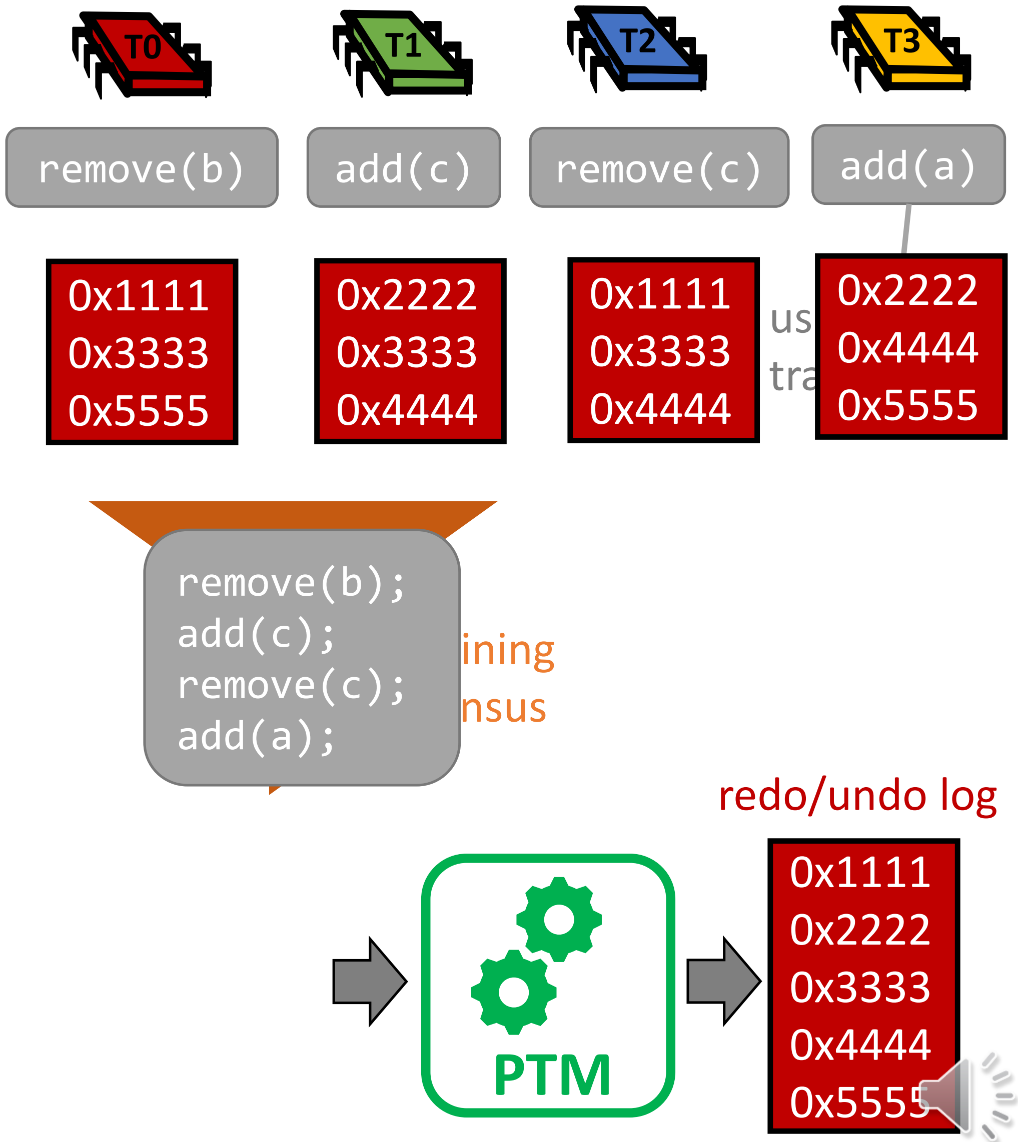
Classic redo-log PTMs (like Mnemosyne) transform the transaction from each thread into a physical redo log.

In Redo-PTM, we use the *combining consensus* to aggregate the operations from multiple in-flight threads, into a single redo/undo log.

With a large number of threads, the likelihood increases that many operations will touch the same addresses.

Each address is written into, a single time, **reducing write amplification**.

Also, in classic redo-log the log is persistent. In Redo-PTM the redo-log is volatile.



What makes Redo-PTM fast
3. *Flush aggregation*

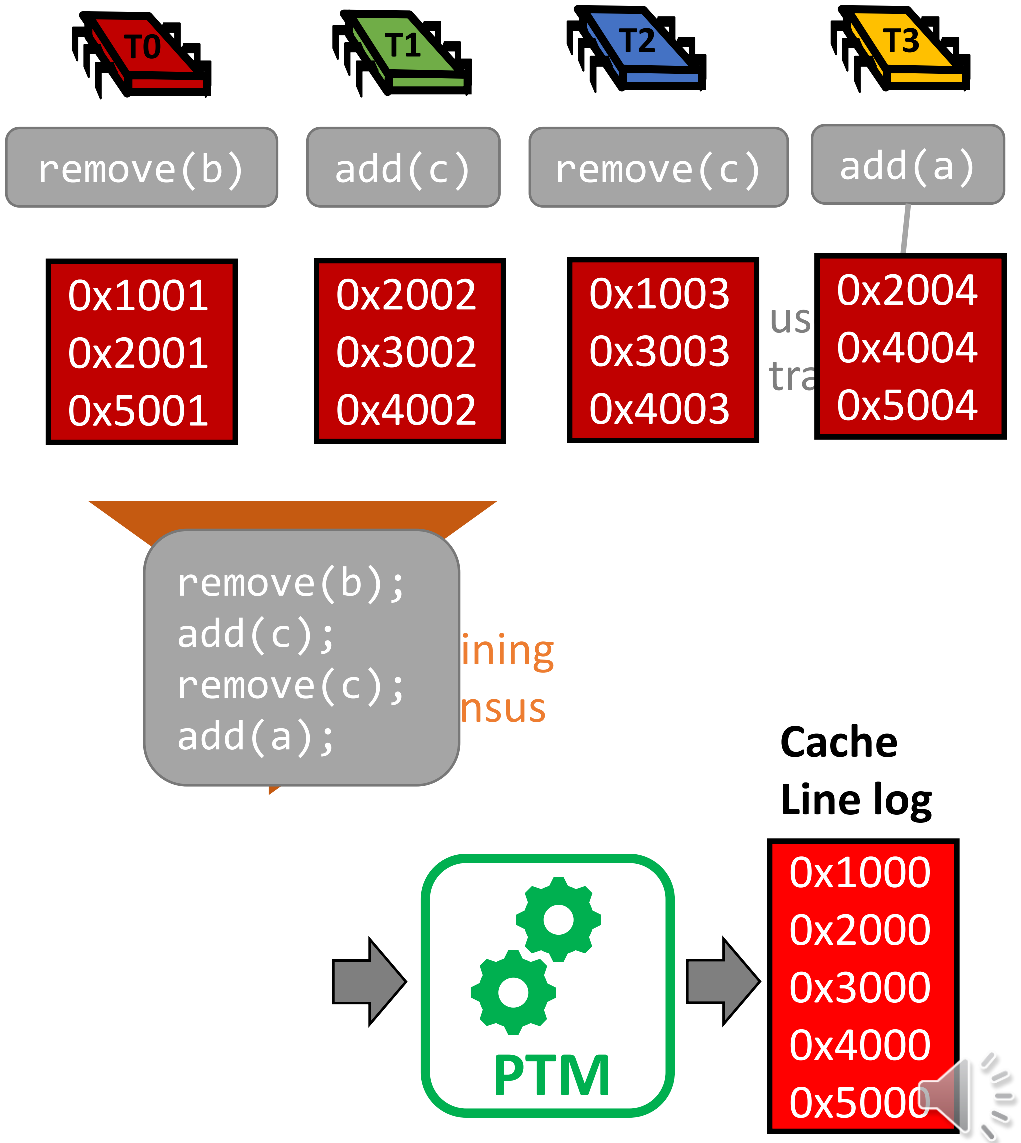
Classic redo-log PTMs (like Mnemosyne and OneFile) flush the **persistent** redo log, and later flush each modified cache line in memory.

In Redo-PTM, the *combining consensus* aggregates the operations from multiple in-flight threads, and the Redo PTM creates a **volatile** redo log and a **volatile** cache line log.

With a large number of threads, the likelihood that many operations will touch the same cache lines is higher.

This is particularly true for allocator metadata modifications.

Each cache line is flushed a single time, improving performance.



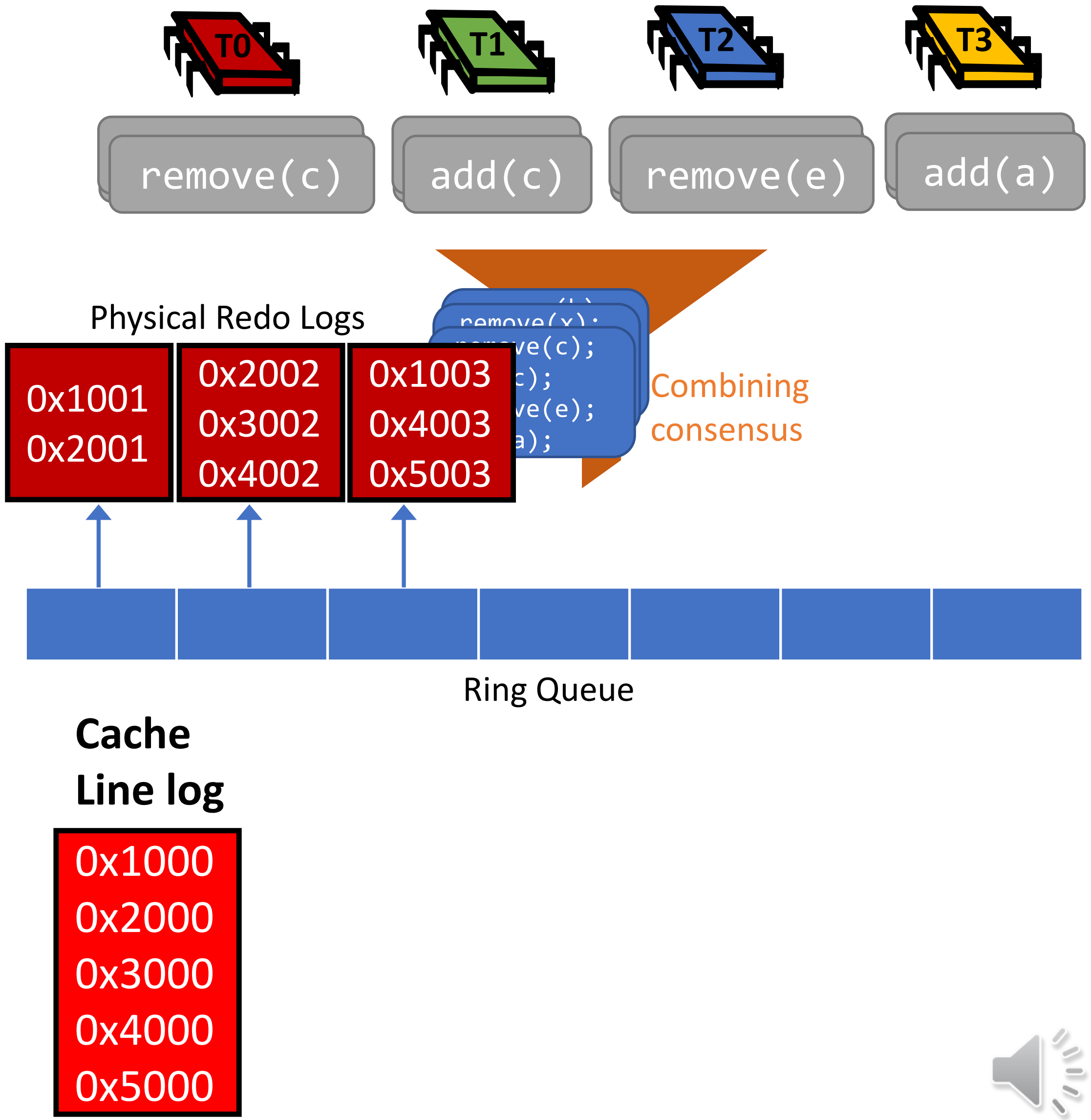
What makes Redo-PTM fast
4. *Flush deferral*

In Redo-PTM, a thread executes modifications on its own private instance and only issues the flushes immediately before attempting to change `curComb` with a CAS.

If another thread has in the meantime changed `curComb`, then no flushes are issued. The Cache Line log remains associated with a replica, for another thread to later aggregate further modifications.

This technique allows Redo-PTM to **aggregate flushes across consecutive transactions**.

If the Cache Line log grows beyond 1/10 of the number of cache lines in the replica, we clear the log and set a flag to flush the entire replica (before becoming the next `curComb`).

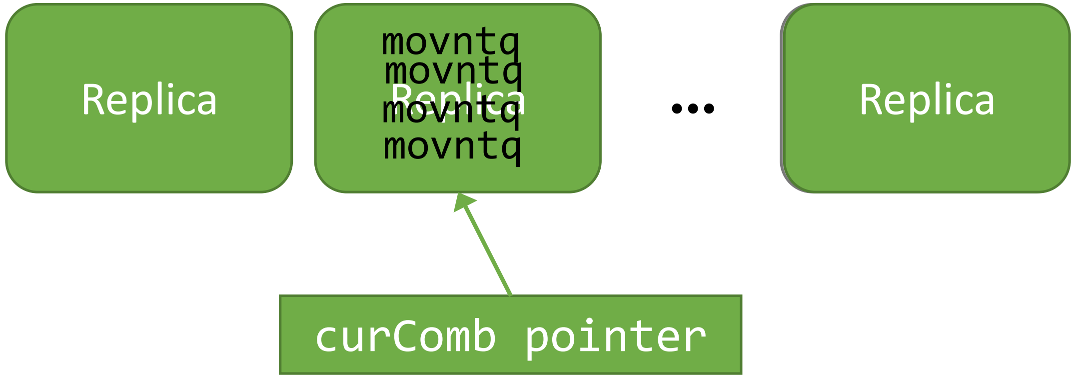


What makes Redo-PTM fast

5. *Replica copy with non-temporal stores*

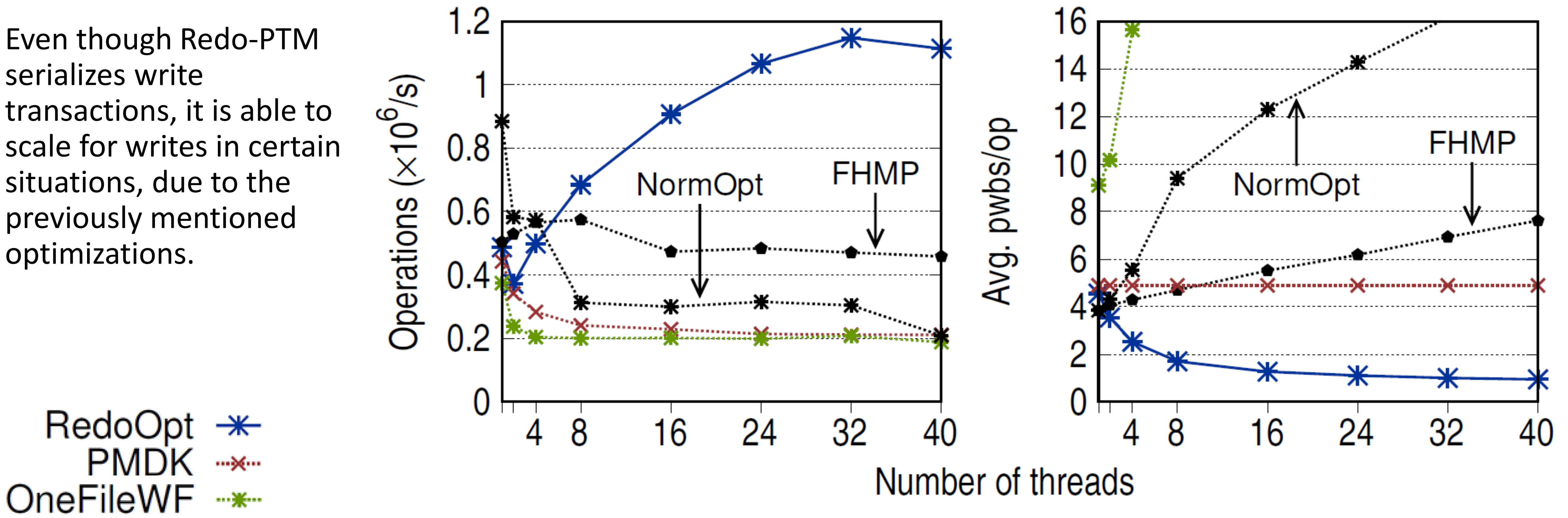
In Redo-PTM, when a full copy of the replica needs to be made, instead of doing a `memcpy()` and then flushing the entire range, **we use non-temporal stores to execute the copy** and forego the need to issue `CLWB` instructions.

This approach provides and extra improvement in performance for such (rare) large copies.



Sequential Linked List Queue transformed into a Wait-Free Persistent Queue

Even though Redo-PTM serializes write transactions, it is able to scale for writes in certain situations, due to the previously mentioned optimizations.



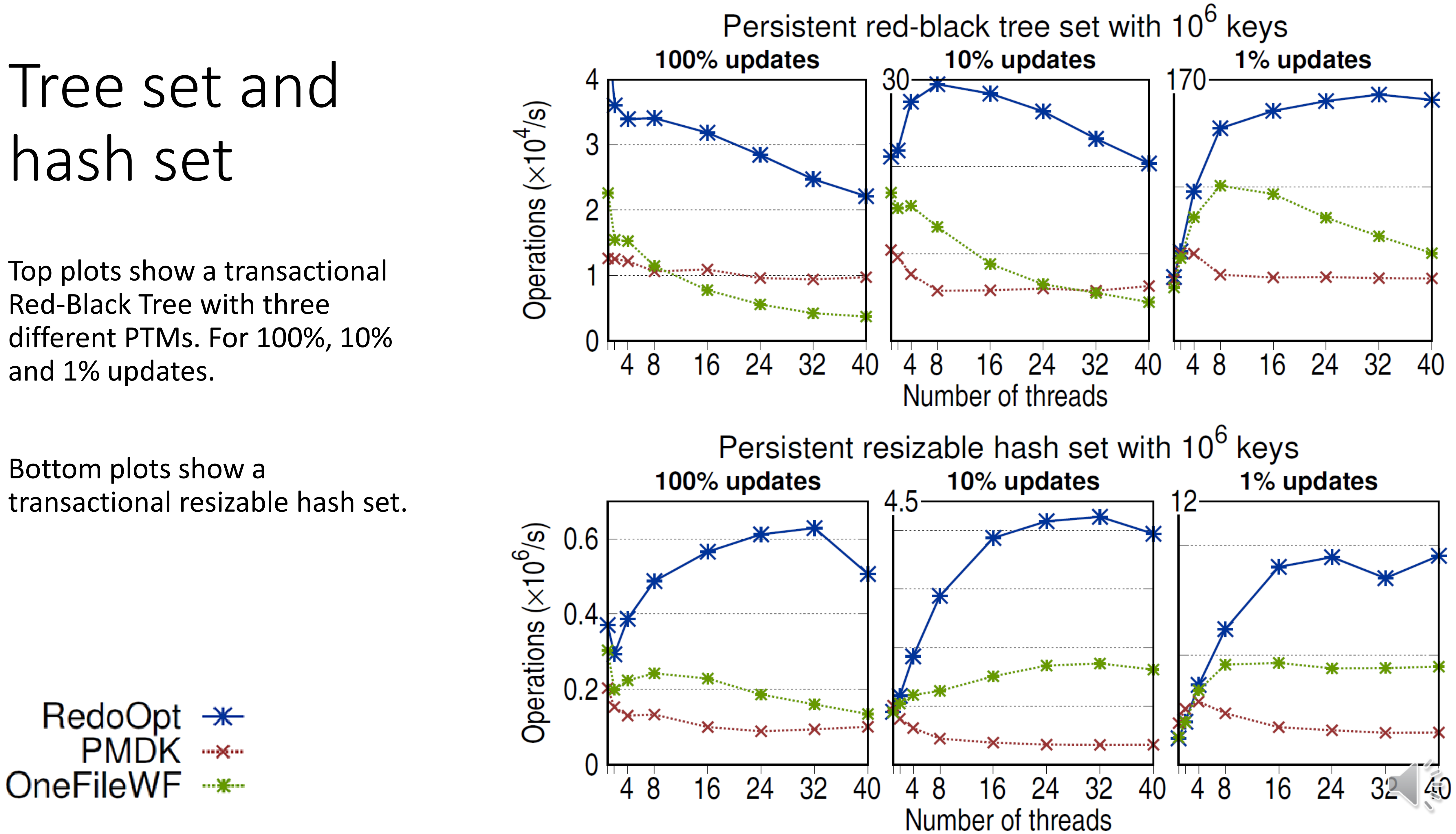
FHMP: Friedman et al, PPoPP 2018

NormOpt: Ben-David et al, SPAA 2019

Tree set and hash set

Top plots show a transactional Red-Black Tree with three different PTMs. For 100%, 10% and 1% updates.

Bottom plots show a transactional resizable hash set.



Sequential queue annotated to be used with Redo-PTM (**wait-free** and persistent)

```cpp
bool enqueue(T* item) {
    return TM::template updateTx<bool>([=] () {
        Node* newNode = TM::template tmNew<Node>(item);
        tail->next = newNode;
        tail = newNode;
        return true;
    });
}


T* dequeue() {
    return TM::template updateTx<T*>([=] () -> T* {
        Node* lhead = head;
        if (lhead == tail) return nullptr;
        head = lhead->next;
        TM::tmDelete(lhead);
        return head->item;
    });
}
```

Handmade queue (**lock-free** and persistent)

```cpp
void enqueue(T item, const int tid) {
    Node* node = internalNew<Node>(item);
```

```cpp
T dequeue(const int tid) {
    T* newReturnedValue = internalNew<T>();
```

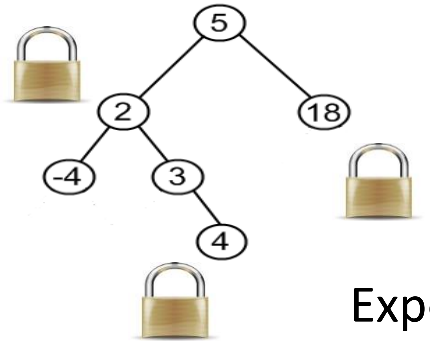
```cpp
        } else {
            T value = next->value;
```

```cpp
void recover() {
    if (destructorInProgress) {
        if (head.load(std::memory_order_relaxed) != nullptr) {
            while (dequeue(0) != EMPTY); // Drain the queue
            head.store(nullptr, std::memory_order_relaxed);
            PWB(&head);
            PFENCE();
            internalDelete(head.load(std::memory_order_relaxed));  // Delete the last node
        }
        PSYNC();
        return;
    }
    hp = new HazardPointers<Node>{2, maxThreads,mydeleter};
    // If both head is null then a failure occurred during constructor
    if (head.load(std::memory_order_relaxed) == nullptr) {
        Node* sentinelNode = internalNew<Node>(T{});
        head.store(sentinelNode, std::memory_order_relaxed);
        PWB(&head);
        PFENCE();
    }
    // If tail is null, then fix it by setting it to head
    if (tail.load(std::memory_order_relaxed) == nullptr) {
        tail.store(head.load(std::memory_order_relaxed), std::memory_order_relaxed);
        PWB(&tail);
        PFENCE();
    }
    // Advance the tail if needed
    Node* ltail = tail.load(std::memory_order_relaxed);
    Node* lnext = ltail->next.load(std::memory_order_relaxed);
    if (lnext != nullptr) {
        tail.store(lnext, std::memory_order_relaxed);
        PWB(&tail);
    }
    PSYNC();
}
```

How KV stores are made today…

Two-Phase Locking
(+ MVCC)

Concurrent Indexing
Data Structure



+

Redo Log

+

Years of
development

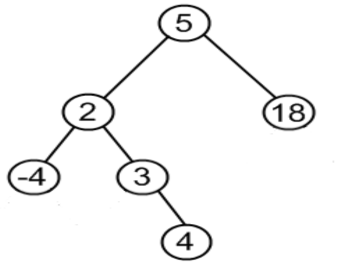
=

KV Store
(blocking)

Expert Developers in Concurrency,
Durability and DBs

How we made a KV store with Redo-PTM

Sequential Indexing Data Structure



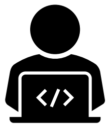
+

Redo-PTM

+

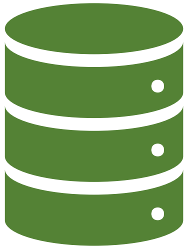
Months of development

Expert DB Developer



=

Redo DB

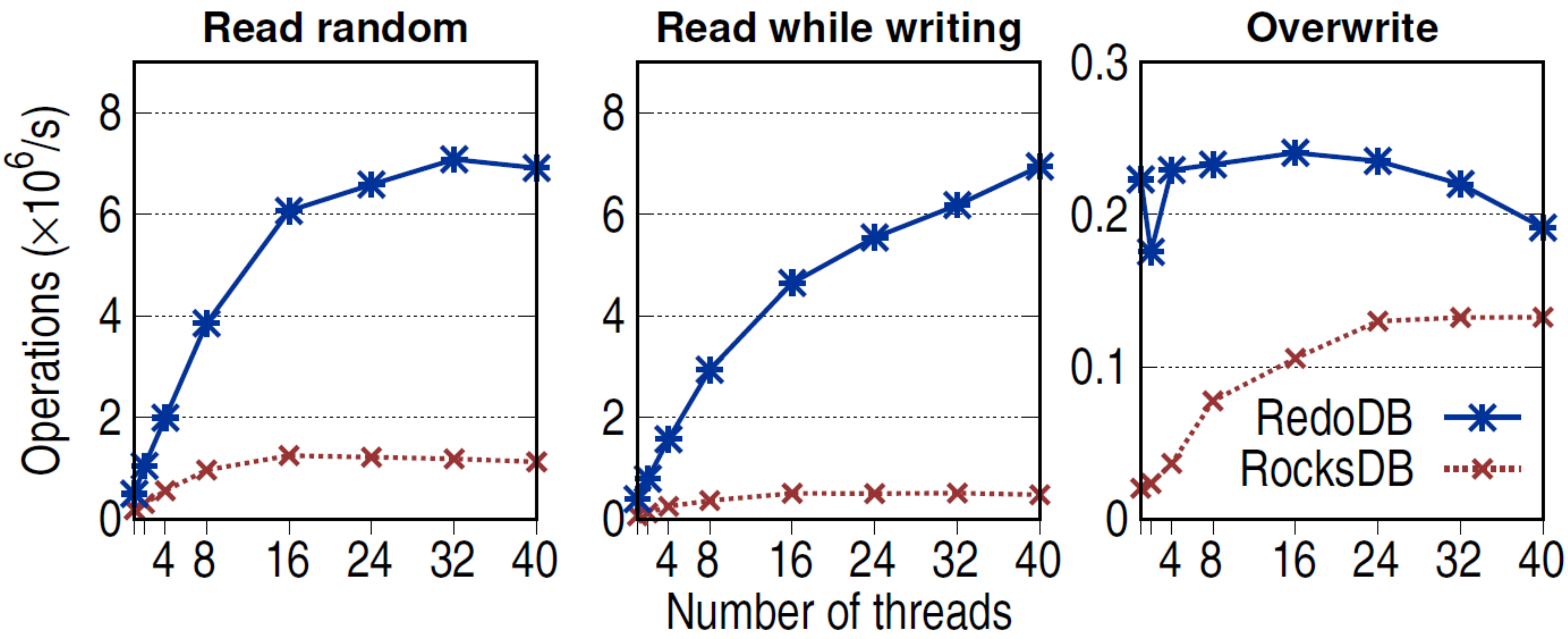


- Wait-Free progress
- Null recovery
- Non-abortable reads
- ACID transactions

RedoDB - KV Store

Because of the non-abortable reads, read-only transactions scale regardless of the existence or not of ongoing write transactions

DB with 10 million keys.



End

Thank you for watching

More links at the Eurosys 2020 program page:

https://www.eurosys2020.org/program/

https://dl.acm.org/doi/abs/10.1145/3342195.3387515

Source code available at:

https://github.com/pramalhe/RedoDB